EUROGRAPHICS 2002 / G. Drettrakis and H.-P. Seidel
(Guest Editors)

Volume 21 (2002), Number 3

Transparency in Interactive Technical Illustrations

J. Diepstraten D. Weiskopf T. Ertl

Visualization and Interactive Systems Group, University of Stuttgart

Abstract

This paper describes how technical illustrations containing opaque and non-opaque objects can be automatically generated. Traditional methods to show transparency in manual drawings are evaluated to extract a small and effective set of rules for computer-based rendering of technical illustrations, leading to a novel view-dependent transparency model. We propose a hardware-accelerated depth sorting algorithm in image-space which specifically meets the requirements of our transparency model. In this way, real-time rendering of semi-transparent technical illustrations is achieved. Finally, it is described how our approach can be combined with other methods in the field of non-photorealistic rendering in order to enhance the visual perception of technical illustrations.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Color, shading, shadowing and texture

1. Introduction

Although a lot of research in computer graphics has been conducted on photorealistic rendering, manuals, advertisements, text and science books still make great use of non-photorealistic illustrations. A major advantage of technical illustrations is that they provide a selective view on important details while extraneous details can be omitted[6]. Technical illustration are better suited to communicate the shape and structure of complex objects and they provide an improved feeling for depth, occlusion, and spatial relationships[2, 3, 4].

This paper is focused on one specific problem in color-shaded technical illustrations: transparency. Neither has transparency yet been addressed in automatic technical illustrations, nor has it been addressed extensively in other fields of non-photorealistic rendering (NPR). To the authors' knowledge, the paper by Hamel et al.[13] is the only work specifically dealing with transparency in NPR. They concentrate on transparency in line drawings, whereas this paper is focused on illustrations consisting of smoothly shaded, colored surfaces.

It is quite remarkable that transparency is widely neglected in computer-based illustrations because books on traditional manual illustrations do provide effective techniques and rules for handling transparency[14, 29] in order to communicate the location of occluding and occluded objects. In this paper, some of these rules are presented and adapted to allow for computer-generated images. Here, we introduce the concept of view-dependent transparency,

Figure 1: *An example of traditional technical drawing, showing transparency effects according to Hodges' rules[14]. (taken from Maier[21]).*

which is widely used in traditional technical illustrations, into the field of computer graphics. Furthermore, we propose two different hardware-accelerated depth sorting algorithms in image-space which meet the requirements of our transparency model. This model can be applied to existing approaches for non-transparent technical illustrations in order to further enhance the visual perception.

The paper is organized as follows. The subsequent section focuses on related and previous work. Section 3 discusses traditional techniques for color-shaded technical illustrations. In the following section, a short overview of our rendering approach is presented. Section 5 describes the concept of view-dependent transparency and how it can be implemented. In Section 6, hardware-accelerated depth sorting algorithms in image-space are presented. Section 7 shows results and performance measurements. The paper closes with a brief conclusion and an outlook on possible future work.

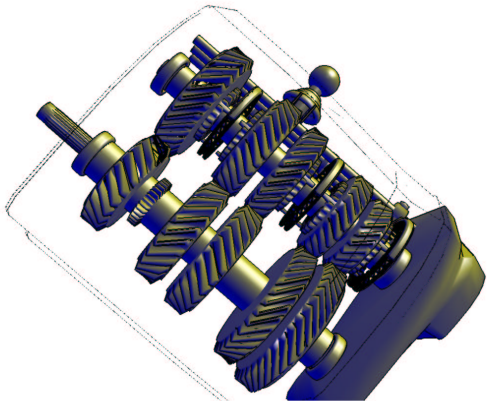


Figure 2: *Phantom lines showing transparent objects.*

2. Related and Previous Work

Although transparency in NPR has not been considered very widely yet, there are still some papers related to our work. Hamel et al.[13] describe how transparency can be handled in line drawings and they present a semi-automatic system to do this. Also in the context of NPR, Meier[25] employs several blended layers of differently shaped and oriented strokes to achieve diffuse object boundaries. This painterly rendering approach generates stroke-based transparent appearance between the boundary of foreground objects and the background. In an early fundamental work, Kay and Greenberg[19] introduce transparency into computer graphics both in its popular linear form and in a more complex non-linear approach to simulate the falling off of transparency at the edges of thin curved surfaces.

Interrante and co-workers[16, 17, 18] render several semi-transparent depth layers with a stroke-based approach. They visualize the 3D shape of smoothly curving transparent surfaces by utilizing the principal curvature directions of surfaces. In a subsequent work[15], they extend this approach to 3D line integral convolution in order to illustrate surface shape in volume data.

Gooch and co-workers[11, 12] introduce the concept of tone-based cool/warm shading combined with silhouette rendering for interactive technical illustrations. Their shading model is the basis for shading in our implementation. Seligman and Feiner[27] describe a rule-based illustration system (Intent-Based Illustration System IBIS) for rendering photorealistic illustrations and Markosian et al. [23] introduce a real-time non-photorealistic rendering system.

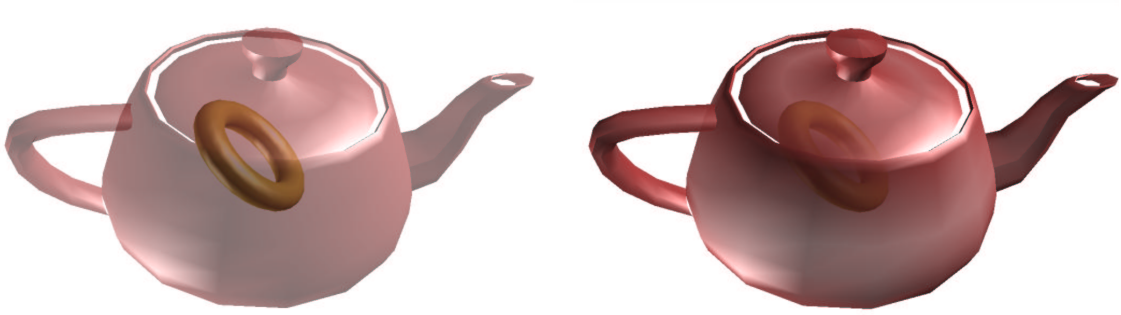
Another field of research related to this paper deals with issues of spatial sorting and visibility, which are crucial for correctly rendering transparent surfaces. For a survey on this well-established topic we refer, for example, to Foley et al.[9] or Durand[7]. Ghali[10] especially deals with object-space visibility. Recently, Snyder and Lengyel[28] proposed visibility sorting for image layer decomposition which could be used for transparent rendering. One of the depth-sorting algorithms of this paper is related to the *depth-peeling* approach by Everitt[8], facilitating the ideas of virtual pixel maps by Mammen[22] and of dual depth buffers by Diefenbach[5].

3. Traditional Visualization of Transparency in Technical Illustrations

Books describing techniques for technical and scientific illustrations[14, 24, 29] provide various rules for how to visualize transparency artistically. A simple method is to draw only the outlines of transparent objects. These outlines are rendered in a linestyle different to the other outlines to make them distinct from the opaque objects. This linestyle is often described as phantom lines[29] and is demonstrated in Figure 2. Although this technique can be applied to a wide variety of drawing styles, ranging from color illustrations to simple line or sketch drawings, there are certain drawbacks:

- Details of the transparent objects are lost as only their outlines are drawn.
- Material and surface information of transparent objects is ignored.
- There are only two transparency states: fully opaque or fully non-opaque; semi-transparency cannot be visualized.

A different approach can be found in “The Guild Handbook of Scientific Illustration” by Hodges[14]. For color illustrations, Hodges recommends to lighten the color of object regions which are occluded by transparent objects. Figure 1 demonstrates this approach. Hodges proposes the following basic rules:



(a) (b)

Figure 3: *Difference between standard transparency blending in (a) and view-dependent transparency blending in (b).*

- Strengthen the shade where an opaque object enters a non-opaque object.
- Set the intensity of the opaque object to zero at the edge of the surrounding object and slowly increase its intensity with increasing distance from the edge.

To put it another way, transparency falls off close to the edges of transparent objects and increases with the distance to edges. Besides these fundamental rules in visualizing transparent objects, there are others which are not directly described by Hodges and which focus on the correlation between objects. These precepts often are obvious for an illustrator, but cannot directly be transformed into rules that are appropriate for a computer-based implementation. However, by analyzing real color-shaded technical drawings, such as Figure 1, the following simplified rules can be identified:

- Back faces or front faces from the same non-opaque object never shine through.
- Opaque objects which are occluded by two transparent objects do not shine through to the closer transparent object.
- Two transparent objects are only blended with each other if they do not distract the viewer or if they are very close to each other and belong to the same semantic group.

This set of rules is based on the fact that in technical drawings transparency is used to look into objects and to show objects which lie inside or go through non-opaque ones. Often these object are opaque in reality.

As semantic grouping is something which can not be achieved without additional user interaction in pre or post processing steps, we propose to change the last rule to:

- Two transparent objects never shine through each other.

4. Basic Rendering Approach

From the above traditional methods to show transparency in manual drawings, we extract the following small and effective set of rules for computer-based rendering:

- Faces of transparent objects never shine through.
- Opaque objects which are occluded by two transparent objects do not shine through.
- Transparency falls off close to the edges of transparent objects and increases with the distance to edges.

Based on these rules, our rendering approach is as follows. First, the objects which are blended have to be determined, following the guidelines of the first two rules. This task essentially corresponds to a view-dependent spatial sorting. An efficient and adapted solution to this problem is described in Section 6.

Secondly, transparency values have to be computed for a correct blending between transparent and opaque objects according to the third rule. In the subsequent section, a corresponding algorithm is proposed. In what follows, we frequently use an α value instead of a transparency value. Note that transparency is $1 - \alpha$.

5. View-Dependent Transparency

To simulate our third rule, we introduce the concept of view-dependent transparency: The α value for blending between transparent and opaque objects depends on the distance to the outline of the transparent object; the outlines of an object projected onto a 2D screen consist of silhouettes lines and thus depend on the position of the viewer.

Silhouette edges can either be determined in 2D image-space or in 3D world/object-space. Hamel et al.[13] recommend to use an image-space method by establishing an object ID buffer and an edge extraction filter. However, this

method is very time-consuming as you first have to render the objects with an ID tag, find the edges, vectorize these edges to 2D lines, and—with the help of these lines—calculate the distance of each pixel to the outline.

In this paper, we rather pursue a 3D object/world-space approach. Here, all silhouettes are determined before the rasterization stage. With this approach most of the previously mentioned steps can be avoided. In 3D space, a silhouette is an edge connecting two faces, where one face of the edge is back facing and the other one is front facing. This classification can be formulated as

$$(\vec{n}_1 \cdot (\vec{p} - \vec{o}))(\vec{n}_2 \cdot (\vec{p} - \vec{o})) < 0 \quad , \tag{1}$$

where $\vec{p}$ is an arbitary, yet fixed point on the edge, $\vec{n}_i$ are the outward facing surface normal vectors of the two faces sharing the edge, and $\vec{o}$ is the position of the camera. In our implementation, all edges are checked for the above criterion and the detected silhouette edges are stored in a list. This is based on Appel's original algorithm[1]. The performance of silhouette detection could be improved by more sophisticated techniques, such as fast back face culling described by Zhang et al.[30], the Gauss map method[12], or by exploiting frame–to–frame coherence.

Then, the distance of each vertex to the closest silhouette edge is computed. First, the distances d of the vertex to all silhouette edges is determined according to

$$d = \left\| \vec{v} - \left(\vec{p} + \frac{(\vec{v} - \vec{p}) \cdot \vec{e}}{\|\vec{e}\|} \vec{e} \right) \right\| \quad ,$$

where $\vec{v}$ is the position of the vertex, $\vec{p}$ is an arbitrary, yet fixed point on the silhouette edge, and $\vec{e}$ is the silhouette edge. The minimum distance of a vertex to the silhouettes, $d_{\min}$, is used to calculate the α value for the respective vertex of the transparent object. In our implementation the following approach is used:

$$\alpha = 1 - \left(\frac{d_{\min}}{d_{\text{object,max}}} \right)^k \quad , \tag{2}$$

where $d_{\text{object,max}}$ denotes the maximum distance between the center of the object and all surface points; $k \in [0, 1]$ is a user-specified falling off term.

As weights for blending between transparent and opaque objects, we use the above α value for the transparent surface and another, fixed value for the opaque object. Therefore, the α value of the opaque object determines this object's weight in the final image. Note that the background color itself may be blended with transparent surfaces as well. In our implementation, different α values for opaque objects and for the background can be specified by the user.

Figure 3 (a) and 3 (b) compare view-dependent transparency to standard view-independent transparency.

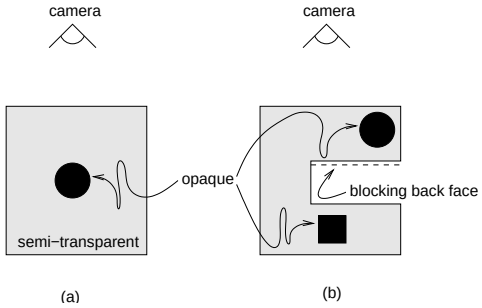


Figure 4: *Scenario with opaque objects embedded into the volume of a transparent object. The left image shows a convex transparent object, the right image shows a concave transparent object.*

6. Depth Sorting

Now that we know how to blend transparent and opaque objects, we need to determine which objects have to be blended and which objects are hidden and do not contribute to the final image. In full generality, rendering semi-transparent scenes would require view-dependent depth sorting. A large body of research has been conducted on these issues of spatial sorting and visibility[7, 9]. The two main approaches to depth sorting are either based on screen-space or on object-space.

In this section, we focus on screen-space algorithms. A major advantage of this approach is that we can exploit dedicated graphics hardware to efficiently solve this problem and can thus avoid operations on the slower CPU. Screen-space algorithms also benefit from the fact that our specific application does not require complete depth sorting—we only need to determine the closest transparent objects and the opaque objects directly behind these transparent surfaces. In all of our depth-sorting approaches, we assume that volumetric objects are defined by boundary surface representations.

6.1. Implicit Interior Boundaries

Let us start with a first, quite common scenario. Here, opaque objects may be contained inside the volume of a transparent object. These opaque objects should be visualized by transparent rendering. Figure 4 illustrates this scenario. Here, the interior boundary of the transparent volume is given implicitly by the surface of the surrounded opaque object.

First, let us consider only convex transparent objects as shown in Figure 4 (a). Opaque objects, however, may be of arbitrary shape. Following our rules from section 4, the front-facing surface of the transparent object closest to the camera has to be blended with opaque objects contained within the volume of this transparent object. All objects—

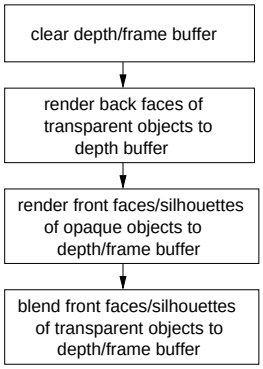


Figure 5: *Rendering pipeline for opaque objects embedded into the volumes of transparent objects.*

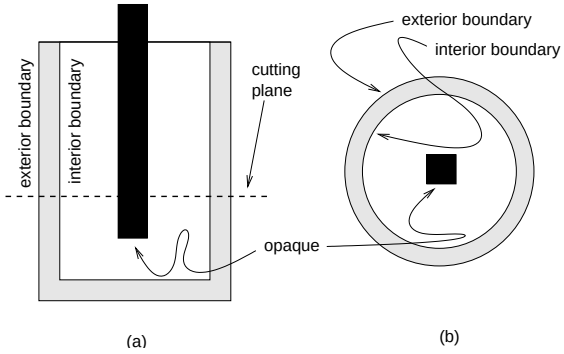


Figure 6: *Scenario with an opaque parallelepiped inside a transparent mug-like object, whose interior boundary is explicitly modeled. The left image shows a side view with a horizontal cutting plane, the right image shows a top view onto the cutting plane.*

both opaque and transparent—which are located further away should be hidden by this nearest transparent object. Note that these invisible objects are not included in Figure 4 (a). Finally, opaque objects in the foreground are drawn on top of all other objects further behind.

In a slightly more complicated scenario, concave transparent objects are permitted, as illustrated in Figure 4 (b). Surrounded opaque objects should be visible only up to the first back-facing part of the transparent object in order to blend out unnecessary and distracting visual information. To put it another way, only those opaque objects are visible which are closer than the nearest back face of the surrounding transparent object. In the example of Figure 4 (b), the opaque circular object is visible, whereas the squared object is hidden by a back-facing surface of the surrounding object.

Figure 5 shows the rendering pipeline for this first scenario. The depth buffer is used to select the correct objects for blending and to hide the unwanted objects. In the first part, back faces of all transparent objects are rendered into the depth buffer only. In this way, the depth buffer contains the distance of the closest back-facing transparent surfaces. In the second part, the front faces and the silhouettes of all opaque objects are rendered to both frame and depth buffers. The depth test rejects all opaque objects lying behind any "transparent" back-facing surface. Finally, the front faces and silhouettes of the transparent objects are rendered and blended into the depth and frame buffers, respectively. Here, the depth test rejects all parts of the transparent objects hidden by an opaque foreground object. Blending is applied only at those parts of the frame buffer where a transparent surface is directly visible to the user.

The algorithm can be implemented by only using standard OpenGL 1.2[26]. Writing to the depth or frame buffers can be enabled and disabled by `glDepthMask` or `glColorMask`, respectively. This rendering approach comprises only "one and a half rendering passes" because front faces of the opaque objects and both front and back faces of the transparent objects have to be rendered.

6.2. Explicit Interior Boundaries

Now let us consider a more complex scenario. Volumetric objects are still represented by boundary surfaces. However, objects may no longer be contained inside the volume of another object. In fact, surrounding transparent objects have to be modeled with respect to both the outside and the inside boundary. Figure 6 illustrates this scenario for the example of an opaque parallelepiped inside a transparent cylindrical, mug-shaped object; see color section(a) shows a color-shaded rendering of a similar scene. This scenario better reflects the properties of many technical 3D data sets, which explicitly represent all boundaries—both inside and outside.

The algorithm from Section 6.1 fails for this scenario, as all surrounded opaque objects are hidden by a back-facing transparent surface. To overcome this problem, another classification of the visibility of opaque objects is necessary: Only those opaque objects located between the closest and second-closest front-facing transparent surfaces are visible. Objects (transparent or opaque) further behind are hidden.

Two depth buffers are necessary to perform this depth selection. However, only one depth buffer is directly supported on available graphics hardware. Fortunately, the required behavior can be emulated by means of texture mapping and per-fragment depth operations on modern hardware, such as NVidia's GeForce3.

The basic algorithm is as follows. In the first step, all front-facing transparent surfaces are rendered to the depth buffer; afterwards, the depth buffer contains the depth values of the closest transparent surfaces. As second step, the

depth buffer is stored in a high-resolution texture and then the depth buffer is cleared. In the third step, the front-facing transparent surfaces are rendered to the depth buffer except the foremost ones, virtually peeling off the closest surfaces. After this step, the second-closest transparent front faces are stored in the depth buffer. In the fourth step, opaque objects are rendered. The depth test rejects all surfaces, but those lying in front of the second-closest transparent front faces. Finally, just the foremost transparent surfaces are blended into the frame buffer.

The principal idea behind step three is related to Everitt's *depth-peeling* approach[8], making use of dual depth buffers[5] and virtual pixel maps[22]. Everitt's and our implementations, however, differ significantly, which will be explained shortly.

Figure 7 shows the details of the actual rendering pipeline. The left part contains the rendering steps, the right part conveys the intended behavior. For the following description, terminology from standard OpenGL specifications[26] and NVidia-specific extensions[20] is employed. Only front-facing polygons are drawn in all rendering steps.

The first four boxes implement parts one and two of the basic algorithm. The depth buffer is read to main memory, after having rendered the transparent objects to the depth buffer. The depth values are transferred as 32 bit unsigned integers. Subsequently, a so-called HILO texture object containing these depth values is defined. A HILO texture is a high-resolution 2D texture, consisting of two 16 bit unsigned integers per texel. The original 32 bit depth component can be regarded as a HILO pair of two 16 bit short integers. In this way, a time-consuming remapping of depth values can be avoided—the content of the depth buffer is transferred from main memory to texture memory as is.

The next two boxes realize part three of the basic algorithm. A texture shader program is enabled to virtually clip away all surfaces that have equal or smaller depth values than those given by the above HILO texture. Essentially, this texture shader program replaces the z value of a fragment by $z - z_{\text{shift}}$, where z_{shift} represents the z value stored in the HILO texture. The gray boxes in Figure 7 indicate the scope of the depth transformation. The details of the texture shader program will be explained shortly. By shifting z values by $-z_{\text{shift}}$, only the fragments with $z > z_{\text{shift}}$ stay in a valid range of depth values—all other fragments are clipped away. As a consequence, the foremost transparent surfaces are "peeled off" and only the depth values of the second-closest surfaces are rendered into the depth buffer.

The last two gray boxes implement part four of the basic algorithm. With the texture shader program still being enabled, the opaque surfaces and corresponding silhouette lines are rendered into both frame and depth buffer. Only the fragments lying in front of the second-closest transparent surfaces pass the depth test, i.e., only those parts that are supposed to be blended with the transparent surrounding.

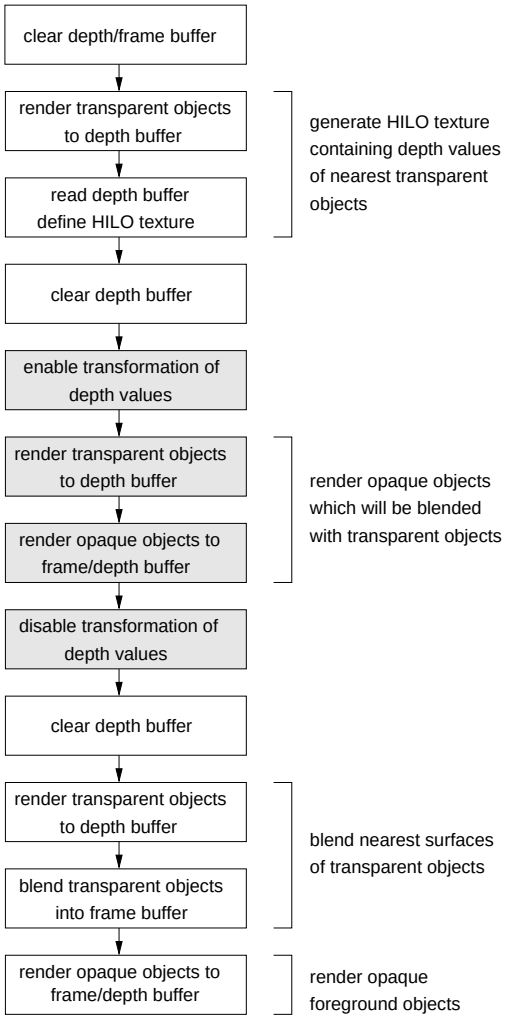


Figure 7: *Rendering pipeline for explicitly modeled inside boundaries of surrounding objects.*

Finally, the last four boxes realize part five of the basic algorithm. The first two steps once again initialize the depth buffer with the original z values of the closest transparent surfaces. Based on a depth test for equal z values, just these closest surfaces and silhouettes are blended into the frame buffer. Ultimately, the missing opaque foreground objects are rendered.

The texture shader program is based on NVidia's `DotProductDepthReplace`. Figure 8 illustrates the structure of the depth replace program, which always comprises three texture stages. Stage zero performs a standard 2D texture lookup in a HILO texture. Stage one and two compute two dot products, Z and W, between the respective texture coordinates and the previously fetched values from the

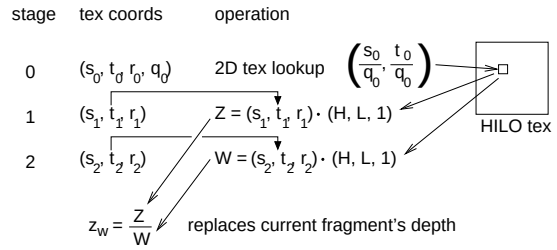


Figure 8: *Texture shader for dot product depth replace.*

HILO texture. Still in stage two, the current fragment's depth is replaced by Z/W.

The texture coordinates for each texture stage have to be set properly in order to achieve the required shift of depth values by $-z_{\text{shift}}$. In our implementation, texture coordinates are issued on a per-vertex basis within a vertex program. For the following discussion, three different coordinate systems have to be considered: homogeneous clip coordinates (x_c, y_c, z_c, w_c), normalized device coordinates $(x_n, y_n, z_n) = (x_c/w_c, y_c/w_c, z_c/w_c)$, and window coordinates (x_w, y_w, z_w). Valid normalized device coordinates are from $[-1,1]^3$ and, here, valid window coordinates are assumed to be from $[0,1]^3$.

The standard transform and lighting part of the vertex program computes homogeneous clip coordinates from the original object coordinates (i.e., after modeling, viewing, and perspective transformations). Clip coordinates are linearly interpolated between vertices during the scan-conversion of triangles, yielding a hyperbolic interpolation in normalized device coordinates and in window coordinates. The division by w_c is performed on a per-fragment basis. For perspectively correct texture mapping, texture coordinates analogous to the above clip coordinates have to be used. Consequently, the vertex program may only assign clip coordinates per vertex, but not device or window coordinates.

The texture coordinates for stage zero are set to

$$\begin{aligned} s_0 &= \frac{x_c + w_c}{2} \quad , \\ t_0 &= \frac{y_c + w_c}{2} \quad , \\ q_0 &= w_c \quad . \end{aligned}$$

In this way, the 2D lookup in the HILO texture is based on the coordinates,

$$\left(\frac{s_0}{q_0}, \frac{t_0}{q_0}\right) = \left(\frac{x_n+1}{2}, \frac{y_n+1}{2}\right) = (x_w, y_w) \quad ,$$

allowing for a mapping to the range of texture coordinates, $[0,1]^2$, after the division by q_0. Therefore, a one-to-one correspondence between xy coordinates in the frame buffer and texels in the HILO texture is established.

The texture coordinates for stage one and two are set to

$$\begin{aligned} (s_1, t_1, r_1, q_1) &= \left(\frac{-w_c}{2^{16}}, -w_c, \frac{z_c + w_c}{2}\right) \quad , \\ (s_2, t_2, r_2, q_2) &= (0, 0, w_c) \quad , \end{aligned}$$

yielding the intermediate homogeneous coordinate Z from the dot product in texture stage one,

$$\begin{aligned} Z &= \left(\frac{-w_c}{2^{16}}, -w_c, \frac{z_c + w_c}{2}\right) \cdot (\mathrm{H}, \mathrm{L}, 1) \\ &= w_c \left\{ -z_{w,\text{shift}} + \frac{z_n + 1}{2} \right\} \\ &= w_c \left\{ -z_{w,\text{shift}} + z_w \right\} \quad , \end{aligned}$$

where $2^{-16}\mathrm{H} + \mathrm{L}$ is the high-resolution depth $z_{w,\text{shift}}$ because the pairs of unsigned short integers in the HILO texture are arranged in low–high order. Similarly, the intermediate homogeneous coordinate W is computed by texture stage two as

$$W = (0, 0, w_c) \cdot (\mathrm{H}, \mathrm{L}, 1) = w_c \quad .$$

Finally, the depth value of the fragment in window coordinates is set by the texture shader to the value

$$z_{w,\text{final}} = \frac{Z}{W} = z_w - z_{w,\text{shift}} \quad ,$$

still in texture stage two. In this way, the required shift of depth values by $-z_{w,\text{shift}}$ is achieved at a very high resolution. Depth tests and clipping are sensitive with respect to the provided accuracy and otherwise could not be realized, for example, by using low resolution standard textures with only eight or twelves bits per channel.

Although the basic ideas of Everitt's *depth-peeling*[8] and our approach are closely related, the two implementations differ significantly. In addition to the three stages for dot product depth replace, Everitt needs another texture stage for a lookup in a depth texture (`SGIX_depth_texture` and `SGIX_shadow`), i.e., his implementation requires four texture stages instead of only three in ours and does not allow any other further texture fetch, e.g., for standard texturing. (Note that the maximum number of texture stages is four.)

7. Results

Our implementation of transparency for technical illustrations is based on OpenGL[26] and on NVidia-specific extensions[20]. User-interaction and the management of rendering contexts in our C++ application are handled by GLUT.

Cool/warm tone-based shading[11] is implemented as a vertex program. Tone-based shading is combined with black line silhouettes to facilitate the recognition of the outlines of both transparent and opaque objects. The silhouettes are rendered using a hardware approach, as described by Gooch and Gooch[12].

Table 1: *Performance measurements.*

scene	# polygons		lines?	FPS	
	total	non-opaque		I	II
Fig. 9(f)	25,192	192	no	21.4	9.0
Fig. 9(f)	25,192	192	yes	13.7	5.7
Fig. 9(d)	40,000	835	no	11.4	5.6
Fig. 2	70,955	3,112	no	—	1.2

Additionally, silhouette edges of transparent objects are computed in 3D object/world space according to Section 5 to determine the α values. These values are calculated per vertex according to Eq. (2). As blending function we use `GL_SRC_ALPHA` for source and `GL_DST_ALPHA` for destination. The source α is set to the above value, the destination α is the fixed value for the opaque object. Note that the background color itself may be blended with transparent surfaces as well. In our implementation, different α values for opaque objects and for the background can be specified by the user. A value $\alpha = 0.5$ for the background is a good choice, making sure that the background color does not shine completely through non-opaque objects. The rendering pipelines for both implicitly modeled interior boundaries (Section 6.1) and explicitly modeled interior boundaries (Section 6.2) have been implemented. For the implementation of the latter, both vertex programs and texture shaders are used, as described in Section 6.2.

The figures in the color section show results generated by our implementation. (a) displays a transparent mug with two boundaries surrounding an opaque box, similarly to the scenario described in Section 6.2. (b) and (c) show a similar scene, including multiple transparent mugs which are rendered according to the rules in Section 4. (d) and (e) show different parts of a Lancia engine block as a typical example of a technical illustration. Finally, (f) displays a crank within a transparent cylinder.

Table 1 shows performance measurements for both depth-sorting approaches. The method for implicitly modeled interior boundaries is denoted “I”, the method for explicitly modeled interior boundaries is denoted “II”. All tests were carried out on a Windows2000 PC with AMD Athlon 900MHz CPU and GeForce3 Ti200. Window size was 512^2. The first column refers to the figure in which is respective scene is depicted. The second and third columns contain the number of either all polygons or transparent polygons. The fourth column indicates whether silhouette lines are rendered. The fifth and sixth columns reflect the frame rates for methods “I” and “II”, respectively. Note that method “I” does not render the scene in row four correctly and thus the corresponding frame rate is omitted.

These performance figures indicate that most of the rendering time is spent in the computation of silhouettes, as frame rate drops rapidly with the complexity of the scene. We expect that speedup techniques previously described in Section 5 will lead to improved frame rates. Although the complex rendering pipeline for method “II” needs a lot more passes than the pipeline for method “I”, the overall performance difference is only an approximate factor of two. This shows that although the number of passes is twice than in Method “I”, a readback of the Z-Buffer is necessary and we have three texture shader stages, the performance is not as worse for these kind of application as one would probably think.

8. Conclusions

We have shown how transparency is employed in traditional renderings of color-shaded technical illustrations to greatly improve the perception of spatial structures. We have proposed an approach to simulate the same effect using computer graphics techniques. In particular, a transparency model which depends on viewing direction has been introduced. Furthermore, two screen-space methods have been proposed to allow for adapted and efficient depth-sorting. By exploiting modern consumer graphics hardware, we achieve transparent rendering of technical illustrations in real time. Our transparency model can readily be combined with other techniques for technical illustration, such as cool/warm tone shading or silhouette rendering, in order to enhance the visual perception and understanding.

In future work, we will apply better silhouette detection routines to improve rendering speed. The mach-banding effects caused by linearly interpolating α values between vertices could be avoided by either per-pixel calculation of α values or by adaptive surface subdivision. In addition, the benefits of appropriate user control could be exploited in order to include semantic classifications. Furthermore, a user-specified blending color which may differ from the background color could be introduced to completely avoid shining through of the background.

Acknowledgements

Thanks to the unknown reviewers for many helpful comments and to Albert Maier for letting us use his example image.

References

1. A. Appel. The notion of quantitative invisibility and the machine rendering of solids. In *Proceedings of ACM National Conference*, pages 387–393, 167. 4

2. I. Biederman and G. Ju. Surface versus edge-based determinants of visual recognition. *Cognitive Psychology*, 20:38–64, 1988. 1

3. W. Braje, B. Tjan, and G. Legge. Human efficiency for recognizing and detecting low-pass filtered objects. *Vision Research*, 35(21):2955–2996, 1995. 1

4. C. Christou, J. Koenderink, and A. can Doorn. Surface gradients contours and the perception of surface attitude in images of complex scenes. *Perception*, 25:701–713, 1996. 1

5. P. J. Diefenbach. *Pipeline Rendering: Interaction and Realism through Hardware-Based Multi-Pass Rendering*. PhD thesis, University of Pennsylvania, 1996. 2, 6

6. D. Dooley and M. Cohen. Automatic illustration of 3D geometric models: Lines. *Computer Graphics*, 24(2):77–82, 1990. 1

7. F. Durand. *3D Visibility: Analytical Study and Applications*. PhD thesis, Université Joseph Fourier, Grenoble I, July 1999. 2, 4

8. C. Everitt. Interactive order-independent transparency. White paper, NVidia, 2001. 2, 6, 7

9. J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley, Reading, Massachusetts, 1990. 2, 4

10. S. Ghali. *SIGGRAPH 2001 Course 6:* Object space visibility, 2001. 2

11. A. Gooch, B. Gooch, P. Shirley, and E. Cohen. A non-photorealistic lighting model for automatic technical illustration. In *SIGGRAPH 1998 Proceedings*, pages 101–108, July 1998. 2, 7

12. B. Gooch, P.-P. Sloan, A. Gooch, P. Shirley, and R. Riesenfeld. Interactive technical illustration. In *1999 ACM Symposium on Interactive 3D Graphics*, pages 31–38, April 1999. 2, 4, 7

13. J. Hamel, S. Schlechtweg, and T. Strothotte. An approach to visualizing transparency in computer-generated line drawings. In *IEEE Proceedings of Information Visualization*, pages 151–156, 1998. 1, 2, 3

14. E. Hodges. *The Guild Handbook of Scientific Illustration*. Van Nostrand Reinhold, New York, 1989. 1, 2

15. V. L. Interrante. Illustrating surface shape in volume data via principal direction-driven 3D line integral convolution. In *SIGGRAPH 1997 Conference Proceedings*, pages 109–116, 1997. 2

16. V. L. Interrante, H. Fuchs, and S. Pizer. Enhancing transparent skin surfaces with ridge and valley lines. In *IEEE Visualization 1995 Proceedings*, pages 52–59, 1995. 2

17. V. L. Interrante, H. Fuchs, and S. Pizer. Illustrating transparent surfaces with curvature-directed strokes. In *IEEE Visualization 1996 Proceedings*, pages 211–218, 1996. 2

18. V. L. 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), Apr.–June 1997. 2

19. D. S. Kay and D. Greenberg. Transparency for computer synthesized images. *Computer Graphics (SIGGRAPH 1979)*, 13(2):158–164, 1979. 2

20. M. J. Kilgard, editor. *NVIDIA OpenGL Extension Specifications*. NVIDIA Corporation, 2001. 6, 7

21. A. Maier. *Technical illustration.* Web Site: http://www.illustrationz.co.nz, 2001. 1

22. A. Mammen. Transparency and antialiasing algorithms implemented with the virtual pixel maps technique. *IEEE Computer Graphics and Applications*, 9(4):43–55, July 1989. 2, 6

23. L. Markosian, M. Kowalski, S. Trychin, and J. Hughes. Real-time non-photorealistic rendering. In *SIGGRAPH 1997 Proccedings*, pages 415–420, Aug. 1997. 2

24. J. Martin. *Technical Illustration: Material, Methods, and Techniques*, volume 1. Macdonald and Co Publishers, 1989. 2

25. B. J. Meier. Painterly rendering for animation. In *SIGGRAPH 1996 Conference Proceedings*, pages 477–484, 1996. 2

26. M. Segal and K. Akeley, editors. *The OpenGL Graphics System: A Specification (Version 1.2.1)*. Silicon Graphics, 1999. 5, 6, 7

27. D. D. Seligmann and S. Feiner. Automated generation of intent-based 3D-illustrations. In *SIGGRAPH 1991 Conference Proccedings*, pages 123–132, 1991. 2

28. J. Snyder and J. Lengyel. Visibility sorting and compositing without splitting for image layer decomposition. In *SIGGRAPH 1998 Conference Proceedings*, pages 219–230, 1998. 2

29. T. Thomas. *Technical Illustration.* McGraw-Hill, New York, second edition, 1968. 1, 2

30. H. Zhang and K. Hoff. Fast backface culling using normal masks. In *Proceedings 1997 Symposium on Interactive 3D Graphics*, pages 103–106, Apr. 1997. 4

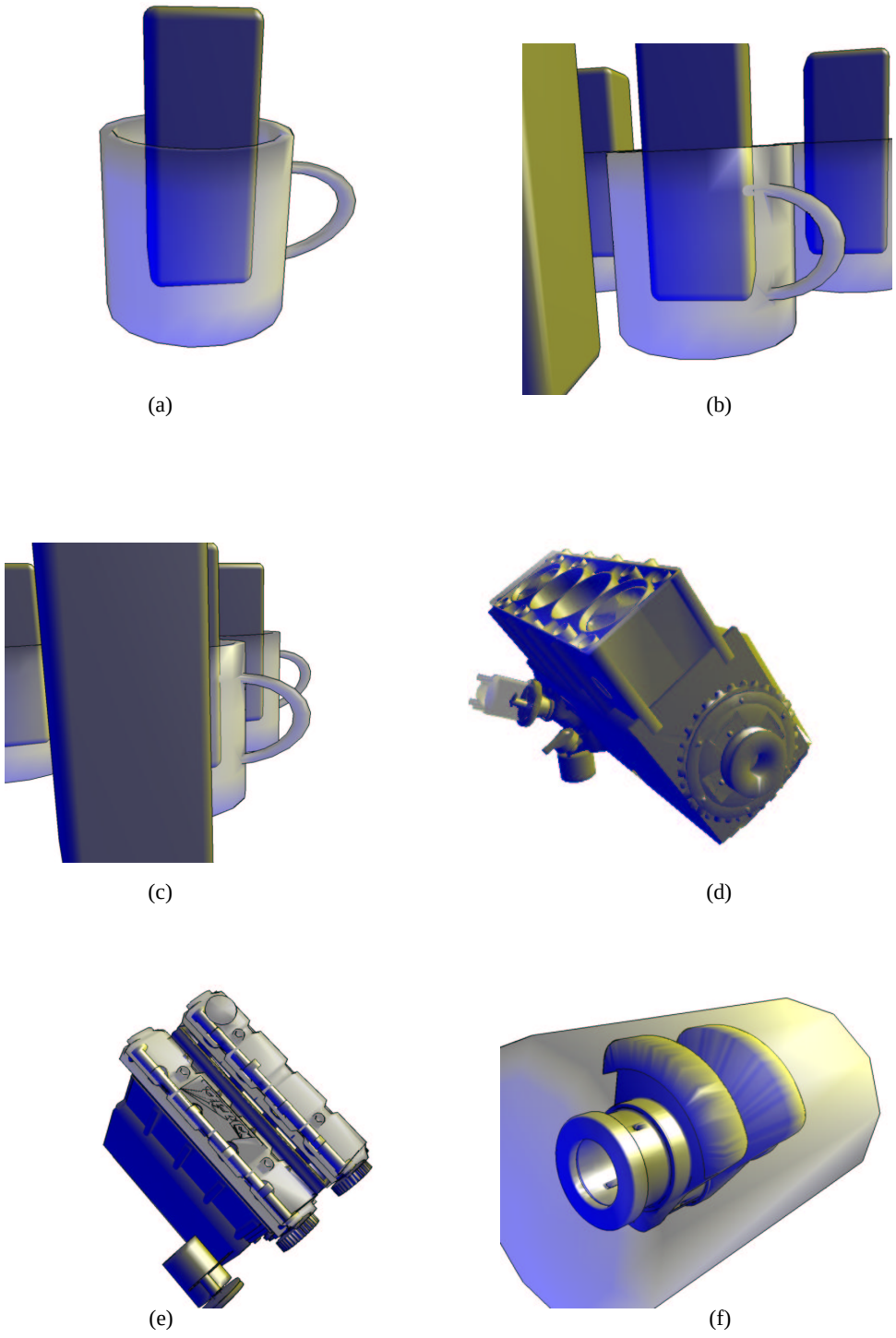


Figure 9: *these pictures show different results from our implementation. Image (a) shows a transparent mug with two boundaries surrounding an opaque box, similarly to the scenario described in Section 6.2. Images (b) and (c) show a similar scene, including multiple transparent mugs which are rendered according to the rules in Section 3. Pictures (d) and (e) show different parts of a Lancia engine block as a typical example of a technical illustration. Image (f) displays a crank within a transparent cylinder.*

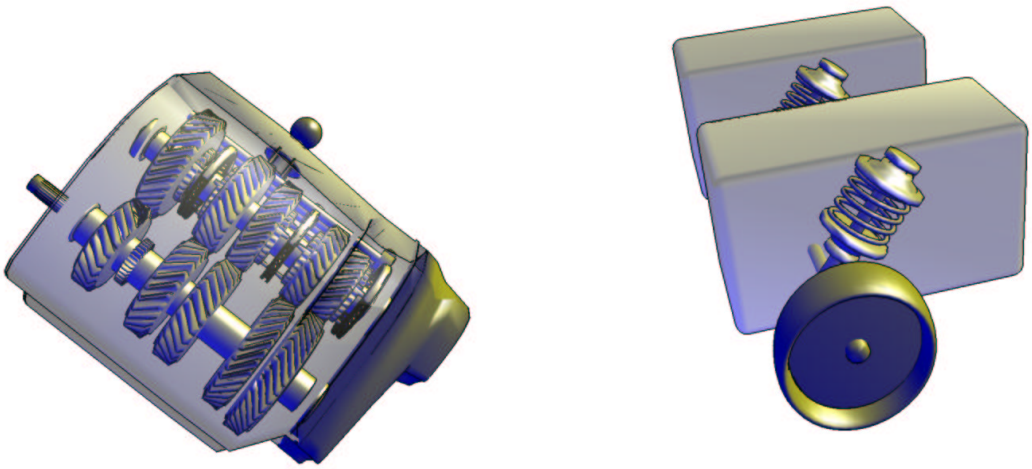


Figure 10: *these picture show more results from our rendering approach. The left image shows the same scenario as in Figure 2 of our paper, rendered using Method II. The right picture shows a simplified wheel axis*