2000

P-Buffer: Hidden-line rendering with a dynamic p-buffer

Xiaobu Yuan
University of Windsor

Sun Hanqiu
The Chinese University of Hong Kong

P-Buffer: A Hidden-Line Algorithm In Image-Space

Xiaobu Yuan*[1] and Hanqiu Sun[2]

[1]Department of Computer Science, Memorial University of Newfoundland, St. John's, Newfoundland, Canada A1B 3X5
[2]Department of Computer Science & Engineering, The Chinese University of Hong Kong, Shatin, N. T., Hong Kong

Abstract

Despite the emergence of highly realistic computer-generated images, line-drawing images are still a common practice in showing the shapes and movements of three-dimensional objects. It is especially true when rendering time is critical in interactive applications such as the modeling and testing stage of computer aided design/manufacturing, computer animation, and virtual reality. Hence much effort has been devoted to provide sufficient information of the displayed objects with the least amount of time. While the techniques that determine visible surfaces in an image-space have the advantages on rendering speed and processable shapes, those that decide visible lines or line segments in an object-space are more suitable for showing hidden lines.

The P-buffer algorithm introduced in this paper is a method for rendering line-drawing images with dashed hidden-lines. Being an image-space method, this algorithm preserves the low computational cost and works on a wide range of object shapes; as an extension to the Z-buffer algorithm it, moreover, discloses hidden surfaces by showing them with dashed lines. After a discussion on rendering techniques of line-drawing images, this paper presents this algorithm with pseudocode in C++ programming language and shows some experiment results as well. This image-space algorithm can be used as a compromise approach that reveals the concealed information of hidden-surface-removed views for time-critical rendering.

*Author for correspondence

1 Introduction

Line drawing is a classical technique of describing three-dimensional objects on two-dimensional surfaces. It has been used, even long before the invention of computers, in engineering and architecture to show the shape and structure of machinery parts and buildings [1]. Even with the emergence of highly realistic computer-generated images in the past decade [2], it remains an active topic and draws continuous attention from different research areas [3, 4, 5].

Given a set of objects and the specification of a view, the objects' surface boundaries are projected onto an image plane along a viewing direction [6]. A wire-frame picture is thus obtained if the created picture contains all the boundary lines. Otherwise, a line-drawing picture presumes opaque object surfaces and shows only those boundary lines or segments that are visible in the view. More preferably, the complementary set of hidden lines and line segments is also displayed but in a different style, such as dotted or dashed lines [7].

The main advantage of a line-drawing picture is that it provides sufficient information of three-dimensional objects at a significantly low computational cost. For instance, two images of a machinery part are shown in Fig. 1, one rendered by ray-tracing and the other by line-drawing. In a test conducted on a Pentium 100 PC, Fig. 1(a) took *470* seconds while Fig. 1(b) took less than one second. In this particular comparison, to provide the same shape information of the object, the ray-tracing image took 700 times longer than the line-drawing image.

Between ray-tracing and line-drawing, there are also other techniques at the intermediate levels. Phong-based rendering that uses a local illumination model, for example, may produce much better images than Fig. 1(b) but with substantially less computation than that spent on Fig. 1(a). However, in the attempt of disclosing the internal structure of this object, it becomes fairly easy in line-drawings when the set of hidden lines is available; but with the other techniques, making the object transparent is the only way of achieving the expected result. It had taken *6.5* times more time to render such a ray-tracing image in the test.

The high efficiency of conveying the information of three-dimensional objects makes line-drawing the best choice for real-time interaction [8, 9] and low-cost applications [10]. When constraints on time are significant, the success of an exercise depends heavily on how fast it is to display objects. Hence, line-drawings are widely used in the modeling stage and interfacing components of computer animation, virtual reality, and computer-aided design [11, 12]. This practice will carry on unless vital progresses are materialized in the cost and power of computer hardware as well as the speed of rendering high-quality images.

Generally speaking, the algorithms used to render line-drawing images fall into two major categories. The first category includes image-space methods that determine which one of the objects to display at a pixel according the objects' distances along an imaginary viewing ray through that pixel. The second category, on the other hand, gathers in object-space methods. In this group the rendering algorithms compare objects directly with each other to determine the list of visible lines and line segments in a picture.

Since object-space methods keep track of boundary intersections in the process of determining visible lines, it is easy for an algorithm in this group to obtain a complementary list of hidden lines, which is essential when the concealed information of a hidden-surface-removed picture has to be provided in an application. However, finding out intersections and determining the visibility of surfaces and lines are computationally expensive especially when objects exhibit complex shapes. On the contrast, image-space methods do not need any time-consuming intersection checks. They are fast because they decide which object to display at a pixel by projecting objects onto the image plane.

Image-space algorithms unfortunately cannot display hidden lines for the lack of facility to retain a list of the surfaces that impact upon each pixel. To overcome this problem, this paper presents a modified image-space method, namely the P-buffer algorithm. Being an image-space method, it can be used to render line-drawing images with a wide variety of three-dimensional objects at a low computational cost. By introducing a pattern buffer, this method is also capable of showing hidden lines with dashed lines. It is therefore useful to reveal the concealed information for time-critical rendering. In the following discussion, this paper presents the P-buffer algorithm with C++ pseudocode, results of experiment, and an example of possible application in off-line robot programming.

2 Rendering of Line-Drawings

Objects are represented constructively by geometric primitives in a computer. They are object models in numerical formats. Given a viewing coordinate system, object models are first transformed to the new coordinate system; and then the visibility of the surface boundaries is decided for this particular view. Consequently, a line-

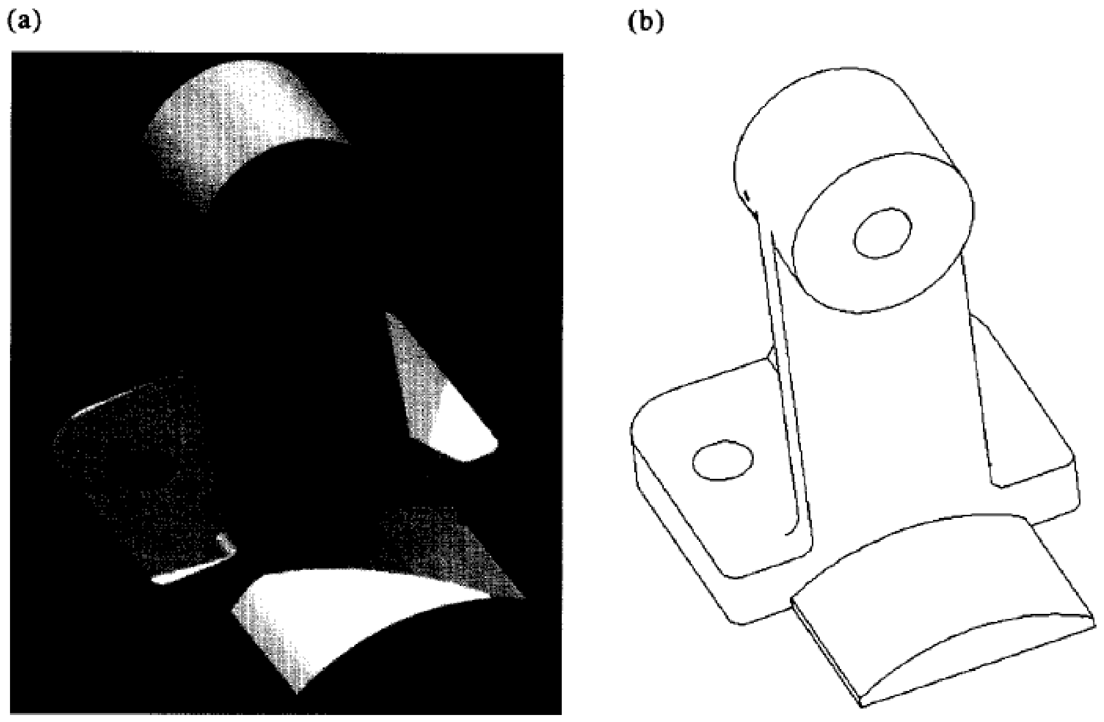

Figure 1: The Ray-traced Image and Line Drawing of an Object

drawing picture is generated which displays with solid lines the visible boundaries and boundary segments on the image plane. When needed, the picture also shows invisible boundaries but with a different style, such as dashed lines.

Which part of an object's surface boundary is visible in a viewing direction relies on its orientation, i.e., the surface normal direction, in the viewing coordinate system as well as its geometrical relationship with the other parts of the object. A boundary may be entirely visible, partially visible, or entirely invisible. To determine the visibility of surfaces for line-drawing pictures, two typical approaches have been developed. They are the object-space methods and image-space methods.

2.1 Object-space Methods

Suppose the set of objects in a scene has a total of m object surfaces in the three-dimensional space; and each surface is defined implicitly with an equation that takes the following form after being transformed to the viewing coordinate system:

$$f_k(x, y, z) = 0 \qquad\qquad 0 \leq k < m \qquad (1)$$

Assuming that there is no change of visibility for each individual surface in a given viewing direction, the m surfaces can be divided into two sets — one for the completely invisible surfaces and the other for the rest, which contains supposedly n surfaces and $n < m$. This separation can be easily accomplished by checking surface normals.

Afterwards, it has been a prolonged effort to find out if the n non-invisible surfaces are visible in the viewing direction [3, 13, 14, 15]. Given any pair of the n surfaces, the visibility of their boundaries depends on the geometric shapes and topological relations of the two entities. If their numerical representation is $f_i(x, y, z) = 0$ and $f_j(x, y, z) = 0$ respectively, $0 \leq i, j < n$ and $i \neq j$, the intersection of their projected two-dimensional equations on the image plane determines if one surface overlaps the other in the viewing direction,

$$\begin{cases} f_i'(x,y) &= 0 \\ f_j'(x,y) &= 0 \end{cases} \tag{2}$$

If there exists any occlusion, the depth value in two original equations decides which surface is in the front and therefore whose boundary could be visible.

The earliest object-space algorithm worked on convex polyhedra [16]. With only straight lines and planar surfaces involved in the rendering process, it is fairly easy to obtain the list of visible and invisible endpoints from parametric line equations, and to find out the depth relation of overlapped polygons according to their z values. The basic idea also applies to objects of polygons in concave relationships [17, 18]. Recent works on how to remove hidden polygonal surfaces in the object space can be found in [12, 19, 20, 21, 13].

If f_i and/or f_j contains curved surfaces, an object-space line drawing algorithm may still work provided that the curved surfaces are approximated by many small facets. Alternatively if there is a close-form solution of Eq. 2 that describes the intersection of the two curved surfaces, the strategy of finding intersection and then overlapping relationships can be principally applied to create a line-drawing image. However, the complexity and difficulty of intersecting two general surfaces unfavorably limit object-space methods to the applications with at most some of the quadric surfaces [22, 23, 24, 25, 26].

2.2 Image-space Methods

Instead of using any explicit algorithm to decide surface intersections, image-space methods only require the z values from each of the surfaces $f_k(x, y, z) = 0$, $k = 0, \cdots, n-1$, for every $< x, y >$ coordinate of an image. Because of the simplicity, an image-space line drawing algorithm has a low computational complexity [27, 28] and also is capable of rendering line-drawing pictures that contain objects with a wide range of shapes.

The typical and famous image-space method is the Z-buffer algorithm [29]. It employs two buffers: one frame buffer to keep the image and one depth buffer to determine which object's boundary should be kept in the image. Initialized to a distant value, the contents of the depth buffer is updated to the closest depth values when the set of n objects is processed one by one. Meanwhile the algorithm updates the frame buffer with the pixels from the objects that contribute the closest values in the depth buffer, and it creates a line-drawing image when all objects are processed.

Directly related to the Z-buffer algorithm is the painter's algorithm, which was simplified from the depth-sort algorithm [30]. This algorithm substitutes the depth buffer with a sequence of pre-sorted objects and renders one at a time from the sequence into the frame buffer. While the painter's algorithm is better for a limited number of object, the Z-buffer algorithm is better for a larger number of objects. In addition, the Z-buffer algorithm also is more predictable regarding to the computational time required to render an image.

The image-space algorithms can also be combined with object-space methods into a scan-line algorithm [31]. In this algorithm, a scan-line covers all the pixels in the frame buffer. Again, a sequence of pre-sorted objects tells which object is in the front when the scan-line encounters several objects. Both of the algorithms have to spend more time on sorting and work mainly with objects of polygonal surfaces.

In comparison to the other rendering algorithms for line-drawing images, the Z-buffer algorithm has the advantages of easy implementation, less computational cost, and hardware supports. It has no limits on the shape of objects either. Z-buffer algorithm yet has some noticeable deficiencies. One of its major problems on anti-aliasing has been solved with the A-buffer algorithm [32], but its inability to show the hidden lines degrades its efficiency of providing sufficient information of the displayed objects as required in applications. The following discussion concentrates on how to solve this problem with an additional pattern buffer.

3 The P-Buffer Algorithm

Image-space algorithms have the advantage in rendering speed because they, instead of conducting surface-sorting and intersection-checking, use the depth of surfaces to decide which boundary lines should be visible in a viewing direction. In the Z-buffer algorithm, for example, a depth-buffer tells if the pixels belonging to a surface is in front and therefore should cover those behind it. The paint-over mechanism, though fast in rendering, unfortunately also eliminates the chances of displaying hidden lines. To reveal the concealed information, a pattern buffer, or P-buffer for short, is introduced in this paper. With a little more operations or some extra working space, the introduced algorithm is capable of displaying hidden lines with dash-lines. Experiment results are given with a testing pattern to show how the algorithm operates.

3.1 The Algorithm

As described in Section 2.2, the Z-buffer algorithm uses two buffers, a frame buffer and a depth buffer. They are defined by the two two-dimensional arrays `frame_buffer[][]` and `depth_buffer[][]` in the C++ pseudocode listed in the appendix of this paper. Both of them are the same size as the image. Any declaration of an object as an instance of class `Z_Buffer` will automatically initialize the two buffers by the default constructor `Z_Buffer()` to the background color and the maximal depth respectively.

In addition, at the request of a call to `image_Render(`**geom**$*$`)`, the only accessible member function of class `Z_Buffer`, the first while-loop in this function processes one by one all the geometric items in the object list with a private function `object_Retrieve(`**geom**$*$, **geom**$*$`)`. Each individual pixel `(x, y)` of every retrieved geometric item is then calculated by `pixel_Determine(`**geom, point2D**$*$, **float**$*$, **boolean**$*$`)`. The third (float) parameter of the pixel-determining function keeps the depth at the pixel; and the last (boolean) parameter is a tag indicating if this pixel is on a boundary line, which returns a `TRUE` value only if this pixel fits into the projected boundary of an object or the object's surface normals around the pixel point to both positive and negative directions in the viewing direction.

The virtual function `this_Algorithm(`**geom, int, int, float, boolean**`)` of class `Z_Buffer` is an implementation of the z_buffer algorithm. It has two conditional statements. The first one checks if the new depth is closer than what the depth buffer has at the given pixel; and the second one, after updating `depth_buffer[x][y]` with the new value, sets `frame_buffer[x][y]` to a proper value because the new pixel belongs to a geometric item in front. Depending on if this is a boundary pixel, the new value of `frame_buffer[x][y]` can be either `LINE_COLOR` or `BACKGROUND_COLOR`. The result is a line-drawing image without hidden-lines, for instance, as shown in Fig. 1`(b)`.

The Z-buffer algorithm does not keep track of the geometric items that make contributions to the frame buffer. What it can do with the boundary lines belonging to the hidden surfaces is to erase them with `BACKGROUND_COLOR` or alternatively not to touch them after they are generated, which actually results in a wireframe picture (Fig. 2`(a)`) though more straight forward methods are available to render such images.

Let a pattern buffer define a grid of filtering patterns, which is also the same size of the image as the depth and frame buffers in the Z-buffer algorithm. The value of all elements in the pattern buffer can only be either '1' or '0'. At a position `(x, y)`, $0 \leq x, y <$ `image_width`, the action of keeping or updating the content of `frame_buffer[x][y]` depends on not only if the new depth is closer than `depth_buffer[x][y]`, as does the Z-buffer algo-

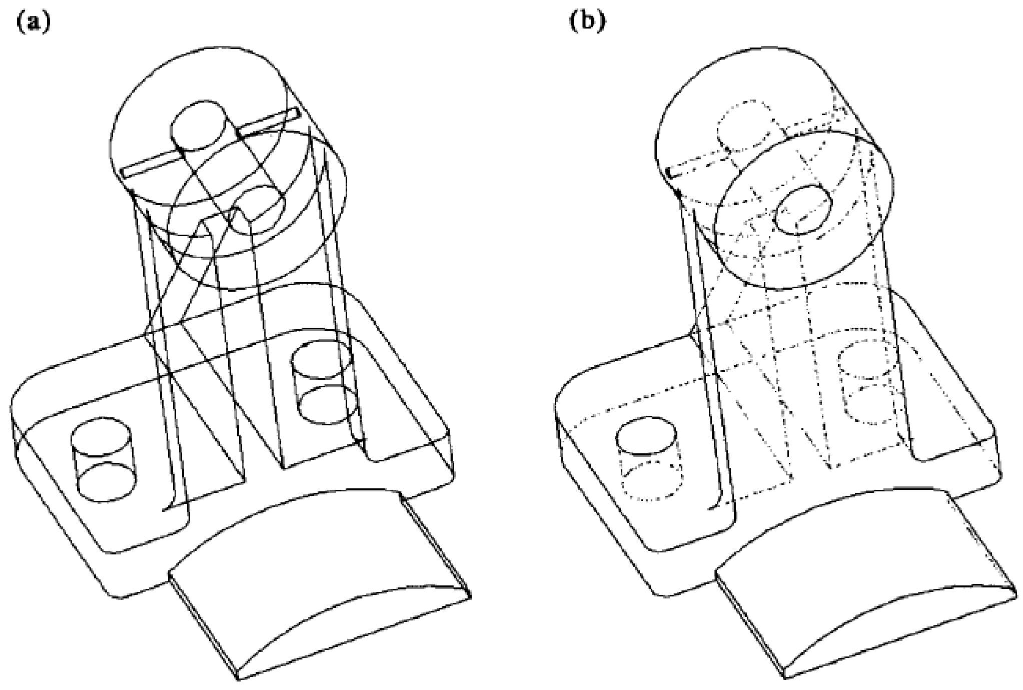

Figure 2: A Wireframe Image and Line-drawing with Dashed Hidden-lines

rithm, but if `pattern_buffer[x][y]` also allows the change of `frame_buffer[x][y]`. In such a way, a modified rendering algorithm, i.e., the P-buffer algorithm, is developed that is capable of retaining those previously displayed but later overwritten hidden-lines while also changing their style to distinguish them from the visible lines.

In this algorithm, a boundary pixel of a front geometric item always overwrites the content of `frame_buffer[x][y]`; but an interior pixel of a front geometric item updates `frame_buffer[x][y]` only if the filtering pattern blocks `frame_buffer[x][y]`, which happens when `pattern_buffer[x][y] = 0`. Otherwise, when `pattern_buffer[x][y] = 1` the content of `frame_buffer[x][y]` should be left unchanged. If, on the other hand, the pixel being processed is a boundary pixel and it belongs to a geometric item whose depth value is farther than `depth_buffer[x][y]`, `frame_buffer[x][y]` still needs to be updated to show the hidden information whenever the pattern buffer allows it to pass the filter. Hence the introduction of the filtering pattern shows up the hidden-lines with the '1's while also breaking them into dashed lines with the '0's.

The derived `P_Buffer` class in the C++ pseudo-code further explains how the P-buffer algorithm works. The default constructor `P_Buffer()`, though blank itself, activates the default constructor `Z_Buffer()` of its base class to initialize the frame and depth buffers for any instance of this class. When the new `this_Algorithm(`**geom, int, int, float, boolean**`)` is invoked in an access to `image_Render(object_list)`, it immediately checks the depth information of the current geometric item with the value in `depth_buffer`. If any boundary portion of the current geometric item is in the behind, segments of this portion are still rendered into `frame_buffer` on the condition that the filtering pattern does not block them. Otherwise, the function `this_Algorithm(`**geom, int, int, float, boolean**`)` of class `P_Buffer` creates the boundary of the front-sitting object and breaks with the filtering-pattern anything overlapped by the projected region of this geometric item.

The `Z_Buffer` class and `P_Buffer` class can be used separately to achieve the hidden-line removal or the dashed hidden-line results although neither of them can achieve both the results at the same time. Or alternatively, by taking advantage of the polymorphism provided by the object-oriented programming language, the two classes can be used together to render a line-drawing image with or without dashed hidden-lines. This can be accomplished by first declaring a pointer `*image_space` of the base class `Z_Buffer` and then pointing `image_space` to a new instance of either `Z_Buffer` itself or its derived class `P_Buffer`. At running time, a function call such as `image_space.image_Render(object_list)` generates the desired line-drawing image.

3.2 The Filtering Pattern

The selection of a filtering pattern plays an important role in the final result of the line-drawing images rendered with the P-buffer algorithm. An idea filtering pattern is a two-dimensional binary array whose elements can be either '1' or '0'. Given any hidden line, the pattern should be so arranged that the '1's shows up this line while the '0's breaks it into dashed segments. Since complicated objects usually have surfaces whose boundary exhibits a wide range of shapes, it is not guaranteed that any grid of filtering patterns can be used in this algorithm to show the hidden-lines. For instance, a chess board consists of black and white squares. If the black squares block and the white squares pass hidden-lines, a filtering pattern of the chess board has the best performance on vertical and horizontal lines. However, when a straight line runs cross the diagonals of the squares, this line will be either completely eliminated or totally untouched depending on what kind of squares it runs through.

The P-buffer algorithm has been tested with 38 standard Macintosh patterns. Most of them are not useful because they are either uniformly colored (white or black) or do not work with boundary lines in horizontal, vertical, or diagonal directions. The 8×8 bitmap shown in Fig. 3`(a)` is one of the remaining patterns that yield acceptable results. The line-drawing images in Fig. 2`(b)` and Fig. 4`(b)`, for example, were rendered by using a grid of this pattern (Fig. 3`(b)`) in the P-buffer algorithm. To make the filtering pattern virtually as big as the frame and depth buffer, the basic pattern repeats itself in both the horizontal and vertical directions until reaching the desired size. Therefore, given a pixel at the x and y coordinates, the following equation determines what value is returned by the member function `value_Pattern(int,int)`,

$$value = \texttt{basic\_pattern}[x\%l][y\%l] \qquad (3)$$

where `basic_pattern[][]` is a two dimensional array that specifies the basic pattern, l is the dimension of the basic pattern, e.g., 8 for Fig. 3`(a)`, and % is the modulo operator.

In implementation, the basic pattern may be different in size or shape from Fig. 3`(a)`. Any pattern could be useful for the P-buffer algorithm as long as it satisfies the requirements, i.e., to break lines while showing them. If speed becomes so critical in an application that it counts in each function call and each operation, a full size pattern buffer can be initialized in the default constructor of class `P_Buffer` to save the function call to `value_Pattern(int, int)` and the two modulo operations in it. Since the redefined `this_Algorithm(···)` adds at most one more conditional statement and a boolean operation to the one defined in the base class `Z_Buffer`, the P-buffer algorithm has the same computational complexity as the classical Z-buffer algorithm.

3.3 An Example

The P-buffer algorithm renders in image space line-drawing images with dashed hidden-lines. Featuring the major advantages of line-drawing algorithms in high-speed rendering, wide range of

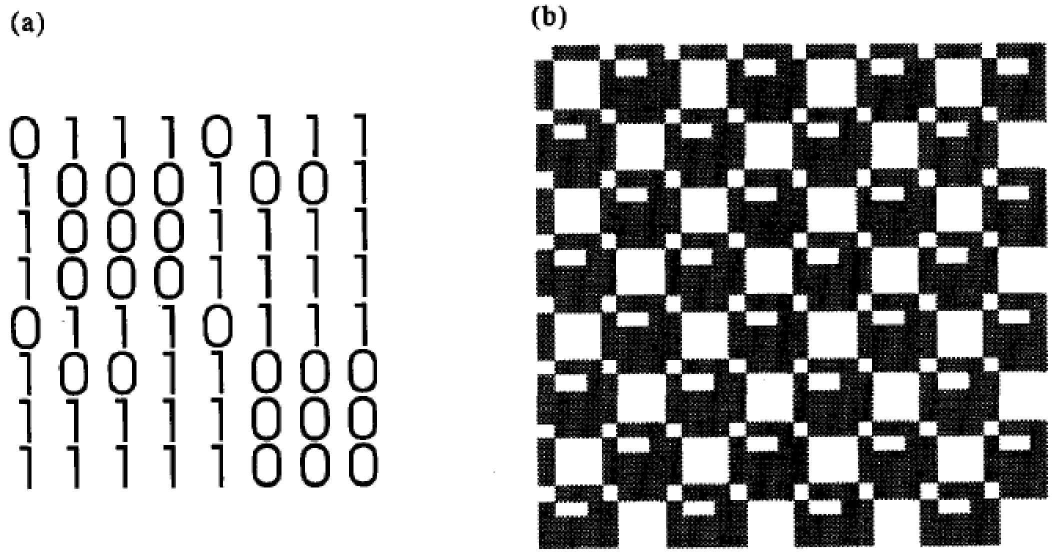

Figure 3: The Filtering Pattern

processable shapes, and hidden-line generation, this algorithm is particularly useful in interactive applications where rendering time is critical. For instance, off-line programming is a technology that offers cost-effectively programming robots to perform diverse assembly tasks in industry automation. To increase programming flexibility and shorten manufacturing cycles, product engineers manipulate on, instead of physical machinery parts, CAD data and defines robot assembly tasks in a virtual environment [33].

Given a task of assembling a sliding-door guide, a sequence of operations has to be specified to place a wheel on top of a bracket and then secure it with a bushing and two pins. One of the assembly operations that a human operator needs to do is to define reference coordinate systems on the machinery parts for alignment. Let one reference system be on the front cylindrical surface of the bracket (Fig. 4) and the other on the inner cylindrical surface of the wheel. The operator then has to interact via an input device, e.g., a three-dimensional data glove, to select points on the two surfaces to define the reference systems.

When the two machinery parts are arranged as in Fig. 4`(a)`, both the surfaces to work on are visible to the operator. In such a case, he/she has no problem defining the reference systems with a view that shows no hidden-lines. If, however, the wheel happens to be in front of the bracket, the operator cannot see the bracket's cylindrical surface in a hidden-surface-removed view. As a result, he/she has to use additional operations to move aside the wheel first before the reference system on the bracket can be specified.

Furthermore, it is difficult to decide when the wheel touches the cylindrical surface of the bracket as one must be in front of the other one, thus hiding the needed information when hidden-lines are removed. Since the P-buffer algorithm reveals the hidden information by displaying hidden-surfaces with dashed lines, a view rendered with this algorithm (Fig. 4`(b)`) is always ready for the operator to pick hidden lines for reference association during interactive sessions to manipulate models. He/she can, therefore, operate in a predictable manner and use much less interactions to accomplish the same job.

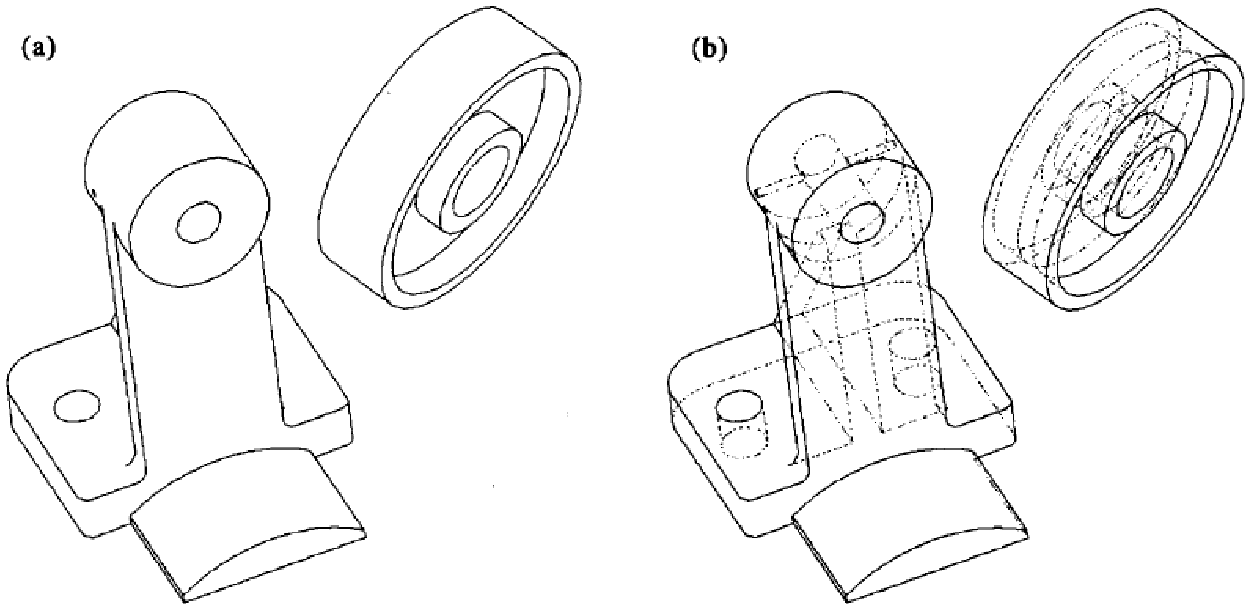

Figure 4: An Example of P-Buffer Application

4 Conclusion

Image-space line-drawing algorithms are suitable for time-critical rendering. Though simple, the pictures rendered with these algorithms provide sufficient information of a wide range of three-dimensional shapes at a fairly low computational cost. To compromise the inability of showing hidden lines with image-space line drawing algorithms, this paper introduces a modified Z-buffer algorithm — P-buffer algorithm. The P-buffer algorithm has the major advantages of image-space algorithms but it is also capable of revealing the concealed information. Together with the C++ pseudocode of the P-buffer algorithm, the paper also shows some experiment results conducted with a testing pattern. The presented algorithm exposes a new approach to uncover the hidden information for time-critical rendering. Since the performance of the P-buffer algorithm relies on filtering patterns to generate dashed hidden-lines and standard bitmap patterns usually do not fit into the need, special patterns is being designed for use as filtering patterns. Meanwhile, active research has started to investigate the use of this algorithm in applications that involves object manipulation via human-computer interactions, e.g., virtual manufacturing.

Acknowledgment

This work was supported by the Natural Sciences and Engineering Research Council of Canada.

References

[1] T. E. French. *Engineering Drawing and Graphic Technology.* McGraw-Hill, New York, 14th edition (1993).

[2] J. Arvo and D. Kirk. A survey of ray tracing acceleration techniques. In A. S. Glassner, editor, *An Introduction to Ray Tracing*, pages 201–262. Academic Press, New York (1989).

[3] M. Deberg. Generalized hidden surface removal. *Computational Geometry-Theory and Applications*, 5(5):249–276 (1996).

[4] M.E. Brown. Visualization of 3-dimensional structure during computer-aided-design. *International Journal of Human-Computer Interaction*, 7(1):37–56 (1995).

[5] R. Cowie. From line-drawings to impressions of 3d objects - developing a model to account for the shapes that people see. *Image and Vision Computing*, 11(6):342–352 (1993).

[6] 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).

[7] D. Hearn and M. P. Baker. *Computer Graphics.* Prentice Hall, second edition (1994).

[8] Xin Li and J. Michael Moshell. Modeling soil: Realtime dynamic models for soil slippage and manipulation. In James T. Kajiya, editor, *Computer Graphics (SIGGRAPH '93 Proceedings)*, volume 27, pages 361–368 (August 1993).

[9] H. Plantinga, C.R. Dyer, and W.B. Seales. Real-time hidden-line elimination for a rotating polyhedral scene using the aspect representation. In *Proceedings of Graphics Interface '90*, pages 9–16 (May 1990).

[10] Luis Serra and Rodney Lionel Rhodes. Low-cost hardware platform for developing real-time 3D graphics. *The Visual Computer*, 6(5):254–265 (November 1990).

[11] L. Kjelldahl. Study on how depth-perception is affected by different presentation methods of 3D objects on a 2D display. *Computers & Graphics*, 19(2):199–202 (1995).

[12] 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).

[13] 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).

[14] J.G. Griffiths. A bibliography of hidden-line and hidden-surface algorithms. *Computer-Aided Design*, 10(3):203–206 (May 1978).

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

[16] L. G. Roberts. Machine perception of three dimensional solid. In *Optical and Electro-Optical Information Processing*, pages 159–197. MIT Press, Cambridge (1964).

[17] P. P. Loutrel. A solution to the hidden-line problem for computer-drawn polyhedra. *IEEE Trans. on Computers*, 19(3):205–213 (1970).

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

[19] J.H. Reif. An efficient output-sensitive hidden-surface removal algorithm for polyhedral terrains. *Mathematical and Computer Modelling*, 21(5):89–104 (1995).

[20] B.R. Vatti. A generic solution to polygon clipping. *Communications of the ACM*, 35(7):56–63 (1992).

[21] M.T. Goodrich. A polygonal approach to hidden-line and hidden-surface elimination. *CVGIP: Graphical Models and Image Processing*, 54(1):1–12 (January 1992).

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

[23] C.K. Shene and J. Johnstone. On the lower degree intersections of two natural quadrics. *ACM Transactions on Graphics*, 13(4):400–424 (1994).

[24] I. Wilf. Quadric-surface intersection curves - shape and structure. *Computer-Aided Design*, 25(10):633–643 (1993).

[25] J. Miller. Geometric approaches to nonlinear quadric surface intersection curves. *ACM Transactions on Graphics*, 6(4):274–307 (1987).

[26] R. F. Sarraga. Algebraic methods for intersections of quadric surfaces in GMSOLID. *CVGIP*, 22(2):222–238 (1983).

[27] E. Fiume. *The Mathematical Structure of Raster Graphics.* Academic Press, San Diego (1989).

[28] A. Fournier and D. Fussell. On the power of the frame buffer. *ACM Transactions on Graphics*, 7(2):103–128 (April 1988).

[29] E. Catmull. *A Subdivision Algorithm for Computer Display of Curved Splines.* PhD thesis, Computer Science Department, University of Utah (1974).

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

[31] W. Bouknight. A procedure for generation of three-dimensional half-toned computer graphics presentations. *Communications of the ACM*, 13(9):527–536 (September 1970).

[32] Loren Carpenter. The A-buffer, an antialiased hidden surface method. In Hank Christiansen, editor, *Computer Graphics (SIGGRAPH '84 Proceedings)*, volume 18, pages 103–108 (July 1984).

[33] M. Smith. An environment for more easily programming a robot. In *Proc. the 1992 IEEE International Conference on Robotics and Automation*, volume 1, pages 10–16 (1992).

APPENDIX

```
class Z_Buffer {
public:
  Z_Buffer() {
    int i, j;
    for ( i = 0; i < image_width; i++ )                // initialize
      for ( j = 0; j < image_height; j++ ) {
        frame_buffer[i][j] = BACKGROUND_COLOR;
        depth_buffer[i][j] = 0.0;
      }
  }
  void image_Render(geom* object_list) {               // render the image
    while ( object_Retrieve(object_list, &object) )
      while ( pixel_Determine(object, &pixel, &depth, &boundary) )
        this_Algorithm(object, pixel.x, pixel.y, depth, boundary);
  }
private:
  virtual this_Algorithm(geom object, int x, int y,
                         float depth, boolean boundary_key) {
    if ( depth > depth_buffer[x][y] ) {                // update both buffers
      depth_buffer[x][y] = depth;
      if ( boundary_key == TRUE )
        frame_buffer[x][y] = LINE_COLOR;               // draw the line
      else
        frame_buffer[x][y] = BACKGROUND_COLOR;         // remove surfaces
    }
  }
  geom object; point2D pixel; float depth; boolean boundary;
  // TRUE if an object is retrieved or NULL when the list is finished
  boolean   object_Retrieve(geom* object_list, geom* object);
  // FALSE if all pixels are processed; otherwise, return a pixel,
  //  the depth, and a key indicating if the pixel is on a line.
  boolean   pixel_Determine(geom object, point2D* pixel,
                            float* depth, boolean* boundary);
};

class P_Buffer : public Z_Buffer {
public:
  P_Buffer() : Z_Buffer() {}                           // initialize
private:
  virtual this_Algorithm(geom object, int x, int y,
                          float depth, boolean boundary_key) {
    if ( depth > depth_buffer[x][y] ) {                // update both buffers
      depth_buffer[x][y] = depth;
      if ( boundary_key )
        frame_buffer[x][y] = LINE_COLOR;               // draw the line
      else
        if ( pattern_buffer[x][y] == 0 )               // dash the others
          frame_buffer[x][y] = BACKGROUND_COLOR;
    }
    else                                               // update image buffer
      if ( boundary_key && value_Pattern(x, y) )
        frame_buffer[x][y] = LINE_COLOR;               // draw a dash-line
  }
  // return the pattern's value at pixel [x,y], 1 = PASS and 0 = BLOCK
  boolean value_Pattern(int x, int y);
};
```