COMBINATORICS OF HIDDEN SURFACE ALGORITHMS

A Thesis presented

by

William Randolph Franklin

to

The Division of Applied Sciences

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

in the subject of

Applied Mathematics

Harvard University

Cambridge, Massachusetts

May, 1978

PREFACE

I wish to thank Harry Lewis, my advisor, for forcing me to think and write logically and rigourously and encouraging me to finish. Tom Cheatham and Christos Papadimitriou, my other readers, also provided valuable suggestions. I wish to thank the members of the Lab for Computer Graphics and Spatial Analysis in the Graduate School of Design at Harvard, particularly Nicholas Chrisman and Denis White, where I obtained valuable ideas and support while implementing some of the algorithms described in this thesis. Allan Schmidt and Eric Teicholz provided encouragement and support during the years it took to get useful results. This thesis was partly supported by work at the Lab for Computer Graphics implementing the results.

He had bought a large map representing the sea,
Without the least vestige of land:
And the crew were much pleased when they found it to be
A map they could all understand.

"What´s the use of Mercator´s North Poles and Equators,
Tropics, Zones and Meridian Lines?"
So the Bellman would cry and the crew would reply,
"They are merely conventional signs!.

"Other maps are such shapes with their islands and capes!
But we´ve got out brave Captain to thank"
(So the crew would protest) "that he´s brought us the best ---
A perfect and absolute blank!"

Fit the Second,
The Hunting of the Snark,
Lewis Carroll

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF FIGURES

NOTATION

1. $f(n) = \text{Omega}(g(n))$ means that there exists a positive c such that there exists n_0 such that $n>n_0 \Rightarrow f(n) \geq c\, g(n)$.

2. $f(n) = O(g(n))$ means that there exists a positive c such that there exists n_0 such that $n>n_0 \Rightarrow f(n) \leq c\, g(n)$.

3. $f(n) = \Theta(g(n))$ means that $f(n) = O(g(n))$ and $f(n) = \text{Omega}(g(n))$.

4. In the thesis, variables may be lowercase or uppercase. Lowercase variables are always one letter long and so juxtaposition indicaes multiplication. Uppercase variable names can be several letters long and so are separated by '*' when multiplied. Notwithstanding the above point, a few common multiletter lowercase variable and function names such as 'pi' and 'log' are allowed. There should be no confusion.

5. 'log' indicates logarithm to the base 2 and 'ln' indicates logarithm to the base 2.71828...

6. $\doteq$ means approximatly equal.

SYNOPSIS

This thesis analyzes various aspects of object space hidden surface algorithms *. It builds on the classic work of Sutherland, Sproull and Schumacker [1974b]. Three new algorithms are presented and analyzed in order to exhibit the implications of two general principles. The first, which is new, is the concept of a _variable grid_. This is a grid superimposed on the screen. As the scene gets more complex, the grid gets finer.

---

* An object space hidden surface algorithm calculates the visible and hidden surfaces accurately to the arithmetic precision of the machine. This is opposed to image space algorithms which calculate the plot only to the precision of the display device. Object space algorithms tend to be suited for vector plotters and image space algorithms for raster display devices.

CHAPTER 1

INTRODUCTION

1.1 THE HIDDEN SURFACE PROBLEM

This thesis analyzes several aspects of the combinatorics of the hidden surface problem in computer graphics. Among the aspects considered are 1) The designing of a faster object space algorithm, 2) analysis of the efficiencies obtainable by restricting the input data to special cases, 3) shading of 3-D schematic plots to enhance comprehensibility, and 4) analysis of the problem of efficient conversion of vector plotter plots for a raster plotter.

Chapter 2 summarizes the hidden surface problem which is about 15 years old and gives its history.

Existing hidden surface algorithms have been divided into image space algorithms and object space algorithms by Sutherland, Sproull & Schumacker [1974b]. The image space methods work more directly on the plotter screen (in "image space") while the object space methods work more on the objects in the scene (in "object space"). Consequently the image space algorithms calculate the plot only to the resolution of the display device while the object space ones generally calculate it exactly (up to the arithmetic precision of the machine performing the computations). Since the image space algorithms produce less information, they are generally considered to be inherently faster. In fact all the object space algorithms that have been discovered so far exhibit very unfavourable asymptotic growth in time while the image space algorithms that have been discovered are far faster.

One of the results of this thesis is to refute the hypothesis that object space algorithms necessarily perform so slowly. Chapter 3 presents an object space algorithm that on scenes containing X intersections of the projected edges takes time O(X) to calculate the hidden surfaces, provided that X grows at least as fast as log(N), where N is the number of edges. Previous object space algorithms grew faster than $O(N^2)$. Thus the fact that object space algorithms calculate more information does not require them to run as slowly as the known algorithms do. I have implemented part of this algorithm in a 12000 line Fortran program, VIEWPLOT. A program summary, user's manual and detailed program logic manual are included as Appendix A of this thesis.

The hidden surface problem cannot be given a simple, crisp, mathematical definition in such a way that algorithms for it can be stated cleanly and elegantly. The problem has many aspects which interact in complex ways. Implementation details always threaten to outrun victories won by theoretically advantageous methods. The field of applications is so broad that it is useful to consider specialized subproblems. This not only allows these special cases to be solved by more efficient algorithms but gives a better feel for the general case. Indeed for every hidden surface algorithm, there is a notion of what is the reasonable or proper input on which it would execute fastest. Each also has possible inputs on which it would execute quite slowly. I consider two important special cases in this thesis: prism maps in geography and ball models of molecules in chemistry.

1.2 THE PRISM-MAP SPECIAL CASE

Geographers are in the process of gradually computerizing cartography. The first step is to digitize a map. An operator pins the map on the bed of a digitizer and then moves a handheld cursor, tracing along all the map's lines. Meanwhile the digitizer is continuously recording the cursor's position by writing its coordinates to a magnetic tape at intervals determined by some criterion. Possible criteria to record a point are whenever the cursor is moved .01 inch, every .01 seconds or whenever the operator steps on a foot pedal. In any case, each border on the map is approximated by a chain of short straight line segments. If the chains represent boundaries, they partition the map into polygons, and, in general, each chain has a left and a right polygon.

Now assume a scalar attribute of the polygons of a map for which each polygon has a value. For instance the polygons might be states and the attribute per capita alcohol consumption. Consider the map in the X-Y plane and erect in the positive Z direction a prism[*] on each polygon with height being that polygon´s value of the attribute.

The result is a 3-D scene that can be processed by a general purpose hidden surface algorithm. In fact the only existing program for drawing such scenes, that by Waldo Tobler [19??], does just this. But because the objects are prisms instead of general polyhedra, a much faster algorithm is possible. Indeed most of the calculations can be done on the input map without even knowing the prism heights. Once the preprocessing is done, scenes with different relative heights and shading can be plotted quickly. I describe this algorithm in chapter 4 and an implementation (summary, users´ guide and program logic manual) in Appendix B.

---

[*] A prism has a horizontal top and bottom. The top is congruent to and directly above the bottom which is a general polygon that does not intersect itself. The prism has a vertical rectangular side corresponding to each of the polygon´s edges.

1.2.1 Shading

Various people, principally at the U. of Utah - Blinn [1976], Crow [1976], [1977a], and Phong [1975] - have investigated how to shade the computer generated scenes. They have made discoveries in the laws of reflection from surfaces that are neither fully diffuse nor completely specular, and produced output that is remarkably realistic, involving highlights, shadows, and reflections of lighted windows on the object. However, these methods use large amounts of computer time. My goal is more modest: the scenes being plotted have no physical existence but are schematic. The purpose of shading them is not physical correctness but the enhancement of comprehensibility - to make it easier for a casual observer to take in the scene at a glance. By way of analogy, note that it is easier to learn to recognize a person from a skilled caricature than from a photograph. The cartoon emphasizes the important features while the photograph presents everything impartially.

In chapter 4, I also consider a shading problem arising from the prism plotting algorithm. This is the problem of having the shading grade smoothly from a face to its neighbour when they are not being shaded one directly after another, the faces are smaller than the shading cross-hatch spacing, and the shading algorithm is left partly unspecified until plot time.

1.2.2 Statistical Analysis

To design and analyze hidden surface problems properly, it is necessary to know the statistical distribution of the input data. This is a difficult problem, not least since the algorithm will affect the distribution - people will use a program more often on those problems that it handles best. In designing the algorithms mentioned above, what I think were reasonable assumptions were made. Of course others may disagree.

One case is analyzed more deeply, but is too complicated to consider completely. This is the distribution of chains in prism-maps arising from natural geographic boundaries such as shorelines and rivers. It is generally accepted that shorelines are scale independent - that is that statistically an outline at a scale of 1 inch to 1 mile is indistinguishable from one at a scale of 1 inch to 1 foot. However the implications of this assumption are vast, extending possibly even to the fractional dimension curves described in Mandlebrot [1977]. There is a complication caused by the fact that the original digitized chains are rarely used since they are too detailed. Instead they are "generalized" by the omission of points so as to produce new chains that never deviate from the original chains by more than some given error. The usual algorithm as given in Douglas [1973], while fast, produces output that is no longer scale invariant. Some aspects of this generalization problem are considered in Chapter 4, also.

1.3 THE HIDDEN SPHERE SPECIAL CASE

Chemistry students buy ball-&-stick kits to make and study models of simple molecules. Professional chemists do the same with more complex molecules like insulin in order to better understand such problems as whether different parts of the molecule are so close to one another as to interfere. However the models are tedious to construct and to change mechanically. The only extant algorithms for computer generation of pictures of such molecules use BFI (brute force and ignorance) number crunching. Chapter 5 presents a faster algorithm for nonintersecting ball models that takes time N*log(N) under reasonable assumptions that are detailed there.

1.4 GENERAL PRINCIPLES

These algorithms illustate the usefulness of two general principles. One, the variable grid, is used in the general algorithm and in the hidden spheres problem. It is a new concept, borrowed from the "buckets" used in partial match hash function retrieval, Rivest [1976]. The purpose is to find quickly which of a set of elements such as edges or spheres coincide.

The other principle is the utility of being able to sort elements by whether or not one hides another. This concept has been used before, for example in Newell´s algorithm. In that case, objects sometimes had to be split in order for such an ordering to exist. I consider two special cases, prism-maps and spheres, where the ordering exists naturally and is easy to determine.

1.5 VECTOR TO RASTER CONVERSION

The algorithms I describe are designed for a vector plotter, such as a Calcomp, Tektronix or Milgo. A primitive step of a vector plotter is to draw a straight line between two given points. However, many people, such as Negroponte [1977], believe that the trend in applications is towards raster scan devices such as the Evans and Sutherland Picture System 3, a TV monitor, an electrostatic printer/plotter such as Versatec, Xerox or Gould, or an ink-jet plotter such as Applicon. There are many programs to convert the point pairs that comprise the vector plotter commands for a raster plotter but they are not always efficient. In chapter 6 I analyze the conversion problem to determine optimal algorithms under different assumptions of the amount of main memory available, number of lines drawn, total length of those lines, and resolution of the raster device.

1.6 SUMMARY

The final chapter summarizes the thesis and describes some of the related remaining open problems.

1.7 APPENDICES

Appendix A describes the implementation of the faster object space algorithm described in Chapter 3. There is a quick summary of the program, an internal logic manual and a users´ guide. This implementation also serves to illustrate some of the problems that have kept the art of implementation from becoming a science. Numerous messy cases must be considered. The problems are exacerbated by the finite precision of floating point numbers. This can cause problems such as making a point near the border of a polygon appear to be inside or outside it depending in the orientation of the axes.

Appendix B gives the implementation details of PRISM, an implementation of the algorithm described in chapter 5.

CHAPTER 2
HISTORY

2.1 INTRODUCTION

Computer graphics is the branch of computer science that deals with the manipulation of pictures and graphics by computer. It is a relatively new field, even as computer science goes, since it did not become practical until computers became rather large and fast. As computing power becomes cheaper, applications where constructs at a higher level than simple numbers are manipulated become more important. Thus computer graphics is just one of the new expanded applications oriented areas of computer science. For a summary of where the field was only eleven years ago, see Skinner [1966], Sutherland [1966], and Coons [1966].

A computer can treat pictures in two supplementary ways: It can take an existing picture such as a photograph and extract the meaning from it. This is picture processing, and is not further considered here. On the other hand, the computer might be given the meaning or intended content of a picture that does not yet exist and create it. This includes, among other subjects such as animation and computer assisted design (CAD), the hidden surface problem. Some samples of recent work in CAD are given in Crow [1977b] and Clark [1976a]. Braid [1975] synthesizes solid objects from intersections and unions of cubes, wedges, and cylinders. Parent [1977] gives the user various natural "sculpture" tools for cutting, shaping, and joining objects. Tanimoto [1977] gives another real time editing technique for environments.

Various aspects of the hidden surface problem are the subject of this thesis. The hidden surface problem concerns the calculation of a picture of a scene composed of translucent and opaque objects. Since the objects are opaque, those that are in front hide those behind them. The problem is to determine what is visible and what is hidden. The problem has been attacked with varying degrees of success for about fifteen years. Roberts [1963] had the first known solution.

The long term driving force behind this research is the desire that concepts that are simple to think about should be simple to manipulate by computer - this after all is what the computer revolution is all about. Now it is easy to imagine a scene such as a room with furniture and even easy for an artist to sketch an outline of the scene. However producing an exact photographic quality painting of what an observer would see can be quite difficult. This is especially marked if there is significant perspective and many objects with different levels of detail. If it is desired to produce another drawing of the same scene from a different angle, all the work must be repeated, even though only a simple change is being made. The problem is analogous to that of producing machine code. Before high level assemblers and compilers were invented, a conceptually small change required recoding the whole program; now the small change is made to the higher level source and the compiler handles the bookkeeping of the changes.

This problem is illustrated in another way by the difference between the difficulty of writing a story and the difficulty of producing a movie. An author can set the mood by sketching a scene with a few sentences; but a producer must build a million dollar set to achieve the same effect. Computer graphics aims eventually to lessen this disparity by allowing nontechnical users to handle such pictures intuitively. A good survey paper of the progress in realism of computer generated images is Newell [1977].

A good general reference on computer graphics and hidden surface algorithms is Newman & Sproull [1973]. Rogers & Adams [1976] is another general work on graphics techniques and projections. The canonical paper on hidden surface algorithms is Sutherland, Sproull & Schumacker [1974b]. Pooch [1976] has a large bibliography on computer graphics in general and Orr [1978] describes different graphic output devices. The analysis of the algorithms requires techniques in geometrical probability such as described in three recent survey papers by Moran [1966], [1969] and Little [1974]. Donath [1968] shows more useful geometric statistics. Melzak [1973] contains many practical techniques and tricks for solving geometric problems such as can arise in analyzing the behaviour of hidden surface algorithms. There are also relationships between some of the concepts in hidden surface algorithms and recent work in databases such as partial matching by Rivest [1976], [1974].

2.1.1 Terminology

Different authors in the field tend to use different terminology, if only because they are addressing different aspects of the problem. The terminology here is chosen to facilitate the presentation of this thesis. Only terminology pertinent to more than one chapter is given here - words of only localized use are defined in the individual chapters.

The scene is the ensemble of three dimensional objects being plotted. In an aircraft flight simulator, it is the airport, other planes and surrounding landscape. In a view of the city, it is the buildings and vehicles. Depending on the scale, it may include windows and individual rooms of the buildings. In the PRISM-MAP algorithm given in chapter 4, it is the set of prisms. For convenience, the scene can be divided into objects. An object has no formal definition but is just one conceptual part of a scene. If the scene is composed of only straight lines and flat planes then it has vertices, edges, and faces. A vertex is a 3-D cartesian point. An edge is a straight line segment between two vertices. A face is a flat polygon in 3-space that is defined by the ordered list of its vertices.

2.1.2 Restrictions And Special Cases

Since producing a computer generated picture of a complicated scene containing shadows, highlights, internal reflections, refractions and translucent objects is even now impossible, various workers have restricted the set of allowable scenes with which they worked. The early algorithms generally required the scenes to contain only straight lines and flat planes. All the objects were opaque and shading, shadows and highlights were not even considered. Even the flat faced objects were restricted by requiring them to be closed convex polyhedra. Sometimes the polyhedra were required to obey a partial ordering where the relation was whether one hid another as seen from the viewpoint. However if there were violations, then the offending polygons could be split until there was a partial order. Generally polyhedra are not allowed to intersect or cut through each other since this complicates calculations considerably.

2.1.3 Curved Surfaces

When algorithms allow the scene to contain curved surfaces, they have to model the curves somehow. Thus this is closely related to CAD (Computer Assisted Design) which models 3-D objects. One method is to provide a menu of stock objects such as spheres and cylinders and allow the user to compose a scene of these pieces, possibly with pieces cut off by intersecting planes. In other cases, general curves with many parameters are used. One favorite is the cubic spline which is extended into three dimensions as the patch. A cubic spline is a method of interpolating a smooth functional curve through a set of points (x_i, y_i). Between any consecutive pair of points the curve is a cubic polynomial and the two polynomials that meet at every point except the endpoints are related by continuity conditions - they have the same value and first derivative. Splines can be extended into 3-D by dividing the surface of an object into patches and using a separate function on each area. Another popular class of patch functions is the set of conic functions which gives rise to quadric patches. A quadric patch is a surface satisfying the equation:

$$Q(x,y,z) = a_1x^2+a_2y^2+a_3z^2+b_1xy+b_2yz+b_3xz+c_1x+c_2y+c_3z+d$$

= Ø

Greville [1969] is a general theoretical mathematical book on splines. Cline [1974] considers something called splines under tension. If the spline is considered to be made of rubber, this models a tension applied to the ends and has the effect of reducing the curves. Manning considers the use of splines in modelling. Gordon [1974] considers the case where smoothness is more desired than accuracy. For this he uses B-splines which are composed of Bernstein polynomials. They converge very slowly but are quite smooth. Pavlidis [1976] considers the problem of piecewise approximations to a function where the number of pieces is changed by splitting and merging. At Cornell, Wu [1977], has an interactive approach to modelling curved surfaces by interpolating Cardinal splines between lofts composed of B-splines. Brewer [1977] has an interactive surface design system that uses points on the surface instead of parametric curves.

Woon [1972] at Rensselaer has designed an algorithm and Potmesil [1977] has implemented it in a program called QUADRAW. QUADRAW solves for hidden surfaces where the surfaces are quadric patches. However it requires the user to enter 1) the explicit equations of the surfaces 2) the equations of the intersections and boundaries of the surfaces and 3) the coordinates of the points where 3 or more surfaces meet. In the next version, however, these intersections will be calculated automatically.

Catmull [1974] uses hierarchical subdivisions to display curved surfaces.

Freeman [1974] is a good survey paper that discusses, among other matters, different ways of efficiently encoding curved lines. Burton [1977] gives a method of representing chains of many short lines such as might approximate a geographic boundary.

Levin [1976] considers objects composed of quadric surfaces and how to calculate the intersections of the surfaces. However his hidden surface algorithm simply uses brute force.

2.1.4 Very Large Databases

As a scene comes to represent a conceptual picture with ever greater accuracy, it begins to contain more and more data that are almost never used. For instance, if a city is modelled to a sufficient accuracy that the paperclips on a desk in a room in a building are included, then the volume of data will be enormous and no possible way of looking at the scene will ever use more than a small fraction of it. For instance a bird´s eye view of the city will be unable to see anything inside the buildings except what is close to the windows and even that will be too small to be resolved. On the other hand, an observer looking at a room in a building from close up will be unable to see anything outside his particular room. To handle this special case of extreme generality, hierarchical databases are used. This technique allows whole large groups of irrelevent data to be excluded in one step so that the algorithm can spend its time analyzing the data that is more likely to be visible. Clark [1976b] considers hierarchical databases.

Fuchs [1977b] considers another special case, that where the computation of the hidden surfaces is distributed over many processors. He finds that this can be done and the separate results pieced together efficiently.

2.2 THE PERSPECTIVE PROJECTION

The scene is being viewed in <u>perspective</u> <u>projection</u>. This is a means of mapping the 3-D scene into a 2-D plotter drawing. This is what is done in an ideal camera, assuming that a wide-angle or fisheye lense is not used. The projection is defined by a <u>viewpoint</u> which is a point in 3-space defined by its coordinates and a <u>perspective</u> <u>plane</u>. The perspective plane is a plane in 3-space that is most conveniently defined by its <u>centerpoint</u> or the point on the perspective plane closest to the viewpoint. A vector from the centerpoint to the viewpoint is normal to the perspective plane, that is, is perpendicular to any line in the plane. Straight lines of sight are extended from the viewpoint through the perspective plane to the points of the scene. Any given point of the scene is projected to the point where its line of sight intersects the perspective plane. Since straight lines in the scene project to straight lines on the perspective plane, it is sufficient to project the endpoints of any edge in the scene and then draw the projected line between the projected endpoints. The perspective projection is shown in Figure 2-1.

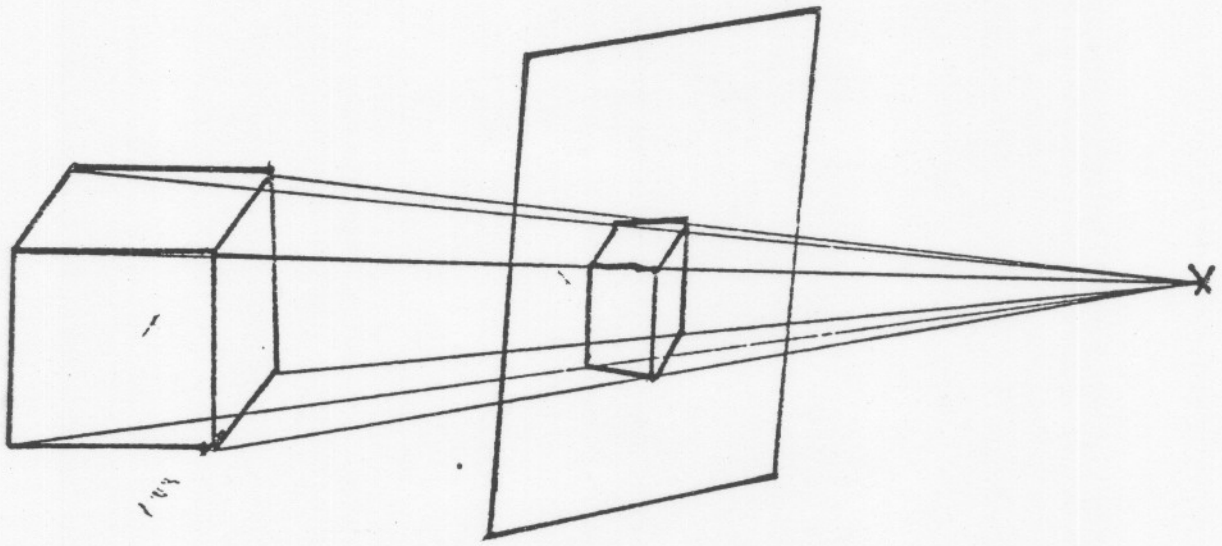


Figure 2-1: Perspective projection

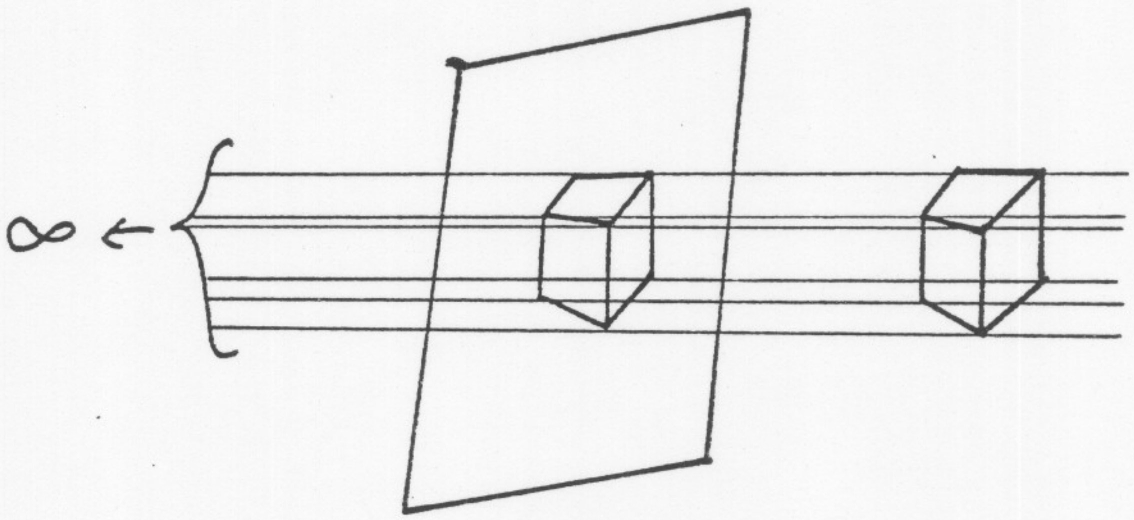


Figure 2-2: Perspective projection transformed to an orthogonal projection

There are various special cases of this general projection, which is also called a _three point projection_. For example, the viewpoint might be at infinity, in which case the lines of sight are parallel. When the viewpoint is at infinity, it can be conveniently defined by the direction cosines of a ray from the origin towards it. This is called an _orthogonal projection_ or an _isometric projection_. There are also two in-between projections: _one point projection_ and _two point projection_. Here the lines of sight converge in only one or two directions and remain parallel in the remaining direction(s). This is equivalent to having one or two, respectively, of the coordinates of the viewpoint being finite and the other two or one of the three infinite. If a coordinate is zero, it can be considered either finite or infinite. Architects and designers treat these as four separate cases of different projections although mathematically they are essentially identical.

2.2.1 _Normalizing The Projection_

It is convenient to bring the scene to a standard projection so that viewpoint and centerpoint do not have to be brought explicitly into every calculation. The goal is an orthogonal projection with the perspective plane's equation Z=Ø and the viewpoint at (Ø,Ø,infinity). In this case the point (x,y,z) is projected to (x,y) which is a simple operation. Further, the Z coordinate of a point expresses its distance from the viewpoint: a higher Z means the point is closer. Since the viewpoint is an infinite distance away, we are dealing with different infinite quantities differing by a finite quantity, but this arithmetic can be rigourously and consistently axiomatized, should it ever prove necessary. If the projection is already orthogonal, the scene need only be rotated to make the viewpoint correct. No shifting is necessary since all projection planes Z=c for any c are equivalent.

However the case where the initial projection is three point is a little more complicated. Assume that the initial viewpoint is V and the centerpoint C. A rotation of the scene followed by a dilatation (or scaling) can make V=(Ø,Ø,Ø) and C=(Ø,Ø,1). Then the following transformation will convert the projection to an orthogonal projection:

$$X' = X/Z$$

$$Y' = Y/Z$$

$$Z' = 1/Z$$

This transformation is applied to every point of the scene to produce a transformed scene. Now the original scene projected with the original V and C will produce the same result on the perspective plane as the transformed scene projected with the new viewpoint V=(Ø,Ø,infinity) and centerpoint C=(Ø,Ø,Ø). This is just the standard orthogonal projection mentioned above. This transformation preserves straight lines and flat planes; otherwise it would be rather useless. Hence it is necessary to transform only the endpoints of any straight edge and only the vertices of any flat polygon, and then draw the projected line or polygon on the perspective plane between the projected points. This transformation, like all perspective transformations in projective geometry, transforms conics to conics, so that one class of curved objects is handled as well. Figure 2-2 shows the transformation of the perspective projection of Figure 2-1 to an orthogonal projection.

This transformation has the property of reversing the sense or parity of a scene so that a right handed coordinate system becomes a left handed one. Some algorithms depend on the parity of the scene. For these a reflection can be applied to the scene to restore the parity.

One point and two point perspectives can be normalized by similar techniques.

2.2.2 Inverted Perspective

It is sometimes convenient to use an *inverted perspective* where the viewpoint is farther away than infinity. Hence the lines of sight diverge as they approach the horizon instead of converging. This means that nearer objects are smaller than distant objects with the result that more of the scene is visible. A normalized perspective projection can be transformed to a normalized orthogonal projection thus:

$$X' = X/Z$$

$$Y' = Y/Z$$

$$Z' = -1/Z$$

Unlike the previous transformation, this one doesn't reverse the parity of the scene.

2.2.3 Clipping The Scene

Continuing the analogy with a camera, it is usually desirable to show only those parts of the scene that fall within a given square on the perspective plane. If the viewpoint is inside the scene then even though the parts of the scene behind the viewpoint also project onto the perspective plane, they should never be shown. The operation of removing the parts that are not to be shown is called clipping. The parts of the scene that project to a square on the perspective plane centered on the centerpoint are those within a certain square cone in 3-space in the original scene. The interior of this cone satisfies the inequalities:

$$\mathrm{abs}(X) < c\ Z$$
$$\mathrm{abs}(Y) < c\ Z$$
$$Z > 0.$$

The constant c determines how big the square on the perspective plane is. This square cone transforms to a semi-infinite rectangular solid when the projection is made orthogonal. Its equation is:

$$\mathrm{abs}(X) < d$$
$$\mathrm{abs}(Y) < d$$

$Z > 0$

for some d. Hence it is easier to clip the scene after it has been transformed. Jarvis [1975] gives two algorithms for clipping which exploit the hardware of the PDP-9 and PDP-11 to run very fast. One of them does not use even multiplication or division.

2.3 ALGORITHMS

2.3.1 Various Taxonomies

There are many hidden surface algorithms that are more or less closely related to each other and they have been classified in various ways.

One such taxonomy is due to Myer [1975]. It is:

I. Realistic
   A. Fast
      1. Clever
      2. Industrious
   B. Easy
   C. Accurate
   D. Compatible
II. Symbolic

A. Contours

B. Isograms

Realistic algorithms model real scenes as opposed to symbolic algorithms which display data such as contours graphically. Fast algorithms are designed for real time and are either smart or efficient. Easy algorithms use brute force and ignorance. Accurate algorithms concentrate on realistic shading. Compatible algorithms are utilities that are used by another program and which are only required to work, not to work very well. He also divides scenes into <u>direct</u> and <u>indirect</u> depending on whether they are models of reality or models of models of reality.

Another classification, which the the one most commonly used, is due to Sutherland, Sproull & Schumacker [1974b]. Here the class of algorithms is divided into <u>object space</u> algorithms, <u>image space</u> algorithms and <u>list priority</u> algorithms.

This paper by Sutherland, Sproull & Schumacker is an important summary of progress in hidden surface algorithms up to until about five years ago. It gives a history of the subject and briefly describes the important algorithms. The algorithms are classified according to how they sort the input data since they propose sorting as the fundamental means of differentiating among the algorithms. All algorithms sort the data along a scan line (the X direction), by scan line number (Y) and in distance (Z). They also propose that coherence is a prime property of scenes that all efficient algorithms use. Coherence means that the scene changes very little as some parameter varies slowly. For instance the set of edges whose projections intersect a scan line is almost the same as the set intersecting the adjacent scan line. In an aircraft flight simulator, as the viewpoint changes, the relationships in the scene such as which faces hide which faces change slowly and smoothly. Because of their systematic classification, they are able to suggest new approaches that no one seems to have tried.

Object space algorithms take a part of the scene such as an edge or face and determine whether or not is is visible. They operate to the precision of the CPU and if the resulting picture could be plotted to a greater resolution than the particular display device happens to provide, it would still be accurate. Since the results are the visible pieces of the edges and faces, edge by edge and face by face, the natural output plotters are vector plotters. Since raster technology is becoming more important, Negroponte [1977], in Chapter 6 I analyze efficient algorithms to convert vector output for raster devices.

In contrast, image space algorithms iterate over the pixels* of the screen and determine how to colour each one. They calculate only to the accuracy of the device and if a new display with greater resolution were used, more calculations would be needed. These algorithms are naturally compatible with raster scan devices. The list priority algorithms fall in the middle since they do some of their processing in object space but finish in image space and calculate the picture only to the precision of the display device.

---

* A pixel is the smallest addressable part of a raster screen. It is displayed as one dot of light that has the attributes of intensity or brightness, and in colour devices, hue and saturation.

Since image space algorithms calculate the plot only to the necessary precision while object space algorithms do so exactly (to the machine´s limits), there is an impression that the latter are inherently much slower. In chapter 3, I show by counterexample that this is not invariably true.

2.3.2 Object Space Algorithms

First I describe a general method for object space algorithms that is a mixture of various published algorithms and then I describe various published algorithms.

2.3.2.1 General Method -

2.3.2.1.1 Edges -

Whether or not an edge is hidden is determined by whether or not there are any faces hiding it. An edge, E, from the set of edges SE={E_i}, changes its visibility when it goes behind, or comes out from behind, a face from the set of faces SF={F_i}. This occurs when it crosses a face boundary, which is when it crosses another edge since these algorithms either require the face boundaries to be in the edge database, or calculate them and insert them there. Thus if E is partitioned into a set $SS=\{S_i\}$ of edge segments by the places where E intersects all the other edges in SE then each segment S_i in SS is either totally visible or else totally hidden.

In more detail: Let E be projected into edge E′ on the perspective plane. Let another edge E_2 be projected into edge E_2'. Assume that E′ and E_2' intersect at point Q on the plane. Since Q is on E′ it is the projection of some point R on E. R is the point at which another segment, S, will be cut off E. It is sufficient to take one point P (say the centre) of S and test it against all the faces F in SF. If any F hides P then S is hidden; otherwise S is visible.

Thus the naive algorithm is:

```
SV <- {};
REPEAT for all E in SE
   SX <- {intersections of projection of E with projections
          of other edges in SE};
   SX <- Sort(SX along projection of E);
   SS <- {segments into which projection of E is cut by
          members of SX};
   REPEAT for all S in SS
      P <- Midpoint(S);
      Flag <- TRUE;
      REPEAT for all F in SF
         IF Hides(F,P)
            THEN Flag <- FALSE;
      ENDREPEAT;
      IF Flag
         THEN
            Plot(S);
            Addelement(SV,S);
         ENDIF;
   ENDREPEAT;
ENDREPEAT;
```

Hides(Face,Point) is a routine that returns a flag saying whether a face hides a point. Addelement(Set,Element) adds an element to a set. The set of visible segments is accumulated in SV for later use.

2.3.2.1.2 Faces -

The above algorithm determines the visible edges ("the hidden line problem"); now we must determine the visible faces ("the hidden surface problem"). A given face, F, can be either totally hidden or can have one or more visible connected polygons on the plot. However these polygons are delimited by the visible segments in SV. These segments determine a planar graph and partition the plane into a set of polygons $SR=\{R_i\}$.

Given the set of NVIS visible edges, where the only attributes of each edge are the coordinates of its two endpoints, the explicit polygons in the partition can be determined thus:

1) Sort the edges by endpoint so as to identify the common endpoints of the different edges.

2) For each point, make a list of the edges ending on that point.

3) It is necessary that the graph be connected since otherwise there may be one connected component subgraph inside a polygon of another subgraph. If that outside polygon does not know about the subgraph inside it, then when the outside polygon is shaded, the shading will cover the inside component and all its faces also, as shown in Figure 3-5. This is erroneous. So pick a point and traverse these lists to determine whether the graph is connected. If not, add edges between the disconnected components to connect it. As each edge is added, test it against all other edges to see whether it intersects them and if so add a new point at the intersection and replace the two edges by four edges.

4) Sort the edges by angle around each point.

5) Pick an edge at random and follow from edge to point to edge always turning in one direction until back at the starting edge. This finds one polygon. Each edge will eventually be traversed exactly once in each direction so as each edge is traversed, mark it used in that direction.

6) Pick an unused (in one direction at least) edge, if any remain, and repeat step 5.

7) The result of this will be a set of polygons, one of which will be the external polygon around the outside of the whole graph. Identify it by its signed area which will be negative and delete it.

Because of step 3 above, this algorithm takes time $\Theta(\mathrm{NVIS}^2)$ in the worst case. However the expected time seems to be dominated by the sorting stage so it is $\Theta(\mathrm{NVIS}*\log(\mathrm{NVIS}))$. This is because in practice the graph is almost always connected.

Now each polygon, R, of the graph corresponds to one face. Tthe converse may be false since one face may have several polygons if its visible region is split by another face in front. So it is sufficient to take a point, P, in R and test it against all faces in SF. It is not so easy to

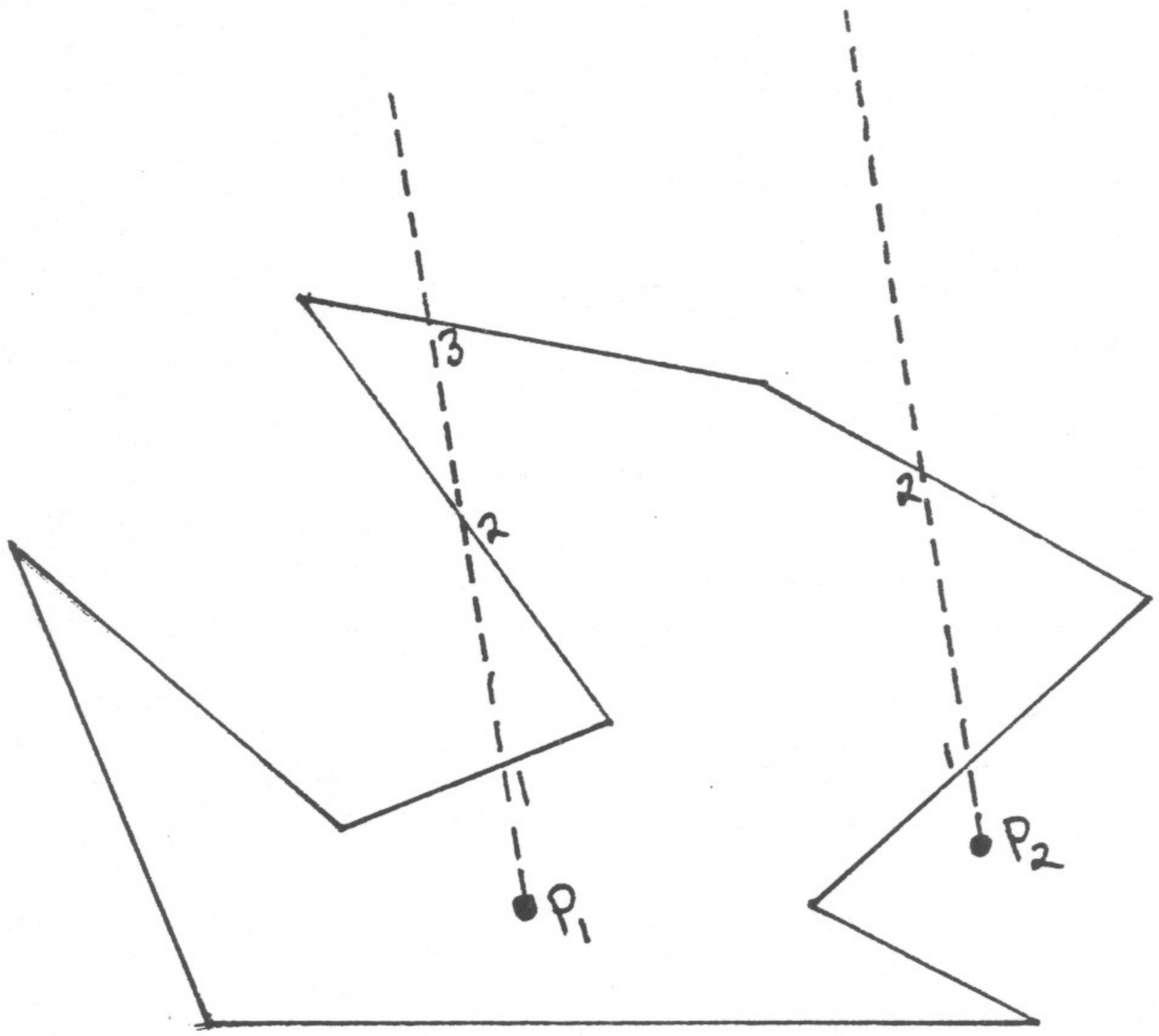


Figure 2-3:
Determining whether a point is in a polygon

find P as before with the edges. This is because if R is not convex, then the centroid of R, or the centre of its enclosing box, may be outside R. Generally either of these points will be inside R, so it is worthwhile to test such a point first with a point-in-polygon routine such as shown in Figure 2-3. This algorithm draws a semi-infinite ray up from the point and counts the number of intersections with the sides of the polygon. If this number is even, the point is outside, and if odd then it is inside. The only tricky special case is if the ray goes through the vertex; then it must be determined whether the ray is passing through or just grazing the polygon´s perimeter. P_1 is inside the polygon since its ray cuts the polygon three time but P_2 is outside. This algorithm takes linear time in the polygon´s size which is adequate. The multiplicative constant in the time is also very small so that it is not only good asymptotically but also good for small polygons in actual implementations. There are also faster algorithms using Voronoi polygon nets that take time $\Theta(\log(N =$ number of edges in polygon$))$ to test a given point, provided there is $\Theta(N*\log(N))$ time to preprocess it.

Nevertheless we need some other points in case the centroid is outside. Luckily, if all nonadjacent vertices in the boundary of R are joined by lines and the midpoints of those lines are found, then at least one such midpoint will be inside R.

2.3.2.1.3 Timing -

If there are N edges, it takes time $\Theta(N^2)$ to find the intersections. Under certain statistical assumptions detailed in chapter 3, there will be $\Theta(N^{4/3})$ intersections. Comparing each of these against all the $\Theta(N)$ faces takes time $\Theta(N^{7/3})$. The exact time depends on the statistical assumptions but is Omega(N^2) even if there are no intersections (since every pair of edges must still be compared) and Omega(N^3) if every pair of edges intersects.

2.3.2.2 Other Object Space Algorithms -

Some algorithms, such as those by Appel [1967], Galimberti [1969], and Loutrel [1967] and [1970] employ additional refinements. For instance, if there are N faces hiding edge E at some point and E crosses another edge, E_2, then E is now hidden by N+1 or N-1 faces. Therefore if N is large enough, there is no need to determine the visibility of this segment of E; we know it must be hidden. We can thus propagate the minimum number of faces hiding an edge segment and only actually check the segment when this falls to zero or below. We can also propagate it across vertices since all the edge segments incident on one vertex will in general be hidden by the same faces.

However this refinement has problems that render it problematical whether it is really an improvement. First it is no longer sufficient to determine whether an edge segment is visible or hidden; we must also know by how many faces it is hidden. In large scenes, this takes much longer. Second, there are many messy special cases to consider, especially when propagating through a vertex to another edge. If a third edge should run through the vertex then the number of faces hiding the first segment will change without the segment intersecting anything (unless being adjacent on an end is considered to be an intersection).

The Loutrel algorithm [1970] restricts the scene to be composed of polyhedra, convex or concave. An implementation by Potmesil in 1976, [1976], allows the faces to have holes. It uses a complicated data structure, requiring about 50 words per vertex storage and is implemented on a CDC-6600. The asymptotic time growth is not given but the time required seems to grow rapidly with the scene complexity.

2.3.3 Image Space Algorithms

These algorithms are generally suited to raster devices. They ask what should be drawn at given points on the screen instead of asking how a given part of the scene should be drawn. Here is just a brief summary, including some of the more interesting implementations.

Warnock's algorithm [1969] and [1970] proceeds recursively on the screen and the faces that cover parts of it. It asks if the situation is too complex to draw, and if so splits the rectangle into two smaller ones. It puts each face into that rectangle or rectangles with which it has nonempty intersections. This process of subdivision is repeated until it is easy to see how to shade the rectangle, or until the rectangle is smaller than a pixel. In fact this can be continued until the rectangle is one quarter the size of a pixel and the all the rectangles in a pixel can be combined to give an average colour. This tends to reduce "aliasing" problems which occur because of the finite resolution of the display device. This process is fast, but produces output in a random order, not easily convertible for a raster scan device.

The algorithm by Watkins [1970] has the scan line as the central point instead of the face. For each scan line, the set of edges whose projections onto the display screen intersect it is determined. The intersections of the edges with the scan line are used to divide it into segments, each of which corresponds to only one face. Then the nearest face corresponding to each segment is found from those faces intersecting the scan line. This algorithm desn´t need to store a picture buffer in core but only the segments of the current scan line. From this, the actual pixels can be calculated when they have to be displayed. This algorithm is very fast, owing partly to the fact that adjacent scan lines are very similar with respect to the edges and order of the edges intersecting them.

Archuleta [1972] gives an implementation of an algorithm that is similar to Watkins. Hamlin [1977] describes some image space algorithms and gives improvements.

2.3.4 List Priority Algorithms

These algorithms start their calculations in object space and then finish in image space. The two major algorithms are due to Schumacker [1969] and Newell [1972a], [1972b]. Schumacker's was the first real time hidden surface algorithm of any type. It is used in an aircraft flight simulator where the scene stays constant but the viewpoint changes. The scene has restrictions on it - for instance the faces must be grouped into clusters that are linearly separable (separable by a plane between them). Extensive preprocessing is done in object space on the scene. The real time calculations are made with special purpose hardware.

Donald Greenberg at Cornell [1974], [1977a], [1977b] has implemented an interactive system involving colour with shading and shadows. It uses an algorithm similar to

Schumacker. Since it does not operate in real time, it can handle more complex scenes.

Newell´s algorithm sorts the faces into a priority list ordered by distance from the viewpoint. If two faces are not comparable because they each overlap the other then one must be split until a partial ordering exists. Then the faces are written into a picture buffer, the farthest face first. A nearer face simply overwrites a more distant face in the same location. A transparent face doesn´t simply overwrite a face it hides; instead some combination of the two intensities is stored.

C.M. Brown [1977] developed an algorithm for displaying scenes composed of well tesselated polyhedra on a raster device. Such a polyhedron has triangular faces that may be transparent. It also has an interior point, P, from which all its faces are completely visible. Finally, if the polyhedron is projected onto the surface of a sphere centred on P, then the angle between any two adjacent edges incident on the same vertex is less than $9Ø^{o}$ and the angle between any two nonadjacent edges incident on the same vertex is greater than $9Ø^{o}$. A well tesselated polyhedron remains well tesselated under a radial transformation wherein the distances of its vertices from P are arbitrarily changed (provided their altitudes and azimuths remain constant). Brown sorts the faces as Newell does. The resulting order is not changed by radial transformations of the vertices. This algorithm takes time $\Theta(N*\log(N))$.

2.3.5 Two And A Half Dimensional Algorithms

By this I mean algorithms that plot the surfaces of functions Z=F(X,Y). Usually they place a fishnet or grid of lines over the function surface and plot its visible portions. There are many similar algorithms and they generally employ a device called a horizon line. This is a piecewise straight line running from left to right on the screen that never doubles back and that lies initially along the bottom edge. As the plot progresses, it moves up the screen. The algorithm, briefly, is:

1. As always, normalize the viewpoint by rotating and scaling the scene.

2. Now split the fishnet up into individual edges and sort them from from to back based on their distance from the viewpoint.

3. Take the edges in sorted order. For each edge, draw as much of it as is above the horizon line and raise the horizon line to meet the edge wherever it is below it.

A more detailed explanation of horizon lines can be found in the chapter on PRISM.

The horizon line was first used before 1968 by Rens and Tobler in the SYMVU program described in Lab for Computer Graphics [1977] and [1978]. Other references to uses of a horizon line are Brauer [1968], Veen [1977], S. Watkins [1974], and Williamson [1972].

Wright [1974] uses the same type of algorithm to plot electron orbital clouds, although they are not bivariate functions. He represents them by a 3-D bit array and then plots them front to back using a generalized version of the horizon line.

Computer axial tomography scanner algorithms also use the same techniques. They also use various interesting smoothing techniques on the resulting pictures. Fuchs [1977a] considers objects like human heads and intersects a system of parallel planes with them to create a set of cross-sectional contours. Various interpolation and smoothing techniques are used but the hidden surface method has a similar spirit. Edelheit [1977] and Herman [1977a] and [1977b] show some recent work in CAT algorithms.

2.4 OUTPUT

2.4.1 Devices

The output of hidden surface algorithms can be displayed on a variety of devices. In general, object space algorithms output on vector plotters such as pen plotters, vector CRT´s or microfilm plotters while image space and list priority algorithms output on raster devices such as raster CRT´s and graphic printer/plotters. Each mode of output has its own advantages and problems. For instance, a CRT is fast but is not as accurate as a pen plotter. Also the usual hardcopy devices for CRT´s produce copies that are not permanent (they fade in time). Pen plotters produce accurate copies but are slow and are subject to problems such as the pen skipping a little at the start of every line before the ink starts flowing. On the other hand it is easy to change ink colours in a pen plotter, in contrast to microfilm plotters which while fast and accurate, produce only black and white unless they are very expensive models. A raster device is better suited to shading since it can fill areas directly while a vector device must draw many close parallel lines. Also a raster device can do halftone shading easily. However, raster devices often have a very low resolution such as 256 by 256. If the resolution is not high enough, problems wth aliasing, Crow [1977a], will appear. Also oblique lines will have an unpleasant stepped appearance.

2.4.2 Shading

There are many different aspects to the realistic shading of computer generated images. Much of the work has been done at the U. of Utah by Blinn [1976], [1977], Crow [1976] & [1977a], and Phong [1975]. Newell [1977] is a recent summary of progress.

One of the aspects of shading is the micro-complexity versus the macro-complexity of the scene. The macro-complexity concerns the large scale description of the scene such as the faces themselves while the micro-complexity concerns such properties such as the texture of the faces. This affects the properties of the reflected light. Initially, the reflection was considered to be diffuse; later specular components and other components due to highlights were added. Shadows, Crow [1978] and Weiler [1977], are handled by first calculating the hidden surfaces from the point of view of the light surface. (This paper, "Shaded computer graphics in the entertainment industry", illustrates some of the powerful economic forces, such as the search for techniques to create more effective TV commercials and science fiction movies, that are driving research in computer graphics today.) The visible surfaces are those exposed to the light. They are marked and the hidden surface algorithm is applied again with the correct viewpoint. When the faces that are visible this time are shaded, the parts of the visible faces that were hidden from the light source are shaded differently. Transparency is handled easily in some algorithms such as Newell [1972a] that proceed by overwriting faces in the picture buffer. Here a transparent face is not allowed to completely overwrite a face behind it but instead some combination of the two faces is stored. Highlights can be included by using several light sources. If the light sources are not point sources then the light will be more muted and shadows will have smoother boundaries. For extra accuracy, the light diffraction around sharp corners and refraction through transparent objects should be handled; but so far no one knows how to do this efficiently. Colours can be handled by repeating the shading calculations three times, once for each of the primary colours. This allows the faces and light sources to have different colours. The calculations will be slightly different depending on whether an additive (as for a CRT) or subtractive (as for a print) colour scheme is used.

2.5 SUMMARY

As we have seen, the hidden surface problem is very broad and has no clean-cut natural separations between it and the rest of computer science but rather blends continuously into approximations of functions, databases, modelling and even into the physics of light reflection. This thesis attempts the elucidation of a small area involving object space algorithms and vector to raster plotter conversion.

CHAPTER 3

A FAST OBJECT SPACE ALGORITHM

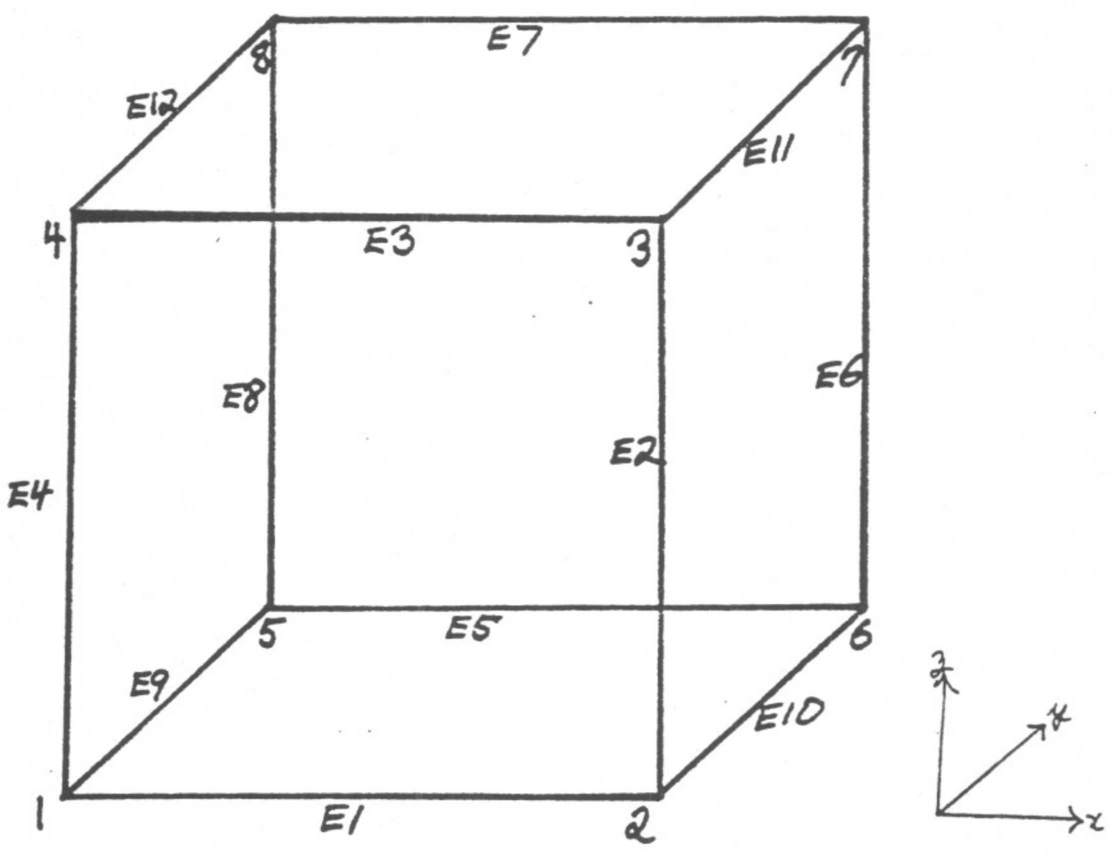
3.1 INTRODUCTION

This chapter describes a fast object space hidden surface algorithm. For scenes with NE edges satisfying certain reasonable statistics (to be defined in section 3.3), this algorithm calculates the hidden surfaces in time $T=\Theta(NE^{4/3})$. In general, a useful measure of the complexity of a given scene is the number of intersections among the projected edges in that scene, which we will call C. We will show below, that within broad limits, this algorithm takes time = $\Theta(C)$. One reasonable value for C is $\Theta(NE^{4/3})$ whence the statement above. This is an improvement since existing object space algorithms such as those described in Sutherland, Sproull & Schumacker, [1974b], take time $T=\mathrm{Omega}(NE^{7/3})$ under these statistical assumptions and time $T=\mathrm{Omega}(NE^{2})$ under any statistical assumptions. Now object space algorithms calculate the resulting visible scene accurately to the floating point precision of the scene and not just to the resolution of one pixel as image space algorithms do. Because of this, object space algorithms have been considered to be inherently very slow. The algorithm described here shows that the extra information calculated by object space algorithms does not constrain them to run so much more slowly than image space algorithms.

This algorithm only handles scenes in which all the faces are flat and the edges are straight. Curved surfaces are a topic for future research.

3.2 DATABASE FORMAT

The scene is assumed to consist of vertices, edges, and faces. A vertex is a 3-D point defined by its Cartesian coordinates. The vertices are numbered. An edge is a finite straight line between two vertices that is defined by the set of those two vertices. A face is a polygon on a plane in 3-space that is defined by an ordered list of vertices. Such a polygon may not intersect itself. The edges of a face are also edges in the database. See Figure



Vertices	
1:	(Ø,Ø,Ø)
2:	(1,Ø,Ø)
3:	(1,Ø,1)
4:	(Ø,Ø,1)
5:	(Ø,1,Ø)
6:	(1,Ø,Ø)
7:	(1,1,1)
8:	(Ø,1,1)

Edges	
1:	(1,2)
2:	(2,3)
3:	(3,4)
4:	(4,1)
5:	(5,6)
6:	(6,7)
7:	(7,8)
8:	(8,1)
9:	(1,5)
1Ø:	(2,6)
11:	(3,7)
12:	(4,8)

Faces	
1:	(1,2,3,4)
2:	(5,8,7,6)
3:	(2,6,7,3)
4:	(3,8,7,4)
5:	(4,8,5,1)
6:	(1,5,6,2)

Figure 3-1: Data structure for a cube showing vertices, edges, and faces

3-1 for an example of how a scene consisting of a cube is encoded. The faces are assumed to be opaque and to hide whatever is behind them.

3.3 STATISTICAL ASSUMPTIONS

Before any algorithm can be analyzed, it is necessary to know the statistical distribution of the input scenes that it is to be used on. Worst case analysis is possible, but inappropriate. All existing implementations of hidden surface algorithms above the naive level have some concept of a "normal" input and are optimized with respect to it. An adversary can make them run much more slowly by choosing proper input. The task is to define what input scenes are normal. This is difficult since there is no obvious a priori probability distribution on the complete set of all possible scenes. Instead, statistics should be kept of what scenes are actually drawn. However this presupposes an implemented algorithm; and even then the data would be biassed since people would tend to use such an algorithm more on those scenes that it plots more efficiently.

However some attempt must be made. As is usual, only orders of complexity are used since the actual multiplicative constants are dependent on the implementation of the algorithm and on the machine used. The measure of size of the input scene is taken to be NE which is the number of edges. Other measures could be taken such as the number of vertices (NV) or faces (NF) which are normally proportional to NE. To be precise, however, there do exist infinite sequences of scenes with the ratios $\frac{NE}{NV}$ or $\frac{NE}{NF}$ monotonically increasing without limit.

The following assumptions are made:

1. The edges are uniformly, randomly and independently distributed in location and angle of inclination. That is, the centre of each edge is drawn from the distribution $U[0,1]^2$ and the angle of inclination from U[0,2*pi). Any edge that would lie partly outside the square screen is deleted from the ensemble of possible edges. This effect of the border becomes relatively smaller as NE tends to infinity since the edges become shorter and a smaller fraction would cross the border if not deleted.

2. For any scene, all the edges are the same length.

3. The basic measure of complexity of a scene is the number of intersections among the projected edges.

Assumption 1, that the edges are independently and randomly distributed, is not exactly true since the edges are joined by the vertices. However, it becomes more nearly true as the scene becomes bigger, and is always a good working approximation. If the number of edges incident on a vertex is bounded, then the number of other edges adjacent to any given edge, and thus strongly correlated with it, becomes a decreasing fraction of the total number of edges. As the scene size tends to infinity, this fraction tends to zero.

Assume the scene is composed of fixed size objects such as cubes. If the different cubes are randomly oriented relative to one another then as the scene gets bigger, most of the edges (those from different cubes) are totally uncorrelated. Even if the scene is one highly ordered object such as a 3-D grid, the effect on a pair of distant edges is small. Instead of the probability distribution of their distance and angle being smooth, it has bumps at the allowable separations. These bumps in the probability distribution become relatively smaller as the scene gets bigger and thus don´t affect which grid cells the edges fall in. Further, under either a smooth or bumpy distribution, most of the edges will not intersect.

There is still another effect tending to smooth out the probability distribution of the edges. Viewing from a random angle and projection onto 2-D both act as convolutions that tend to smooth out the distribution and destroy the order. This loss of order effect is visually apparent in the loss of clarity when a photograph of a complicated scene is viewed, compared with seeing the original scene in stereo.

There may also be violations in assumption 1 due to the edges being predominantly vertical and horizontal, and due to the edges clustering in the centre of the scene. These are unimportant since they cause only a change in the multiplicative constant of the algorithm´s speed.

Assumption 2, that the edges are all the same length, simplifies analysis. If necessary, a scene can be changed to conform by splitting the longer edges until they are some average length. This will only change NE by a constant and so not affect the rate of growth of the time.

Assumption 3 is used since all object space algorithms must determine which projected edges intersect and calculate the intersections. The complexity of other calculations they perform such as determining the visible segments is related to the number of intersections.

The first problem is to determine the number of intersections among the projected edges. Normalize the screen to be of size 1 by 1. Two projected edges E_1 and E_2 with lengths l_i, centres (x_i, y_i) and angles of inclination a_i will intersect iff

$$((x_1-x_2)\sin(a_2)-(y_2-y_2)\cos(a_2))^2-((\sin(a_1-a_2))*l_1)^2 < 0$$

and

$$((x_2-x_1)\sin(a_1)-(y_1-y_1)\cos(a_1))^2-((\sin(a_2-a_1))*l_2)^2 < 0.$$

This formula is derived from the fact that E_1 and E_2 intersect iff the endpoints of E_1 are on opposite sides of

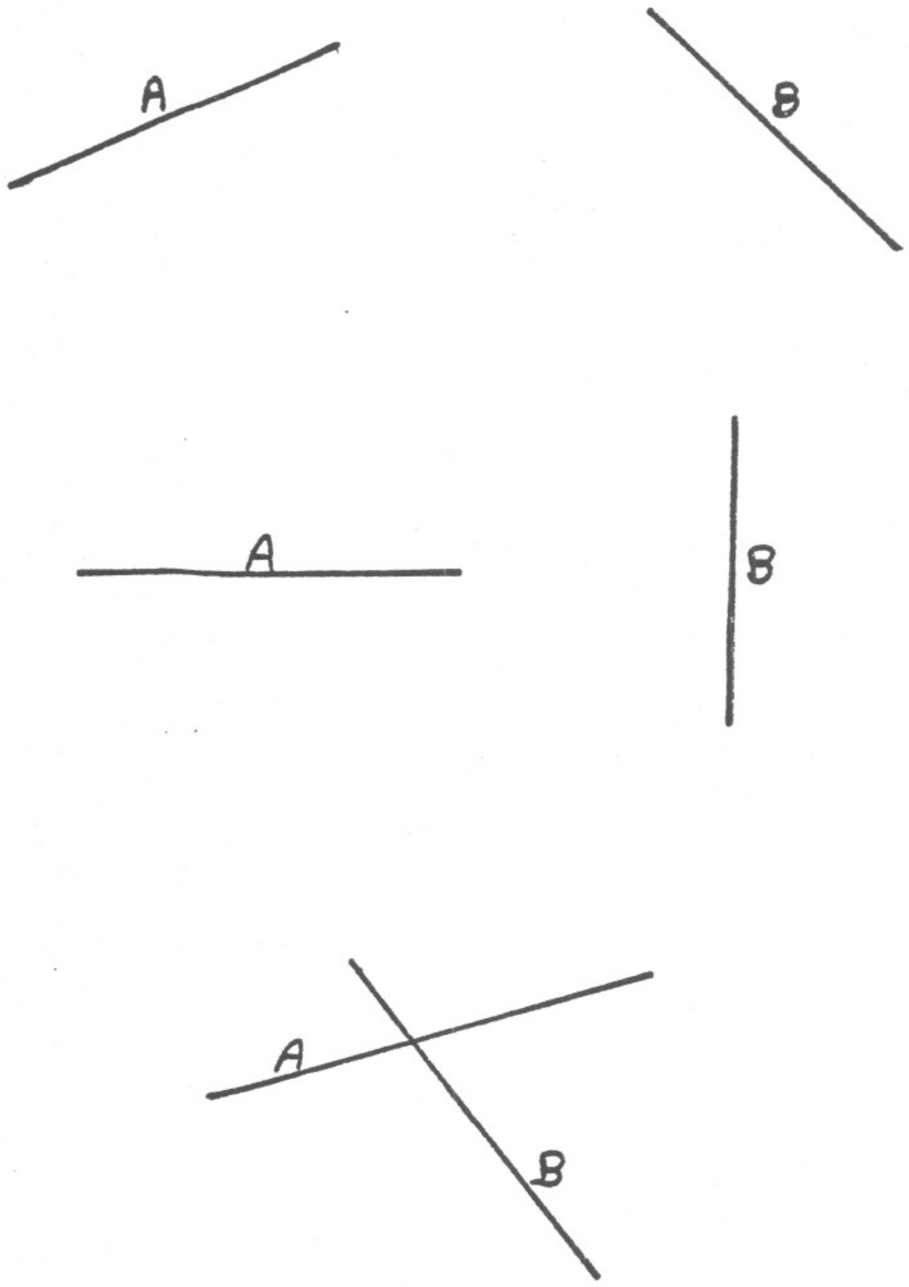


Figure 3-2: Determining when two edges intersect

E_2 extended to infinity, and vice-versa. This is illustrated in Figure 3-2. In the first case there, neither of the edges A and B crosses the path of the other. In the second case, only one does. In the third case, each crosses the other so the edges intersect. If the edges have centres (x_i, y_i) uniformly distributed in $[Ø,1]^2$ and angles a_i uniformly distributed in [Ø,2*pi), then the probability of their intersecting is $O(l_1*l_2)$.

If there are NE edges,

let L $= L_{NE}$ be their length (c.f. assumption 2).

Then NX = expected number of intersections

$= \Theta(NE^2L^2)$. (eqn 3-1)

TEL = total edge length

$= \Theta(NE*L)$.

Thus the scene complexity, which is the number of edge intersections, is the square of the total edge length.

AFA = average face area

$= \Theta(L^2)$ since the faces have sides L long.

TFA = total face area

= NF*AFA

$= \Theta(NE*L^2)$ since $NF = \Theta(NE)$.

Now, Appel´s algorithm [1967], which intersects all pairs of edges and then tests all edge segments against all faces, takes time

$$T_{Appel} = \text{Omega}(NE^2 + NX*NF)$$

$$= \text{Omega}(NE^2 + NE^3L^2)$$

$$= \text{Omega}(NE^2)$$

To be useful the form of L_{NE} has to be further specified in terms of NE. A typical infinite sequence s_i of scenes with increasing numbers of edges has to be found. One such sequence might be a cubical array of cubes. Let scene s_i have i^3 cubes containing $NE = 12*i^3$ edges. Since the scene´s total size is one by one by one and there are i cubes in a row, an edge of a cube has length i^{-1}. Then

$$L = NE^{-1/3}. \qquad \text{(eqn 3-2)}$$

So, substituting this into equation (3-1),

$$NX = \Theta(NE^{4/3})$$

and $T_{Appel} = \text{Omega}(NE^{7/3})$

Another possible sequence of scenes would have s_i containing not only the i^3 cubes of length i^{-1} but also all previous cubes. Thus there would be j^3 edges of length j^{-1} for $1 \le j \le i$. Thus, summing,

$$TEL = \Theta(i^3)$$

and splitting all the edges to a constant length of i^{-1},

$$L = NE^{-1/4}, \text{ so}$$

T_{Appel} = Omega($NE^{2.5}$) which is worse than before.

These times are actually lower bounds since clearly Appel's algorithm must take at least this time (for instance testing a pair of edges for intersection takes at least constant time). However the algorithm may actually take more time. In particular, various sorting operations will probably add a log(NE) factor.

3.4 THE ALGORITHM

3.4.1 Summary

It is assumed that the standard normalizations and perspective transformation defined in chapter 2 have been performed. This algorithm resembles that of Appel. In brief it is:

1. Perform various preprocessing steps such as deleting "back" edges and faces.
2. Determine which projected edges intersect.
3. Partition each edge at its intersections with other edges into segments.
4. Determine which segments are visible and plot them.
5. Use the visible segments to partition the screen into polygons.
6. Determine which face each polygon corresponds to and shade it accordingly.

3.4.2 In Detail

There are various preprocessing steps that speed up the algorithm by a constant factor of about two. Thus they are useful in an implementation but do not affect the asymptotic rate of growth. For instance, if the scene contains a closed polyhedron, then the faces and edges on its back can never be visible so they might as well be deleted. If we assume that every face of such a closed polyhedron knows which side of itself is inside the polyhedron and which side is outside, then the back faces are those with the viewpoint on the inside of the face. This applies whether or not the polyhedron is convex or not. Any edge that is adjacent to only back faces must also be on the back of the polyhedron and can be deleted. For example, in Figure 3-1, the back edges are 5, 8, and 9 and the back faces are 2, 5, and 6. An easy way to record the orientation of a face is to specify that its vertices run in a positive direction when seen from the outside. This is why chapter 2 mentioned that the projection normalization reverses the parity of a scene. After such a normalization, vertices that ran in a positive direction now run in a negative direction.

An average polyhedron has half its faces on its back. The fraction of back edges depends on the number of edges in the polyhedron but for a cube averages 1/4 and for a large polyhedron 1/2. Thus a significant amount of time is saved by deleting them. If the scene contains other objects besides closed polyhedra, they do not prevent this optimization from being applied to the polyhedra that are there.

An edge, E, can change its visibility only when it passes behind or comes out from behind a face. Since each face's edges are required to be edges in the database, E can change its visibility only when its projection onto the screen crosses the projection of another edge. Thus the projected edge intersections must be determined. One way is to test every pair of edges by the method mentioned earlier and illustrated in Figure 3-2.

If the intersections of edge E with other edges are sorted in order of occurrence along E and used to divide E into segments, each segment will be either wholly visible or else wholly hidden. Thus these segments must be determined. Since these intersections are in 2-D on the screen, they must first be "deprojected" to the original 3-D edge E.

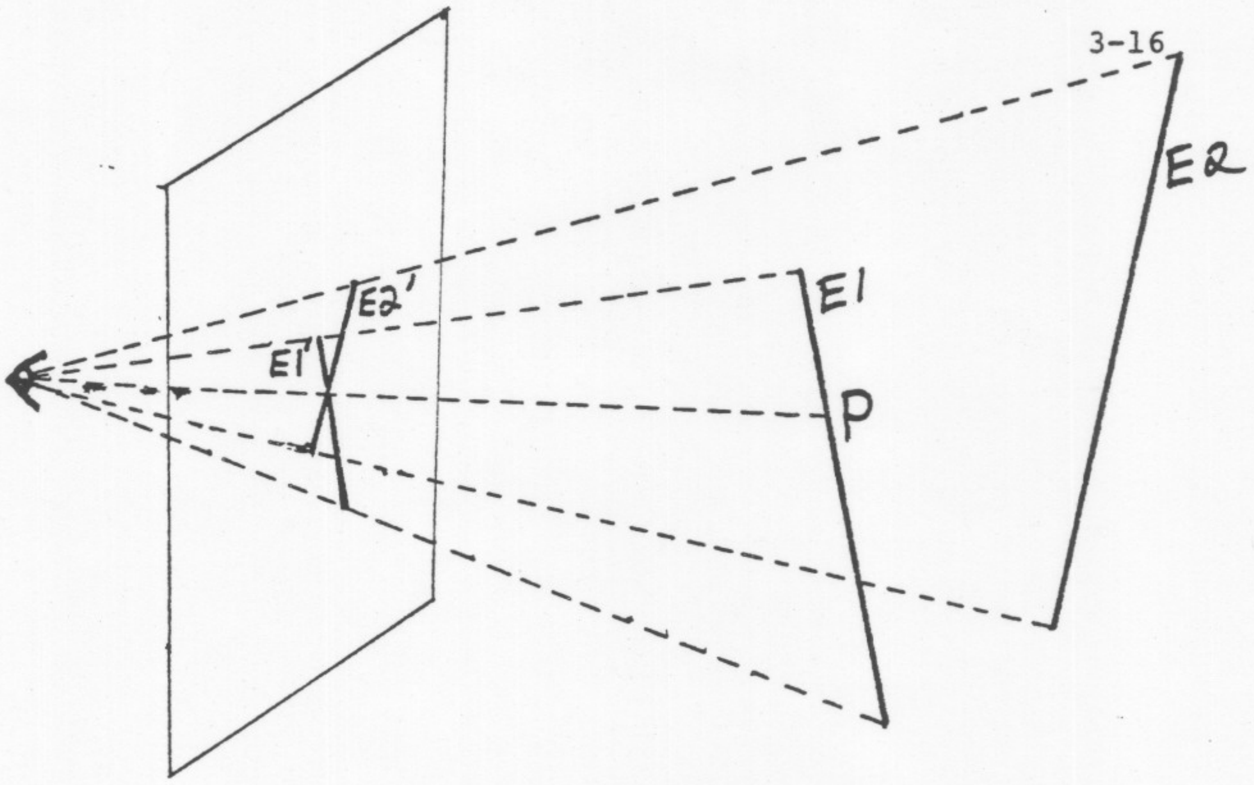


Figure 3-3: Deprojecting a projected edge intersection to the original 3-D edge

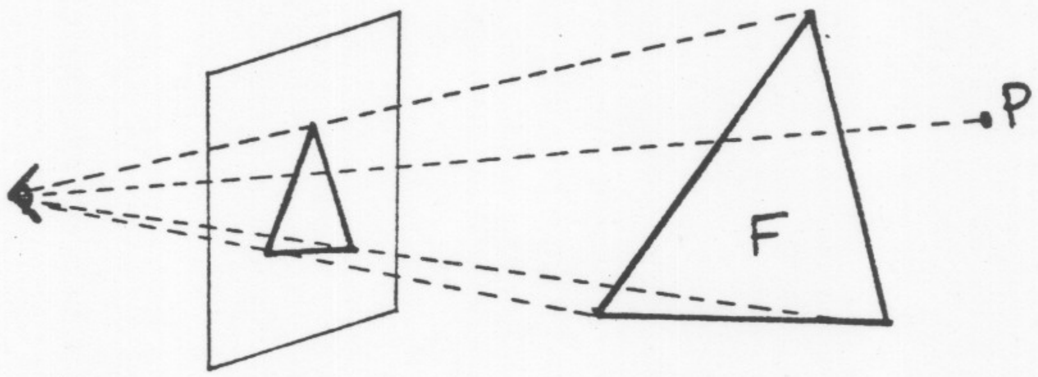


Figure 3-4: A face hiding a point

This is done by extending a line from the viewpoint through the 2-D intersection to meet E in 3-space at a point which is the 3-D intersection. See for example Figure 3-3 where the intersection of the projections of E_1 and E_2 is deprojected to P on E_1.

The visibility of a segment, S, is the same as the visibility of any point along it, say its midpoint, P. P is a point in 3-space. Compare it with all the faces to determine which it is behind. Being "behind" a face, F, as in Figure 3-4, means that a line from the viewpoint through P passes through F. There are two parts to this: P must be inside the projection of F and the line from the viewpoint to P must pass through the plane of F. The first part can be performed easily since projecting F consists of suppressing the Z coordinates of its vertices and since testing whether a point is in a polygon is easy. One such algorithm is given in chapter 2 and illustrated in figure 2-6. The second step can be done by substituting P into the equation of F to see which side of the plane of F, P is on. It is behind the plane if it is on the opposite side from the viewpoint. If P is behind no faces then P and also the corresponding segment is visible. The visible segments can

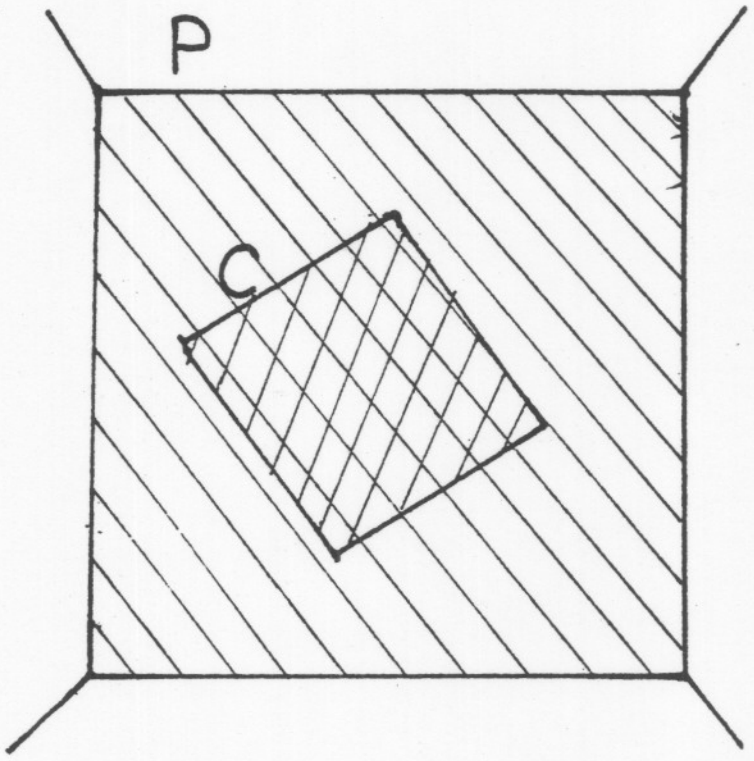


Figure 3-5: Disconnected planar graph causing overshading

be plotted.

The parts of the faces that are visible form polygons on the screen that are delimited by visible segments. Hence we wish to partition the screen into polygons with the visible segments. This means to produce an explicit list of polygons. If there are NS visible segments, this takes time $T = \Theta(NS*\log(NS))$ since it is essentially a sorting operation. Here we are dealing with the projected segments again.

This explicit list of polygons is insufficient if the planar graph is disconnected and has components inside polygons of other components. These cases must be marked since if polygon P contains component C without that fact being recorded, when P is shaded all of C will be overdrawn as shown in Figure 3-5. These inclusions can be detected with a generalized point-in-polygon routine such as given in chapter 2. Next one solution is to connect the disconnected components with extra segments that will not be drawn, so as to make the graph connected. Another way is to calculate the tree describing the inclusion relations. The tree nodes are connected components and component C_1 is a son of C_2 iff C_1 is immediately contained (without any intermediate

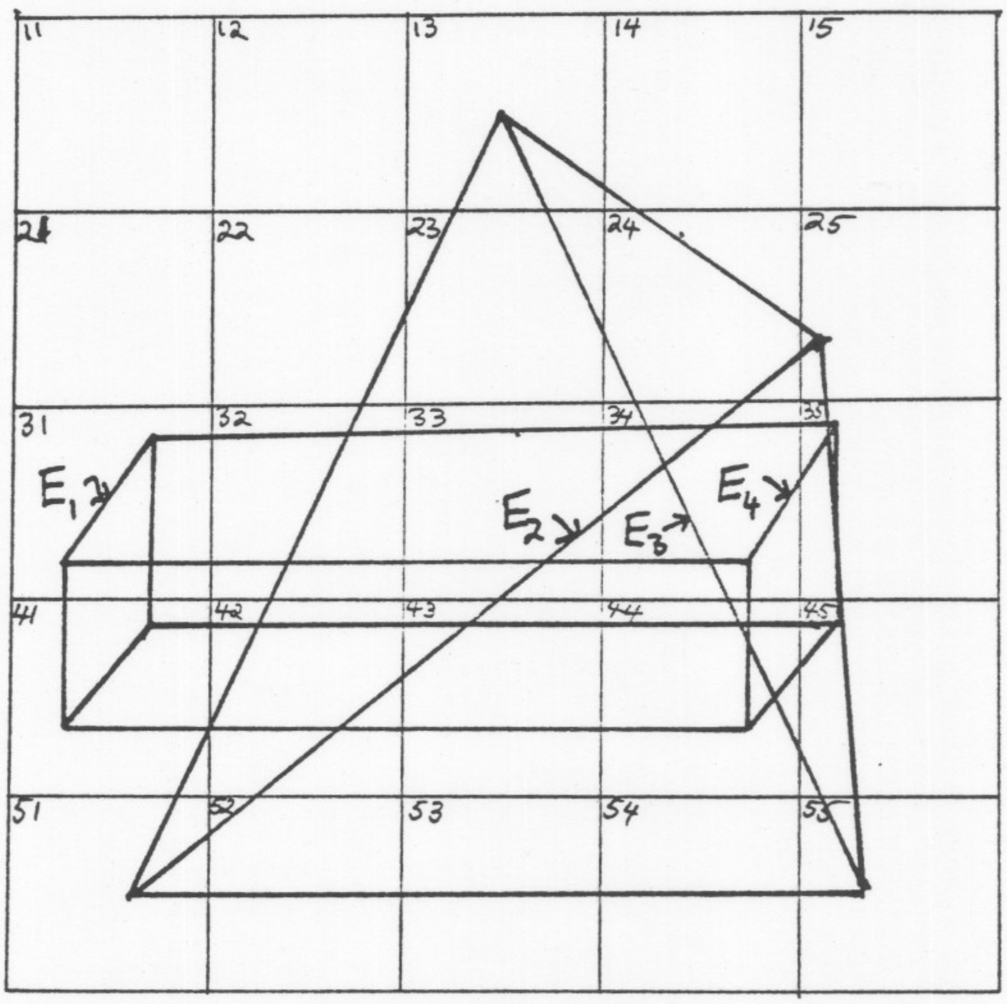


Figure 3-6: Finding edge intersections with a variable grid

components) inside C_2. Then when a polygon of C_2 is shaded, any components inside it can be excluded.

Now each polygon corresponds to only one visible face. However, one visible face may have two or more polygons since it may have several disjoint visible parts.

1. To shade a polygon, Q, it is necessary to know which face Q corresponds to. Since all points of Q correspond to the same face, choose a representative point, P. Either the centroid of Q or the centre of a box enclosing it are natural choices. However if Q is not convex, they may not be inside it. Nevertheless, if all the nonadjacent vertices of Q are joined, at least one of those lines will fall totally inside Q. So if a point-in-polygon test fails with the centroid, the midpoints of these lines can be tried in turn until one is found that is inside Q.

3.4.3 Finding Edge Intersections

3.4.3.1 Method -

The problem of determining which of the possible edge intersections actually occur is a form of relational database problem. Each edge can be considered to be a relation, that is a set of ordered pairs (x_i,y_i). Two edges that intersect are equivalent to two relations having a non zero intersection. Finding all intersections is equivalent to retrieving all the records or ordered pairs that satisfy simultaneously any two from a set of relations. In the edge intersection problem however, each relation is satisfied by an infinite set of records (all the points on the edge).

Finding which edges intersect is also related to the partial match retrieval problem, Rivest [1976], [1974]. The partial match retrieval problem concerns retrieving all the records that satisfy a certain criterion from a set. It is more general than retrieving the i-th record (array lookup) or the record with key #i (hashing). In partial match retrieval, an n-bit key is given, but only k of the bits are specified and it is desired to retrieve all the records whose keys match the target key in the k specified bits and have anything in the (n-k) remaining bits. The naive way is to simply hash and check all 2^{n-k} possibilities but there are faster methods that involve dividing key space into "buckets" or groups and hashing each record by its group number. Then all the satisfactory records can be retrieved by reading a small number of buckets (much smaller than 2^{n-k}).

It was the partial match retrieval buckets that provided the idea for the data dependent grids that are the basis of the fast object space algorithm given in this chapter.

Since the number of edge pairs is $\Theta(NE^2)$, while the number of intersections is $\Theta(NE^{4/3})$, no method of comparing the edges pair by pair can be asymptotically linear in the number of intersections to be found. There are various fast pretests such as first testing whether the pair of edges overlap in both X and Y coordinates before doing the detailed calculation. This causes only a constant speedup in the time, and speeds the algorithm up at all only if it causes a sufficient number of edge pairs to be rejected. Something better is needed.

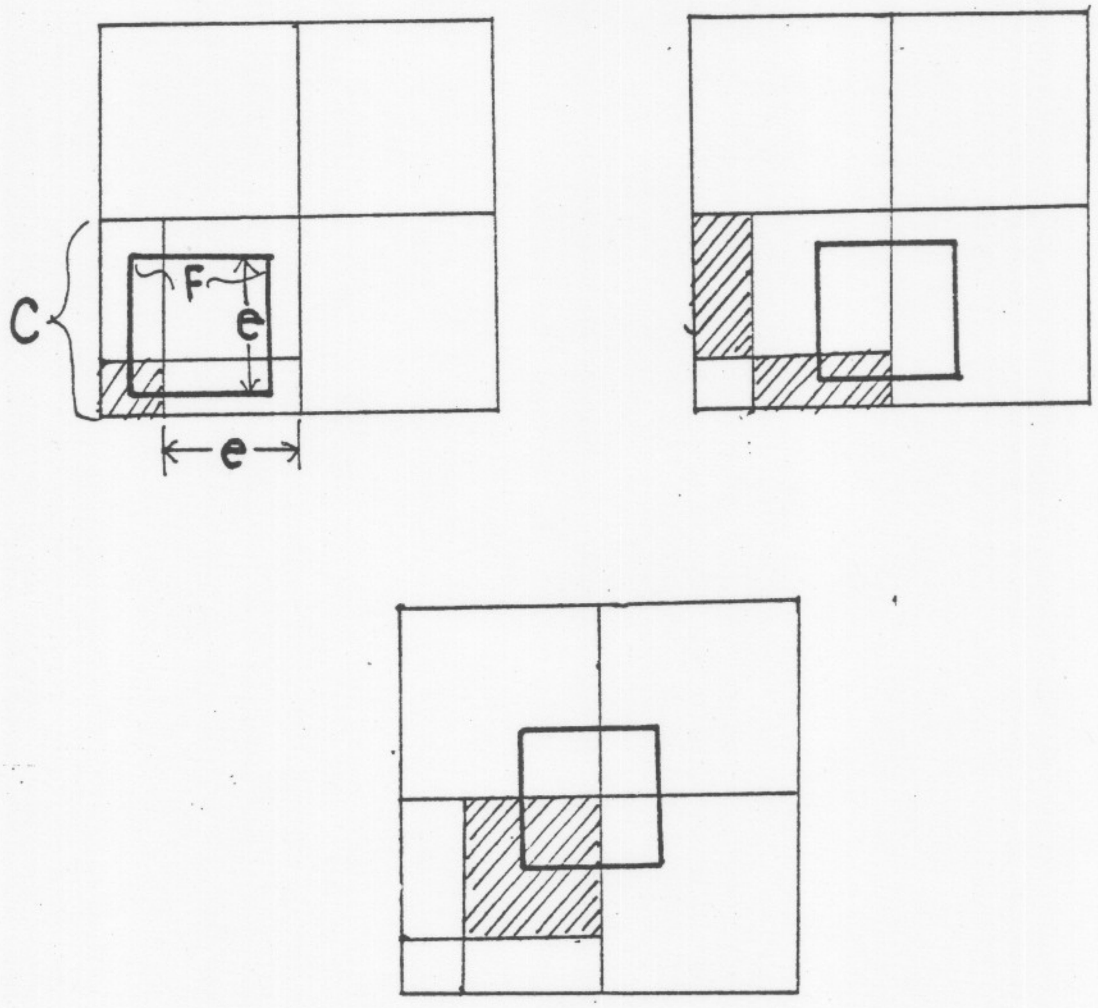


Figure 3-7: The number of grid cells hidden by a face

One possibility would be to use the k-d and quad trees of Bentley [1975a] and [1975b] and Finkel [1974]. These trees are generalizations of the traditional 1-D binary tree. They can be used to partition multi-dimensional key spaces to find matches and neighbours. However they are difficult to implement and inefficient for small databases. So I decided to use a <u>variable grid</u>.

Therefore do the following to find edge projected intersections, as shown in Figure 3-6:

1. Divide the screen into a grid of GE by GE cells where GE varies with NE in a way to be determined later.

2. For each edge E, calculate the cells $\{C_i\}$ that E passes through.

3. Write a file of ordered pairs (C_i, E_j) giving these inclusion relationships. As each E is processed in order, its ordered pairs will be written.

4. Sort this file by cell number.

5. For each cell, read into memory those ordered pairs of edges passing through it. Test all the pairs of edges for intersections. Since a pair of edges that intersects must

do so in some cell, this finds all intersections.

6. As the intersections are found, write them out in a file for future processing.

Thus in Figure 3-6, edges E_1 and E_2 are never tested for intersection since they never fall in the same cell. Since E_2 and E_3 both pass through cell 34, they are tested and do actually intersect. E_3 and E_4 are also tested but this pair does not intersect.

3.4.3.2 Notes -

1. This division of the screen bears a surface similarity to Warnock's algorithm but is actually quite different since here the division is fixed instead of being subdivided recursively by the data, and here the intersections are found exactly instead of just to the accuracy of a pixel.

2. The grid size need not be constant across the screen but might take advantage of the greater density of edges in the centre. However

1. This would produce only a small constant factor improvement, and

2. Calculating which cells a given edge fell in would be much slower.

3. An intersection between two edges, one of which is in general farther from the viewpoint than the other, is only useful to the farther edge. This is because the farther edge can be hidden by a face adjacent to the nearer edge but not vice-versa. If this fact is ignored, the nearer edge will be split into an extra unnecessary segment that must be tested for visibility. On the other hand, testing the distances of the edges at the intersection also takes time. Which effect predominates depends on the detailed implementation of the algorithm.

3.4.3.3 Timing -

Consider first the qualitive relationship between GE and the time it takes to determine edge intersections: If GE is large (cell size small): The cells themselves use no storage unless they contain edges so there is no overhead in this respect from making the cell size small. However there will then be more (cell, edge) pairs to write and sort.

If GE is small (cell size is big): There will be fewer pairs, but each cell will have more edges all of whose pairs will have to be tested. Also, a smaller fraction of those pairs will actually intersect. In the limit as GE->infinity, the only cells with more than one edge will have two edges that intersect. On the other hand, if GE=1, we are back at the case of worse than quadratic growth.

Let A = number of cells an edge of length L is in.

= Θ(max(GE*L,1)) since it is in at least 1 cell.

Then B = total number of (cell, edge) ordered pairs

= NE*A

= Θ(NE*max(GE*L,1))

= Θ(max(NE*GE*L,NE)).

And T_1 = time to calculate, write and sort (cell, edge) ordered pairs

= O(B*log(B)).

These B ordered pairs are distributed among GE^2 cells for an average of

C = B/GE^2 pairs per cell.

$$= O(\max(NE*L/GE, NE/GE^2))$$

Let P_i = probability that a cell has i edges in it. Because of the assumption that the edges are independently distributed among the cells, P which is the distribution of the number in any given cell, is Poisson distributed with mean C. This is because whether or not any given cell contains any given edge is independent of any other edges that might be in that cell. The average number of edge pairs per cell is the sum of $i(i-1)P_i/2$ from 1 to infinity. This sum is $\Theta(C^2)$. Thus the time to test each the edges of each of the GE^2 cells for intersections is $\Theta(C^2)$ for a total time of

$$\begin{aligned} T_2 &= \Theta(GE^2*C^2) \\ &= \Theta(\max(NE^2*L^2, NE^2/GE^2)) \end{aligned}$$

The total time to find the edge intersections is

$$T = T_1 + T_2$$

and it is desired to choose GE as a function of NE to minimize T. Now there are two cases depending on whether GE $< L^{-1}$ or not.

Case 1: $GE < L^{-1}$

$$\begin{aligned} B &= \Theta(NE) \\ T &= \Theta(NE*\log(NE) + NE^2/GE^2) \end{aligned}$$

This is minimized when GE is as large as possible, that is when

$GE = L^{-1}$ which gives

$T = \Theta(NE*\log(NE) + NE^2*L^2)$

Case 2: $GE > L^{-1}$

$B = \Theta(NE*L*GE)$

$T = \Theta(NE*L*GE*\log(NE*L*GE) + NE^2*L^2)$

which is minimized by minimizing GE, that is for

$GE = L^{-1}$ as before.

So in either case, $T = \Theta(NE*\log(NE)+NE^2*L^2)$.

But NX = expected number of intersections

$= \Theta(NE^2*L^2)$

so $T = \Theta(NE*\log(NE)+NX)$.

Thus for any statistical measure such that NX grows at least as fast as NE*log(NE),

$T = \Theta(NX)$

which is certainly optimal.

For the special case mentioned before, $L=NE^{-1/3}$ so

$T = \Theta(NE^{4/3})$

3.4.4 Splitting The Edges Into Segments

Now these intersections can be used to split the edges up into segments that are each either wholly visible or else wholly hidden. The method is:

1. As each intersection point is found, say between edges E_i and E_j, write two ordered pairs (E_i, E_j) and (E_j, E_i) to a temporary file.

2. Sort this file by the first member of each pair.

3. For each edge, E, in order, read in the ordered pairs with that edge as the first element.

4. Calculate the 2-D intersection points between E projected and the intersecting edges, projected.

5. Sort these points along E.

6. "Deproject" each point, P, to the 3-D edge by finding the intersection of a ray from the viewpoint through P and the line E, as shown in Figure 3-3. Because of roundoff error, these two lines in 3-space may not intersect exactly in which case the point on E closest to the ray is sufficient.

7. Use the 3-D intersection points to split E into segments.

8. Write out a file of segments $\{S_i\}$.

3.4.5 Determining Visibility Of The Segments

3.4.5.1 Introduction -

The naive way is to compare each segment against all the faces to see whether any hide it. But with $\Theta(NE^{4/3})$ segments and $\Theta(NE)$ faces this could take time = Omega$(NE^{7/3})$. So the same device as before of splitting the screen into a grid is used. But this grid of GF by GF cells is a different size than the GE by GE edge grid because a different quantity is being optimized.

3.4.5.2 The Algorithm -

1. Split the screen into a grid of GF by GF cells, where GF will be determined later.

2. For each face, F_j, determine which cells C_i, F_j intersects, that is which cells F_j falls at least partly in.

3. Write a file of the ordered pairs (C_i, F_j) determined in step 2.

4. Sort this file by cell number.

5. Within each cell, deproject the centre point of the cell onto the planes of all the faces in that cell. Sort the faces in the cell by the Z coordinates of those deprojected points. The greater the Z coordinate the closer the point is to the viewpoint. The deprojected point for a face may or may not be inside the face; this is immaterial.

6. For each cell, consider all the faces in it: If any completely covers the cell, delete in that cell only, all the faces completely behind it. An easy test to determine whether face F_1 is behind F_2 is to consider the intersections of the four corners of the cell projected onto each face plane. If F_1 is behind F_2 at these four points then it will be behind it at every linear combination of them, i.e. everywhere in the cell. (However F_1 may be in front of F_2 somewhere else in another cell. This is alright.) This test is sufficient but not necessary since it

misses the case where F_2 covers the cell and F_1 does not and although F_1 is behind F_2, F_1 extended over the whole cell is in front of F_2 at some point. Whether it is worth making this test exact and eliminating some more faces from the cells depends on the exact timing in the implementation.

7. Put this file into a form where all the faces in a given cell can be found quickly:

   1. If memory is available, read the file into core. It is unnecessary to store the ordered pairs explicitly; a list of face numbers for each cell is equivalent and more compact.

   2. If the file is too large, arrange it on disk in some convenient tree structure. The desired operation is the retrieval of varying length records by key. Since published random access methods generally require fixed length records, each record for the set of faces in a given cell can be split into several fixed length records. Since there will be only retrievals, no insertions and deletions, the tree can be optimized.

8. Read each segment, S_i, in sequence from the segment file.

9. Find the midpoint, P, of S_i.

10. Determine which grid cell, C_i, P falls into.

11. Compare P against every face, F_j, in C_i to see whether the face hides P by:

    1. Testing whether P is behind the plane of F_j, and

    2. testing whether P projected is inside F_j projected.

12. If P is visible, plot S_i and add it to a file of visible segments.

3.4.5.3 Timing -

Assume the faces are squares of side L with horizontal and vertical sides (that is not obliquely oriented). Rectangular and oblique faces would affect the results by a constant factor only.

Let $L = GF^{-1}(N+e)$ for nonnegative integer N and $0 \leq e < 1$. Then a face, F, covers $(N+1)^2$, $(N+1)(N+2)$, or $(N+2)^2$ cells with probabilities:

$$P[(N+1)^2] = (1-e)^2$$
$$P[(N+1)(N+2)] = 2e(1-e)$$
$$P[(N+2)^2] = e^2.$$

To see this, consider Figure 3-7. Here, N=0 and e=0.7. Assume without loss of generality that the lower left corner of F is in cell C. Then in the first case, F occupies one grid cell if its lower left corner is in the shaded (1-e)(1-e) subsquare of C. In the second case, F occupies two cells and in the third case four.

Thus ACF = average number of cells covered by each face is obtained by summing the above so

$$\begin{aligned} ACF &= (N+1+e)^2 \\ &= (L*GF+1)^2. \qquad \text{(eqn 3-3)} \end{aligned}$$

Let TFP = total number of (cell, face) ordered pairs

$$= NF*ACF.$$

Let AFC = average number of faces per cell, for the moment not deleting faces that are completely behind another face that covers the whole cell.

$$\begin{aligned} AFC &= TFP / GF^2 \\ &= NF*ACF/GF^2 \\ &= NE*ACF/GF^2 \qquad \text{since } NF=\Theta(NE) \end{aligned}$$

The times here are:

T_1 = time to calculate the faces in the cells

= time to sort the TFP (cell, face) ordered pairs

$= \Theta(TFP*\log(TFP))$

$= \Theta(NE*ACF*\log(NE*ACF))$ (eqn 3-4)

and T_2 = total time to test all the NE^2L^2 segments for visibility, assuming each segment is compared against all the faces in that cell.

$$\begin{aligned} T_2 &= \Theta(NE^2*L^2*AFC) \qquad \text{(eqn 3-5)} \\ &= \Theta(NE^3*L^2*ACF/GF^2) \\ &= \Theta(NE^3L^4+2*NE^3L^3/GF+NE^3L^2/GF^2) \end{aligned}$$

Finally T = total time

$= T_1 + T_2$ (eqn 3-6)

Now T_2 is minimized when GF is chosen so that T_2's second and third terms grow no faster than its first term.

Therefore GF = Omega(L^{-1})

So $T_1 = \Theta(NE*\log(NE))$

and $T_2 = NE^3L^4$

For the sample statistics used before,

$L = NE^{-1/3}$ (eqn 3-2, repeated)

so $T = NE*\log(NE) + NE^{5/3}$

This result is slower than the time taken to determine the edge intersections so it will dominate the total time. The problem is that making the face grid finer doesn't reduce the number of faces per cell which in this case is still $\Theta(NE^{1/3})$ as GF→infinity. In contrast, the number of edges per cell tended to zero as GE->infinity. This is why the algorithm has the refinement (section 3.4.5.2, points 4 and 5) of sorting the faces in each cell and deleting those behind any face that covers the whole cell. How great an improvement does this give? The next two pages of calculations will determine this.

Assuming, as before, that the faces are squares of side L that are oriented orthogonally,

ACF = average number of cells covered by each face

$= (L*GF+1)^2$ (eqn 3-3, repeated)

By a calculation similar to that used to calculate ACF, ACC

= average number of cells completely covered by each face

$= (L*GF-1)^2$

Thus of the AFC faces in each cell, a fraction

$r=ACC/ACF$ (eqn 3-7)

of them completely cover the cell. Since the faces are assumed to be the same size, the faces in a given cell that completely cover it are randomly distributed among all the faces in that cell, when those faces are sorted. Thus it is easy to calculate the expected number of faces encountered before the first face to completely cover the cell. Assume that AFC >> 1 and thus L*GF >> 1.

Let AFC′ = average number of faces per cell, up to the first face completely covering the cell.

$= 1/r$

$= ACF/ACC$ (from eqn 3-7)

Note that the total number of faces in the cell (AFC) is irrelevent; only the fraction of faces that completely covers the cell matters.

Thus AFC′ = $(L*GF+1)^2/(L*GF-1)^2$

But so long as GF = Omega(L^{-1}), AFC′ has an upper bound and so

T_2 = total time to test all NE^2L^2 segments for visibility

$= \Theta(NE^2L^2)$ (eqn 3-8, from eqn 3-5)

Now $ACF = (L*GF+1)^2$ (eqn 3-3, repeated)

$= \Theta(L^2GF^2)$

so, substituting equations 3-5 and 3-8 into 3-6,

$T = \Theta(NE*L^2*GF^2*\log(L*GF)+NE^2L^2)$

$= NE^2L^2*\Theta(GF^2*\log(L*GF)+1)$

This is minimized by any GF such that $GF^2*\log(L*GF)$ doesn´t grow faster than 1, or if this is impossible by minimizing GF.

If $GF = \Theta(L^{-1})$

then $T = NE+NE^2L^2$

$= NE^2L^2$ since L = Omega(NE)

Thus we have shown that the refinements given in section 3.4.5.2, points 4 and 5, allow us to determine the visible segments in time linear in the number of edge intersections, which is also linear in the number of segments.

3.4.6 Shading The Polygons

Once the visible segments have been found, they can be used to partition the plane into polygons. The faces corresponding to these polygons must be found. That is, if a ray is extended from the viewpoint through any point in the polygon (which is in the perspective plane) to infinity, of the faces it intersects (which will be different depending on which point in the polygon is chosen), then the same face will always be first. This is the face that the polygon corresponds to. This face´s properties such as angle of inclination relative to the light and texture will be used to shade the polygon.

Find the face corresponding to the polygon P thus:

1. Pick a point, X, in P. If P is convex, the centroid is sufficient but if not then it may be necessary to join all pairs of nonadjacent vertices and test the midpoints of those lines to find an acceptable point.

2. Determine which face cell X is in.

3. Test the faces $\{F_i\}$ in that cell. For each F_i that contains X inside its projection onto the perspective plane, extend a ray from the viewpoint through X to the plane of F_i and find the distance from the perspective plane to the face plane along it. The F_i with the smallest distance is the desired face.

Since the $\Theta(NE^2L^2)$ visible segments produce $\Theta(NE^2L^2)$ polygons and each takes constant time to find the face for (by reasoning the same way as in the preceding section), this takes time

$$T = \Theta(NE^2L^2)$$

3.4.7 Overall Time

Thus the total time for this algorithm, adding the times to find the edge intersections, test the segments for visibility, and find the faces corresponding to the polygons, is

$$T = \Theta(NE^2L^2)$$

(assuming L = Omega(NE*log(NE))) and using the example statistics given before,

$$T = \Theta(NE^{4/3})$$

3.5 IMPLEMENTATION

The hidden line part of this algorithm has been implemented in a 12ØØØ line Fortran program, VIEWPLOT on the PDP-1Ø. It is described in Appendix A. Section A.1 contain a brief summary of the program. Section A.2 is a logic manual giving a routine by rutine description of VIEWPLOT. Section A.3 is a users´ guide.

The program logic manuals for VIEWPLOT and PRISM-MAP in the appendices illustrate some of the problems that arise when graphics algorithms are implemented. Numerous messy special cases must be considered. The problem is exacerbated by the finite precision of floating point numbers. For instance two lines may intersect but still be nearly enough parallel that the determinant of the equations is nearly zero. Then the standard equation for the intersection point gives a floating overflow error. In this case, I project the two edges´ four endpoints onto an axis, sort them to identify the middle two and then use the average of those two as the intersection point. Again, a routine that decides whether a point is in a polygon may return that a point that is near the perimiter is inside at one time but outside if everything is rotated about the origin because both the point and the polygon vertices will have small errors now. A vertex of a polygon in 3-D that satisfies the polygon's plane equation no longer satisfies it exactly after both the vertices and the equation are transformed perspectively. Addition and multiplication do not associate. On poorly designed machines such as IBM-370, multiplication and division are not exactly inverse operations. These last two points can cause errors with an over-optimizing optimizing compiler. I generally do not require that calculations be exact to the last bit, only that they be reproducible to the last bit.

There are also numerous low level problems caused by differences between machines, restrictions of Fortran, etc. Both these algorithms are implemented in standard Fortran so that they are transportable.

These sorts of problems have kept the art of implementation, not only in graphics but also in other areas of computer science, from becoming a science.