On the Computational Requirements of Virtual Reality Systems

Frank Dévai

School of Computing & Mathematics, University of Ulster
Londonderry, BT48 7JL, UK

Abstract

The computational requirements of high-quality, real-time rendering exceeds the limits of generally available computing power. However illumination effects, except shadows, are less noticeable on moving pictures. Shadows can be produced with the same techniques used for visibility computations, therefore the basic requirements of real-time rendering are transformations, pre-selection of the part of the scene to be displayed and visibility computations. Transformations scale well, ie, their time requirement grows linearly with the input size. Pre-selection, if implemented by the traditional way of polygon clipping, has a growing rate of $N \log N$ in the worst case, where N is the total number of edges in the scene. Visibility computations, exhibiting a quadratic growing rate, are the bottleneck from a theoretical point of view. Three approaches are discussed to speed up visibility computations: (i) reducing the expected running time to $O(N \log N)$ (ii) using approximation algorithms with $O(NK)$ worst-case time, where K is the linear resolution of the image, and (iii) applying parallel techniques leading to logarithmic time in the worst-case. Though the growing rate of the time requirement of pre-selection is significantly slower than that of visibility, it is demonstrated that pre-selection has to deal with a significantly higher amount of data than visibility computations, as the average clipping volume is 1/27 of the volume of the model.

1. Introduction

Virtual reality (VR) is a new human-computer interface paradigm to create the effect of a three-dimensional environment in which the user directly interacts with virtual objects. An *immersive virtual environment* allows human participants to engage their perceptual skills in solving problems [47, 83]. Immersive systems require special equipment, eg, a head-mounted display. *Desktop, or non-immersive* systems use a normal visual display unit that displays the image of the environment. The user interacts with input devices, such as a data glove or a three-dimensional mouse.

Despite recent advances in computer-graphics hardware, complex virtual environments cannot be displayed with a sufficiently high frame rate because of limitations in the available rendering performance. The necessary frame rate is around 25 frames/sec, though some researchers would tolerate lower rates. With frame rates less than 20 frames/sec scenes appear as a series of separate frames, and even frame rates between 20 an 60 frames sec may produce ghosting effects, ie, multiple images of the same object [47] Another impediment is *lag*, the delay between performing an action and seeing the result of that action. Lag is critical when trying to achieve immersion [86].

In spite of the shortcomings of contemporary VR systems, a wide range of application areas are reported in the literature: flight [71] and driving [5] simulation, scientific visualisation [9], medicine [4, 44], walk-through and fly-through of complex environments [28, 48, 88, 89], lighting-design [28] and even performance analysis of parallel computer systems [70]. Cobb *et al* [14] examined the feasibility of VR as a tool for the UK manufacturing industry. Education and training is another area with vast potential: students can fly through landscapes for a geography lesson, or travel down blood vessels in an anatomy class [78].

Spectacular applications are a training model for the repair mission of the Hubble Space Telescope and the reconstruction of the Dresden Frauenkirche. Shortly after NASA launched the Hubble Space Telescope in 1990, astronomers discovered flaws in its optical system. A preparation and the crew training for a repair and maintenance mission became a major NASA project. More than 100 members of the ground-support flight team were trained in immersive virtual environments, and the repair mission was successfully completed in December 1993 [56]. The Dresden Frauenkirche was destroyed when the city was bombed by the Allied forces in 1945. During the reconstruction of the church a model was created from the original building plans. A software package developed at the IBM T. J. Watson Research Center was used to view and walk through the model using both immersive and non-immersive technologies [48].

Another potentially spectacular application area is *telesensation*, a sort of a three-dimensional photography, when a scene from a remote location, eg, from nature or from a museum, is transmitted to a viewer. Then the scene is regenerated at the viewer's location, who can enter the scene, walk around there, and touch the objects found there [82].

Though VR is based on traditional computer-graphics technology, some new techniques developed specifically for VR also emerged recently. One of these techniques is *object pre-selection* or *culling*, when simple mechanisms are used to reject most of the objects. As a result, only a very small portion of the model has to go through the time-consuming process of visibility computations. Actually the concept is well known in computer graphics as clipping, but considering the huge amount of input data, some preprocessing is justified. Yagel and Ray [88] report on such a culling mechanism based on regular space subdivision. Only objects in the potentially visible set of cells are actually submitted to the hidden object removal algorithm. Schaufler and Stürzlinger [73] propose a three-dimensional image cache.

Another group of new techniques are *hierarchical* or *level-of-detail algorithms* and *object simplification* [43, 47, 72, 80, 81]. VR applications also increased the practical significance of research on reducing the *growing rate* of visibility algorithms [19, 20, 21, 18, 58, 59] both in the worst case and on the average. Though the concept *output-sensitive visibility algorithms* has been raised as early as the 1986 Computational Geometry conference [20, 66] Sudarsky and Gotsman [77] recently reported the application of output-sensitive visibility algorithms to dynamic scenes in VR.

On the hardware level graphics accelerators [50, 51], logic-enhanced memories [36, 51, 60], texture mapping [51] and *scaleable architectures* [27, 51, 60] are the new developments. Coppen *et al* [17] describe a distributed frame buffer architecture, designed to achieve fast display updates in response to dynamic transformations of graphical objects. As a matter of fact, developing scaleable architectures are basically the same concept as reducing the growing rate of algorithms.

Slater and Usoh [76] propose an alternative viewing pipeline simulating *peripheral vision* in immersive virtual environments. Peripheral vision offers important cues for direction of gaze and movement. Relatively few papers [9, 47] deal with the design and computational requirements of VR systems.

Development has traditionally been extensive in computer graphics: bigger memories and faster processors are becoming available due to increasingly faster electronic components. However, there are two inherent difficulties with this way of development. On one hand, this approach has already been pushed almost as far as it will go: simply the speed of light imposes a limit that cannot be surpassed by any electronic component. On the other hand, bigger memory capacity and computing power lead to bigger problems to be solved and more functionality requirements.

Unfortunately the prevailing theoretical background for three-dimensional computer graphics is inherently wrong. Indeed, this theoretical background cannot even predict or explain the performance of the most widely used hidden-surface technique, the z-buffer algorithm. The running time of the z-buffer algorithm is often claimed to be a linear function of the input size, or even constant [29, 31, 32, 33, 65, 85]. On the other hand, Schmitt [74] demonstrated how vertical and horizontal rectangles can force any hidden-line or hidden-surface algorithm to take at least quadratic time in the worst case. (This result is wrongly attributed to Fiume [30] by Foley et al [32, 33].) The quadratic lower bound can be demonstrated even if the input is only one simple polyhedron [20].

A constant running time, ie, a running time independent of the size of the input is a nonsense, which is impossible to achieve even with parallel processing. We will demonstrate in section 5 that the hidden-line and hidden-surface problems cannot be solved in faster than logarithmic time under a widely accepted parallel model of computation even if arbitrarily many processors were available.

The false assumption of the constant running time of the z-buffer algorithm is the result of a gross misunderstanding of some speculations made more than 20 years ago by Sutherland et al [79]. As the underestimated — and not experimentally obtained — timing results were tabulated, authors of textbooks took them as experimental data. It is regrettable that new textbooks on computer graphics are usually based on older ones, and not on research publications, therefore practitioners, system designers and even researchers work under the delusion that the z-buffer algorithm takes constant time.

This paper offers a new theoretical background for the real-time, realistic rendering of static scenes in general, and for the computational requirements of virtual-reality and CAD systems in particular. In section 2 first three fundamental computational problems of rendering of static scenes are identified. These are transformations, clipping and visibility computations. Then it is demonstrated that any transformation can be performed in time proportional to the total number N of the edges of the model, clipping in time at most proportional to $N \log N$, and that visibility computations need time at least proportional to N^2 in the worst case. Though for small N the cost of visibility computations can be negligible due to a small constant of proportionality imposed by a simple hidden-surface algorithm, as N increases visibility computations are becoming a bottleneck. In section 2.1 we demonstrate that the average clipping volume is 1/27 of the volume of the model, and section 2.1 that polygon clipping can be solved in linear time if $O(N \log N)$ time of preprocessing is allowed.

In section 3 a surprising fact is revealed that many hidden-line and hidden-surface algorithms thought to be efficient earlier actually take time proportional to N^3 in the worst case. Though the growing rate of the worst-case time is not easy to reduce below the function $N^2 \log N$ for any practical algorithm, the first approach reported here to deal with the visibility bottleneck is a hidden-surface algorithm with an expected running time proportional to $N \log N$.

In section 4 the possibilities of the exploitation of the finite resolution of the rendered image are investigated. A new analysis method is proposed that takes into account also the linear resolution K of the image. The traditional classification of visibility computations as object-space and image-space algorithms is challenged by distinguishing exact and approximation algorithms. The z-buffer algorithm is demonstrated to take time proportional to NK^2 both in the worst case and on the average. Then a second method is proposed to speed up visibility computations by using an approximation algorithm generating a data structure in $O(NK)$ time in the worst case that can be displayed in $O(K^2)$ time.

In section 5 the application of parallel algorithms that take $O(\log N)$ time in the worst case is proposed as a third approach. While it is well known that approximation methods such as the z-buffer, and ray tracing algorithms are relatively easy to implement on parallel computers, the parallel complexity of the exact hidden-line problem has been established only recently [22, 25]. Though these results are based on a theoretical model of parallel computation, the proposed algorithms can also be executed on real parallel machines in $O(\log^d N)$ time, where d is a small positive constant depending on the particular machine.

A more practical approach with the technology available in the foreseeable future is to assign a processor to each row of picture elements of a raster-scan image, in order to compute the image of that particular row [27]. Then the dominant computational problem is the determination of the visibility of a planar set of line segments. In section 6 distributed-memory parallel algorithms are considered. First a proof is provided that the planar visibility problem in itself takes $\Omega(n \log n)$ time even if the output is not required in a sorted order. Then four new algorithms, including two Las-Vegas type probability ones, are proposed and compared with five existing algorithms. A computational complexity analysis is provided in terms of time and space requirements for each algorithm. None of the new algorithms require sorting, merging or advanced data structures such as priority queues or segment trees. Segment trees are used only for the proof of the upper bound on the deterministic algorithms, while all four algorithms are based on elementary data structures, hence amenable to hardware implementation.

In section 7 a method and a test-data generation algorithm for the experimental performance evaluations of planar visibility algorithms are proposed. Finally in section 8 the practical significance of the proposed theoretical background is evaluated, and directions for further work are recommended.

2. Rendering three-dimensional scenes

For the description of three-dimensional objects polygon-mesh models are most widely used [47, 72, 84]. These models provide an exact description for objects modelled by polyhedra, and an approximation for objects with curved surfaces. A polygon-mesh model is a collection of simple polygons possibly with holes, such that the polygons can intersect only at their edges. In image synthesis the polygons can be treated separately, and the model to be displayed is often called a scene. Therefore, we can assume that the scene is a collection of pairwise disjoint simple polygons possibly with holes. In a static scene the distance between any pair of vertices is fixed, though the observer's position, called the viewpoint, is allowed to move.

In practice the viewpoint is fixed. To provide the illusion of movement, the system should be able to change the position and the dimensions of the model. For example, when the observer is given the illusion of moving around an object, the scene is rotated in the opposite direction. The illusion of perspective is also required in many applications. All the above functionality can be provided by the transformations of the model.

In image synthesis usually a left-handed coordinate system is used such that the x-axis points to the right, the y-axis upwards and the z-axis away from the observer. The perspective transformation moves the viewpoint to infinity, therefore a viewpoint of $u = (0, 0, -\infty)$ can be assumed for the remainder of the image synthesis process.

The models of practical importance are usually very large, and the system is required to render only a part of the model, eg, a field of view in a virtual-reality environment. Similarly in a CAD system the user most often concentrates only on a detail, and the parts of the model falling outside the range of the display device should be discarded. The above functionality is achieved by a process called clipping in computer graphics.

Finally the system should provide the illusion that objects nearer to the observer may hide objects farther from the observer. This functionality is provided by visibility computations. There is evidence that the human visual system recognises solids by extracting edges in an image [78]. Indeed, line-drawing images often used for visualisation of solids in CAD systems, in addition to shaded, realistic images. Therefore two types of visibility problems are distinguished.

Given a set S of pairwise disjoint, opaque and planar simple polygons possibly with holes and with a total of N edges in three-dimensional space, and a viewpoint u, $u = (0,0,-\infty)$.

- If one wishes to find each interval ι of all the boundaries of the polygons in S, such that all points of ι are visible from u, this problem is called the *hidden-line problem.*
- If one is interested to find each region ρ of each polygon in S, such that all points of ρ are visible from u, then the problem is referred to as the *hidden-surface problem.*

Since $u = (0,0,-\infty)$, a point p, $p = (x_p, y_p, z_p)$, of S is visible if z_p is smaller than the z-coordinate of any other point of S along the line through p parallel to the z-axis. In practice we require somewhat less than stated above. As we wish to generate a two-dimensional image, we need only a projection of the visible points onto a *projection plane* π.

Once the facilities for transformations, clipping and visibility computations have been provided, any two-dimensional image of a three-dimensional scene illuminated from the viewpoint can be generated. Visibility computations can also be used for shadow calculations: the parts of the scene are in shadow which are not visible from a given light source. In a global illumination model, such as the radiosity method [40] all light interactions in the scene can be determined in advance in a view-independent way. Theoretically this would require the determination of the visibility of the scene from every point of the scene. In practice an approximation is sufficient. Then a hidden-surface algorithm can be used to determine what is visible form the viewpoint. The result is photo-realistic images produced at the speed of a hidden-surface algorithm.

Our purpose is the analysis of image synthesis algorithms in a machine-independent way. Since polygon-mesh models typically contain a total of 10^4–10^6 edges, ie, the size of the input is large, the growing rate of time and space requirements is a good measure of efficiency. The following notation is used: If f and g are functions of nonnegative variables $n, m, \ldots$, we say 'f is $O(g)$' if there are positive constants c_1 and c_2 such that

$$f(n,m,...) \le c_1 g(n,m,...) + c_2$$

for all n, m, ... We say 'f is $\Omega(g)$' if g is $O(f)$, and 'f is $\Theta(g)$' if f is both $O(g)$ and $\Omega(g)$.

Any transformation can be implemented by the multiplication of each vector corresponding to a vertex of the scene with a 4 by 4 matrix. This matrix multiplication requires 16 multiplications and 12 additions for each one of the vertices. The number of vertices is the same as the number of edges for a set of polygons, and proportional to the number of edges of a polygon-mesh model. All transformations can be combined in a single transformation matrix, therefore transformations take $\Theta(N)$ time in the worst case. Clipping can be reduced to the problem of determining the intersection of a line segment with a set of polygons that can be solved in $O(N \log N)$ time in the worst case [24]. In section 2.2 we will demonstrate that clipping can also be done in linear time if preprocessing is allowed.

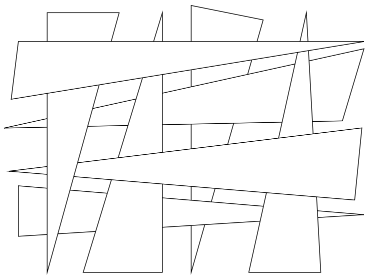
Figure 1: *A worst-case scene for visibility*

To determine the time requirement of visibility computations consider two groups of triangles such that each triangle in one group intersects every triangle in the other group as shown in Figure 1. If the number of triangles is $N/6$ in each group, the total number of edges is N, and the total number of edge intersections is $2N/6$ by $2N/6$ which is $N^2/9$ intersection points. Each intersection point is the endpoint of a visible line segment which must be reported in the output of any hidden-line algorithm. Also each intersection point is the vertex of a visible region which must be reported in the output of any hidden-surface algorithm. Therefore there exists an input for any hidden-line or hidden-surface algorithm that forces the algorithm to determine at least $N^2/9$ intersection points for N edges. We can conclude that $\Omega(N^2)$ is a lower bound for the visibility problem, ie, the time requirement of any visibility algorithm grows at least as fast as the N^2 function in the worst case. The time requirements of the visualisation of a polygon-mesh model with a total of N edges are summarised in Table 1.

function	time requirement
transformations	$\Theta(N)$
clipping	$O(N \log N)$
visibility computations	$\Omega(N^2)$

Table 1: *Computational requirements for displaying 3D scenes*

One can argue that the constant factor obtained for visibility computations is very small compared to the constant for the transformations. Note, however, that merely the number of intersection points is counted for the visibility problem, while the number of actual operations for the transformations.

For the sake of argument let us suppose that the calculation of one intersection point takes at least one time unit, eg, the time of one multiplication. Allowing one time unit also for an addition, $28N$ is an upper bound on the time requirement of transformations. The break-even point can be obtained from the formula $28N = N^2/9$, which gives $N = 252$. If $N > 252$, visibility calculations take more time than transformations.

On one hand it should be noted, however, that we made conservative estimates both on the lower bound for the visibility problem and on the upper bound for the transformation. On the other hand, our estimates apply to the time requirements in the worst case. One could find a more efficient algorithm for the average case. We follow this approach in section 3.

2.1. The average clipping volume

With object pre-selection or culling it is assume that only a small portion of the model has to go through the time-consuming process of visibility computations. The question how small this portion actually is naturally arises. In other words, what is the size of the average clipping volume?

Two answer this question, first we determine the size and the position of the average clipping window in two dimensions. Though small windows are probably more often used in practice, it will result in conservative estimates if we assume that windows with all sizes and positions are equally likely. We restrict ourselves to a model M with sides parallel to the coordinate axes.

Let M be a rectangle determined by the diagonal with the endpoints $(0,0)$ and (m,n), where $m,n > 0$. Now we determine the size and the location of the average window. For simplicity let m and n be integers. All windows being equally likely is the same as if the endpoints of their diagonals were chosen uniformly, independently at random from M. Choosing a point uniformly at random from M can be done by choosing an x-coordinate uniformly at random from the interval $[0,m]$ and then choosing a y-coordinate uniformly at random from the interval $[0,n]$ independently of the x-coordinate.

For simplifying the presentation, consider only windows with integer coordinates. Then the x-extents, the x-coordinates x_L of the left-hand sides of the possible windows and the appropriate number of windows can be given by the following table.

x-extent	x_L	number of windows
1	$0,1,2,\ldots,m-1$	m
2	$0,1,2,\ldots,m-2$	$m-1$
$\vdots$	$\vdots$	$\vdots$
i	$0,1,2,\ldots,m-i$	$m-i+1$
$\vdots$	$\vdots$	$\vdots$
$m-1$	$0,1$	2
m	0	1

The total number of windows with different x-coordinates is the sum of the third column of the table:

$$\sum_{k=1}^{m} k = \frac{m(m+1)}{2}.$$

The possible x_L values for a window of x-extent i are $0,1,2,\ldots,m-i$. Let a be the average value of x_L. Then

$$\begin{aligned} a &= \frac{2}{m(m+1)}\sum_{i=1}^{m}\sum_{j=0}^{m-i} j = \frac{2}{m(m+1)}\sum_{i=1}^{m}(m-i+1)\frac{m-i}{2} \\ &= \frac{1}{m(m+1)}\sum_{i=1}^{m}\left((m-i)^2+m-i\right). \end{aligned}$$

$\sum_{i=1}^{m}(m-i)^2$ can be rewritten as $\sum_{i=1}^{m-1} i^2$, and it can be demonstrated by mathematical induction that $\sum_{i=1}^{n} i^2 = n(n+1)(2n+1)/6$, hence we obtain

$$\begin{aligned} a &= \frac{1}{m(m+1)}\left(\frac{(m-1)m(2(m-1)+1)}{6}+\frac{m(m-1)}{2}\right) \\ &= \frac{(m-1)(2m-1)+3(m-1)}{6(m+1)} = \frac{m^2-1}{3(m+1)} = \frac{m-1}{3}. \end{aligned}$$

As m increases, a approaches $m/3$. We can make a similar argument for the y-dimension, then it follows that the bottom-left corner of the average window approaches the point $(m/3,n/3)$ if m and n get large.

Now let c and d respectively be the x- and y-dimensions of the average window. There are $m-i+1$ windows of x-extent i with different x-coordinates, therefore

$$c = \frac{2}{m(m+1)}\sum_{i=1}^{m}(m-i+1)i.$$

The sum can be rewritten as follows.

$$\sum_{i=1}^{m}(m-i+1)i = m\sum_{i=1}^{m} i - \sum_{i=1}^{m} i^2 + \sum_{i=1}^{m} i = (m+1)\sum_{i=1}^{m} i - \sum_{i=1}^{m} i^2.$$

Substituting $\sum_{i=1}^{m} i = m(m+1)/2$ and $\sum_{i=1}^{m} i^2 = m(m+1)(2m+1)/6$, we obtain

$$\begin{aligned} c &= \frac{2}{m(m+1)}\left(\frac{m(m+1)^2}{2}-\frac{m(m+1)(2m+1)}{6}\right) \\ &= m+1-\frac{2m+1}{3} = \frac{m+2}{3}. \end{aligned}$$

With a similar reasoning for d we can conclude that the size of the average window approaches $m/3$ by $n/3$ if the dimensions m and n of the model get large. Our derivations generalise in three dimensions with the important consequence that the average clipping volume is 1/27 of the volume of the model, assuming that all clipping volumes are equally likely.

2.2. Polygon clipping

As we have already mentioned, polygon clipping can be solved in $O(N \log N)$ time in the worst case, where N is the total number of edges in the scene. In this section we will demonstrate that it can also be done in linear time if some preprocessing is allowed [26].

The intersection of an arbitrary polygon with any face of the clipping volume can be obtained by determining the intersection of a line segment and a polygon. Indeed, if we project the polygon into a plane perpendicular to the face of the clipping volume, then it is sufficient to determine the intersection of the image of the polygon with the image of the face of the clipping volume — which is a line segment. Then the intersection points of the edges of the image of the polygon are projected back to the original polygon to obtain the intersection points of its edges with the face of the clipping volume.

A related problem we are going to solve is called the *line-polygon classification* (LPC) problem, and it can be formulated as follows. Given a line segment L and a polygon P with N edges in the plane, find their intersection. The result is a classification of the points of L in three subsets, such as L_{inP} containing the points of L lying in the interior of P, the subset L_{onP} of the points of L lying on the boundary of P and finally a subset L_{outP} lying outside P.

We begin with some definitions, and introduce the notion of *ordinary polygons*. A *path* is a sequence of points $p_1, p_2, \ldots, p_n$, and line segments $\overline{p_1, p_2}$, $\overline{p_2, p_3}$, $\ldots$, $\overline{p_{n-1}, p_n}$ connecting the appropriate point pairs. If the last point of the path is the same as its first point, the path is called a *closed path.*

A *polygon* is a subset P of the plane, such that P does not contain a half-line, and the boundary of P is a finite set of closed paths. The points defining the closed paths on the boundary of P are called the *vertices* of P, and the line segments of the closed paths connecting the vertices of P are called the *edges* of P.

A polygon P is an *ordinary polygon* if P is a connected subset of the plane, the closed paths defining the boundary of P are disjoint, and no non-consecutive edges of any closed path intersect.

If P is an ordinary polygon, then each of its vertices is shared by exactly two edges. The subdivision of the plane induced by the boundary of P may have some regions which do not contain a half-line, but do not belong to P. Such a region is called a *hole*.

An ordinary polygon can be described by the set of closed paths defining its boundary. Each closed path can be given by the sequence of its vertices. One of the closed paths will describe the *outer boundary* of the polygon, and the remaining ones (if any) will specify holes. We will adopt the convention that the vertices of the outer boundary are given in counter-clockwise order, and the vertices of a hole in clockwise order. Then the *interior* of an ordinary polygon will always lie to the left as its boundary is traversed.

An ordinary polygon which is a simply connected subset of the plane is called a *simple polygon*. A polygon P is said to be *convex* if any line segment connecting two points inside P is itself entirely inside P. There is a hierarchy strictly ordered by the subset relation

$$\text{convex} \subset \text{simple} \subset \text{ordinary polygons},$$

that is, the class of ordinary polygons include all convex polygons, all simple polygons, and all simple polygons with holes. Now we will prove a lower bound.

Lemma 1 $\Omega(N \log N)$ is a lower bound on worst-case time for determining the intersection of a line segment and an ordinary polygon with N edges, assuming the algebraic tree model of computation.

Proof: We will demonstrate that any algorithm that determines the intersection of a line segment and an ordinary polygon with N edges can decide the ε-*closeness problem* by using $O(N)$ additional algebraic operations. The ε-closeness problem is as follows. Given $N+1$ real numbers $r_1, r_2, \ldots, r_N$ and $\varepsilon > 0$, decide if any pair r_i and r_j are at a distance less than ε from each other, i.e., there are i and j, $1 \le i, j \le N$, such that $i \ne j$ and $|r_i - r_j| < \varepsilon$.

Let A be an arbitrary LPC algorithm, and let $x_1, x_2, \ldots, x_N$ be a set of N real numbers such that

$$|x_i - x_j| \ge \varepsilon \text{ for all } i \ne j, 1 \le i, j \le N.$$

We construct an ordinary polygon P with boundaries $((a, -\varepsilon), (b, -\varepsilon), (b, \varepsilon), (a, \varepsilon))$ and $((x_i, 0), (x_i + \delta, \delta), (x_i + \delta, -\delta))$, as shown in Figure 1, where $a = \min\{x_i\} - \varepsilon$ and $b = \max\{x_i\} + \varepsilon$ for $1 \le i \le N$, and δ, $0 < \delta < \varepsilon$, is an arbitrarily small real number.

Let a candidate line segment L be defined by the endpoints $(a-1, 0)$ and $(b+1, 0)$. Then the set L_{inP} returned by A will contain $N+1$ line segments of length $\ge \varepsilon - \delta$.

Now let δ approach zero, and let

$$x_1, x_2, \ldots, x_N$$

be an instance of the ε-closeness problem. Classify L with respect to P using algorithm A. Then if any interval in L_{inP} has a length less than ε, return a YES, otherwise a NO answer for the ε-closeness problem.

The ε-closeness problem takes $\Omega(N \log N)$ time in the worst case, assuming the algebraic tree model of computation [6]. Let $T_A(N)$ be the running time of algorithm A. There exists a positive constant c such that the construction of P and the examination of the intervals in L_{inP} together can be done in at most cN algebraic operations, therefore

$$T_A(N) + cN = \Omega(N \log N)$$

from which the lemma follows. □

First we describe a preprocessing algorithm that converts polygon P into a *planar straight-line graph* G. G will contain

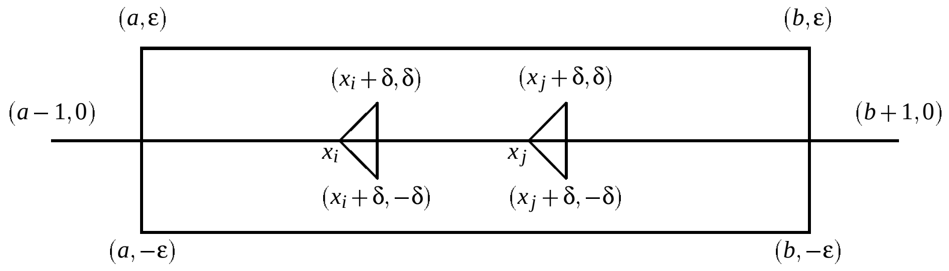

Figure 2: *The ε-closeness problem is reducible to the LPC problem*

all vertices of P, and some additional vertices and edges. For simplifying the presentation, we will assume that all vertices of P have distinct x- and y-coordinates, and the half-line h (to be specified later) containing L does not go through any vertices of G. From here it follows that no edge of P will be vertical or horizontal, and L_{onP} is a set of measure zero. Without the above restrictions algorithms longer in detail but not in asymptotic time can be given.

We say that two edges e and f of P are *comparable at abscissa* ξ if the vertical line $x = \xi$ intersects both e and f. Then the relation *above at* ξ can be defined as follows: e is above f at ξ if e and f are comparable at ξ, and the intersection of e with the line $x = \xi$ has an ordinate greater than the ordinate of the intersection of f with the line $x = \xi$. Note that the relation above at ξ is a linear order on the set of edges intersected by the same vertical line. We will use an abstract data type, called a *linearly ordered set, T*, to maintain the order of edges. An important technical detail, as we will see later, that the edges in T are represented by the equation of the line containing the particular edge.

Let v_i, $1 \le i \le N$, be the set of vertices of P, and let $a = \min\{x_i\}$, $b = \max\{x_i\}$, $c = \min\{y_i\}$ and $d = \max\{y_i\}$, where x_i and y_i are the x- and y-coordinates of vertex v_i. Let $v_t = (x_t, y_t)$ and $v_b = (x_b, y_b)$ respectively denote the top and bottom extreme vertices of P, i.e., $y_t = d$ and $y_b = c$. According to our first assumption, the vertices v_t and v_b are unique. We will introduce four new vertices in G, such as $v_{3N+1} = (a-1, c)$, $v_{3N+2} = (b+1, c)$, $v_{3N+3} = (a-1, d)$ and $v_{3N+4} = (b+1, d)$. (See Figure 2.) Once the preparation of G has been finished, we will remove these vertices together with the edges incident on them. At each vertex of G we imagine a line parallel to the y-axis, and call the edges incident with the vertex and left to the imaginary line the *incoming edges* and those right to the line the *outgoing edges*. Then the preprocessing algorithm can be stated as follows.

1. Let G be P initially. Add vertices $v_{3N+1}, \cdots, v_{3N+4}$ to G together with the edges $\overline{v_{3N+1}, v_b}$, $\overline{v_b, v_{3N+2}}$, $\overline{v_{3N+3}, v_t}$ and $\overline{v_t, v_{3N+4}}$, and initialise T with edges $\overline{v_{3N+1}, v_b}$ and $\overline{v_{3N+3}, v_t}$. Sort the vertices of P by their x-coordinate in increasing order, and initialise two N-element arrays TOP and BOTTOM.
2. Examine the vertices v_i of P in turn from left to right.
   a. Delete from T the incoming edges of G incident on v_i, and insert in T the outgoing edges of G incident on v_i.
   b. Let v_{2i} and v_{3i} respectively be the intersection points of the vertical line through v_i with the edge $\overline{v_j, v_k}$ above v_i in T and with the edge $\overline{v_l, v_m}$ below v_i in T, where $1 \le j, k, l, m \le 3N+4$.
   Add vertices v_{2i} and v_{3i} and edges $\overline{v_i, v_{2i}}$ and $\overline{v_i, v_{3i}}$ to G, and replace edge $\overline{v_j, v_k}$ by edges $\overline{v_j, v_{2i}}$ and $\overline{v_{2i}, v_k}$, and edge $\overline{v_l, v_m}$ by edges $\overline{v_l, v_{3i}}$ and $\overline{v_{3i}, v_m}$.
   Whenever a horizontal edge $\overline{v_j, v_t}$ or $\overline{v_t, v_k}$ is replaced, write the x-coordinate x_{2i} of v_{2i} in the next element of the array TOP, and whenever a horizontal edge $\overline{v_l, v_b}$ or $\overline{v_b, v_m}$ is replaced, write the x-coordinate x_{3i} of v_{3i} in the next element of the array BOTTOM.
3. Remove vertices v_{3N+1} to v_{3N+4} from G together with the edges incident on them.

The faces of G will be trapezoids which may degenerate into triangles. An example is given in Figure 2. G is shown after step 2 of the above algorithm. The edges of G which are also edges of P or contained by the edges of P are shown in heavy lines.

Although some of the edges of P may be replaced by $\Theta(N)$ edges in G, we can demonstrate that G has asymptotically the same size as P.

Lemma 2 The planar subdivision G has at most $3N$ vertices and at most $5N$ edges.

Proof: Initially G will have the same number of vertices and edges as P, i.e., N vertices and N edges. At each vertex v_i of P, $1 \le i \le N$, we introduce at most two new vertices in step 2.b of the above algorithm. The four extra vertices introduced in step 1 will be removed in step 3, therefore the total number of vertices of G is at most $3N$. Similarly, at each vertex v_i we introduce at most two new edges, and replace each of at most two existing edges of G by two new edges. Therefore, the total number of edges of G is at most $5N$. □

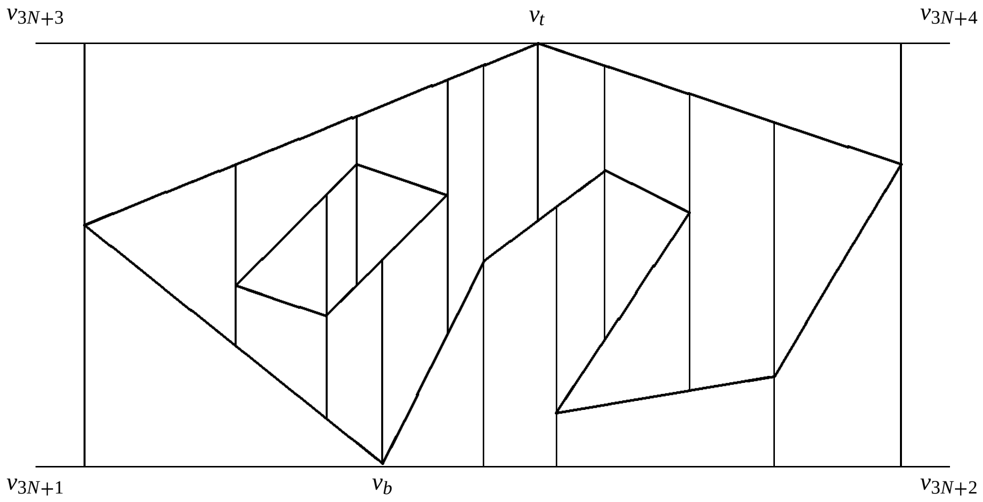

Figure 3: *The preprocessing of an ordinary polygon*

The linearly ordered set T can be realised by a balanced tree [1,2] such that the leaf nodes of the tree are labelled by line equations, and are also threaded by a doubly linked list. Then the operations *insert* and *delete* can be implemented in $O(\log N)$ time, while *above* and *below* in constant time. Thus step 2.a takes $O(N \log N)$ time, and step 2.b $O(N)$ time. Step 1, including the sorting, can be implemented in $O(N \log N)$ time, and step 3 takes constant time, therefore we obtain the following upper bound on the preprocessing time.

Lemma 3 An ordinary polygon with N edges can be converted into a planar subdivision with $\Theta(N)$ trapezoidal faces in $O(N \log N)$ time in the worst case.

The planar subdivision G can be represented by doubly linked adjacency lists. Then any half-line can be classified with respect to G, and therefore P, by a linear-time incremental algorithm. This algorithm traverses the boundary of each trapezoid intersected by the half-line in turn. Let R be the rectangle defined by the four coordinate pairs (a,c), (b,c), (b,d) and (a,d), and let p and q be the endpoints of L, and finally let h be the half-line starting from p and containing L. The equation of h in the parametric form is

$$r = p + (q-p)t \quad \text{for } 0 \leq t \leq \infty,$$

where r is an arbitrary point of h, and t is a scalar parameter. Then L can be classified with respect to P as follows.

1. If $L \cap R = \emptyset$, L and P must be disjoint, the algorithm terminates.
2. Otherwise if h has one intersection point with the boundary of R, let s be the intersection point. If h has two intersection points, rename the intersection point nearer to p as p, and let s be the other intersection point.
3. Find the trapezoid Z that h entering at point s. If s is on a vertical side of R, this means the selection of one of two edges, and if s is on a horizontal side of R, find Z by binary searching the arrays TOP or BOTTOM.
4. Starting from s, traverse the boundary of Z to find the intersection point w where $\overline{p,s}$ leaves Z if such a w exists.
5. Repeat this procedure from step 4 by renaming w as s until an intersection point w cannot be found, or w coincides with p.

The algorithm requires only elementary data structures, such as lists and arrays. We obtain the intersection points of L with the edges of G in decreasing order of the values of the parameter t, and consider only the intersection points where the edge of G is also an edge of P or contained by an edge of P. Then we can prove the following result.

Theorem 1 The intersection of a line segment and an ordinary polygon with N edges can be determined in $O(N)$ time, assuming $\Theta(N \log N)$ preprocessing time is allowed.

Proof: According to Lemma 3, G can be prepared in $O(N \log N)$ time in the worst case. The first two steps of the classification algorithm take constant time, and step 3 can be executed in $O(\log N)$ time in the worst case. In step 4 and step 5 we traverse any edge of G at most two times. According to Lemma 2, G has at most $5N$ edges, which results in an $O(N)$ bound for the classification algorithm, and from here the theorem follows. $\square$

Since all the operations of the above algorithms are available in the algebraic tree model of computation, considering Lemma 1, we obtain a tight bound on the complexity of the LPC problem.

Corollary 1 The line-polygon classification problem for an ordinary polygon with N edges takes $\Theta(N \log N)$ time in the worst case, and this time cannot be reduced in the algebraic tree model of computation.

Given two ordinary polygons P and Q with m and n vertices, respectively. Their intersection $P \cap Q$ can be constructed as follows. Let a_i, $1 \leq i \leq m$, be the set of edges of P, and let a_i^{inQ} be the set of closed intervals along which a_i is contained by polygon Q. Let we denote by A^{inQ} the part of the boundary of P contained by Q, and by B^{inP} the part of the boundary of Q contained by P. Then the boundary of $P \cap Q$ is the union of A^{inQ} and B^{inP}. We use the following algorithm.

1. Prepare the trapezoidal decomposition of Q. For each edge a_i of P, find a_i^{inQ} by using the line-polygon intersection algorithm proposed here. Then set $A^{inQ} = \bigcup_i a_i^{inQ}$, for $1 \leq i \leq m$.
2. Find B^{inP} analogously.
3. Obtain $P \cap Q$ as the polygon whose boundary is $A^{inQ} \cup B^{inP}$.

Now, we can prove a tight bound on the complexity of the polygon intersection problem.

Theorem 2 Determining the intersection of an ordinary polygon of m edges with another ordinary polygon of n edges takes $\Theta(m \log m + mn + n \log n)$ time in the worst case, and this cannot be further improved in the algebraic tree model of computation.

Proof: The set of intervals a_i^{inQ} for all $1 \leq i \leq m$ can be found in $O(mn)$ time, after preprocessing Q in $O(n \log n)$ time. Therefore, step 1 takes $O(mn + n \log n)$ time. Similarly, step 2 takes $O(mn + m \log m)$ time. Then $P \cap Q$ can be specified with the enumeration of its boundary by alternately traversing the edges of A^{inQ} and B^{inP} according to the orientation of P and Q. Therefore step 3 takes $O(mn)$ time, and the whole algorithm $O(m \log m + mn + n \log n)$ time. An $\Omega(m \log m + n \log n)$ lower bound follows from Lemma 1, and an $\Omega(mn)$ bound from the fact that the intersection of a simple m-gon with a simple n-gon may have mn vertices [69]. □

Note that $P \cap Q$ may be disconnected, and therefore not necessarily an ordinary polygon. Once A^{inQ} and B^{inP} has been determined, $P \cup Q$, $P - Q$ and $Q - P$ can also be determined by traversing edges in $O(mn)$ additional time.

For clipping a set of three-dimensional polygons, as we have already seen, we have to solve the following problem. Given a line segment L and a set of M possibly non-disjoint ordinary polygons $P_1, P_2, \ldots, P_M$ with a total of N edges in the plane. Find the set of (possibly non-disjoint) intervals of L corresponding to the intersection of L with all P_i, $1 \leq i \leq M$.

After the preprocessing of each polygon in trapezoids as given above, the set of intervals can be found in $O(N)$ time. For the preprocessing $O(N \log N)$ total time will be sufficient. Indeed, let $n_1, n_2, \ldots, n_M$ be respectively the number of edges of $P_1, P_2, \ldots, P_M$. Then $\sum_i n_i = N$, and for each polygon P_i there exists a positive constant c such that the preprocessing of P_i can be obtained in time $c(n_i \log n_i)$. Since $\log n_i \leq \log N$ for any n_i, the total preprocessing time for the set of M polygons is

$$c \sum_{i=1}^{M} (n_i \log n_i) \leq c(\log N) \sum_{i=1}^{M} n_i = cN \log N.$$

Hence we can conclude that the polygon-clipping problem can be solved in linear time if $O(N \log N)$ time of preprocessing is allowed.

3. Improving the expected running time

Even if one can find all intersections in $O(N^2)$ time, it is not sufficient to determine visibility. Most of the algorithms proposed in the literature [34, 37, 45, 46, 57] divide edges into line segments at the intersection points, then test each line segment for visibility against each polygon. As we have seen earlier, the total number of intersection points — and therefore the total number of line segments — is $\Omega(N^2)$ in the worst case.

If the input polygons form the faces of a collection of simple polyhedra, it can be demonstrated that the total number of polygons is at least $N/3$. Indeed, let m be the total number of polyhedra. Then for each polyhedron $p_i, 1 \leq i \leq m$, Euler's polyhedron theorem

$$v_i + f_i = e_i + 2 \tag{1}$$

holds, where v_i, f_i and e_i, respectively, are the number of vertices, faces and edges of polyhedron p_i. It follows from 1 that $f_i > e_i - v_i$ for each p_i, then for the total number of faces, $\sum f_i$, one can write

$$\sum_{i=1}^{m} f_i > \sum_{i=1}^{m} e_i - \sum_{i=1}^{m} v_i = N - \sum_{i=1}^{m} v_i \tag{2}$$

where $\sum e_i$ is the total number of edges, and $\sum v_i$ is the total number of vertices. There must be at least three edges emanating from each vertex, and each edge is incident on exactly two vertices, therefore $\sum v_i \leq 2N/3$. Then it follows from 2 that $\sum f_i > N/3$. Testing $\Omega(N^2)$ line segments against $N/3$ faces takes $\Omega(N^3)$ time. It is also possible to demonstrate, however, that $\Theta(N^3)$ time is sufficient for the above-mentioned algorithms in the worst case.

For an improvement it should be noted that any visibility algorithm has to determine the union of $\Theta(N)$ hidden intervals on $\Theta(N)$ edges in the worst case. Since $\Omega(N \log N)$ is a lower bound for determining the union of N intervals [69], it appears that the best one can hope to achieve is a $\Theta(N^2 \log N)$ worst-case time. Though it has been demonstrated that $\Theta(N^2)$ worst-case time can actually be attained both for the hidden-line [20] and for the hidden-surface problem [59] it is not easy to reduce below $\Theta(N^2 \log N)$ for any practical algorithm.

Our worst-case lower bound is based on the fact that there can be $\Omega(N^2)$ intersection points. Then the question naturally arises whether a better algorithm exists for the cases when there are a smaller number of intersections. Indeed, the algorithm given below takes $O((N+k)\log N)$ time, where k, $k < 0 < N(N-1)/2$, is the total number of intersection points.

The following observations are used [21]. If all polygons are projected into the viewing plane π, the edges of the polygons induce a planar subdivision G of π. To avoid confusion, the vertices, edges and faces of G will be referred to as nodes, arcs and regions respectively. When moving from one region into a neighbouring region by crossing an arc of G, one either enters the projection of a polygon P or leave the projection of a polygon Q. Within each region of G the polygons can be ordered according to their distance from the observer even if the polygons cyclically overlap. The ordering can be maintained by a priority queue, and the algorithm, called the *priority-queue method*, is stated as follows.

1. Determine the planar subdivision G of the projection plane π induced by the images of the N edges in the input.
2. Visit all regions of G systematically by moving from one region into a neighbouring region by crossing an arc of G. If the projection of a polygon P is entered, insert P in a priority queue H, and if the projection of a polygon Q is left, delete Q from H. The polygon corresponding to the minimum element of H will be visible within the currently visited region.

Step (1) can be implemented in $O((N+k)\log N)$ time [7, 10, 64]. While traversing the regions of G, one has to account at most one insertion and at most one deletion for each crossing of an arc of G. Both an insertion and a deletion can be done in $O(\log N)$ time [2, 52, 75] and the number of arcs is O(N + k), therefore step (2), and also the whole algorithm, can be implemented in $O((N+k)\log N)$ time.

Having an upper bound on the running time as a function of k, we can now prove that if the expected number of intersections is $O(N)$, the expected running time of the algorithm is $O(N\log N)$, regardless of the underlying probability distribution of the input data. Indeed, let $E\{k\}$ be the expected number of intersection points, and let $t(N)$ be the expected running time of the algorithm. An upper bound on the running time $T(N)$ of the algorithm can be expressed as $c(N+k)\log N$ for some $c > 0$. Now, assume that $E\{k\} = O(N)$. Then

$$\begin{aligned} t(N) &= E\{T(N)\} \le E\{c(N+k)\log N\} \\ &= cN\log N + c(\log N)E\{k\} \\ &= O(N\log N), \end{aligned}$$

which was to be demonstrated. Note that no assumption on the distribution of the input data was made, the only requirement is that the expected number of edge intersections is $O(N)$.

4. Approximation algorithms

A widely used classification of visibility algorithms distinguishes two main classes: object-space and image-space algorithms [32, 33, 65, 79]. Object-space algorithms are supposed to make calculations on the three-dimensional scene, while image-space algorithms on the two-dimensional image. (Foley et al [32] use a slightly different terminology: image- and object-precision algorithms, actually with the same meaning.) Image-space algorithms are also supposed to exploit the finite resolution of the image, while object-space algorithms are assumed to be independent of the display device.

This classification, however, is inappropriate in some respects. First, as visibility can only be decided by the comparison of z-coordinates, it is not possible to perform all calculations in image space. Thus, there is no pure image-space algorithm. Second, performing any possible amount of calculations in image space does not necessarily mean the exploitation of the finite resolution of the image.

The algorithm presented in section 3 does most of its calculations in image space (finding edge intersections, determining and traversing the subdivision of π) but the resolution of the image is not even mentioned. We propose a more appropriate classification.

An algorithm is called an *exact algorithm* if it determines each visible point of the scene, and maps them onto the projection plane π. Then the algorithm in section 3 should be classified as an exact algorithm. Another class of algorithms, called *approximation algorithms*, as opposed to exact algorithms, determine and map onto the projection plane only a subset of the visible points of the scene. In other words, these algorithms compute approximations to the visible set of points.

In raster displays picture elements, called *pixels* for short, of ε by ε size are used to approximate the exact image. As the aspect ratio of images is usually constant, we can assume without loss of generality that the number of pixels, $K = 1/\varepsilon$, is the same in both the horizontal and vertical directions.

The visibility algorithms traditionally classified as image-space algorithms are actually approximation algorithms in most of the cases. The dominant visibility algorithms in use nowadays are z-buffer scan-line algorithms and ray casting [41]. It easy to see that the z-buffer and the ray casting methods take $\Theta(K^2N)$ time in the worst case. It is also not hard to demonstrate that the expected running time of the z-buffer algorithm is still $\Theta(K^2N)$, eg, for a set of orthogonal rectangles, assuming that the coordinates of their vertices are taken uniformly, independently at random from a rectangular parallelepiped. Indeed, if one chooses two points uniformly, independently at random from a unit square in the plane, the expected value of the area of the orthogonal rectangle determined by the two points is 1/9.

Assuming just as many pixels as edges, ie, $K^2 = N$, would suggest that accepting an approximation does not pay: exact algorithms, eg, Goodrich's algorithm [38], are asymptotically faster. Perhaps using a hierarchical approach that can compare more than one pixel at a time to each polygon would result in a faster algorithm. Warnock developed an algorithm that uses the observation that pixels within large areas of the image are coherent in the sense that they represent a single polygon [65, 32]. However, it has been demonstrated [24] that Warnock's algorithm also takes $\Theta(K^2N)$ time in the worst case. Greene et al [41] describe a heuristic method similar to Warnock's algorithm, and report that a hierarchical approach can be faster in practice than the ordinary z-buffer algorithm. In the remainder of this section we demonstrate that hierarchical data structures, called z-trees, can be used to develop approximation algorithms with running times of sublinear functions of the total number of pixels.

A scan-line variant of the z-buffer algorithm takes $\Theta(KN)$ time, while the scan-line variant of the z-tree method takes $O(N\log K + K)$ time in the worst case. A z-tree for a two-dimensional problem is a binary tree, where the root node represents the whole scan line of K pixels, the left son of the root the pixels numbered from 1 to $\lfloor(K+1)/2\rfloor$, and the right son the pixels numbered from $\lfloor(K+1)/2\rfloor + 1$ to K. In general, if a node represents pixels from l to r, its left son will represent those from l to $\lfloor(r+l)/2\rfloor$, and its right son from $\lfloor(r+l)/2\rfloor + 1$ to r.

Associated with each nonterminal node V is a vertical line L such that the pixels represented by the left son of V are to the left of L, and those represented by the right son of V are to the right of L. There are also associated with each node two integers z_1 and z_2. The line segment B determined by points (l, z_1) and (r, z_2) is called the *blocker* of V. In Figure 4 a z-tree representation of a line segment is given: blockers assigned to leaves denoted by solid squares generate the original segment.

The introduction of blockers allows for the representation of all the N line segments in the scan plane by a single z-tree. If several blockers occur with the same node, then only the nearest to the viewpoint is retained. Assuming K pixels, a binary z-tree has at most K leaf nodes and $K-1$ internal nodes, regardless of the number of line segments inserted in the tree. That is, the information for the invisible parts of the segments is lost.

The z-tree conversion of a line segment can be accomplished recursively as follows. Let V be the root node of the tree initially.

(1) If the current line segment X is totally hidden by the blocker B of node V, then discard X and return.
(2) Otherwise if B is totally hidden by X, substitute X for B, and return.
(3) If node V is not a leaf, then

(3.1) if the right endpoint of X is to the left of the vertical line L assigned to V, execute the algorithm on the left subtree, and if the left endpoint of X is to the right of L, execute the algorithm on the right subtree of V;
(3.2) otherwise split X is by L, and execute the algorithm on both the left and the right subtrees of V.

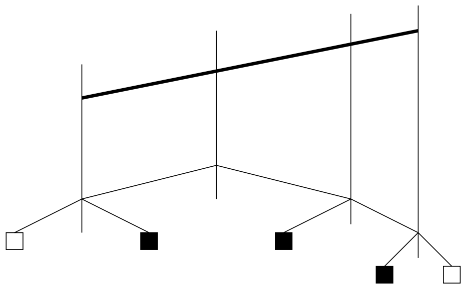
Figure 4: *A z-tree representation of a line segment*

A binary z-tree for N line segments and K pixels can be built in $O(N\log K)$ time, and takes $O(K)$ space in the worst case [23]. The visibility of the scan line can be obtained by a preorder traversal [1] of the z-tree. Let V be initially the root node of the tree, and let F be a background segment which is farther from the viewpoint u than any other segment in the scan plane. Then a preorder traversal of a binary z-tree can be done recursively as follows.

(1) Visit node V. If F is nearer to u than the blocker B of V, substitute the appropriate part of F for B.
(2) Let F be the blocker of V obtained in step (1). Visit in preorder the subtree with root V_1, then the subtree with root V_2, where V_1 and V_2 are the left and right sons of V respectively.

The above algorithm takes $O(K)$ time, which is asymptotically the same as the output time of a z-buffer. Therefore, the scan line variant of the z-tree method takes $O(N\log K + K)$ time in the worst case. A three-dimensional generalisation of the z-tree method generates a hierarchical data structure in $O(NK)$ time that can be displayed in $O(K^2)$ time [23].

5. Parallel complexity

Distinguishing exact and approximation methods is also important when considering parallel solutions to the visibility problem. Most parallel approaches recommended until recently are based on the z-buffer [36, 61] or ray-tracing [49, 68] methods that classify as approximation algorithms. With these methods usually a processor is assigned to each pixel. In general, by using approximation methods the visibility problem can be divided conveniently among as many processors as are available.

It is more difficult to find solutions to the exact hidden-line and hidden-surface problems. The most widely accepted theoretical models of parallel computation are the variants of the *Parallel Random Access Machine* (PRAM) model.

The PRAM model is a collection of random access machines and a global memory. All the processors have access to the global memory, and run synchronously. The global memory accesses are assumed to take unit time. The variants of the PRAM model handle concurrent reads and writes to the global memory cells differently. The major variants are the exclusive read, exclusive write (EREW), concurrent read, exclusive write (CREW) and concurrent read, concurrent write (CRCW) models. The most often used variant is the CREW PRAM model. In this model any number of processors can read a given global memory cell at once, but at most one processor is allowed to write into a given memory cell in one step. If more than one attempts to write, the computation is invalid.

The generally accepted definition for the fast solvability of a problem by parallel algorithms is if it can be solved in time polynomial in $\log N$ by using a number of processors polynomial in N, where N is the problem size. This class of problems is commonly referred to as $\mathcal{NC}$. One reason why $\mathcal{NC}$ is broadly accepted as the class of problems amenable to parallelization is that this class remains the same whether it is defined in terms of any variant of the PRAM model, or in terms of any other reasonable model, eg, uniform circuits. To convert one model to another, a slow-down or speed-up of a factor of $\log N$ emerges. The advantage of $\mathcal{NC}$ is that it allows us to ignore the factors of $\log N$ that separate the various models.

It has been demonstrated that the exact hidden-line problem can be solved in $\Theta(\log N)$ time in the worst case with N^2 processors, and that the $\Theta(\log N)$ time cannot be further improved in the CREW PRAM model, even if arbitrarily many processors are available [22]. We prove the same result for the EREW model [25]. The EREW model is the variant of PRAM closest to real machines.

First we propose a parallel algorithm for determining the union of a set of intervals, and then we use this algorithm to develop a parallel hidden-line algorithm. No parallel algorithm is known in the literature for the interval-union problem, though it has been demonstrated [22] that the hidden-line problem can be solved in $\Theta(\log N)$ time with N^2 processors under the CREW model. The parallel hidden-line algorithm proposed earlier [22] uses a parallel sorting algorithm as a preprocessing step, and parallel sorting seems to be a promising starting point also for the interval-union problem. We can use an optimal EREW parallel sorting algorithm proposed by Cole [15]. Unfortunately, the hidden-line problem is significantly more difficult than sorting, and the earlier parallel hidden-line algorithm [22] relies heavily on concurrent-read operations.

In section 5.1 we introduce the problem of union of point sets. In particular, if the point sets are N intervals of the real line, we demonstrate that the complexity of the problem is $\Theta(N \log N)$ under the algebraic tree model of computation. Then we present a parallel algorithm that takes $O(\log N)$ time and N processors. The product of time and processor number is equal to the sequential complexity of the problem, therefore the algorithm achieves a perfect speedup.

In section 5.2 the parallel interval-union algorithm is used to develop an algorithm for hidden-line elimination. The hidden-line algorithm takes $O(\log N)$ time with N^2 processors, and therefore achieves a linear speedup on the $O(N^2 \log N)$ worst-case time of the best known practicable sequential algorithms.

5.1. The interval-union problem

In a wide range of application areas such as computer-aided design, geographic information systems, data processing and computer graphics we are often required to find the *union of point sets*: Given a collection $R_1, R_2, \ldots, R_N$ of N point sets, determine the set S defined by $R_1 \cup R_2 \cup \ldots \cup R_N$. In particular, the hidden-line problem requires the determination of a subset of a line segment L contained by a collection of point sets $R_1, R_2, \ldots, R_k$. More precisely, if L is the image of an edge, and $R_1, R_2, \ldots, R_k$ are images of polygons lying between the edge and an observer in three-dimensional space, then the visible subset V of L to be displayed is

$$V = L - \{R_1 \cup R_2 \cup \ldots \cup R_k\} \tag{3}$$

where $R_1, R_2, \ldots, R_k$ are also polygons. Surprisingly, the computation of equation 3 according to the definition would be both excessive and insufficient at the same time. It is insufficient, because a particular polygon R_i, $1 \leq i \leq k$, may cover L along some intervals, but may not cover it along some other intervals, eg, if R_i is a simple polygon with holes. On the other hand, the computation of equation 3 is excessive, since $R_1 \cup R_2 \cup \ldots \cup R_k$ is not required; what we only need is the union of the hidden intervals of L.

Then the *interval-union problem*, as a special case of the problem of the union of point sets, can be formulated as follows: Given a list of $2N$ real numbers representing the endpoints of N intervals, compute the union of these intervals.

Fredman and Weide [35] have established the complexity of a similar problem, ie, the *measure* of the union of a set of intervals, under the linear decision tree model of computation. It is relatively straightforward to establish the complexity of the interval-union problem under a more general model of computation, called the *algebraic tree* [6] by demonstrating that any algorithm that can find the union of N intervals can also decide the element distinctness problem. The *element distinctness problem* is stated as follows [69]. Given N real numbers, $x_1, x_2, \ldots, x_N$, decide if all are different (ie, there are no i and j, $1 \leq i, j \leq N$, such that $i \neq j$ and $x_i = x_j$).

Indeed, given $x_1, x_2, \ldots, x_N$ as an input for the element distinctness problem, form the intervals $[x_i, x_i], 1 \leq i \leq N$, and find their union. If the number of output intervals is exactly N, the numbers $x_1, x_2, \ldots, x_N$ were all different, and the answer is ‘yes’ to the element distinctness problem, otherwise 'no'. From here it follows that the $\Omega(N \log N)$ lower bound for the element distinctness problem is also a lower bound for determining the union of N intervals.

To devise an optimal algorithm which we will attempt to parallelize, we only need a counter c, initialised $c = 0$. For simplifying the presentation we assume that all the endpoints of the input intervals are disjoint. First we sort the endpoints of the intervals in increasing order, relabel them such that $x_1, x_2, ..., x_{2N}$ is the sorted sequence. Then we scan that sequence starting with x_1, and increment c by 1 if $x_i, 1 \le i \le 2N$, is a left endpoint, decrement c by 1 if x_i is a right endpoint of an input interval. Whenever $c = 1$, we record x_i as the left endpoint of an output interval, and whenever $c = 0$, we record x_i as the right endpoint of an output interval.

It is not hard to demonstrate that whenever $c = 1$, x_i must be the left endpoint, and if $c = 0$, x_i must be the right endpoint of an input interval. Also if x_i is the left endpoint of an input interval and $c > 1$, or if x_i is the right endpoint of an input interval and $c > 0$, x_i must be overlapped by one or more intervals. The running time of the above algorithm is dominated by the sorting step, therefore we can summarise our results as follows.

Lemma 4 The complexity of determining the union of N intervals of the real line is $\Theta(N \log N)$ in the worst case, assuming the algebraic tree model of computation.

Though the proposed algorithm is quite simple, there are two difficulties with its parallelization. First, scanning the sorted list is inherently sequential. Second, even if we know the endpoints of the output intervals, it is not easy to store them in the memory parallely. We will use a linked list such that the elements of the list are stored in an array with mappings *pred* and *succ*, where *pred* provides the element preceding a given element, and *succ* provides the element subsequent to a given element in the list. Then overlapped endpoints are simply removed from the list.

We apply an efficient technique proposed by Kruskal *et al* [54]. The *parallel prefix problem* is to compute all initial prefixes $x_1, x_1 \circ x_2, ..., x_1 \circ x_2 \circ ... \circ x_N$ of N items $x_1, x_2, ..., x_N$, where $\circ$ is an associative binary operation. By the solution of the parallel prefix problem we not only can assign the values of the counter c to the endpoints of the intervals, but also can attach ranks $1, 2, ..., N$ to the elements of a linked list, eg, 1 to the first, 2 to the second element etc, and the elements can be placed in an array by simply using the rank of each element as its index. Then the parallel interval-union algorithm is stated as follows.

```
(1) Sort the endpoints of the intervals in increasing order,
    relabel them, and prepare a doubly-linked list $D$ such
    that $x_1, x_2, ..., x_{2N}$ is the sorted sequence, $pred(x_i) = x_{i-1}$,
    $succ(x_i) = x_{i+1}$, $2 \le i \le 2N-1$, $pred(x_1) =$ nil and
    $succ(x_{2N}) =$ nil;
(2) Assign weights $w_i$ to $x_i$, $1 \le i \le 2N$, such that if $x_i$ is a
    left endpoint, then $w_i = 1$, and if $x_i$ is a right endpoint,
    then $w_i = -1$;
(3) Compute the parallel prefix sum
        $c_i = w_1 + w_2 + ... + w_i$
    for all $x_i$, $1 \le i \le 2N$;
(4) for all $x_j$, $j = 1, 3, ..., 2N-1$, do in parallel
        if (($x_j$ is a left endpoint and $c_j > 1$) or
            ($x_j$ is a right endpoint and $c_j > 0$)) then
                remove $x_j$ from the doubly linked list $D$
        endif
    endfor;
(5) Repeat step (4) for all $x_j$, $j = 2, 4, ..., 2N$, in parallel;
(6) Rank the doubly linked list $D$, and write the endpoints of
    the $M \le N$ output intervals parallely into $2M$ consecutive
    cells of the global memory.
```

The correctness of the above algorithm is based on the same observations as we made for the sequential algorithm, therefore we can state the following.

Theorem 3 The union of N intervals of the real line can be computed in $O(\log N)$ time in the worst case by using N processors, assuming the EREW PRAM model of computation.

Proof: Step (1) can be implemented in $O(\log N)$ time by using N processors under the EREW model [15]. Step (3) and therefore step (6) take $O(\log N)$ time and $N/\log N$ processors assuming the EREW model [54]. Steps (2), (4) and (5) take constant time and N processors. There are no memory conflicts in step (2), and we can avoid memory conflicts by examining and, if necessary, removing first the odd elements of D in step (4), then the even elements in step (5). Therefore the whole algorithm can be implemented in $O(\log N)$ time in the worst case by using N processors, assuming the EREW PRAM model of parallel computation. □

5.2. Hidden-line elimination

As we said earlier, the input to a hidden-line algorithm is a set S of pairwise disjoint ordinary polygons (simple polygons possibly with holes). Let N be the total number of edges, and let u be a viewpoint, $u = (0, 0, -\infty)$. We will adopt the convention that the vertices of the outer boundary of a polygon are given in counter-clockwise order, and the vertices of a hole in clockwise order. Then the interior of a polygon will always lie to the left as its boundary is traversed.

If we wish to achieve a sublinear running time, it follows from the sequential complexity of the problem that we need $\Omega(N)$ processors, which may have memory conflicts while processing the N edges of the input. Therefore, we have to make copies of the input first if we assume an EREW model. We can make use of the following observation.

Lemma 5 The content of any cell of the shared memory can be copied into any block of N consecutive cells in $O(\log N)$ time by using $N/\log N$ EREW PRAM processors.

Let e_i be the image of edge $\mathbf{e}_i$ in the projection plane, and let l_i be the straight line containing e_i, $1 \leq i \leq N$. We can assume without loss of generality that l_i coincides with the x-axis of the coordinate system. Then a parallel hidden-line algorithm can be formulated as follows.

```
(1) Make $N$ copies of the description of each edge $\mathbf{e}_i$, $1 \leq i \leq N$, in $N$ consecutive blocks of memory cells.
(2) **for all** edge $\mathbf{e}_i$, $1 \leq i \leq N$, **do in parallel**
    (2.1) Find the intersection points $x_j$ of $l_i$ with all $e_j$, $1 \leq j \leq N$, $j \neq i$, such that $\mathbf{e}_j$ is nearer to the observer than $\mathbf{e}_i$ at the intersection point $x_j$.
    (2.2) Let $a_j$ and $b_j$ be the endpoints of $e_j$, $1 \leq j \leq N$; $j \neq i$, and let $e_j$ be oriented from $a_j$ to $b_j$. If $a_j$ is above $l_i$, label $x_j$ as a left, otherwise as a right endpoint.
    (2.3) Let $x_l$ be a point of $l_i$ to the left of the leftmost $x_j$, let $x_r$ be a point of $l_i$ to the right of the rightmost $x_j$, $x_a$ be the left endpoint of $e_i$, and $x_b$ be the right endpoint of $e_i$. Label $x_l$ and $x_b$ as left, $x_a$, and $x_r$ as right endpoints.
    (2.4) Determine the union of the intervals specified by the endpoints $x_l$, $x_a$, $x_b$, $x_r$ and $x_j$, $1 \leq j \leq N$; $j \neq i$.
    (2.5) Insert $x_a$ and $x_b$ into the list $L$ obtained as a result of step (2.4). If the insertion of $x_a$ fails, ie, $x_a$ is already in $L$, then the interval $[x_a, succ(x_a))$ is a visible segment of $e_i$, otherwise else $[x_a, succ(x_a)]$ is a hidden interval of $e_i$. Similarly, if the insertion of $x_b$ fails, then the interval $(pred(x_b), x_b]$ is a visible segment of $e_i$, otherwise $[pred(x_b), x_b]$ is a hidden interval of $e_i$. Discard the elements of $L$ left to $x_a$ and those right to $x_b$.
    **end**
```

Using the notation of the algorithm, it is relatively straightforward to demonstrate the following.

Lemma 6 If two consecutive elements x_j and x_k, $j \neq a$, $k \neq b$, of L are a left and a right endpoint respectively, then $[x_j, x_k]$ is a hidden interval. Otherwise if x_j is a right, and x_k is a left endpoint, then (x_j, x_k) is a visible segment of e_i.

Then we can state the main result of this section.

Theorem 4 The hidden-line problem for a set of pairwise disjoint polygons with a total of N edges can be solved in $O(\log N)$ parallel time and $O(N^2)$ space by using N^2 processors, assuming the EREW PRAM model of parallel computation. $\Theta(\log N)$ time is the best possible under both the EREW and the CREW models with arbitrarily many processors.

Proof: Step (1) of the above algorithm can be implemented in $O(\log N)$ time by using $N^2/\log N$ processors according to Lemma 5. Steps (2.1) and (2.2) take constant time and N processors (or $O(\log N)$ time and $N/\log N$ processors). Step (2.3) takes $O(\log N)$ time and $N/\log N$ processors. According to Theorem 3 step (2.4) can be computed in $O(\log N)$ time in the worst case by using N processors. In step (2.5) x_a and x_b can be inserted in L in $O(\log N)$ serial time, discarding the elements of L left to x_a and those right to x_b take $O(\log N)$ time and $N/\log N$ processors by ranking L. Therefore step (2) of the above algorithm can be executed in $O(\log N)$ time for a single edge by using N processors. Using N^2 processors, the algorithm can be executed for N edges within the same time under the EREW model.

It follows from the definition of visibility that finding the maximum of N integers is constant-time reducible to the hidden line problem by using N processors. Cook and Dwork [16] have given an $\Omega(\log N)$ lower bound for finding the maximum of N integers allowing infinitely many processors of a CREW PRAM model. From here the theorem follows. □

While the proposed hidden-line algorithm is optimal in a stronger sense, ie, its running time cannot be further improved, an interesting question arises: would $N^2/\log N$ processors be sufficient to maintain $O(\log N)$ time? The proof of the $O(N^2)$ sequential complexity of the hidden-line problem [20] is based on an optimal algorithm for the arrangement of N lines in the plane. Recently Goodrich [39] proposed an optimal parallel algorithm for constructing line arrangements. Combining Goodrich's result with the techniques presented here and in [20] the question can be answered affirmatively. The resulting algorithm, however, is significantly more complicated than the one presented here.

The theoretical significance of the above results is the demonstration that the exact hidden-line problem is amenable to parallelization. The practical consequence is that the exact hidden-line problem can be solved on real machines in $O(\log^d N)$ time, where d is a small positive constant depending on the particular machine.

6. Distributed-memory parallel algorithms

Unfortunately, the a global shared memory required by the PRAM models is not feasible in practice. A practical approach with the technology available in the foreseeable future is to assign a processor to each row of picture elements of a raster-scan image, in order to compute the image of that particular row [27]. Then the dominant computational problem is the determination of the visibility of a planar set of line segments.

Given a point u and a set S of n opaque line segments in the plane, the planar visibility problem requires to find the line segments or part of the line segments in S visible from u. Recall that in computer graphics the (x,z) Cartesian coordinate system and $u = (0, -\infty)$ are assumed, the visible image is projected into the x-axis, and the algorithms for the solution of the problem are referred to as *scan-line algorithms* [32, 65].

An early algorithm, which takes $\Theta(n^2)$ time in the worst case, was given by Watkins [32, 65]. In practice the *z-buffer algorithm* — an approximation method — is often used, where the image is divided into r equal picture elements. The visibility of each picture element is approximated by the visibility of a sample point, usually taken at the middle of the pixel.

A linear array of size r, called the z-buffer, is maintained. Each element of the array is used to record the z-coordinate of the input line segment nearest to u along the line through the sample point. Then the algorithm simply subdivides the projection of each input line segments into pixels, and updates the appropriate elements of the z-buffer in $\Theta(nr)$ time in the worst case.

The z-buffer algorithm is easy to implement in hardware [60, 51], and the speed of that system is difficult to surpass if there are few overlapping surfaces in the scene. However, in virtual-reality applications the depth of the scene increases; there are multiple overlapping surfaces, and the speed of the z-buffer algorithm quickly deteriorates. One of the objectives of this section is to provide alternatives to the z-buffer algorithm.

If the output for the exact solution is required in a sorted order, it is easy to provide an $\Omega(n \log n)$ lower bound by demonstrating that sorting is reducible to the planar visibility problem [3]. However, this is the time requirement for sorting the output, which is not inherently required. In section 6.1 a proof is provided that the planar visibility problem in itself takes $\Omega(n \log n)$ time even if the output is not required in a sorted order.

6.1. Lower bound

We demonstrate that an $\Omega(n \log n)$ lower bound applies to the exact solution of the planar visibility problem even if the output is not required in a sorted order. In particular, we prove that any algorithm that determines the visibility of n line segments in the plane can be used to solve the element distinctness problem by using $O(n)$ additional operations.

Let us suppose that given an input $x_1, x_2, ..., x_n$ for the element distinctness problem [69], and we are allowed to use any algorithm for determining the visibility of a planar set of line segments. Let $u = (a, b)$ be the observer's position with arbitrary a and $b < 0$. Associate with each x_i, $1 \le i \le n$, the closed line segment $[(x'_i, i), (x'_i, i)]$, where (x'_i, i) is the intersection point of a ray from u through $(x_i, 0)$ with the horizontal line $z = i$. This set of line segments can be constructed in $O(n)$ operations, and $x_i = x_j$ holds for $1 \le i, j \le n$ and $i \ne j$ if and only if one of the points (x'_i, i) and (x'_j, j) is hidden the other, as shown in Figure 5. Then the element distinctness problem can be decided as follows: If the number of visible segments returned by the visibility algorithm is exactly n, the input numbers $x_1, x_2, ..., x_n$ were distinct, otherwise not. $\Omega(n \log n)$ is a lower bound for the element distinctness problem, eg, in the algebraic tree model [69]. It follows that $\Omega(n \log n)$ is a lower bound also for the planar visibility problem in any computational model where $\Omega(n \log n)$ is a lower bound for the element distinctness problem.

The binary-partition-tree method [62] can solve the visibility problem in $O(n \log n)$ expected time after some preprocessing. The *priority-queue method*, proposed earlier by

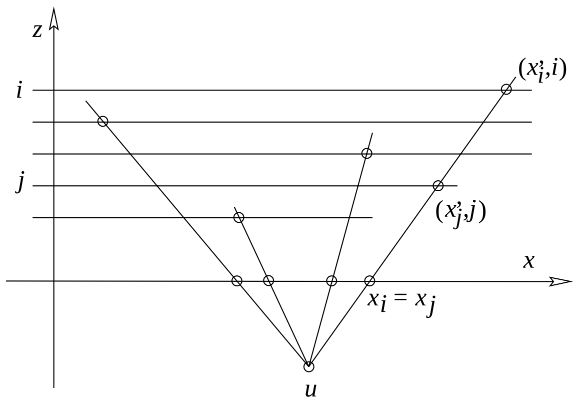

Figure 5: *Element distinctness is reducible to planar visibility*

the author [19], solves the problem in $O(n \log n)$ worst-case time without the need for preprocessing by using a plane-sweep method and maintaining a priority queue of line segments. Another algorithm, based on a divide-and conquer approach and called the *merge* method also achieves the optimal $O(n \log n)$ worst-case time [3].

Early scan-line algorithms [8, 87] take $\Theta(n^2)$ time in the worst case [19], and attempt to exploit coherence [32, 65] which is no longer possible in a parallel environment, where adjacent scan lines may be processed by different processors. Both the priority-queue [19] and the merge [3] methods take $\Theta(n \log n)$ time in the worst case, ie, these methods are best possible in terms of worst-case time, but use sorting, and merging of visible sets of line segments, therefore less appropriate for hardware implementation. The next two sections offer four simple but efficient scan-line algorithms. First two hierarchical methods based on subdivision techniques, then two simple probabilistic algorithms are proposed.

6.2. Hierarchical methods

A scan-line algorithm can be developed by using the ideas proposed by Warnock for determining the visibility of a set of polygons in three-dimensional space [32, 65]. An interval of the x-axis containing the image of the input set will be called the *window*. Warnock's basic idea is to attempt to display the image if it is simple, otherwise subdivide the window until the image is simple enough. Each input line segment t can be classified according to the relation of its image τ to the current window W as follows:

- *contained* by the window, ie, $\tau \subset W$, where '$\subset$' denotes the proper subset relation,
- *totally overlapping* the window, ie, $W \subseteq \tau$,
- *disjoint* from the window, ie, $\tau \cap W = \emptyset$ and
- *intersecting* the window, ie, $\tau \cap W \ne \emptyset \wedge W \not\subseteq \tau \wedge \tau \not\subset W$.

Let S be the set of input line segments, and b a background line segment. Initially W represents the scan line consisting of r pixels. Then a set T is constructed such that T contains all line segments to be processed in the current window. An image is regarded to be simple if only the background segment is visible, ie, $T = \emptyset$, or each element of T is hidden by a single line segment. If the image is not simple, W is subdivided into two equal sub-windows, and the procedure is applied recursively to process T in both sub-windows. The planar visibility algorithm based Warnock's ideas is formally stated as Figure 6.

With reference to the notation of Figure 6, it should be noted that T will always be empty when window W is equal to a pixel, as the image of each segment in T will overlap W, and will be put on list B. Therefore the termination of the algorithm can be assured by testing if T is empty.

The recursive subdivision of the scan-line can be represented by a binary tree of r leaf nodes and $r-1$ internal nodes, both the internal and the leaf nodes corresponding to a window. Then the total number of windows is $2r-1$. There are $O(n)$ operations in any window, therefore an upper bound on the worst-case time of the planar Warnock method is $O(nr)$. As a copy of each element of the input set may be made in each window, the same $O(nr)$ upper bound applies to its space requirement in the worst case.

The actual time and space requirements must be better, since short line segments will not be processed in all windows, and on the other hand if there are several long line segments, all but one will become hidden and removed from T at a certain level of subdivision. To be able to give sharper bounds we develop another algorithm.

The planar Warnock method does not subdivide the set S of line segments, only the window, therefore each line segment should be tested against another sub-window at the same level, even if the line segment is already outside that sub-window. These extra tests are avoided by the second algorithm we propose, called the *subdivision method*, which is similar to the Warnock method, except that the set S is always subdivided according to the window boundaries. The subdivision method is presented as Figure 7.

Note that, though S is subdivided into S_1 and S_2 with the subdivision method, the line segments overlapping the window boundary are not subdivided for efficiency reasons; a copy of these line segments appears both in S_1 and S_2. However, for an upper bound on the running time of the subdivision method we can assume that all the line segments are subdivided at the window boundaries. The crucial observation is that the windows, where a particular line segment t is processed, correspond to the external and internal nodes of a segment tree representing the line segment t. Hence any line segment t could be represented by less than $2\log_2 r$ standard intervals of a segment tree [69], though with the subdivision method t is represented by at most that many copies of itself. From here it follows that an upper bound on the worst-case time is $O(n \log r)$.

An input set can be created such that each segment is processed in $\Theta(\log r)$ sub-windows, therefore the worst-case time requirement is $\Theta(n \log r)$. For similar considerations the worst-case space requirement is also $\Theta(n \log r)$. The slightly more sophisticated *z-tree method* [23] reduces the space requirement to $\Theta(n+r)$ while retaining the $\Theta(n \log r)$ worst-case time bound.

Now let us return to the analysis of the planar Warnock method. First we observe that both methods make the same window subdivisions. Indeed, if a window W is subdivided by the subdivision method because of a line segment t, ie, $\tau \cap W \neq \emptyset$ and $W \not\subseteq \tau$, the same window W will also be subdivided by the Warnock method.

Now the difference between the two methods is in the termination of the processing of the line segments. With the subdivision method the processing of any line segment t can be terminated in two ways: either t is discarded as hidden by another line segment, or t is displayed as a background. With the Warnock method there is also a third way: t is discarded as disjoint from the window. While the first two types of termination can happen several times to the same line segment, ie, the parts of the same line segment will become hidden in different windows, and also several parts of a line segment can be displayed as a background. The third type of termination, however, can only happen once to each line segment; when it becomes disjoint to a window W, and it will never again be considered in the sub-windows of W. Thus the processing of the line segments terminates by only $O(n)$ additional operations, therefore the Warnock method must take $\Theta(n \log r)$ time and space in the worst case.

6.3. Probabilistic Algorithms

We propose two probabilistic algorithms with the advantages of simplicity and good expected running time. The first method, called *random*, is stated as follows. If the input set S of line segments is empty, display the background b and return. Otherwise choose a line segment t at random. Discard all line segments and parts of line segments hidden by t. Subdivide the remaining line segments if necessary along the vertical lines through the endpoints of t. Let S_1, S_2 and S_3 be the set of line segments left and right to t and in front of t, respectively. Apply the procedure recursively for S_1, S_2 and S_3, with the background b for S_1 and S_2, and use t as the background for S_3.

The second algorithm, called the *trapezoid method*, randomly selects k or $|S|$, whichever is the smaller, line segments, where k is a constant, and $|S|$ is the number of line segments in the input set S. Then the algorithm chooses a segment a with the largest area trapezoid $T(a)$ formed by a, the background b and the vertical lines through the endpoints of a. All line segments and parts of line segments hidden by

```
Warnock($S, W, b$)
    initialise $B$ and $T$ as empty lists of line segments;
    for each segment $t \in S$ do
        let $\tau$ be the projection of $t$ into the $x$-axis;
        if $\tau \cap W \neq \emptyset$ then
            if $W \subseteq \tau$ then copy $t$ to list $B$ else copy $t$ to list $T$ endif
        endif
    endfor;
    if $B \neq \emptyset$ then
        find $a \in B$ as the segment nearest to $u$;
        for each segment $t \in T$ do
            if $t$ is hidden by $a$ then remove $t$ from $T$ endif
        endfor;
        substitute $a$ for $b$, and discard list $B$
    endif;
    if $T = \emptyset$ then display $b$ in $W$
    else
        subdivide $W$ into two sub-windows $W_1$ and $W_2$ of equal size;
        Warnock($T, W_1, b$); Warnock($T, W_2, b$)
    endif
end
```

Figure 6: *The two-dimensional variant of Warnock's method*

```
subdiv($S, W, b$)
    initialise $B$ as an empty list of line segments;
    for each segment $t \in S$ do
        let $\tau$ be the projection of $t$ into the $x$-axis;
        if $W \subseteq \tau$ then relocate $t$ from $S$ to list $B$ endif
    endfor;
    if $B \neq \emptyset$ then
        find $a \in B$ as the segment nearest to $u$;
        for each segment $t \in S$ do
            if $t$ is hidden by $a$ then remove $t$ from $S$ endif
        endfor;
        substitute $a$ for $b$, and discard list $B$
    endif;
    if $S = \emptyset$ then display $b$ in $W$
    else
        subdivide $W$ into two sub-windows $W_1$ and $W_2$ of equal size;
        subdivide $S$ into $S_1 = \{t \in S \mid \tau \cap W_1 \neq \emptyset\}$ and $S_2 = \{t \in S \mid \tau \cap W_2 \neq \emptyset\}$;
        subdiv($S_1, W_1, b$); subdiv($S_2, W_2, b$)
    endif
end
```

Figure 7: *The subdivision method*

a are discarded. In other words, the part of the scene within $T(a)$ is removed. Let S_1, S_2 and S_3 be as with the random method, and the procedure is applied recursively for S_1, S_2 and S_3. The trapezoid method is stated formally as Figure 8. The best value of k is determined experimentally.

The random and the trapezoid methods never give wrong results, but may exhibit different running times if applied twice for the same input. Recall that this type of algorithms are classified as Las-Vegas algorithms.

For establishing an upper bound on the worst-case running time of the random and the trapezoid methods, consider a set S of n line segments with non-overlapping images. If always the leftmost or the rightmost line segment is selected for partitioning S into subsets, the running time is $\Omega(n^2)$. On

```
trapezoid($S, W, b$)
    if $S = \emptyset$ then display $b$ in $W$
    else
        for $\min(k, |S|)$ line segments chosen at random from $S$ do
            find segment $a$ with the maximum-area trapezoid $T(a)$;
        endfor;
        subdivide $W$ into sub-windows $W_1$, $W_2$ and $W_3$ such that $W_1$ is to the left,
        $W_2$ is to right of $T(a)$, and $W_3$ is the projection of $a$ into the $x$-axis;
        for each segment $t \in S - \{a\}$ do
            let $\tau$ be the projection of $t$ into the $x$-axis;
            if $\tau$ is to the left of $W_3$ then put $t$ into $S_1$
            else if $\tau$ is to the right of $W_3$ then put $t$ into $S_2$ endif
            else
                cut off the part of $t$ visible in $W_1$ ($W_2$) if any, and put it into $S_1$ ($S_2$);
                if $t$ is in front of $a$ then put $t$ into $S_3$ endif
            endif
        endfor;
        trapezoid($S_1, W_1, b_1$); trapezoid($S_2, W_2, b_2$); trapezoid($S_3, W_3, a$)
    endif
end
```

Figure 8: *The trapezoid method*

the other hand, both methods compare at most n line segments in at most $2n-1$ vertical strips, which takes $O(n^2)$ time, therefore their worst-case time requirement is $\Theta(n^2)$. There can be at least $n/2$ line segments broken into at least $n/2$ parts in the worst case, but there are at most n parts in any of the $2n-1$ vertical strips. This gives a $\Theta(n^2)$ bound on the space requirement in the worst case.

method	time	space
z-buffer [32, 65]	$\Theta(nr)$	$\Theta(n+r)$
Watkins [32, 65]	$\Theta(n^2)$	$\Theta(n)$
priority-queue [19]	$\Theta(n \log n)$	$\Theta(n)$
merge [3]	$\Theta(n \log n)$	$\Theta(n)$
z-tree [23]	$\Theta(n \log r)$	$\Theta(n+r)$
Warnock	$\Theta(n \log r)$	$\Theta(n \log r)$
recursive subdivision	$\Theta(n \log r)$	$\Theta(n \log r)$
random	$\Theta(n^2)$	$\Theta(n^2)$
trapezoid	$\Theta(n^2)$	$\Theta(n^2)$

Table 2: *Worst-case time and space requirements of scan-line algorithms*

Table 2 summarises the scan-line methods investigated, together with the worst-case time and space requirements of the particular method, where n is the number of line segments and r is the number of pixels in the scan line. The average running time can be significantly better for some of the algorithms, and indeed, expected-time analyses of probabilistic algorithms attracted much attention recently [11, 12, 13, 62, 63]. Asymptotic analysis, however, cannot take into consideration constant factors, which can also be different in different environments.

Constant factors can be determined from time measurements in the actual machine environment where the algorithms are to be used. Kremer-Patard [53] used manually generated input to compare some algorithms including the priority-queue method [19]. Another possibility is to extract the input data from real three-dimensional models. Considering the number of possible algorithms together with their variants, the number of time measurements required for conclusive results would be too high, and this method would be too expensive and time consuming. Since the input to a scan-line algorithm is only a planar set of line segments, more efficient test-data generation methods can be developed.

7. Experimental Performance Evaluation

As some algorithms exploit the fact that the input produced by a solid modeler results in a set of non-intersecting line segments, a test-data generation method is required to produce a set of non-intersecting random line segments in the plane. The first idea that would probably spring to mind is to divide an r by r square into m rows and m columns such that $m = \sqrt{r}$, and then choose the endpoints of a line segment at random from each square. This would be a naive approach, however, as it would result in a very sparse scene, ie, relatively short line segments with a lot of empty space among them.

It can be demonstrated that the average distance of two points chosen uniformly and independently at random from the [0,1] interval is 1/3. Therefore, in the one-dimensional case, we would obtain a set of line segments with an average length of gaps of twice the average length of the line segments. Practical scenes are more dense. The reader is encouraged to look at Figure 10 before reading any further, and have a guess how the set of line segments was generated.

The technique we propose is based on a channel-assignment algorithm [42] and can be used to generate x-coordinates of the line segments from arbitrary probability distributions. First the x-coordinates of the left and right endpoints of the line segments are generated, which determine an interval for each line segment. Then we find a minimal partition of this set of intervals into subsets of intervals such that no subset contains overlapping intervals. The subsets will be called *channels*, and each channel will correspond to a horizontal band within an r by r rectangle. As the x-intervals corresponding to the line segments are pairwise disjoint within a channel, we can assign arbitrary z-coordinates to the line segments within the same channel. The algorithm is given in Figure 9, and a sample output in Figure 10.

Step 1 of the algorithm requires $2n$ random numbers, and takes $O(n)$ time. Step 2, dominated by the sorting, can be implemented in $O(n \log n)$ time. Finally step 3 requires $2n$ random numbers and $O(n)$ time. Therefore the proposed method takes, in total, $4n$ random numbers and $O(n \log n)$ time in the worst case to generate $4n$ coordinates for a set of n non-intersecting random line segments.

If $i < j$, the leftmost x-coordinate in channel j cannot be less than the leftmost x-coordinate in channel i. Therefore the top left corner of the r by r rectangle may appear empty. This problem can be eliminated by a random permutation of the channels before step 3. A C-language implementation of the method has been incorporated in a testbed for the performance evaluation of scan-line algorithms.

8. Conclusions

The computational requirements of CAD and virtual-reality systems are often underestimated, and this is the main reason for the inadequate usability and performance of these systems. We have demonstrated that, contrary to the prevailing theoretical background, visibility computations are a bottleneck. Most polygon-based, exact algorithms recommended by the literature have $\Theta(N^3)$ worst-case time, and also their expected running time is $\Theta(N^2)$ even if the total number of edge intersections is $\Theta(N)$. If the expected number of intersections is $O(N)$, the expected running time of the algorithm we recommended is $O(N \log N)$, regardless of the underlying probability distribution of the input data.

We have also demonstrated that most approximation algorithms, including the z-buffer algorithm, take $\Theta(K^2 N)$ time in the worst case, which is actually the same as the time requirement of a brute-force method. We recommended the use of hierarchical data structures, called z-trees, that can reduce the worst-case time of approximation algorithms to $\Theta(KN)$. With a typical resolution of $K = 1024$, this is a promising approach.

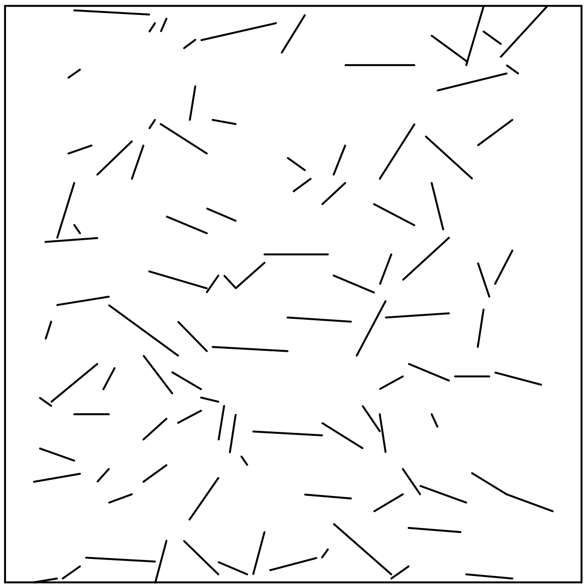
Figure 10: *Non-intersecting random line segments*

The third possibility to speed up visibility computations is parallel processing. Approximation algorithms are inherently appropriate for parallel implementation, but using inefficient algorithms, the power of parallel machines is merely absorbed by compensating the poor performance of the algorithm. The most promising directions for future research are in the applications of approximation and parallel techniques.

Contemporary VR research is driven by applications. Unfortunately, some of the promises of VR applications hyped by the media are unrealistic. These circumstances may do harm to both research and industry. For example, artificial-intelligence research suffered when it failed to live up the hyped promises. Therefore it is very important that VR applications deliver at least the realistic promises, and a solid theoretical background is definitely helpful for this purpose.

References

1. Aho, A. V., Hopcroft, J. E. and Ullman, J. D. *Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass. 1975.

2. Aho, A. V., Hopcroft, J. E. and Ullman, J. D. *Data Structures and Algorithms.* Addison-Wesley, Reading, Mass. 1983.

3. Atallah, M. J., Cole R., Goodrich M. T. Cascading divide-and-conquer — a technique for designing parallel algorithms. *SIAM Journal on Computing* **18**,3 (1989) 499–532.

```
1)  Generate pairs of $x$-coordinates $(a_i, b_i)$, $1 \le i \le n$, for $n$ line segments;
2)  Sort the $2n$ $x$-coordinates $\{x_j\} = \{a_i\} \cup \{b_i\}$ in non-decreasing order, such that
    $x_1, x_2, ..., x_i, ..., x_{2n}$ is the sorted sequence;  Initialize a stack as empty;
    for $1 \le i \le 2n$ do
        if $x_i$ is the left endpoint of a line segment $s$ then
            if the stack is empty then
                allocate a new channel $c$
            else
                get a free channel $c$ from the stack
            endif;
            put segment $s$ into channel $c$
        else
            push the channel containing the segment with the right
            endpoint $x_i$ into the stack as a free channel
        endif
    endfor;
3)  Generate pairs of $z$-coordinates for each line segment within each channel.
```

Figure 9: *Test data generation*

4. Balch, D. C., Tichenor, J. M. Telemedicine expanding the scope of health-care information. *Journal of the American Medical Informatics Association* **4**,1 (1997) 1–5.

5. Bayarri, S., Fernandez, M., Perez, M. Virtual reality for driving simulation. *Communications of the ACM* **39**,5 (1996) 72–76.

6. Ben-Or, M. Lower bounds for algebraic computation trees. Proc. *15th ACM Annual Symp. on Theory of Computing* (Apr. 1983) 80–86.

7. Bentley, J. L., Ottmann, T. Algorithms for reporting and counting geometric intersections. *IEEE Trans. Comput.* **C-28** (Sep. 1979) 643–647.

8. Bouknight, W. J. A procedure for generation of three-dimensional half-toned computer graphics presentations. *Comm. ACM* **13**,9 (Sep. 1970) 527–536.

9. Bryson, S. Virtual reality in scientific visualization. *Communications of the ACM* **39**,5 (1996) 62–71.

10. Chazelle, B., Edelsbrunner H. An optimal algorithm for intersecting line segments in the plane. *Journal of the Association for Computing Machinery* **39**,1 (Jan. 1992) 1–54.

11. Clarkson, K. L., Shor, P. W. Applications of random sampling in computational geometry II. *Discrete and Computational Geometry* **4**,1 1989, 387–421.

12. Clarkson, K. L. Randomized geometric algorithms. *Computers and Euclidean Geometry* 1992.

13. Clarkson, K. L., Cole, R., Tarjan, R. E. Randomized parallel algorithms for trapezoidal diagrams. *Int. J. Comp. Geom. and Applications* 1992, 117–133.

14. Cobb, S. V. G., Dcruz, M. D., Wilson, J. R. Integrated manufacture — a role for virtual-reality. *International Journal of Industrial Ergonomics* **16**,4–6 (1995) 411–425.

15. Cole, R. Parallel merge sort. *SIAM J. Computing* **17**,4 (Aug. 1988) 770–785.

16. Cook, S., Dwork, C. Bounds on the time for parallel RAMs to compute simple functions. Proc. *14th ACM Symp. on Theory of Computing*, San Francisco, California, (May, 1982) 231–233.

17. Coppen, D., Hawes, D., Slater, M., Davison, A. Distributed frame buffer for rapid dynamic changes to 3D scenes. *Computers & Graphics* **19**,2 (1995) 247–250.

18. de Berg, M. Ray shooting, depth orders and hidden-surface removal. *Lecture Notes in Computer Science* **703**, Springer Verlag, Berlin, 1993, 201 pp.

19. Dévai, F. Complexity of two-dimensional visibility computations. Proc. *3rd European Conference on CAD/CAM and Computer Graphics*, Paris, France, Feb. 1984, MICAD'84 Vol. **3**, 827–841.

20. Dévai, F. Quadratic bounds for hidden-line elimination. Proc. *Second Annual ACM Symposium on Computational Geometry*, Yorktown Heights, New York, USA, June 2–4, 1986, 269–275.

21. Dévai, F. An intersection-sensitive hidden-surface algorithm. Proc. EUROGRAPHICS'87, Maréchal, G. (Ed.) Amsterdam, the Netherlands (Aug. 24–28, 1987) 495–502.

22. Dévai, F. An $O(\log N)$ parallel time exact hidden-line algorithm. In: Kuijk, A. A. M., Strasser, W. (Eds)

Advances in Graphics Hardware II, Springer-Verlag, Berlin, Germany, 1988, 65–73.

23. Dévai, F. Approximation algorithms for high-resolution display. Proc. *PIXIM'88, 1st International Conference on Computer Graphics in Paris*, Péroche, B. (Ed) France, Oct. 24–28, 1988, 121–130.

24. Dévai, F. *Computational Geometry and Image Synthesis*. Lecture notes for Course 2, PIXIM'89, 2nd International Conference on Computer Graphics in Paris, France, Sept. 25–29, 1989, 88 pp.

25. Dévai, F. An optimal parallel algorithm for the visualisation of solid models. In: *Applications of Supercomputers in Engineering III,* Elsevier Applied Science, London, 1993, 199–210.

26. Dévai, F. On the complexity of some geometric intersection problems. *Journal of Computing and Information* **1**,1 (May 1995) 333–352.

27. Dévai, F. Scan-line methods for parallel rendering. In: Chen, M., Townsend, P., Vince J. A. (Eds) *High-Performance Computing for Computer Graphics and Visualisation*. Springer Verlag, London, 1996, 88–98.

28. Dobashi, Y., Kaneda, K., Nakatani, H., Yamashita, H., Nishita, T. A quick rendering method using basis functions for interactive lighting-design. *Computer Graphics Forum* **14**,3 (1995) c229.

29. Firebaugh, M. W. *Computer Graphics. Tools for Visualization*. Wm. C. Brown Publishers, Oxford, UK, 1993, 547 pp.

30. Fiume, E. L. *The Mathematical Structure of Raster Graphics*. Academic Press, San Diego, 1989.

31. Foley, J. D., van Dam, A. *Fundamentals of Interactive Computer Graphics*. Addison-Wesley, Reading, Mass., 1982. 664 pp.

32. Foley, J. D., van Dam, A., Feiner, S. K., Hughes, J. F. *Computer Graphics: Principles and Practice*. (Second Edition) Addison-Wesley, Reading, Mass., 1990. 1174 pp.

33. Foley, J. D. et al. *Introduction to Computer Graphics*. Addison-Wesley, Reading, Mass., 1994, 557 pp

34. Franklin, W. R. A linear time exact hidden surface algorithm. *Computer Graphics* **14**,3 (1980) 117–123.

35. Fredman, M. L., Weide, B. On the complexity of computing the measure of $\cup[a_i, b_i]$. *Comm. ACM* **21**,7 (July 1978) 540–544.

36. Fuchs, H. et al. Pixel-planes 5: A heterogeneous multiprocessor graphics system using processor enhanced memories. Proc. SIGGRAPH 89, 79–88.

37. Galimberti, R. Montanari, U. An algorithm for hidden-line elimination. *Comm. ACM* **12**,4 (Apr. 1969) 206–211.

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

39. Goodrich, M. T. Constructing arrangements optimally in parallel. *Discrete & Computational Geometry* **9**,4 (1993) 371–385.

40. Greenberg, D. P. *Global Illumination: The Radiosity Approach*. Lecture notes for Course 14, PIXIM'89, 2nd International Conference on Computer Graphics in Paris, France, Sept. 25–29, 1989, 56 pp.

41. Greene, N., Kass, M., Miller, G. Hierarchical z-buffer visibility. Proc. SIGGRAPH 93, Anaheim, California, August 1993, 231–238.

42. Gupta, U. I., Lee, D. T., Leung, J. Y.-T. An optimal solution for the channel-assignment problem. *IEEE Trans. Comput.* **C-28**,11 (1979) 807–810.

43. He, T. S., Hong, L. C., Kaufman, A., Varshney, A., Wang, S. Voxel based object simplification. Proc. *Visualization*'95 (1995) 296–303.

44. Higgins, G. A., Meglan, D. A., Raju, R., Merril, J. R., Merril, G. L. Teleos(TM): Development of a software toolkit for authoring virtual medical environments. *Presence — Teleoperators and Virtual Environments* **6**,2 (1997) 241–252.

45. Hornung, C. An approach to a calculation-minimized hidden line algorithm. *Comput. & Graphics* **6**,3 (1982) 121–126.

46. Hornung, C. A method for solving the visibility problem. *IEEE Comput. Graphics & Appl.* **4**,7 (July 1984) 26–33.

47. Hubbold, R., Murta, A. West, A. Howard, T. Design issues for virtual reality systems. In: Göbel, M. (Ed) *Virtual Environments* '95, Springer Verlag, Wien, 1995, 224–235.

48. Jalili, R., Kirchner, P. D., Montoya, J., Duncan, S., Genevriez, L., Lipscomb, J. S., Wolfe, R. H., Codella, C. F. A visit to the Dresden Frauenkirche. *Presence — Teleoperators and Virtual Environments* **5**,1 (1995) 87–94.

49. Kedem, G., Ellis, J. L. The raycasting machine. Proc. *1984 Int. Conf. on Computer Design*, October 1984, 533–538.

50. Knittel, G. A scalable architecture for volume rendering. *Computers & Graphics* **19**,5 (1995) 653–665.

51. Knittel, G., Schilling, A., Strasser, W. GRAMMY: High performance graphics using graphics memories. In:

Chen, M., Townsend, P., Vince J. A. (Eds) *High-Performance Computing for Computer Graphics and Visualisation*. Springer Verlag, London, 1996, 33–48.

52. Knuth, D. E. *The Art of Computer Programming*, Vol. 3: *Sorting and Searching*. Addison-Wesley, Reading, Mass., 1973.

53. Kremer-Patard, G. Evaluation d'algorithmes de calcul de la visibilité d'un ensemble de segments du plan. *Revue de CFAO et d'Infographie* **3**,3 (1988) 39–57.

54. Kruskal, C. P., Rudolph, L., Snir, M. Efficient parallel algorithms for graph problems. *Algorithmica* **5** (1990) 43–64.

55. Laszlo, M. J. *Computational Geometry and Computer Graphics in C++*. Prentice Hall, Upper Saddle River, USA, 1996, 266 pp.

56. Loftin, R. B., Kenney, P. J. Training the Hubble Space Telescope flight team. *IEEE Comput. Graphics & Appl.* **15**,5 (1995) 31–37.

57. Loutrel, P. P. A solution to the hidden-line problem for computer drawn polyhedra. *IEEE Trans. Comp.* **C–19**,3 (Mar. 1970) 205–213.

58. Márton, G. *Investigation of the Average Complexity of Ray-Tracing Algorithms*. Ph.D Thesis, Budapest, 1995 (In Hungarian).

59. McKenna, M. Worst-case optimal hidden-surface removal. *ACM Transactions on Graphics* **6**,1 (Jan. 1987) 19–28.

60. Molnar, S., Eyles, J., and Poulton, J. PixelFlow: High-speed rendering using image composition. *Computer Graphics* **26**,2 (Proc. SIGGRAPH 92, July 1992) 231–240.

61. Molnar, S., Cox, M., Ellsworth, D., Fuchs, H. A sorting classification of parallel rendering. *IEEE Comput. Graphics & Appl.* **14**,4 (1994) 23–32.

62. Motwani, R., Raghavan, P. *Randomized Algorithms*. Cambridge University Press, Cambridge, UK, 1995, 476 pp.

63. Mulmuley, K. An efficient algorithm for hidden surface removal 2. *J. Computer and System Sciences* **49**,3 (1994) 427–453.

64. Myers, E. W. An $O(E \log E + I)$ expected time algorithm for the planar segment intersection problem. *SIAM J. Comput.* **14**,3 (Aug. 1985) 625–637.

65. Newman, W. M., Sproull, R. F. *Principles of Interactive Computer Graphics*. (Second Edition) McGraw-Hill Kogakusha Ltd, Tokyo, Japan, 1979, 541 pp.

66. O'Rourke, J. The computational geometry column. *Computer Graphics* **20**,5 (1986) 232–234.

67. O'Rourke, J. *Computational Geometry in C*. Cambridge University Press, Cambridge, UK, 1994, 368 pp.

68. Pili, P. A parallel raycast algorithm of CSG models on CM2. *International J. Modern Physics C-physics & Computers* **4**,1 (1993) 29–40.

69. Preparata, F. P., Shamos, M. I. *Computational Geometry. An Introduction*. Springer-Verlag, Berlin, 1985, 390 pp.

70. Reed, D. A, Shields, K. A, Scullin, W. H, Tavera, L. F, Elford, C. L. Virtual-reality and parallel systems performance analysis. *IEEE Computer* **28**,11 (1995) 57–67.

71. Rimmek, K. Flight simulation, an advanced application of virtual-reality. *IFIP Transactions A — Computer Science and Technology* **53** (1994) 171–176.

72. Schaufler, G., Stürzlinger, W. Generating multiple levels of detail from polygonal geometry models. In: Göbel, M. (Ed) *Virtual Environments* '95, Springer Verlag, Wien, 1995, 33–41.

73. Schaufler, G., Stürzlinger, W. A 3-dimensional image cache for virtual-reality *Computer Graphics Forum* **15**,3 (1996) c227.

74. Schmitt, A. Time and space bounds for hidden line and hidden surface algorithms. Proc. EUROGRAPHICS'81, Darmstadt, Germany, (Sep. 1981) 43–56.

75. Sedgewick, R. *Algorithms in C++*. Addison-Wesley, Reading, Mass., 1992, 658 pp.

76. Slater, M., Usoh, M. Simulating peripheral vision in immersive virtual environments. *Computers & Graphics* **17**,6 (1993) 643–653.

77. Sudarsky, O., Gotsman, C. Output-sensitive visibility algorithms for dynamic scenes with applications to virtual-reality. *Computer Graphics Forum* **15**,3 (1996) c249–c258.

78. Sutcliffe, A. G. *Human-Computer Interface Design*. Macmillan Press Ltd, 1988, Second edition 1955, 326 pp.

79. Sutherland, I. E., Sproull, R. F., Schumaker, R. A. A characterization of ten hidden-surface algorithms. *Computing Surveys* **6**,1 (March 1974) 1–55.

80. Teller, S. J., Sequin, C. H. Visibility preprocessing for interactive walktroughs. Proc. SIGGRAPH 91, 1991, 61–69.

81. Teller, S. J., Hanrahan, P. Global visibility algorithms for illumination computations. Proc. SIGGRAPH 93, Anaheim, California, 1993, 239–246.

82. Terashima, N. Telesensation — distributed interactive virtual reality — overview and prospects. *IFIP Transactions A — Computer Science and Technology* **51**, (1994) 49–59.

83. Usoh, M., Slater, M. An exploration of immersive virtual environments. *Endeavour* **19**,1 (1995) 34–38.

84. Watt, A. *Fundamentals of Three-Dimensional Computer Graphics*. Addison-Wesley, Wokingham, UK, 1989.

85. Watt, A., Watt, M. *Advanced Animation and Rendering Techniques. Theory and Practice*. Addison-Wesley, Wokingham, England, 1992, 455 pp.

86. Wloka, M. M. Lag in multiprocessor virtual-reality. *Presence — Teleoperators and Virtual Environments* **4**,1 (1995) 50–63.

87. Wylie, C., Romney, G. W., Evans, D. C., Erdahl, A. C. Halftone perspective drawings by computer. Proc. *Fall Joint Computer Conference 1967*, Thompson Books, Washington DC, 1967, 49–58.

88. Yagel, R., Ray, W. Visibility computation for efficient walkthrough of complex environments. *Presence — Teleoperators and Virtual Environments* **5**,1 (1995) 45–60.

89. Zobel, R. W. The representation of experience in architectural design. *Presence — Teleoperators and Virtual Environments* **4**,3 (1995) 254–266.