D-BUFFER: A NEW HIDDEN-LINE ALGORITHM IN IMAGE-SPACE

CENTRE FOR NEWFOUNDLAND STUDIES

TOTAL OF 10 PAGES ONLY MAY BE XEROXED

(Without Author's Permission)

XIAOMIN DONG

D-Buffer: A New Hidden-Line Algorithm in Image-Space

by

Xiaomin Dong

A thesis submitted to the

School of Graduate Studies

in partial fulfillment of the

requirements for the degree of

Master of Science

Department of Computer Science
Memorial University of Newfoundland

1999

St. John's Newfoundland

Abstract

In many applications such as computer-aided design (CAD), drafting, descriptive geometry, geometric modeling, computer animation, and virtual reality, etc., real-time or time-critical rendering is required by three-dimensional interaction and manipulation to provide adequate information of complex objects. Traditionally, rendering techniques can be discussed in two major categories: shaded mode and wireframe mode. While shaded rendering algorithms create more realistic pictures, wireframe techniques are more efficient for generating line-drawing images that are often more informative. In wireframe images, it is preferred to display the hidden-lines in a special style to distinguish them from ordinary visible lines. Thus hidden-line algorithms are needed. Though object-space hidden-line algorithms are widely adopted for showing hidden-lines in distinctive styles, image-space algorithms have the advantages of rendering speed and processable shapes for their simplicity. This thesis develops a

new image-space algorithm based on the traditional Z-Buffer algorithm to recover the information loss caused by hidden-line removal. This D-Buffer algorithm improves the Z-Buffer algorithm by drawing the hidden-lines in dotted or dashed style rather than removing them, hence retrieving the concealed information. Some image processing techniques, such as neighborhood operations, are used to generate the dotted or dashed lines. The D-Buffer algorithm is not only as efficient as other image-space algorithms, but is also capable of disclosing more inner structure information for a wide range of three-dimensional objects.

Acknowledgments

I could complete the work presented here thanks to a lot of people who have helped me in some ways. With sincere appreciation, I would like to express my highest gratitude to my supervisors Professor Xiaobu Yuan and Professor Hong Wang for their advice and encouragement in pursuing this research. Their continuous support, guidance, and efforts to arrange adequate financial support made it possible for me to be able to study in Canada. Their valuable comments and suggestions have really improved this work.

I am also grateful to the School of Graduate Studies, the Department of Computer Science, and the Department of Mathematics and Statistics in Memorial University of Newfoundland for providing me with the fellowship and other financial support, without which it would have been impossible for me to finish this work.

Thanks are also due to the graduate students in the Department of Computer Science and the Department of Mathematics and Statistics for making my graduate student experience very pleasurable. Our many vigorous and helpful discussions served to clarify a number of key points.

Finally, a very special note of appreciation is extended to my wife and daughter for their unlimited support and blessings.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Problem Statement

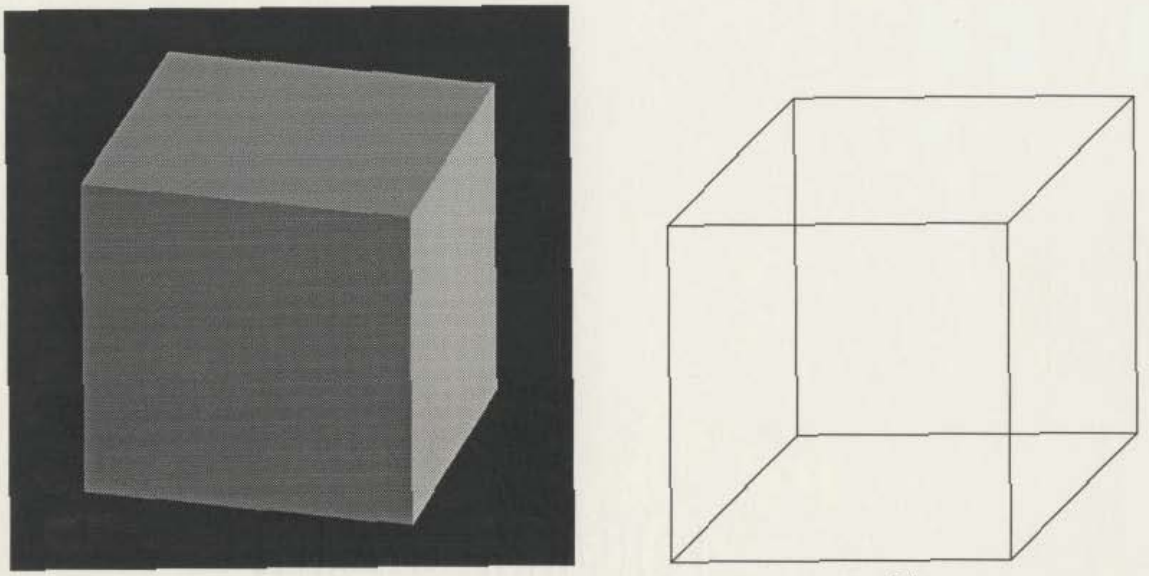
In many applications such as computer-aided design (CAD), drafting, descriptive geometry, geometric modeling, computer animation, and virtual reality, etc., three-dimensional interaction and manipulation are needed, often requiring real-time or time-critical rendering that provides sufficient information of complex objects. The goal of this thesis is to find a method that displays objects with complicated shapes quickly and without loss of information.

1.2 Rendering Modes

Rendering is the process of creating images from models. Rendering techniques can be classified into two modes: shaded and wireframe [12]. In shaded mode, a rendering program "shades" the interior pixels of visible portions of facets. Hidden-surfaces must be removed for a picture to make sense. In wireframe mode, objects are drawn as though made of wires, with only their boundaries showing. Hidden-lines may or may not be removed.

1.2.1 Shaded Mode

Shaded mode algorithms usually generate more realistic images than do wireframe mode ones. The addition of shaded areas to the rendering process, however, increases the complexity significantly, because spatial ordering becomes important — portions of objects that are hidden (because they are obscured by portions of "closer" objects) must not be displayed. Computational inefficiency and incapability of visualizing the rear and internal structure of objects are two major disadvantages of shaded mode rendering algorithms. Furthermore, a more realistic picture is not necessarily more desirable or useful (Figure 1.1(a)). If the ultimate goal of a picture is to convey



(a) (b)

Figure 1.1: Shaded vs. Wireframe

information, then a picture that is free of the complications of shadows and reflections may well be more successful than a photo-realistic image.

For example, ray tracing is one of the most popular and powerful shaded rendering algorithms [1, 14]. The beauty of ray tracing is its extreme simplicity, while one of its greatest challenges is efficient execution. It is often dismissed as being too computationally exorbitant to be useful. To display the internal structure, objects can be made transparent, but that takes even longer rendering time [30].

1.2.2 Wireframe Mode

Though shaded mode (surface-drawing) usually possesses better perceptibility, the conciseness of data representation and accuracy of boundary description make wireframe mode (line-drawing) preferred in many circumstances. It has long been known that information about surface shape is largely conveyed via the curving of boundary edges [13]. The discontinuity at surface boundaries (edges), depicted as lines in two-dimensional drawing, is often a primary source of information about object structure that can be extracted from an image [4]. The line structure represents an irreplaceable basis for any more sophisticated representations of objects.

The main advantage of a wireframe picture is that it provides sufficient information of three-dimensional objects at a significantly low computational cost. It allows the user to "see through" objects and visualize the internal structure and shape of normally invisible surfaces. Besides, when constraint on time is significant, the success of a task depends heavily on how fast it displays objects. Therefore, wireframe rendering algorithms are better choices than shaded ones to meet the time-critical requests and to show the internal structure of objects. However, a new problem arises. If the created picture contains all the boundary lines to show the whole information of the object, it may be hard to distinguish front from back, and the complexity of

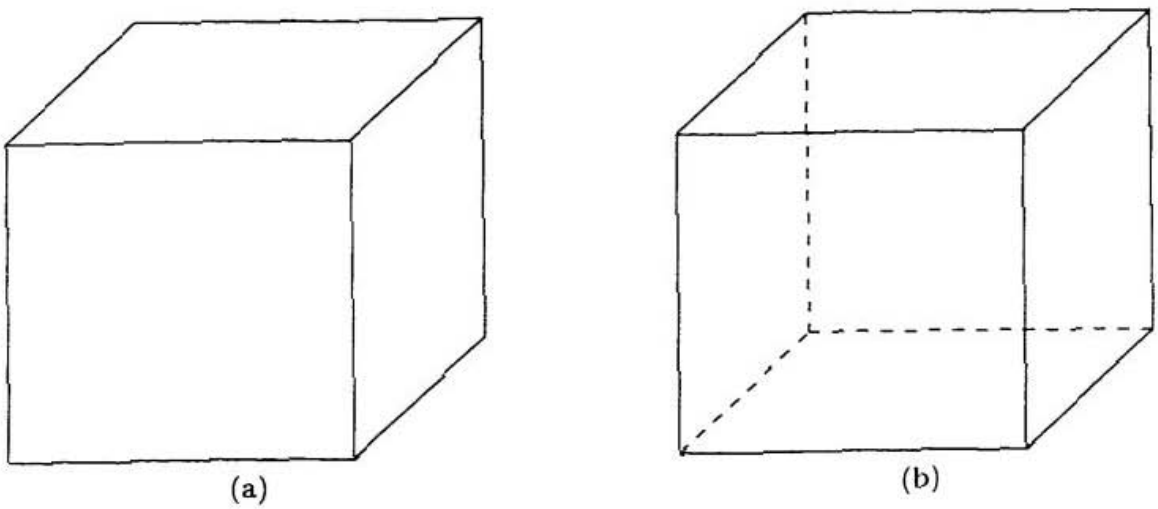


Figure 1.2: Need for distinctive hidden-line

even relatively simple objects soon overwhelms the observer, with insight into shape being lost in the clutter of lines (Figure 1.1(b)). On the other hand, if the picture presumes opaque object surfaces and shows only those boundary lines or segments that are visible in the view, internal structure is concealed as in the shaded mode (Figure 1.2(a)). The ideal picture should contain all the boundary information. but show the visible and invisible lines or segments in different ways (Figure 1.2(b)).

The task now becomes finding hidden-lines or segments and displaying them in distinctive style, such as dotted or dashed lines.

1.3 Hidden-Line Elimination

The hidden-line problem is one of the most difficult in computer graphics. Hidden-line algorithms attempt to determine the lines, edges, surfaces, or volumes that are visible or invisible to an observer located at a specific point in space. There is no best solution of the hidden-line problem. Sutherland et al [26] characterize the hidden-line algorithms as to whether they operate primarily in *object-space* or *image-space* and the different uses of *coherence* that the algorithms employ. Coherence is a term used to describe the process where geometrical units, such as areas or scan line segments, instead of single points are operated on by the hidden-line removal algorithm.

1.3.1 Object-Space Algorithms

The earliest hidden-line algorithm [22] worked in object-space. Object-space algorithms are performed at the precision with which each object is defined, and determine the visibility of each object. They are particularly useful in precise engineering applications. However, Object-space algorithms can only handle objects with at most quadric surfaces so far [17, 19] due to the complexity of intersecting two general surfaces. In addition to the limitation of object shape, they are usually computa-

tional inefficient because they have to perform object-object comparisons, sorting, and difficult intersection finding, which are typically very time-consuming and hard to implement, especially when objects exhibit complexly curved shapes. Therefore, they are not suitable for our goal.

1.3.2 Image-Space Algorithms

Image-space algorithms are typically performed at the resolution of the display device with which the objects are viewed, and determine the visibility at each pixel. Because of their simplicity, they have lower computational complexity and also are capable of rendering pictures that contain objects with a wider range of shapes. Consequently, for time-critical rendering of objects with complex shapes, image-space algorithms are superior to object-space ones for the reason of efficiency and no limitation of object shapes. Among the image-space hidden-line algorithms, the Z-Buffer algorithm stands out for its simplicity and efficiency.

1.4 D-Buffer Algorithm

Though the Z-Buffer algorithm is very fast and can handle objects with complicated shapes, it conceals the internal structure information of objects. This thesis presents a new image-space algorithm based on the Z-Buffer algorithm, namely the D-Buffer algorithm. By using one more boundary buffer, the D-Buffer algorithm can generate dotted or dashed hidden-line segments of any three-dimensional shapes at a fairly low computational cost, hence meeting all the demands set in Section 1.1.

1.5 Summary of the Thesis

The rest of this thesis is structured in the following manner. Chapter 2 gives a brief description of the methods for determining visible lines and discusses their advantages and disadvantages. Chapter 3 presents the new D-Buffer algorithm in detail. Finally, Chapter 4 contains concluding remarks, including the advantages and shortcomings of the D-Buffer algorithm and the directions for future work. Appendix A and B provides the pseudo-code of the D-Buffer algorithm.

Chapter 2

Hidden-Line Elimination

A fundamental problem to computer graphics is to determine the visibility of a scene from a specific viewpoint. This problem is known as *visible line* or *visible surface determination*. or *hidden-line* or *hidden-surface elimination*. Here *surfaces* are assumed to be opaque. They may obscure other surfaces farther from the viewer. In wireframe mode, *lines* are used to present the boundary edges or silhouette lines of surfaces. *Hidden-line elimination* will be used to refer to this problem.

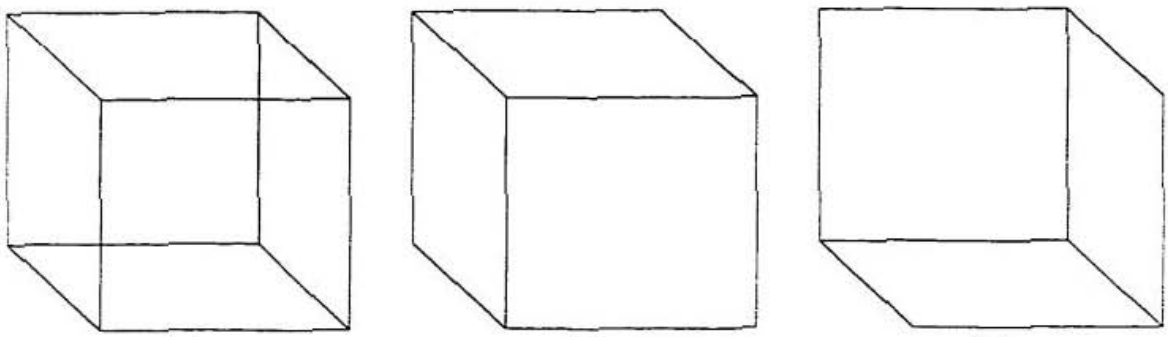


Figure 2.1: Need for hidden-line elimination

2.1 Introduction

The need for eliminating hidden-lines is illustrated in Figure 2.1. Figure 2.1(a) shows a typical wireframe drawing of a cube. It can be interpreted either as a view of the cube from above and to the left or from below and to the right. The alternate views can be seen by blinking and refocusing the eyes. This ambiguity can be eliminated by removing the lines that are invisible from the two alternate viewpoints. The results are shown in Figure 2.1(b) and (c).

The complexity of the hidden-line problem has resulted in a large number of diverse solutions. Many of these are for specialized applications. There is no best solution to the hidden-line problem. Fast algorithms that can provide solutions at video frame rates (30 frames per second) are required for real-time simulations, e.g. in

aircraft simulation. Algorithms that can provide detailed realistic solutions including shadows, transparency, and texture effects, with reflections and refraction in a multitude of subtle shades of color, are also required, e.g. in computer animation. These algorithms are slower, often requiring several minutes or even hours of computation. Technically, transparency, texture, reflection, etc., are not part of the hidden-line problem. They are more appropriately part of picture rendering. However, many of these effects are incorporated into hidden-line algorithms. There is a tradeoff between speed and detail. No single algorithm can provide both with current hardware. As faster algorithms are developed, more rendering detail can be incorporated. However, inevitably more detail will be required.

Hidden-line algorithms can be classified into two groups based on the coordinate system or space in which they operate, namely object-space and image-space algorithms, respectively [26]. Object-space algorithms work in the physical coordinate system in which the objects are described. They compare objects directly with each other, eliminating entire objects or portions of them that are invisible. Very precise results, generally to the precision of the machine, are available. These results can be satisfactorily enlarged many times. Object-space algorithms are particularly useful in precise engineering applications. Image-space algorithms are implemented in the screen coordinate system in which the objects are viewed. They determine which

object is visible at each pixel in the image. Calculations are performed only to the precision of the screen representation. Scenes calculated in image-space and significantly enlarged do not give acceptable results. In effect, object-space algorithms generate an analytic description of a graphics scene while image-space algorithms generate a mere bitmap or raster image of a graphics scene [10]. As a special case, list priority algorithms operate in both object- and image-spaces. (Refer to Section 2.5 for more details.)

To find the hidden-lines or segments, the most straightforward method is to find their endpoints and line equations in object-space. Theoretically, the computation for an object-space algorithm that compares every object in a scene with every other object in the scene grows as the number of objects squared (n^2). Similarly, the work for an image-space algorithm which compares every object in the scene with every pixel location in screen coordinates theoretically grows as nN. Here, n is the number of objects (volumes, planes, or edges) in the scene, and N is the number of pixels. In addition, object-space algorithms require fewer comparisons than image-space algorithms for $n < N$. Since N is over 1 million for a high-resolution display, most algorithms should theoretically be implemented in object-space.

In practice, this is not the case. Object-space algorithms involve a great deal of

intersection computation. Intersections between straight lines and planar surfaces are easy to obtain [22], but become more complex when the objects contain curved surfaces [16, 18]. It is hard to intersect two general surfaces [15]. Object-space algorithms can only handle objects with at most quadric surfaces so far [17, 19].

Image-space algorithms, on the other hand, are capable of rendering pictures that contain objects with a richer range of shapes and usually faster since they do not have to perform object-object comparisons, sorting, and difficult intersection finding, which are typically very time-consuming and difficult to implement, especially when objects exhibit complicated and curved shapes. It is clear that for time-critical rendering, or when the scene contains objects with very complex shapes, image-space algorithms are superior to object-space counterparts for the reason of simplicity.

The following sections examine and compare several major object- and image-space hidden-line elimination algorithms.

2.2 Back-Face Culling

The basic concept in back-face culling involves plotting only surfaces "facing the camera" since the back side of objects are invisible. This technique is an object-space

approach. It can remove approximately 50% of the surfaces in a scene viewed in parallel projection and somewhat greater than 50% of surfaces in perspective projections. The closer the objects are to the center of projection (COP) in perspective projection, the higher percentage of surfaces that the back-face algorithm removes. If an object is convex then all the hidden-lines are removed.

The back-face culling technique is the simplest hidden-line algorithm for single convex polygonal volumes. However, it applies only to objects considered individually. It does not take into consideration the "interaction" between objects, i.e., many surfaces surviving the back-face culling algorithm ("front-faces") may still be obscured by front-face even closer to the viewer.

Further, in a scene containing highly reflective objects, the surface of an object, which could be back-faces if the object is isolated, may be reflected in the front-faces of an adjacent object. Culling or depth sorting techniques cannot be used for such scenes. In other cases, they can be used to eliminate the back-faces from a scene before applying most of the hidden-line algorithms to be discussed in the remaining sections of this chapter.

2.3 Warnock's Area Subdivision Algorithm

The basic ideas behind the Warnock algorithm [27, 28] are very general. They are, by analogy, based on an hypothesis of how the human eye-brain combination processes information contained in a scene. The hypothesis is that very little time or effort is expended on areas that contain little information. The majority of time and effort is spent on areas of high information content. The Warnock algorithm and its derivatives attempt to take advantage of the fact that large areas of a display are similar. This characteristic is known as area coherence; i.e., adjacent areas (pixels) in both x and y directions tend to be similar.

Since the Warnock algorithm is concerned with what is displayed, it works in image-space. It considers a window in image-space and seeks to determine if the window is empty or if the contents of the window are simple enough to display. If not, the window is subdivided until either the contents of a sub-window are simple enough to display or the sub-window size is at the limit of desired resolution. In the latter case, the remaining information in the window is evaluated and the result displayed at a single intensity or color. Anti-aliasing can be incorporated by carrying the subdivision process to less than display pixel resolution and averaging the sub-pixel attributes to determine the display pixel attributes.

2.4 Z-Buffer (Depth-Buffer) Algorithm

The Z-Buffer algorithm is one of the simplest hidden-line algorithms to implement in either software or hardware. The technique was originally proposed by Catmull [9] and is an image-space algorithm. The z buffer is a simple extension of the frame buffer idea. A frame buffer is used to store the attributes (intensity) of each pixel in image-space. The z buffer is a separate depth buffer, with the same number of entries as the frame buffer, used to store the z coordinate or depth of every visible pixel in image-space. In use, the depth or z value of a new pixel to be written to the frame buffer is compared to the depth of that pixel stored in the z buffer which is initialized to a distant value. If the comparison indicates that the new pixel is in front of the pixel stored in the frame buffer, then the new pixel is written to the frame buffer and the z buffer updated with the new z value. If not, no action is taken. Conceptually, the algorithm is a search over x and y for the largest value of $z(x, y)$.

The simplicity of the algorithm is its greatest advantage. In addition, it handles the hidden-line problem and the display of complex surface intersections trivially. Scenes can be of any complexity. As image-space is of fixed size, the increase in computational work with the complexity of the scene is at most linear. Since elements of a scene or picture can be written to the frame or z buffer in arbitrary order, they

do not have to be sorted into depth priority order. Hence, the computation time associated with the object-object comparisons and depth pre-sort which are often very complex is eliminated.

The amount of storage required is the principal disadvantage of this algorithm. The size of the z buffer depends on the accuracy to which the depth value of each point (x, y) is to be stored, which is a function of scene complexity. If the scene is transformed and clipped to a fixed range of z coordinates, then a z buffer of fixed precision can be used. Depth information must be maintained to a higher precision than lateral x, y information; 20–32 bits is usually sufficient. A $512 \times 512 \times 24$ bit frame buffer, in combination with a $512 \times 512 \times 20$ bit z buffer, requires almost 1.5 megabytes of storage. If the requirement taxes the computing resources available, the algorithm may be decomposed into individual scan line arrays and performed one scan line at a time. This approach is called scan line Z-Buffer algorithm. (Refer to Section 2.6.1 for more details.) However, the current decrease in memory costs is making dedicated z buffer memory and associated hardware practical.

A further disadvantage of the z buffer is the difficulty and expense of implementing anti-aliasing, transparency, and translucency effects. Because the algorithm writes pixels to the frame buffer in arbitrary order, the necessary information for pre-filtering

anti-aliasing techniques is not easily available. For transparency and translucency effects, pixels may be written to the frame buffer in incorrect order, leading to local errors. The A-Buffer (anti-aliased, area-averaged, accumulator buffer) algorithm [8] addresses this problem by using a discrete approximation to unweighted area sampling. The significant advantage of this approach is that floating point geometry calculations are avoided.

2.5 List Priority (Depth Sorting) Algorithm

The implementation of all the hidden-line algorithms discussed above involves establishing the priority, i.e., the depth or distance from the viewpoint, of objects in a scene. The list priority algorithms attempt to capitalize on this by performing the depth or priority sort first. The objective of the sorting is to obtain a definitive list of scene elements in depth priority order based on distance from the viewpoint. If the list is definitive, then no two elements overlap in depth. Starting with the scene element farthest from the viewpoint, each element is written to a frame buffer in turn. Closer elements on the list overwrite the contents of the frame buffer. Thus, the hidden-line problem is trivially solved. Transparency effects can be incorporated into the algorithm by only partially overwriting the contents of the frame buffer with

the attributes of the transparent element [21]. This technique is sometimes called the painter's algorithm because it is analogous to that used by an artist in creating a painting.

The list priority algorithms involve an object pre-sorting which may be very complicated when objects overlap in z direction, or cyclically overlap each other, or penetrate each other. In these cases, it will be necessary to split one or more objects to make a linear order possible.

The list priority algorithms operate in both object and image-space. In particular, the priority list calculations are carried out in object-space and the result written to an image-space frame buffer. The use of a frame buffer is critical to this algorithm.

2.6 Scan Line Algorithms

The Warnock, Z-Buffer, and list priority algorithms process scene elements in arbitrary order with respect to the display. The scan line algorithms [31, 5, 6, 29] process the scene in scan line order. Scan line algorithms operate in image-space. They process the image one scan line at a time rather than one pixel at a time. By using area coherence of the polygon, the processing efficiency is improved over the pixel

oriented method.

Using an active edge table, the scan line algorithm keeps track of where the projection beam is at any given time during the scan line sweep. When it enters the projection of a polygon, the beam switches from the background color to the color of the polygon. After the beam leaves the polygon's edge, the color switches back to background color. To this point, no depth information need be calculated at all. However, when the scan line beam finds itself in two or more polygons, it becomes necessary to perform a z-depth sort and select the color of the nearest polygon as the painting color.

2.6.1 Scan Line Z-Buffer Algorithm

One of the simplest scan line algorithms that solves the hidden-line problem is a special case of the Z-Buffer algorithm discussed in Section 2.4. It is called scan line Z-Buffer algorithm [20]. In this algorithm the display window is one scan line high by the horizontal resolution of the display wide. Therefore both the frame buffer and the z buffer need only to be 1 bit high by the horizontal resolution of the display wide by the requisite precision deep. The required depth precision depends on the range of z. Its main advantage lies in the small amount of memory it requires in comparison

to a full-blown Z-Buffer. Both pre- and post-filtering anti-aliasing techniques can be used with the scan line Z-Buffer algorithm.

2.6.2 Spanning Scan Line Algorithm

Rather than solving the hidden-line problem on a pixel-by-pixel basis using incremental z calculation, the spanning scan line algorithm uses spans along the scan line over which there is no depth conflict [29]. The hidden-line removal process uses coherence in x and deals in units of many pixels. The processing implication is that a sort in x is required for each scan line and the spans have to be evaluated. The major drawback is the increase in complexity of the algorithm itself.

2.7 Visible Surface Ray Tracing Algorithm

All the hidden-line algorithms discussed in the previous sections depend on some coherence characteristic of the scene to find the visible portions of a scene. In comparison, ray tracing is a brute force technique. The basic idea underlying the technique is that an observer views an object by means of light from a source that strikes the object and then somehow reaches the observer. The light may reach the observer

by reflection from the surface or by refraction or transmission through the object. If light rays from the source are traced, very few will reach the viewer. Consequently, the process would be computationally inefficient. Appel [1] originally suggested that rays should be traced in the opposite direction, i.e., from the observer to the object.

The most important element of a ray tracing algorithm is the intersection routine. Any object for which an intersection routine can be written may be included in a scene. Determining the intersections of an arbitrary line in space (a ray) with a particular object may be computationally expensive. Since a ray tracing algorithm spends up to 95% of its effort in determining intersections [30], the efficiency of the intersection routine significantly affects the efficiency of the algorithm.

For multiple intersections of the ray being traced and objects in the scene, it is necessary to determine the visible intersection. For the simple opaque visible surface algorithms, the intersection with the maximum z coordinate is the visible surface. For more complex algorithms with reflection and refraction, the intersections must be ordered with respect to the distance from the point of origin of the ray. A transformed coordinate system allows this to be accomplished with a simple z sort.

It should be clear that the normal back-face culling operation commonly used by hidden-line algorithms cannot be used with a ray tracing algorithm. Further, an

initial priority sort to determine visible faces also cannot be used. For example, an object totally obscured by another object may be visible as a reflection in a third object. Since a ray tracing algorithm is a brute force technique, the opaque visible surface algorithm discussed in previous sections are more efficient and should be used.

Roth [25] points out that a ray tracing algorithm can also be used to generate the hidden-line removed wireframe line-drawings for solid objects. The procedure assumes a scan-line-oriented generation of the rays, i.e., top to bottom and left to right. The procedure is

> If the visible surface at Pixel(x, y) is the background or is different from the visible surface at Pixel$(x-1, y)$ or at Pixel$(x, y-1)$, display the pixel. Otherwise, do not display the pixel.

Because of the inherently parallel nature of ray tracing (the process for each ray is the same and independent of the results for any other ray) the algorithm could be implemented in very large scale integrated (VLSI) hardware using parallel processing techniques.

Table 2.1: Complexity comparison

	Input Size	Output Size	Lower Bound	Upper Bound
Image-space	n	image	$\Omega(n)$	$O(n)$
Object-space	n	$O(n^2)$	$\Omega(n^2)$	$O(n^2 \log n)$

2.8 Comparison

It has long been known that there is a fundamental relationship between sorting and the hidden-line problem [26]. However, the hidden-line problem for geometrically complex scenes has a greater computational complexity than sorting, since a large number of visible objects may be produced with respect to a given set of input objects. The complexity of the hidden-line problem thus depends on both the input and output size of the problem instance. Table 2.1 shows that image-space algorithms are more efficient than object-space ones [11].

Implementation of the algorithms as described in the previous sections in the same language on the same computer system for the same scene yields performance ratios of table 2.2 [23]. Another informal estimate is shown in table 2.3 [26].

Table 2.2: Comparison of some hidden-line algorithms

Algorithm	Performance ratio
Ray Tracing	9.2
Warnock	6.2
Spanning Scan Line	2.1
Scan Line Z-Buffer	1.9
Z-Buffer	1

Table 2.3: Comparison of some hidden-line algorithms

Algorithm	Number of Polygonal faces in scene		
	100	2,500	60,000
Depth Sort	1	10	507
Warnock	11	64	307
Scan Line	5	21	100
Z-Buffer	54	54	54

All the formal analyses and the informal estimates of the computational complexity show that from the efficiency and ease of implementation points of view the Z-Buffer algorithm is the best. It has significant memory requirements, particularly for high resolution frame buffers. However, it places no upwards limit on the complexity of scenes, an advantage that is becoming increasingly important.

An important restriction it places on the type of object that can be rendered by the Z-Buffer algorithm is that it cannot deal with transparent objects without costly modification. Besides, anti-aliasing solutions, particularly hardware implementations, are also difficult.

If memory requirements are too prodigious then the scan line Z-Buffer algorithm is the next best solution. Unless a rendering is to work efficiently on simple scenes, it is doubtful whether it is worth contemplating the large increase in complexity that a spanning scan line algorithm demands.

Chapter 3

The D-Buffer Algorithm

3.1 Introduction

The purpose of this thesis is to seek a fast way to display hidden-line segments in a different style, such as dotted or dashed, instead of suppressing them. Unfortunately, none of the algorithms discussed in last chapter can be used directly for this goal.

All object-space hidden-line algorithms can be easily adapted to show hidden-lines as dotted, as dashed, of lower intensity, or with some other rendering style supported by the display device, after they obtain the endpoints and equations of all hidden-line

segments. Appel, Rohlf, and Stein [2] describe another algorithm for rendering haloed lines. Each line is surrounded on both sides by a halo that obscures those parts of lines passing behind it. Lines that pass behind others are obscured only around their intersection on the view plane. The algorithm intersects each line with those passing in front of it, keeps track of those sections that are obscured by halos, and draws the visible sections of each line after the intersections have been calculated. Though this algorithm can partially fulfill our request, it involves a great deal of intersection computation. Besides, as all other object-space hidden-line algorithms, it is also difficult for this algorithm to handle complicated surfaces with curved boundary edges and silhouette lines.

An image-space hidden-line algorithm has to be developed for the reasons of efficiency and the capability of rendering pictures that contain complicated objects. Though the Z-Buffer algorithm is the fastest approach and has no limitation on the shape of objects, it is designed for hidden-line or hidden-surface elimination. It discards all the information except the "closest" surface, therefore cannot provide sufficient structure information of the displayed objects as required in applications. A new algorithm is needed to reveal the concealed information.

3.2 P-Buffer Algorithm

In order to reveal the concealed information, Yuan and Sun develop a modified Z-Buffer algorithm named the P-Buffer algorithm [32]. It is also an image-space algorithm, using an additional pattern buffer which defines a grid of filtering pattern. This buffer has the same size in x and y directions as the depth and frame buffers in the Z-Buffer algorithm. The value of each element in the pattern buffer is either "1" or "0". Hidden-lines are displayed with the "1"s while broken into dashes and dots with the "0"s. By using this pattern buffer, the algorithm is capable of distinguishing dashed hidden-lines from the solid visible lines.

The result image of the P-Buffer algorithm heavily depends on the selection of the filtering pattern. This pattern must be able to handle all kinds of boundary lines of complicated objects that may be almost any shape. Unfortunately, in any given $m \times n$ array of 0 and 1, by the means of 8-neighbors connectedness, there is always at least one path that consists of either all "0"s or all "1"s whose length is at least $\min(m, n)$. (See next section for the exact definitions of 8-neighbors, path, and length.) That means, no matter how well a filtering pattern is generated, there are always chances to create a "dash" (when the path is formed all by "1"s) and/or "space" (when all "0"s) not shorter than $\min(m, n)$, where m and n are the height and width of the

image, respectively. Such a dash or space is too long to be acceptable. In addition, the algorithm cannot guarantee the even dashes and spaces. Therefore, no bitmap can be used in the P-Buffer algorithm as a universal filtering pattern.

The fatal problem of the P-Buffer algorithm is that it tried to use a *static* filtering pattern to deal with various shapes. This proved impossible. In contrast with its fixed pattern, a *dynamic* pattern is needed to generate the dotted and dashed hidden-lines. Here dynamic means auto-adapting to object shapes. Neighborhood operation, an image processing approach, is a powerful tool to perform local adaptation in a digital image.

3.3 Neighborhood Operation

Actually, the name of "image" space algorithm suggests that some image processing concepts and techniques may contribute to finding the required new algorithm. Image processing is usually associated with pattern recognition and is rather treated as a subject outside of the computer graphics interest. Basically computer graphics algorithms are used for the visualization of scenes or models described using some abstract notation, while image processing is used in the opposite way, i.e., when find-

ing an abstract description of an analyzed pattern. However, some image processing methods can be used as a computer graphics tool [24]. Neighborhood operation is one of the most important and most useful image processing approaches that can be used in computer graphics algorithms.

Before discussing the neighborhood operation, some definitions of the basic concepts of adjacency, connectedness, and components have to be given first.

Let (x, y) be a point of a given digital image. Then (x, y) has four horizontal and vertical neighbors, namely the points

$$(x-1, y),\ (x+1, y),\ (x, y-1),\ (x, y+1)$$

These points are called the *4-neighbors* of (x, y), and are said to be *4-adjacent* to (x, y). In addition, (x, y) has four diagonal neighbors, namely

$$(x-1, y-1),\ (x-1, y+1),\ (x+1, y-1),\ (x+1, y+1)$$

Both the diagonal neighbors and the 4-neighbors are called *8-neighbors* of (x, y). If (x, y) is on the border of the image, some of these neighbors will be outside the image. More generally, if S and T are image subsets, we say that S is 4- or 8-adjacent to T

if some point of S is 4- or 8-adjacent to some point of T.

$(x-1, y+1)$	$(x, y+1)$	$(x+1, y+1)$
$(x-1, y)$	(x, y)	$(x+1, y)$
$(x-1, y-1)$	$(x, y-1)$	$(x+1, y-1)$

In the illustration of the 3×3 neighborhood of a point, Cartesian coordinates (x, y) are used, with x increasing to the right and y increasing upward. There are other possibilities; for example, one could use matrix coordinates (m, n), in which m increases downward and n to the right. Note that the diagonal neighbors are $\sqrt{2}$ units away from (x, y), while the horizontal and vertical neighbors are only one unit away. If a pixel is treated as a unit square, the horizontal and vertical neighbors of (x, y) share a side with (x, y), while its diagonal neighbors only touch it at a corner.

A *path* from (i, j) to (h, k) is a sequence of distinct points

$$(i, j) = (x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n) = (h, k)$$

such that (x_m, y_m) is adjacent to (x_{m-1}, y_{m-1}), $1 \leq m \leq n$. Note that there are two versions of this, 4-path or 8-path, depending on whether "adjacent" means "4-adjacent" or "8-adjacent". Here n is called the *length* of the path.

If $p = (i, j)$ and $q = (h, k)$ are points of a image subset S, we say that p is *connected* to q (in S) if there is a path from p to q consisting entirely of points of S. For any point (x, y) of S, the set of points of S that are connected to (x, y) is called a connected *component* of S. If S has only one component, it is called *connected.*

Neighborhood operations are those that in some form or fashion combine a small area of pixels, or neighborhood, to generate an output pixel. Such an operation is thus defined as being "spatially dependent" since it depends on the pixel values at positions other than the pixel under immediate consideration. This is different from *point operations*, which rely only on a single pixel or single pixels from multiple images to perform a function. The uses and consequences of neighborhood operations are wide-ranging. The most important neighborhood operations are *convolution* and *sampling.* The applications of neighborhood operations are diverse, ranging from digital filters to techniques for smoothing, sharpening, transforming, and warping [3, 7].

The neighborhood is often taken to mean the 4-neighbors or 8-neighbors defined above, since a neighborhood of 3×3 pixels is very commonly considered. However, any neighborhood size is theoretically allowable up to the maximum spatial resolution of the system, but the computational expense of processing increases dramatically as the size of the neighborhood increases. 8-adjacent connectedness is used in this thesis.

Neighborhood thus means the 8-neighbors.

3.4 Dotted D-Buffer Algorithm

The neighborhood operations can be applied to the "dynamic" pattern designing. Object-space algorithms can easily draw hidden-lines in various styles because they know the endpoints and the equations of every hidden-line segment. Image-space algorithms, though having no such endpoint and equation information, can "trace" the hidden-lines in the image plane by using neighborhood operations.

Consider the dotted style first. A static chess board pattern can generate ideal horizontal and vertical dotted lines. But it cannot properly handle lines with other angles. Especially when a straight line runs cross the diagonals of the pattern, this line will be either completely eliminated or totally untouched depending on what kind of pattern grids it runs through. In comparison, a dynamic pattern, which is generated along the lines while the image itself is being generated, will perfectly fit arbitrary lines or curves. This thesis first develops a new algorithm, called Dotted D-Buffer, using the concept of neighborhood operations to dynamically generate the dotted hidden-lines. Appendix A shows the algorithm in pseudo-code. Its modified version,

namely Dashed D-Buffer, that can generate dashed hidden-lines is to be presented in the next section.

Here D-Buffer refers to *Dynamical*-Buffer. It is a new image-space algorithm based on the Z-Buffer algorithm, borrowing some ideas from the P-Buffer algorithm. The main point of the D-Buffer algorithm is to generate a dynamic or auto-adaptive filtering pattern by "tracing" the hidden-lines. Using this dynamic pattern, the D-Buffer algorithm can generate perfect dotted or dashed hidden-lines.

In addition of a frame buffer and a depth buffer used by the traditional Z-Buffer algorithm, the Dotted D-Buffer algorithm uses an additional boundary buffer. All three buffers have the same size as the image. The depth buffer has the same meaning and usage as in the Z-Buffer algorithm, maintaining the closest depth value of every pixel of the image. The final content of the frame buffer is the created picture, though each of its entries has one of three possible states before the final result is written. The new boundary buffer contains all the boundaries information together with their depth values, i.e., non-boundary pixels have the background value, while the others (on boundaries) have various closer values.

The D-Buffer algorithm contains two major steps, collect enough depth information and break the hidden-lines accordingly.

In the first step, the algorithm processes every point on every object to find the depth and boundary information needed in the second step. The pseudo-code in Appendix A shows that after initializing the three buffers, the algorithm executes a pair of nested `while` loops. The outer `while` loop processes one by one all the objects in the scene. Function `Object_Retrieve(OBJECT*, OBJECT*)` returns `TRUE` if an object is retrieved or `FALSE` when the object list is finished. Then the inner `while` loop processes one by one every pixel in the projection of the retrieved object. Function `Pixel_Determine(OBJECT, int*, int*, int*, boolean*)` returns `TRUE` if a pixel is calculated or `FALSE` if all pixels are processed. In addition to the projected position `(x, y)` and the depth value `z`, this function also indicates by its last parameter whether this pixel is on a boundary line.

The D-Buffer algorithm does two things inside the nested loops. First, as the same as in the Z-Buffer algorithm, it updates the depth buffer if the depth value of the new determined pixel is closer than what the depth buffer has at this pixel. The second thing, however, is different from that in the Z-Buffer algorithm. Instead of updating the frame buffer directly, the D-Buffer algorithm records the boundary information in the extra boundary buffer. The boundary buffer has the same size and data type as the depth buffer. It stores depth values as well, but only for boundary pixels. All the other entries of the boundary buffer have a distant value indicating background.

	allow to change (1)	NOT allow to change (0)
white (0)	UNDECIDED (0x10)	WHITE (0x00)
black (1)	UNUSED (0x11)	BLACK (0x01)

Table 3.1: Possible values of the frame buffer entries

Every boundary point, if it is not blocked by any other boundary points, has its depth value recorded in the boundary buffer.

After setting the depth buffer to the closest surfaces of all the objects in the scene and the boundary buffer with the information of all boundary lines, the Dotted D-Buffer algorithm begins to dot the hidden-lines.

In this second step, the algorithm generates the image by writing to the frame buffer. A pixel in the frame buffer can be either white (non-boundary) or black (boundary). Besides, the value of each entry of the frame buffer can either be changeable or have to keep its current value. Two bits are used accordingly in the algorithm to indicate the status. The four possible combinations are showed in table 3.1.

The initial status is `UNDECIDED (0x10)`, so at the beginning the frame buffer is

pure white and every pixel can be changed to black. For each pixel, the algorithm determines whether it is on a boundary line and then whether it is in the front by checking its values in the boundary and depth buffers. If it is not on any boundary line, it is left unchanged. If it is on a boundary in front, its value in the frame buffer is changed *unconditionally* to `BLACK (0x01)`, which means this pixel is blackened and can not be changed any more. This is a point operation in terms of image processing. When a pixel is on a boundary in behind, i.e., a hidden-line, the algorithm must then check if it is changeable. If the answer is no, nothing is done; otherwise, it changes this pixel to `BLACK (0x01)` as a dot and the pixels "around" it to `WHITE (0x00)` to make the line dotted since the first bit in `WHITE (0x00)` implies an unchangeable pixel. The term "around" means the "underneath and right" neighbors (P_{5-8} in the following figure) of the pixel being processed (P_0) since the "above and left" neighbors (P_{1-4}) have already been processed. This is a neighborhood operation in the sense that the values of P_5, P_6, P_7, and P_8 are determined by the value of P_0. In this way, the Dotted D-Buffer algorithm produces very evenly dotted hidden-lines (Figure 3.1(a)).

P_2	P_3	P_4
P_1	P_0	P_5
P_8	P_7	P_6

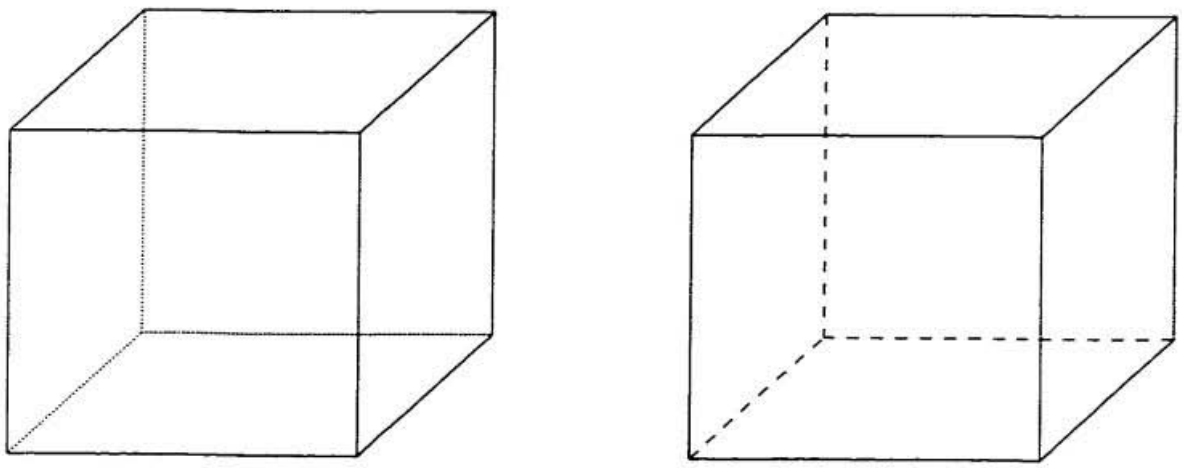


Figure 3.1: Dotted and Dashed Hidden-Line

Note that the algorithm "unconditionally" blackens a pixel when it is in the front and on a boundary line no matter what the first bit of its frame value is. It is more precise to say that the first part of a value in the frame buffer indicates whether or not this pixel can be affected by a hidden-line.

After a pixel has been processed, the first bit of its frame value is no longer useful. The second bit alone describes completely the final created image. Therefore, the last statement in the Dotted D-Buffer algorithm simply eliminates the first part and reduces the frame buffer to the normal one-bit image.

Some image processing operations are intrinsically *parallel* in the sense that the same rule must be applied to many data and the order in which the data are processed

does not matter. Examples are single pixel operations such as contrast manipulation or threshold and neighborhood operations such as convolution, erosion/dilation and non-linear filtering. On the other hand, some other image processing operations are intrinsically *serial* in the sense that the order in which data are processed is important. Examples are searching, measurement and classification. The Dotted D-Buffer algorithm is a serial image processing operation.

3.5 Dashed D-Buffer Algorithm

The Dotted D-Buffer algorithm generates perfect dotted hidden-lines but it can still be improved for other applications. For instance, on today's high resolution display devices, such as monitors or printers, a single pixel is too small to easily distinguish from its neighbors with naked eyes. That is why the hidden-lines in Figure 3.1(a), when viewed from a distance, seem to be grey solid lines rather than black dotted lines. It is because the Dotted D-Buffer algorithm has no flexibility of generating dots and spaces of variable size. In most cases, people prefer dashed hidden-lines over dotted hidden-lines, especially when the lengths of the dashes and spaces are adjustable.

A Dashed D-Buffer algorithm to be presented in this section can fulfill this requirement. It is an improvement of the Dotted D-Buffer algorithm. Instead of setting spaces around every dot on the hidden-lines to generate dotted hidden-lines, the Dashed D-Buffer algorithm really "traces" the hidden-lines to make them evenly dashed. Appendix B shows its pseudo-code.

The Dashed D-Buffer algorithm also containts two major steps, same as the Dotted D-Buffer, collect depth information and break the hidden-lines.

The first step of the two algorithms are exactly the same. They both use the same three buffers: frame buffer, depth buffer, and boundary buffer; and they both use the same 2-bit values (Table 3.1) in the frame buffer which is initialized to `UNDECIDED` (`0x10`) at the beginning of the algorithms. In addition, they both obtain the same depth information of the surfaces (in depth buffer) and boundaries (in boundary buffer) by using the same pair of nested `while` loops. However, the second step of the two algorithms are different in the loops for every pixel. Rather than setting dots and spaces directly, the Dashed D-Buffer algorithm uses a recursive function to make the hidden-lines dashed.

When given an initial pixel on a boundary line, function `dash(int, int, int, int, int**, double**, double**)` traces the line and determines if the line should

be solid or dashed. The first two parameters present the pixel in question. The fourth parameter indicates whether a dash or a space is being drawn if the line is invisible, and the third parameter shows how long it has already been drawn. The rest three parameters are the three buffers. This function is called for every pixel on boundaries. It checks whether the pixel is in front by comparing the values in boundary buffer and depth buffer. If it is in front, its value in the frame buffer is changed to **BLACK** (0x01) without question. If it is in behind, it is set to the current dash line color, either **BLACK** (for dash) or **WHITE** (for space). After deciding the color, function **dash()** checks the length of dash or space. If it is long enough, the color is reversed (from **BLACK** to **WHITE**, or from **WHITE** to **BLACK**) and the length reset. The algorithm uses **DASH_THRESHOLD** to control the length of the dashes and the spaces in between. It can produce different effects by simply adjusting this threshold value. The single threshold make the length of the dashes and spaces the same. Another threshold, suitably named **SPACE_THRESHOLD**, can also be introduced to control the length of the dashes and spaces separately. In this way, the algorithm can create long dashes with short spaces or the reverse.

After the current pixel is processed and the color for the next pixel prepared, function **dash()** recursively calls itself to the 8-neighbors of the current pixel. Some criteria are involved to make sure that the neighbor is actually part of a line and was

not processed before. This ensures that the algorithm does not trace to a previously processed pixel to avoid processing that pixel more than once. By using the recursive calls, the algorithm travels through every boundary line, making visible ones solid and invisible ones dashed. All the non-boundary pixels are never touched, so they remain value **UNDECIDED** in the frame buffer. The last action in the Dashed D-Buffer algorithm simply eliminates the first bit and reduces the frame buffer to the normal one-bit image, i.e., changes **UNDECIDED** to **WHITE**. A result is showed in Figure 3.1(b)).

The Dashed D-Buffer algorithm is also a serial image processing operation.

Chapter 4

Implementation and Discussion

Both Doted and Dashed D-Buffer algorithms are implemented in C on UNIX.

4.1 Discussion

Z-Buffer is an image-space algorithm. Table 2.1 on page 24 shows that its computational complexity is $O(n)$, where n is the number of objects. It calculates the depth value of every pixel of every object's projection, and compares this value to that stored in the depth buffer to determine the visibility. The calculation may be sometimes complicated when the object has a complex shape.

In comparison to the Z-Buffer algorithm, the D-Buffer algorithm uses one more boundary buffer and performs one more operation, i.e., checking every boundary pixel to generate dotted or dashed hidden-lines and solid frontal lines. This operation is only related to the number of boundary pixels which are far fewer than the total pixels processed by the Z-Buffer algorithm, yet has nothing to do with the complexity of the scene such as the number, shape, size, and position of the objects. For each boundary pixel, there is only a couple of conditional and assignment statements within the loop. In the Dashed D-Buffer algorithm, though most boundary pixels can be reached by different neighbors from different directions or different boundary lines, each boundary pixel is processed only once since the algorithm always checks whether it is `UNDECIDED` before calling `dash()` function to trace it. For non-boundary pixels, they are never processed because the algorithm also always checks whether the pixel is on a boundary line. In short, the additional operation does not increase the computational complexity level because it performs simpler processing on fewer pixels. Therefore, the D-Buffer algorithm is in the same computational complexity level as the Z-Buffer algorithm.

4.2 Experiment with Polyhedrons

To represent objects in three-dimensional space, polyhedrons are widely used in computer graphics systems. Every polyhedron consists of a set of smoothly joined polygons. By appropriately selecting polygons' shape and number, polyhedrons are capable of modeling any three-dimensional object to the desired degree of accuracy. Some objects, e.g. tetrahedra and cubes, may be modeled precisely by a set of four equilateral triangles or six squares, respectively. Others, such as spheres and cylinders, may be approximated by combinations of trapezoids, triangles, rectangles, and n-sided polygons. For certain applications a coarse polygon grid may be adequate. For applications requiring greater accuracy, more polygons with smaller grid spacing may be required.

The polyhedra-based representation of scenes has the advantages of simplicity, generality, and computational efficiency. Objects may be manipulated and transformed by operating on the points composing the polygons. Polygon surfaces have well-defined orientations which simplify its computation of visibility and shading. The major weakness of polyhedral representations is their poor approximation to smooth curved surfaces, complex shapes, and life-like forms. Simply increasing the number of polygons to achieve visually realistic representations of complex scenes will not only

overwhelm the storage and computing capacity of even large computers, but also introduce many unwelcome lines when the objects are rendered in wireframe mode.

Since the popularity of the polygon mesh modeling technique in computer graphics is undoubtedly due to its inherent simplicity and the development of inexpensive shading algorithms that work with such models, the D-Buffer algorithm has been tested on polyhedrons first. The Z-Buffer algorithm is also executed with the same set of tested objects to compare the performance of the D-Buffer algorithm. Table 4.1 shows the experiment results, which clearly indicate that the D-Buffer algorithm has the same computational complexity level as the Z-Buffer algorithm. Figure 4.1 to 4.4 show the rendered images.

4.3 Experiment with Curved Objects

Other than the polyhedral representation of three-dimensional objects, more abstract representation is available through parametric curved surfaces. The parametric representation of solids and curves is now an established tool in computer graphics, particularly in CAD. With a relatively small set of parameters, significant portions of object surfaces may be accurately modeled. The whole object may, in turn, be

	Pin Thin mesh	Pin Dense mesh	Wheel Thin mesh	Wheel Dense mesh
Number of vertices	227	227	388	388
Number of polygons	211	332	198	612
Wireframe	0.36	0.38	0.37	0.44
Z-Buffer	2.0	2.2	3.8	4.1
Dotted D-Buffer	2.1	2.3	3.9	4.2
Dashed D-Buffer	2.4	2.5	4.2	4.7

Time unit: second

Table 4.1: Experiment Results

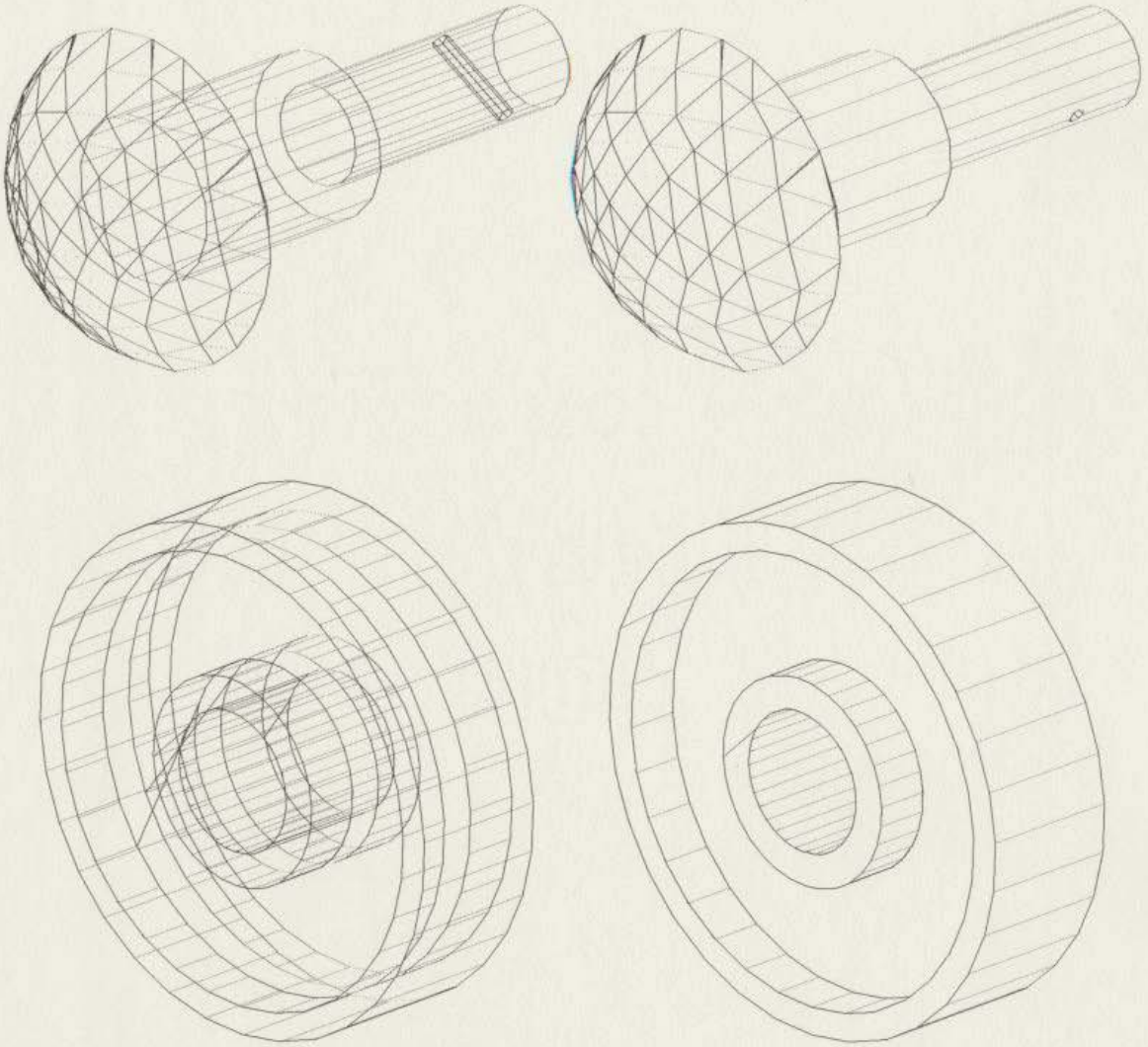


Figure 4.1: Thin mesh: Wireframe vs. Z-Buffer

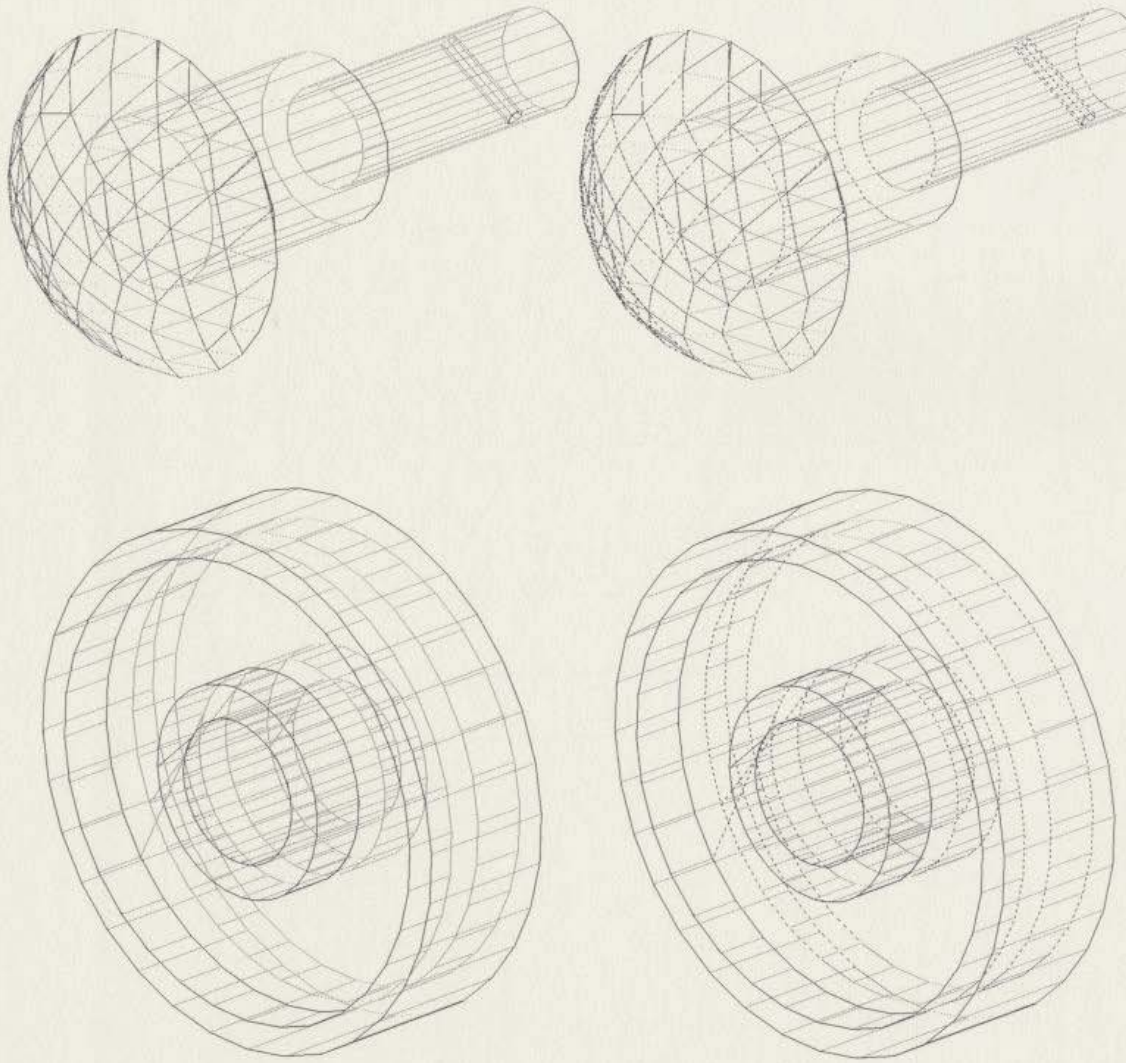


Figure 4.2: Thin mesh: Dotted and Dashed D-Buffer

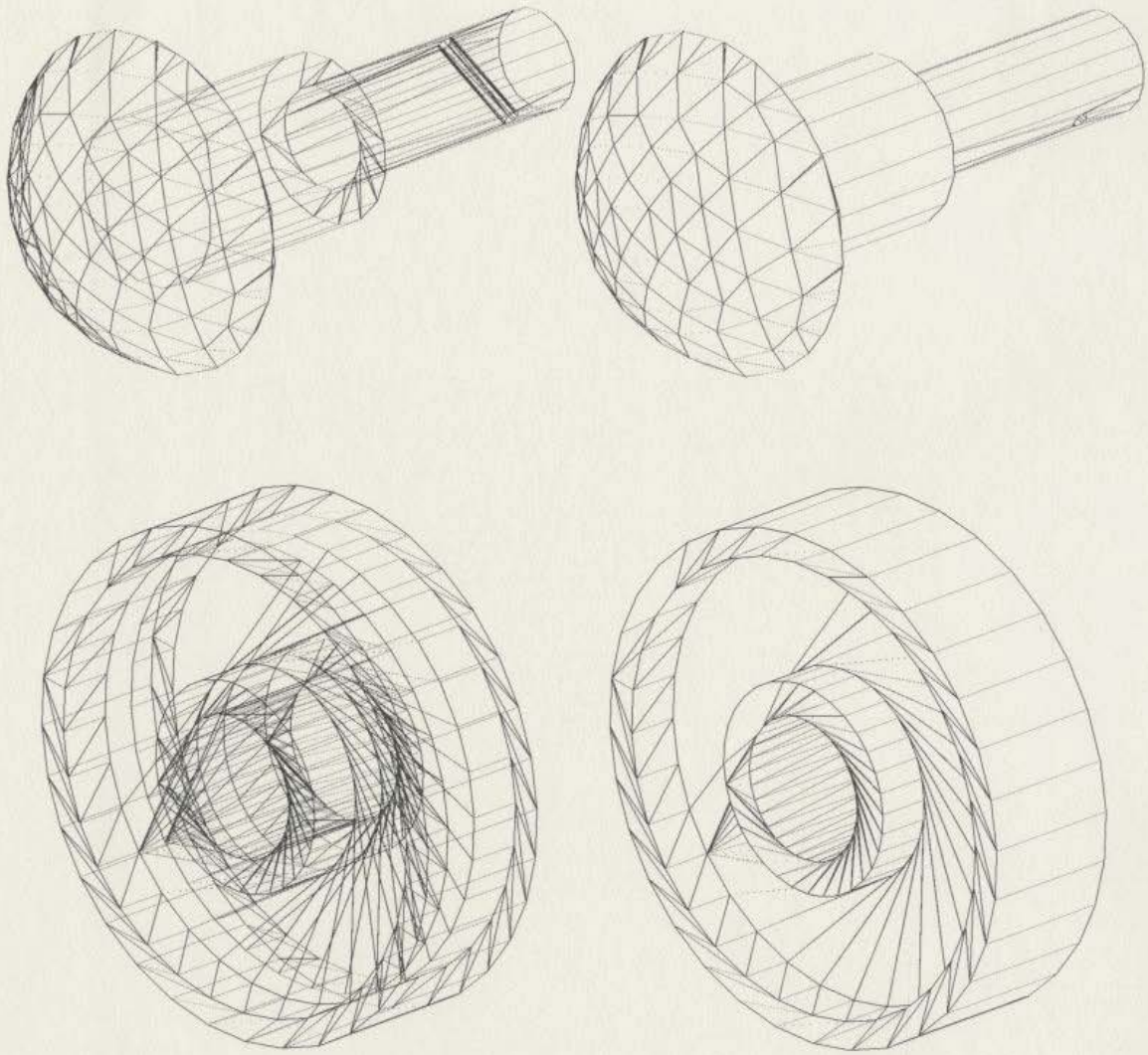


Figure 4.3: Dense mesh: Wireframe vs. Z-Buffer

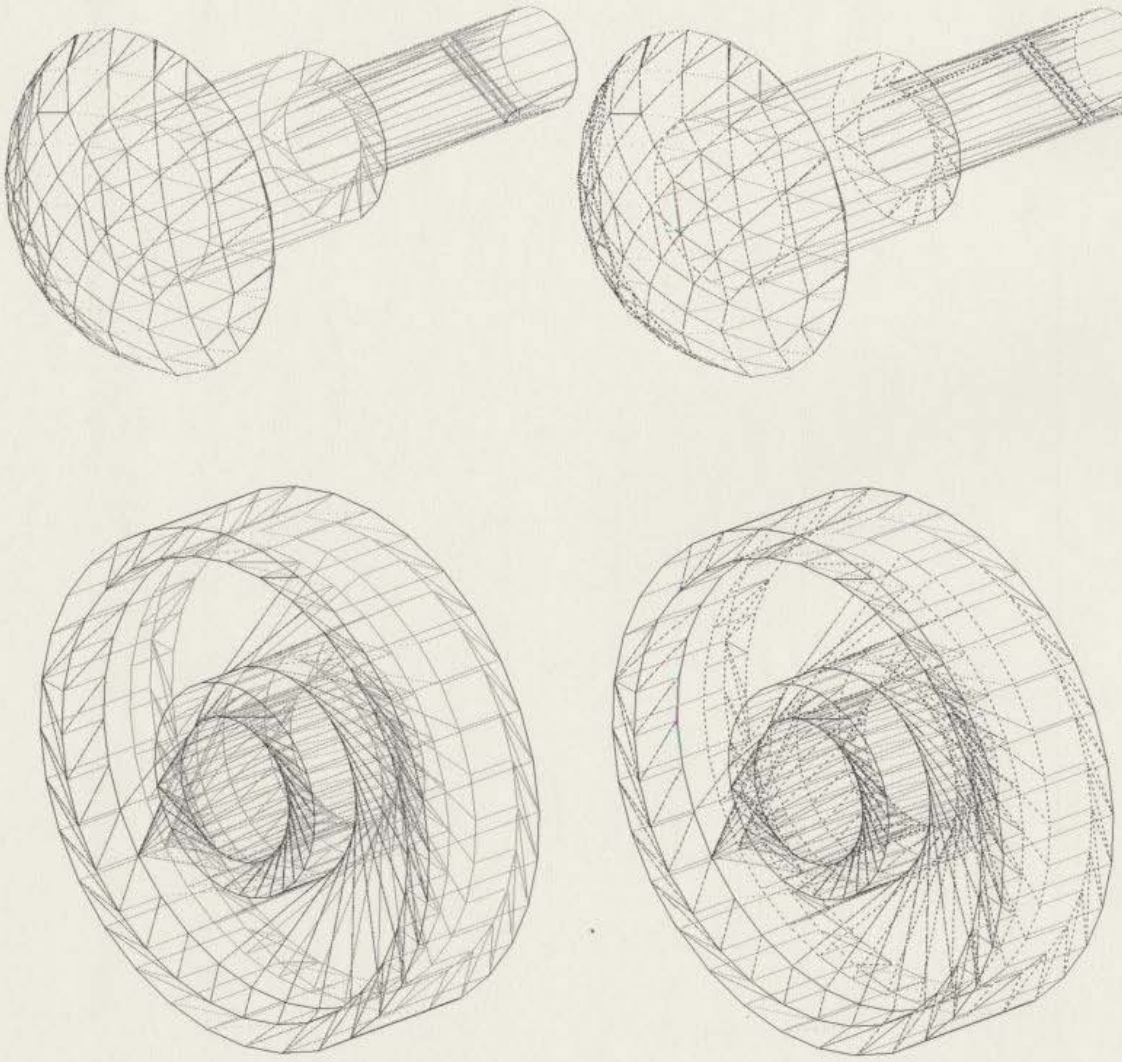


Figure 4.4: Dense mesh: Dotted and Dashed D-Buffer

represented by smoothly joining a set of parametric patches.

The advantages of this representation are efficient storage and a high degree of accuracy. The main disadvantages of parametric curved surface representations stem from the difficulties in determining the correct set of curved patches which model real objects. Another problem arises in wireframe representation of objects with curved surfaces: it is the ambiguity of the interpretation of lines and surfaces with respect to their origins of three-dimensional objects [33]. Edges that are formed from intersections of two surfaces are intrinsic to the object, and are explicitly represented in three-dimensional solid model. The existence of these *real edges* is viewing-direction independent. Some other edges are formed due to the variation of surface normal with respect to projection direction. These *virtual edges* or *silhouette lines* only appear in the process of projecting the three-dimensional curved surfaces to two-dimensional drawings, thus they are viewing-direction dependent. There lacks one-to-one correspondence between the lines in two-dimensional drawing and the edges in three-dimensional objects, especially for those with curved surfaces. The D-Buffer algorithm works well with curved object with provided boundary information. Figures 4.5–4.7 show some results.

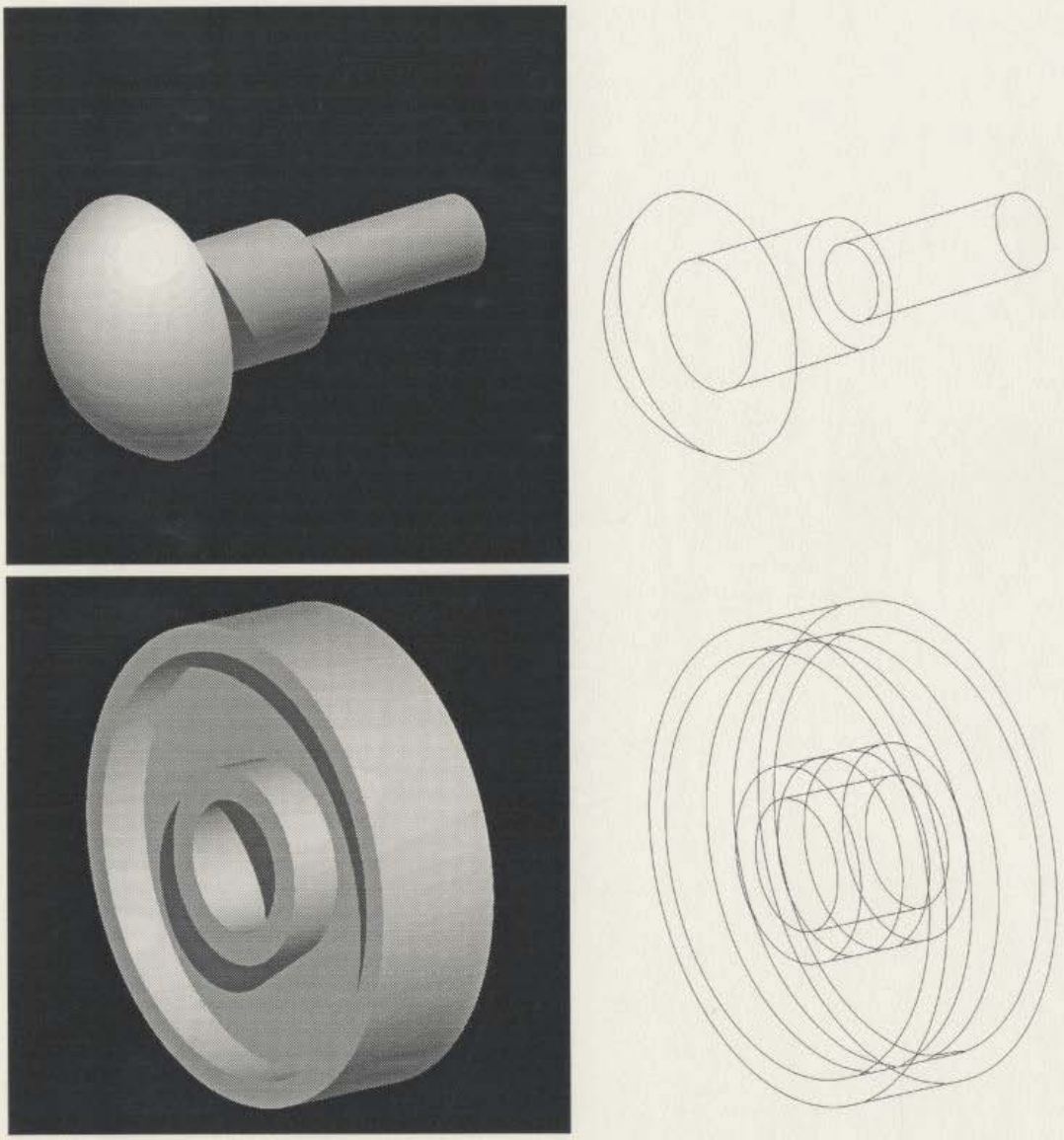


Figure 4.5: Curved: Shaded vs. Wireframe

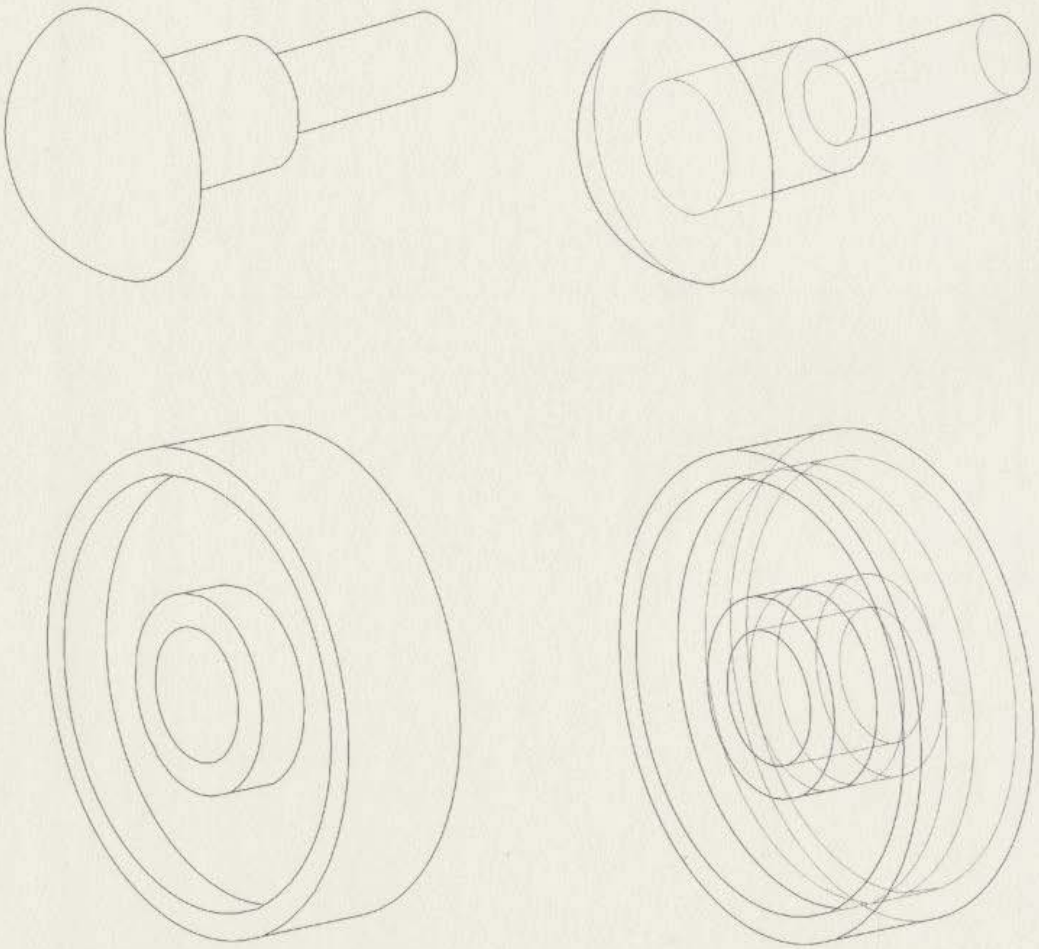


Figure 4.6: Curved: Z-Buffer vs. Dotted D-Buffer

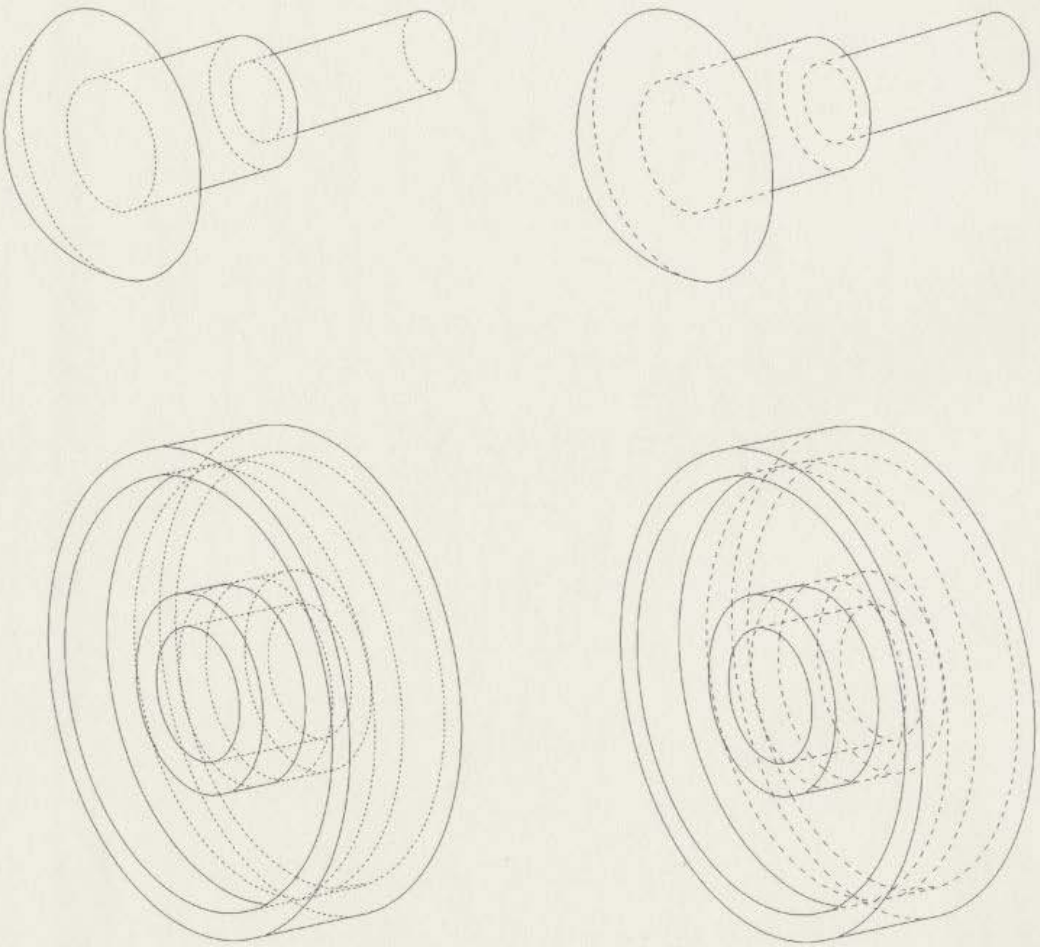


Figure 4.7: Curved: Dashed vs. Longer Dashed Hidden-Line

Chapter 5

Conclusion and Future Research

To display objects with complicated shapes fast and informatively, a hidden-line algorithm is needed to create the line-drawing images that show visible and invisible lines in different styles. After a brief overview of the major hidden-line elimination algorithms, although none of them realizes the purpose, the Z-Buffer algorithm shows its advantages of extreme simplicity, efficiency, and unlimited range of processable shapes. This thesis then presents a new image-space algorithm based on the Z-Buffer algorithm, namely the D-Buffer algorithm. By using one more boundary buffer, the D-Buffer algorithm can generate dotted or dashed (with adjustable length of dashes and spaces) hidden-line segments of any three-dimensional shapes at a fairly low com-

putational cost, hence revealing the information loss caused by hidden-line removal.

While the D-Buffer algorithm can quickly display sufficient information of objects with complicated shapes, there are still several improvements which can be made to enhance the functionality.

- Currently, all the boundary lines are 1-pixel wide, but thicker outline or visible lines may be desired. This may be done by carefully writing some more **BLACK**s to the frame buffer to make some lines thicker but not longer.
- The D-Buffer algorithm can only generate black and white (1 bit) images now. The possible values in the frame buffer may be extended to handle colors.
- The Z-Buffer algorithm is widely implemented in graphics hardware, but it may be more difficult to do the same thing for the D-Buffer algorithm (though it is based on the Z-Buffer algorithm) since it is a serial neighborhood operation instead of a parallel point operation like the Z-Buffer algorithm.

These suggest the direction of the future research.

Bibliography

[1] A. Appel. Some techniques for shading machine renderings of solids. In *Proceedings of the Spring Joint Computer Conference*, pages 37–45, 1968.

[2] A. Appel, F. J. Rohlf, and A. J. Stein. The haloed line effect for hidden line elimination. In *Proceedings SIGGRAPH '79 in Computer Graphics*, volume 13(2), pages 151–157, 1979.

[3] G. J. Awcock and R. Thomas. *Applied Image Processing*. Macmillan, London, 1995.

[4] H. G. Barrow and J. M. Tenebaum. Interpreting line drawing as three-dimensional surfaces. *Artificial Intelligence*, 17(1–3):75–116, 1981.

[5] W. J. Bouknight. A procedure for generation of three-dimensional half-toned computer graphics presentations. *Communications of the ACM*, 13(9):527–536,

1970.

[6] W. J. Bouknight and K. C. Kelly. An algorithm for producing half-tone computer graphics presentations with shadows and movable light sources. In *Proceedings of the Spring Joint Computer Conference*, pages 1–10, 1970.

[7] H. E. Burdick. *Digital Imaging: Theory and Applications*. Computing McGraw-Hill, New York, 1997.

[8] L. C. Carpenter. The a-buffer, an antialiased hidden surface method. In *Proceedings SIGGRAPH '84 in Computer Graphics*, volume 18(3), pages 103–108, 1984.

[9] E. Catmull. *A Subdivision Algorithm for Computer Display of Curved Splines*. PhD thesis, Computer Science Department, University of Utah, Salt Lake City, UT, 1974.

[10] W. H. Chieng. Polygon-to-object boundary clipping in object space for hidden surface removal in computer-aided-design. *Journal of Mechanical Design*, 117(3):374–389, 1995.

[11] E. L. Fiume. *The Mathematical Structure of Raster Graphics*. Academic Press, San Diego, CA, 1989.

[12] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics: Principles and Practice.* Second edition in C. Addison-Wesley, Reading, MA, 1996.

[13] H. Freeman. Computer processing of line-drawing images. *ACM Computing Surveys*, 6(1):57–97, 1974.

[14] A. S. Glassner, editor. *An Introduction to Ray Tracing.* Academic Press, London, 1989.

[15] W. Hsu and J. L. Hock. An algorithm for the general solution of hidden line removal for intersecting solids. *Computers & Graphics*, 15(1):67–86, 1991.

[16] A. Limaiem. Geometric algorithms for the intersection of curves and surfaces. *Computers & Graphics*, 19(3):391–403, 1995.

[17] Y. Liu and P. Zsombormurray. Intersection curves between quadric surfaces of revolution. *Transactions of the Canadian Society for Mechanical Engineering*, 19(4):435–453, 1995.

[18] D. Manocha. Algebraic pruning - a fast technique for curve and surface intersection. *Computer Aided Geometric Design*, 14(9):823–845, 1997.

[19] J. R. Miller and R. N. Goldman. Geometric algorithms for detecting and calculating all conic sections in the intersection of any 2 natural quadric surfaces. *Graphical Models and Image Processing*, 57(1):55–66, 1995.

[20] A. J. Myers. An efficient visible surface program. Report to the National Science Foundation, Computer Graphics Research Group, Ohio State University, Columbus, OH, July 1975.

[21] M. E. Newell, R. G. Newell, and T. L. Sancha. A solution to the hidden surface problem. In *Proceedings of the ACM National Conference*, pages 443–450, 1972.

[22] L. G. Roberts. Machine perception of three dimensional solid. In J. T. Tippet et al., editors, *Optical and Electro-Optical Information Processing*, pages 159–197. MIT Press, Cambridge, MA, 1964.

[23] D. F. Rogers. *Procedural Elements for Computer Graphics*. McGraw-Hill, New York, 1985.

[24] P. Rokita. Application of image processing techniques in computer graphics algorithms. *Computer Networks and ISDN Systems*, 29(14):1705–1714, 1997.

[25] S. D. Roth. Ray casting for modeling solids. *Computer Graphics and Image Processing*, 18(2):109–144, 1982.

[26] I. E. Sutherland, R. F. Sproul, and R. A. Schumacker. A characterization of ten hidden-surface algorithms. *ACM Computing Surveys*, 6(1):1–55, 1974.

[27] J. E. Warnock. A hidden line algorithm for halftone picture representation. Technical Report TR 4-5, NTIS AD 761 995, Computer Science Department, University of Utah, Salt Lake City, UT, May 1968.

[28] J. E. Warnock. A hidden-surface algorithm for computer generated half-tone pictures. Technical Report TR 4-15, NTIS AD 753 671, Computer Science Department, University of Utah, Salt Lake City, UT, June 1969.

[29] G. S. Watkins. *A Real Time Visible Surface Algorithm*. PhD thesis, Computer Science Department, University of Utah, Salt Lake City, UT, 1970.

[30] T. Whitted. An improved illumination model for shaded display. *Communications of the ACM*, 23(6):343–349, 1980.

[31] C. Wylie, G. W. Romney, D. C. Evans, and A. C. Erdahl. Halftone perspective drawings by computer. In *Proceedings of the Fall Joint Computer Conference*, pages 49–58, 1967.

[32] X. Yuan and H. Sun. P-buffer: A hidden-line algorithm in image-space. *Computers & Graphics*, 21(3):359–366, 1997.

[33] Q. Zhu. Virtual edges, viewing faces, and boundary traversal in line drawing representation of objects with curved surfaces. *Computers & Graphics*, 15(2):161-173, 1991.

Appendix A

Pseudo-code of the Dotted D-Buffer Algorithm

```
#define UNDECIDED   0x10
#define BLACK       0x01
#define WHITE       0x00

Dotted_D_Buffer( int image_height, int image_width,
                 int **frame_buffer, OBJECT *object_list )
{
```

```
int x, y, boundary;
double z, depth_buffer[image_height][image_width],
       boundary_buffer[image_height][image_width];
OBJECT object;

  for ( y = 0; y < image_height; y ++ )
    for ( x = 0; x < image_width; x ++ ) {          // initialize
      frame_buffer[y][x] = UNDECIDED;               // changeable
      depth_buffer[y][x] = 0.0;                     // background
      boundary_buffer[y][x] = 0.0;                  // background
    }

  while ( Object_Retrieve ( object_list, &object ) ) // each object
    while ( Pixel_Determine ( object, &y, &x, &z, &boundary ) ) {
                                                    // each pixel in projection
      if ( z >= depth_buffer[y][x] )                // closer point
        depth_buffer[y][x] = z;                     // update depth buffer
      if ( boundary && z >= boundary_buffer[y][x] )
                                                    // closer boundary point
```

```
      boundary_buffer[y][x] = z;       // update boundary buffer
  }

for ( y = 0; y < image_height; y ++ )
  for ( x = 0; x < image_width; x ++ ) {
    if ( boundary_buffer[y][x] > 0.0 )              // boundary pixel
      if ( boundary_buffer[y][x] >= depth_buffer[y][x] )
                                                    // in front
        frame_buffer[y][x] = BLACK;                 // solid
      else                                          // behind
        if ( frame_buffer[y][x] == UNDECIDED ) {    // changeable
          frame_buffer[y][x] = BLACK;               // dot
          if ( x < image_width - 1 )
            frame_buffer[y][x+1] = WHITE;           // space
          if ( y < image_height - 1 ) {             // ...
            frame_buffer[y+1][x] = WHITE;           // around
            if ( x > 0 )                            // ...
              frame_buffer[y+1][x-1] = WHITE;       // the
            if ( x < image_width - 1 )              // ...
```

```
                frame_buffer[y+1][x+1] = WHITE;      // dot
            }
        }
      frame_buffer[y][x] &= BLACK;                   // normalize
    }
}
```

Appendix B

Pseudo-code of the Dashed D-Buffer Algorithm

```
#define UNDECIDED    0x10
#define BLACK        0x01
#define WHITE        0x00

int DASH_THRESHOLD;                                    // dash length

Dashed_D_Buffer( int image_height, int image_width,
```

```
                int **frame_buffer, OBJECT *object_list )
{
int x, y, boundary;
double z, depth_buffer[image_height][image_width],
       boundary_buffer[image_height][image_width];
OBJECT object;

  for ( y = 0; y < image_height; y ++ )
    for ( x = 0; x < image_width; x ++ ) {          // initialize
      frame_buffer[y][x] = UNDECIDED;               // changeable
      depth_buffer[y][x] = 0.0;                     // background
      boundary_buffer[y][x] = 0.0;                  // background
    }

  while ( Object_Retrieve ( object_list, &object ) ) // each object
    while ( Pixel_Determine ( object, &y, &x, &z, &boundary ) ) {
                                        // each pixel in projection
      if ( z >= depth_buffer[y][x] )    // closer point
        depth_buffer[y][x] = z;         // update depth buffer
```

```
      if ( boundary && z >= boundary_buffer[y][x] )
                                          // closer boundary point
        boundary_buffer[y][x] = z;        // update boundary buffer
    }

  for ( y = 0; y < image_height; y ++ )
    for ( x = 0; x < image_width; x ++ ) {
      if ( boundary_buffer[y][x] > 0.0 &&          // boundary pixel
           frame_buffer[y][x] == UNDECIDED )       // unprocessed
        dash ( y, x, 0, BLACK, frame_buffer,      // trace the line
               boundary_buffer, depth_buffer );
      frame_buffer[y][x] &= BLACK;                // normalize
    }
}


dash ( int y, int x, int distance, int color,
       int **d_buffer, double **b_buffer, double **z_buffer )
{
```

```
if ( b_buffer[y][x] >= z_buffer[y][x] )          // in front
  d_buffer[y][x] = BLACK;                        // solid
else                                             // behind
  d_buffer[y][x] = color;                        // dash line color

if ( ++distance >= DASH_THRESHOLD ) {            // long enough
  distance = 0;                                  // reset counter
  color = ( color == BLACK ) ? WHITE : BLACK;    // dash <=> space
}

// recursive calls to 8-neighbors of the current pixel.
if ( b_buffer[y][x+1] > 0.0 &&                   // boundary pixel
     d_buffer[y][x+1] == UNDECIDED )             // unprocessed
  dash ( y, x + 1, distance, color,              // trace the line
         d_buffer, b_buffer, z_buffer );

if ( b_buffer[y][x-1] > 0.0 &&                   // boundary pixel
     d_buffer[y][x-1] == UNDECIDED )             // unprocessed
  dash ( y, x - 1, distance, color,              // trace the line
```

```
        d_buffer, b_buffer, z_buffer );

if ( b_buffer[y+1][x] > 0.0 &&                    // boundary pixel
     d_buffer[y+1][x] == UNDECIDED )              // unprocessed
  dash ( y + 1, x, distance, color,               // trace the line
         d_buffer, b_buffer, z_buffer );

if ( b_buffer[y-1][x] > 0.0 &&                    // boundary pixel
     d_buffer[y-1][x] == UNDECIDED )              // unprocessed
  dash ( y - 1, x, distance, color,               // trace the line
         d_buffer, b_buffer, z_buffer );

if ( b_buffer[y+1][x+1] > 0.0 &&                  // boundary pixel
     d_buffer[y+1][x+1] == UNDECIDED )            // unprocessed
  dash ( y + 1, x + 1, distance, color,           // trace the line
         d_buffer, b_buffer, z_buffer );

if ( b_buffer[y+1][x-1] > 0.0 &&                  // boundary pixel
     d_buffer[y+1][x-1] == UNDECIDED )            // unprocessed
```

```
    dash ( y + 1, x - 1, distance, color,          // trace the line
           d_buffer, b_buffer, z_buffer );

  if ( b_buffer[y-1][x+1] > 0.0 &&                 // boundary pixel
       d_buffer[y-1][x+1] == UNDECIDED )           // unprocessed
    dash ( y - 1, x + 1, distance, color,          // trace the line
           d_buffer, b_buffer, z_buffer );

  if ( b_buffer[y-1][x-1] > 0.0 &&                 // boundary pixel
       d_buffer[y-1][x-1] == UNDECIDED )           // unprocessed
    dash ( y - 1, x - 1, distance, color,          // trace the line
           d_buffer, b_buffer, z_buffer );
}
```