5-28-1985

BRUCE: A Graphics System with Hidden Line and Hidden Surface Algorithms

Keith Vetter
Dartmouth College

Christopher Roche
Dartmouth College

BRUCE: A GRAPHICS SYSTEM WITH HIDDEN LINE AND HIDDEN SURFACE ALGORITHMS

Keith P. Vetter and Christopher C. Roche

Technical Report PCS-TR86-123

Table of Contents

Abstract

Accurately representing the physical world by computer is a topic which has direct benefits to fields like chemistry and architecture, and is a source of much research in computer science. This paper examines the steps necessary to develope and implement a graphical system that will allow for the modeling of physical world objects. In particular, this is a description of BRUCE: a graphical system that will describe a world of three dimensional polyhedra, implementing algorithms for hidden line and hidden surface removal. This paper also deal with the problems incurred along the way and suggestion for further improvement of BRUCE.

1. Introducion

1.1 The Problem and Initial Goals

Our interest in computer graphic system comes from an article in the September, 1985 issue of Scientific American entitled "Computer Graphics." We marveled at the highly realistic and detailed drawings of complicated real life scene like jet fighers and panoramic vistas depicted in the displays. The project set out to create a system which would allow the Dartmouth College user the same privilege of creating and manipulating high resolution, three dimensional pictures like the ones in the magazine. With great expectations, and great inexperience in graphics skills, BRUCE began to take form. And as we researched, we defined our goals.

i. Learn About Graphical Systems: We were starting from scratch, both in terms of our experience and in terms of what was available at the college. Part of the experience of this adventure would be the wealth of knowlege gained from delving into the problem.

ii. Create A Graphical System: To draw these advanced pictures, a graphics system would have to be built. This system would give the user a simple and consistent method for creating and handling graphical data. We would try to make the system as complete as possible and afford the user many options. The system would also allow for later expansions.

iii. Remove Hidden Lines: One of the given features in a complete graphics system is the ability to remove hidden lines from a wire frame drawing. This is a powerful feature, and one that would take some time to implement.

iv. Remove Hidden Surfaces: This is another feature employed by

a complete graphics system which provides the basis for real three dimensional representation of solids. This, like hidden line removal, is a complicated problem .

Initially, we sought to create a shading algorithm, as another goal. Shading is where the realism comes into graphical solid modeling. The most realistic of pictures have quite complex shading techniques to simulate the play of light upon the surfaces of the depicted objects. However, as we got further into the project time grew short and our hope that we would be able to get a terminal with suitable graphic capability had been dashed. Still, BRUCE is planned to be capable of advanced renderings, and in our improvements we will discuss the possiblity of implementing a shading algorithm to the system.

2. Constraints

BRUCE is subject to numerous constraints, both environmental ones and self-imposed ones. The environmental constraints were constraints beyond our control. They affected the direction of development for BRUCE, forcing us to abandon some goals and to adopt others. Self-imposed constraints were ones we imposed while defining the problems BRUCE would address.

2.1 Environmental Constraints

By far the most limiting environmental constraint is the lack of a good graphics terminal. To do hidden line algorithms we only need a black and white graphics terminal. To do hidden surface removal we need a simple color terminal. To do shading we need a color terminal that supports a whole range of colors with various saturation, hue, and intensity.

Dartmouth College has three main types of graphics terminal, none of which, unfortunately, are very good. The HP2648a does not support color. The Tek4006 is a black and white vector terminal making it good for drawing lines but unsatisfactory for filling in areas. The Dec GIGI has color but it has a severe problem with bleeding.

We also looked into the graphics terminals at Thayer School of Engineering. While some of these terminals have better quality, none of them are connected to the Kiewit Network or have been made compatable to the graphics on DCTS 1. Furthermore, Thayer has a heavy load for the use of these terminals for engineering courses and tends to frown upon

people using their computer facilities for non-engineering work.

The third possibility for a graphic terminal that we looked into was the Apple Macintosh. This has the advantage of being independent of Dartmouth College and thus useable anywhere. The Macintosh has two offsetting disadvantages. First, the development environment on the Macintosh is poor; more time would be spent learning "Inside Macintosh" than actually working on BRUCE. Second, the Macintosh is black and white, not color making it no better than the HP2648a. We looked at the possibility of using various shades of grey to simulate color. This idea was rejected because we could only get a small number of different shades and it would be insufficient for the shading problem.

The terminal problem still is not solved. The solution to the hidden line problem could be fully displayed, the hidden surface problem only partially, and the shading problem almost not at all. To get display the results of the hidden surface algorithm we use different linestyles instead of different colors. As a result we decided to tackle on the hidden line and hidden surface problems, to think about but not implement the shading problem, and to add utilities to make BRUCE a graphic system.

Another external constraint is the lack of expertise on three dimensional graphics at Dartmouth College. We had to learn the topics as we developed them. If we hit a thorny problem there was no one to whom we could turn and ask for help. Furthermore, the library contained few books on the subject -- we had to get most of our references on inter-library loans which were slow in arriving.

A third constraint on BRUCE is the limited time we had to design and devolop BRUCE. Since we knew about this before hand, we decided to work

on the project in a familiar developing environment. That is why we choose to use DCTS and the Basic 7/8 graphic routines. This has the added advantage of making BRUCE useable on any graphics terminal supported by the system.

2.3 Self-imposed Constraints

The second type of constraints are the ones we placed in defining what BRUCE should do. There are two types of objects can be depicted in three dimensional graphics: surfaces e.g. F(x, y) = x^2 + y^2, and solids. We decided to do the latter.

Specifically the types of solids we handle are polyhedra--faceted solids with polygonal faces. Requiring objects to be faceted means that objects with curved surfaces like spheres and cylinders have to be approximated. Solutions to the problem of representing curved surfaces is currently one of the main areas of research in three dimensional graphics and is beyond the scope of this project.

Another constraint is that the polygon composing the faces of the polyhedra must be convex. This means that any line intersecting the polygon can intersect in at most two places. This constraint is fairly common since it is not really a limitation. Concave faces can easily be decomposed into convex ones by adding auxilliary edges.

A third constraint is that we do not allow penetrating objects, the intersection between any two polyhedron must be empty. Handling penetrating faces is a complex problem with no good soluion. The commonest approach is to decompose the solids into smaller ones

eliminating the overlapping parts. This turns the problem into one of detecting overlapping and determining where it overlaps.

A final constraint is a software one. Because we must declare array sizes at compiler time the maximum number of vertices, faces and edges of a face for any object are limited to the literal values max_v, max_f, and max_c respectively. If the need ever arises we can change the value of any of these and recompile.

3. Graphics System

3.1 Overview

A graphics system is a collection of procedures designed to ease the input, output and manipulation of graphical picture. One of the goals of BRUCE is to produce a simple yet powerful graphics system. We want a beginner who knows nothing about three dimensional graphics to be able to use BRUCE and get useful results. All the parameters involved in three dimensional graphics ranging from the viewport and the projection screen size to the eye position and the center of interest are set for him. A small subset of commands (table 3.1), which are easy to learn, gives a user a lot of power. At the same time we have over thirty commands (table 3.2) for the expert to precisely control the settings to get the exact result he wants. For both the beginner and the expert there is an extensive built in help command which provides explanations for every command and some key concepts.

Get objects:	Create	Load	
Display objects:	Draw	Hline	Surface
Manipulate objects:	Shift	Rotate	Scale

Table 3.1 Elementary commands

The graphics system consists of three parts: the command parser, the commands and the data structures. The command parser handles the user interface, interpreting the input supplied by the user. The commands are the actions that BRUCE can do. The data structures is how the objects are

represented internally. A poor data structure can severely limit what can be done while a good one can make some tasks easy. The advantages of some of the better data structures must be weighed against the larger memory requirements. The three sections are discussed below.

3.2 Command Parser

The main procedure of BRUCE consists of a tight loop of getting a command, looking it up, and calling the appropriate routine. All the routines called from the command parser take no arguments. It is up to the called procedure, usually a small buffer routine, to set up the variables, files, or whatever as necessary for call to the actual application procedures. This allows the parser to be easily expanded since it is independent of the details on the lower levels.

Adding a new command is very simple. After the appropriate routines for doing the commands have been written, one only needs to add the command word to a list of commands and add another case statement in the command parser. (Ideally one would also create a help file for the command also, and update the help file for commands.)

All input from the user is done through one procedure, get_token. Get_token maintains an input buffer, and returns the first word off the buffer. Tokens are deliminated by spaces, commas, and semi-colons which all have equal precedence. When the buffer is empty, the user is prompted for more input.

Some commands like DRAW take an optional argument. For DRAW to determine if it sould draw ALL the edges or only the VISIBLE ones, it looks

ahead in input buffer for the first token. If the token is oneof the two keywords then DRAW removes it from the buffer by calling *get_token* If the token is not an argument then it leaves it on the buffer for whomever needs it.

This method of handling input has two advantages. First it allows multiple commands per line. Second, it allows commands to be broken over several lines.

The command parser also has a nice facility for handling errors. If a procedure detects an error such as bad input or undefined file, it calls the procedure error and passes an appropriate error message. Error prints out an appropriate error message, flushes the input buffer then signals condition red (I couldn't resist calling it that). The error condition gets trapped by the main procedure which automatically clears the stack frame and returns control to the command parser. This avoids the need of a global error flag and conditionals checking for errors. Another nice feature this allows is for the user to abort a command at any point by typing ABORT at any time. When get_token sees that word it calls error who then returns control back to the command parser aborting the command.

3.3 Commands

Table 3.2 gives a list of all the commands that BRUCE recognizes. The goal here is to have a command set that is simple, consistent, and complete (Newman and Sproull, pp 80-81). Simplicity is achieved in two ways. First there is a small subset of powerful commands (table 3.1) that are easy to learn and with which the user can get useful results. Second, the more

complicated commands has their syntax based upon the way one would say the command in english, ie. to create a cube called my_cube which is centered one uses the command **Create cube my_cube centered**.

A consistent system is one that behaves in a predictable manner. Bruce provides consistency in three ways: its command syntax, its error handling, and the coordinate system. First, all similiar commands have the same syntax. For example, one **shift/scales/rotate** an object with a given **name** with a given **transformation**. Second, all error handling is done by one procedure which prints out an error message, clears all pending commands and returns control to the command parser. Lastly, in graphics applications there are always two coordinate systems to choose between: problem coordinates and screen coordinates. All commands that require numbers in one of the two systems use problem coordinates. The single exception being the viewport command which cannot be done in problem coordinates since it defines the extent of the screen coordinates.

The third aim is completeness. This differs from comprehensiveness in that the system need not provide every imaginable facility but to rather not have omissions in a set of similiar functions. Because it is easy to add new commands, BRUCE has a fairly complete command set as can be seen in table 3.2.

Following is a list of all the commands and what they do. They fall into six categories depending on their effects:

Display: these commands affect the display environment.

Viewport l,r,b,t: defines the extreme left, right, top and bottom of the device viewport which is where on the monitor screen the graphical

output gets placed. The coordinate system is normalized to having the short side of the screen equal to 100 and the longer side equal to 100*aspect ratio. The viewport is initially set to the largest square that can be fit into the upper right hand corner.

Screen l,r,b,t: specifies the distance in problem units of the center of interest from the left, right, bottom and top of the viewport. The screen is initially set to -2, 2, -2, 2.

Fullscreen: sets the viewport to correspond to the whole monitor screen.

Eye x,y,z: specifies in problem units the location of the observer's eye thus varying the perspective. The eye is initially set to (300,100,150).

Fix x,y,z: specifies in problem units the location of the center of interest. This defines which direction the observer's eye is looking. The center of interest is initially set to the origin.

Display commands:	Viewport	Screen	Fullscreen
	Eye	Fix	
Object commands:	Load	Create	Save
	New	Delete	Names
Drawing commands:	Draw	Hline	Surface
	Dotted	Fill	Clear
Transformations:	Shift	Scale	Rotate
Display controls:	Color	Linestyle	Frame
Miscellaneous:	Account	Stop	Pass
	Input	Pause	Examine
	Explain	Help	

Table 3.2 Summary of commands.

Object: these commands all manipulate the data about the objects.

Load <filename>: loads objects defined in a specified file. All

currently defined objects are first deleted.

Create <type> <name> [centered]: allows user to create a new object. The types of objects he can currently create are a cube, pyramid or tetrahedron all with sides of length 1. The object will be located with one vertex at the origin and the rest in the positive x, y, z direction unless centered is specified in which case the object is placed so that its center is at the origin.

Save <filename>: saves the current object definitions in a specified file. If the does not exist one is created else the file gets scratched.

New: deletes all currently defined objects.

Delete <name>: deletes the object with the given name.

Names: prints out a list of the names of all currently defined objects.

Draw: these commands display objects on the screen in some form.

Draw [all | visible]: draws the edges of all currently defined objects with no clipping. If all is specified then all edges will be drawn; if visible is specified then only edges on faces which are visible are drawn; if no option is given then the edges are drawn as they are.

Hline: performs the hidden line algorithm. Only the edges or parts of edges that are visible to the observer are drawn.

Fill: colors in all the visible faces in the scene with no hidden surface removal. No account is taken about who is in front if two faces overlap.

Surface: performs the hidden surface algorithm. Only the faces or parts of the faces that are visible to the observer are drawn.

Dotted [all | visible]: similiar to draw but with dotted lines.

Clear: clears the display screen.

Transformation: these commands all transform objects.

Shift <name> <x, y, z | back>: shifts the object with the given name by adding x,y,z to each vertex. If the name is *all* then every object gets shifted. If back is specified then the object will be moved so that it has one vertex on the origin.

Scale <name> x, y, z: scales the object with the given name by multiplying each vertex by scalars x,y,z. If the name is *all* then every object gets scaled.

Rotate <name> <axis> <degrees>: rotates the object with the given name counter clockwise around either x, y or z axis by the given amount in degrees.

Control: these command affect how drawings appear on the screen.

Color <number>: sets current drawing color to the color with the specified number.

Linestyle <number>: sets the current drawing linestyle to the one with the specified number.

Frame: draws a frame around the current viewport.

Miscellaneous: commands that do various things

Account <on | off>: If on then it prints out how many run time units the session used so far. Then, after each command it prints how many units that command took. Option off turns off this feature.

Stop: terminates the session.

Pass: forces the data to be sent through pass 1 and 2.

Input <filename>: redirects command input from a specified file. All input is echoed on the terminal. If an error occurs while reading from a file then the file is closed and input is taken from the terminal.

Pause: causes the program to stop, print a message and wait for the user to hit return.

Examine <name>: the data structure for the object with the given name is printed out.

Explain [topic]: invokes the on-line help system for the given topic.

Help [topic]: same as explain.

3.3 Data Structure

3.3.1 Modeling Objects

In our design of BRUCE we decided work with polyhedra -- solid objects with flat faces. These include simple objects like cubes, prisms and parallelepipeds. More complex objects can be modelled by combining a number of simpler ones. Curved surfaces can be approximated well by increasing the number of faces. Thus any solid object can be modelled. "This completeness property makes the polyhedron particularly attractive as a primitive representation." (Newman and Sproull, p. 301)

Polyhedron in turn can be modeled by its *vertices* and either its *faces* or its *edges*. A face is defined by listing the vertices that outline the face. An edge is defined by a start and a end vertex. For hidden line removal the edges are of prime importance. For hidden surface and shading, the faces are of prime importance.

Each face also has the qualities of an inside and an outside: the inside faces in towards the center of the object, the other faces out. To distinguish between the two one can either order the vertices in a clockwise order. An alternative method is to the normal vector to the face to specify which direction is out.

Another property of a face is its plane equation. This is given by four numbers (a, b, c, d) which satisfy the equation $ax + by + cz + d = 0$ for any (x, y, z) on the face. This is closely related to the normal since the normal vector is (a, b, c).

Much research has recently gone into representing curved surfaces

with such techniques as cubic splines, parabolic blending, Bezier curves and B-splines. Another area of research in three dimensional modelling is how to represent non-solid objects -- objects that have holes in them. This is beyond the scope of BRUCE but for a good reference on the topic see Rogers, *Mathematical Elements For Computer Graphics.*

Inherent in all these methods is the problem of data entry. Graphical scenes tend to get the data in. Graphic scenes tend to be very complex and entering data is tedious and prone to errors. Data-entry programs, a form of computer-aided design, and have been written. Again this is a problem seperate from what BRUCE is designed to handle. Listed are references to three different approaches:

Braid, I. C.: "Designing with Volumes," 2nd ed., Cantab Press, 97 Hurst Park Ave., Cambridge, U.K., 1974.

Baumgart, B. G.: "GEOMED: A Geometric Editor," *Stanford Univ. Comput. Sci. Dept.* AIM-232, STAN-CS-74-463, October 1974.

Sutherland, I.E.: "Three-dimensional Data Input by Tablet," *Proc. IEEE* **62**(4):64, April 1974.

3.3.2 Represention of Models

Every polyhedron has three classes of information: geometric, topological and auxiliary. Geometric is concerned with position and measurements of the points and object. Topographical is concerned with the structure of the object such as how the vertices define the faces. Auxiliary information is such properties as color, albedo and translucency.

From the geometric information one can get the topological information and vice versa. The form one chooses to store the information depends on the application.

We chose to store the vertices and the faces as a clockwise list of vertices and the color of each face. The data is stored in two ways, one for external use and one for internal use. The external method is how the user sees the data and how it gets stored. The internal method is how BRUCE stores the data and passes the information around.

3.3.2.1 External Format

Object name	
V x y z [sx sy d]	Vertex
F n v_1 v_2 v_3 ... v_n [color l r b t near far]	Face
[**B** l r b t]	Bounding box
[**D** near far]	Depths
End object	

Table 3.3 External data format

The external structure is shown in table 3.3. It is designed for easy use by the user who may want to create a new object by hand or edit an old one. Every object definition must start with **Object** followed by the object's name, and end with **End Object.** The first word of each line defines what type of data the line holds. If the input routine does not recognize the key word then the line is ignored. This allows new information to be stored with the data without having to change the input routines.

Currently four keywords are recognized. **V** stands for vertex data which is followed by the x, y, z position and optionally the screen coordinates and the depth of the point. The order of reading the vertices correspond to the order in which they are stored, hence the first one read is v_1, the second is v_2 and so on. **F** stands for face and is immediately followed by the number of vertices that define the corners of the face. After that comes a list of the vertices listed in a clockwise order. Following this are optional information including the color of the face, the left, right, bottom and top of the bounding box and the minimum and maximum depth of the face. The bounding box is the position of the smallest rectangle that encloses the image of the object on the screen. The bounding box and depth are used to prune objects that cannot obscure another. Both the **V** and **F** commands allow multiple definition per line provided that * is uses as the delimiter. The two other keywords are optional. **B** is the bounding box of the object. **D** is the depth for the object.

The *load* command reads in a file with objects in this format and converts it to the internal format. Similiarly the *save* command writes out objects stored in the internal format to a file in the external format. For editing and debugging purposes all the optional information is written out by the *save* command.

3.3.2.2 Internal Format

Internal data is stored in a structure shown in table 3.4. Included in this representation is room for much more information which BRUCE computes. Associated with each object is how many vertices it has, how

many faces, its maximum and minimum depths and its bounds. The bounds is left, right, bottom, top of the smallest rectangle which would enclose the object in screen coordinates. The depths and bounds are used to prune objects that cannot obscure another object. Associated with each vertex are its problem or world coordinates coordinates, its screen coordinates and depth or distance from the eye. Associated with each face is how many vertices define the face, the list of vertices, the color of the face, a visibility flag, its maximun and minimum depths and its bounds. The depths, bounds, screen coordinates and visibility are all computed by BRUCE from the other data.

```
dcl 1 Object,
      3 Name char (max_n) varying,          /* The object's name */
      3 Bounds (4) fixed,                   /* Bounding box (lrbt) */
      3 Depth (2) float,                    /* Min/max */
      3 Num_v fixed,                        /* How many vertices */
      3 Vertex (max_v),                     /* The points */
        5 Problem,                          /* Problem coordinates */
          7 (px, py, pz) float,
        5 Screen,                           /* Screen coordinates */
          7 (sx, sy) fixed,
        5 Depth float,                      /* How deep it is */
      3 Num_f fixed,                        /* How many faces */
      3 Face (max_f),                       /* Info for each face */
        5 Visible boolean,                  /* Can we see the face? */
        5 Color fixed,                      /* Its color */
        5 Bounds (4) fixed,                 /* Bounding box */
        5 Depth (2) float,                  /* Min/max */
        5 Number fixed,                     /* How many vertices */
        5 V (max_c) fixed;                  /* List of vertices */
```

Table 3.4 Internal data format

3.3.3 Data Passing

Since BRUCE is designed to arbitrarily complex, all the objects in the scene could not be kept in memory. Instead, the data is kept in files and objects are only read into memory when needed. At any time, no more than two objects are ever in memory at once. Information is passed to a routine by passing two files: an input file containing the current data, and an output file to hold the results.

The problem with using files is that accessing them is slow. Furthermore, storing the internal format of the information in a file usually requires a lot of converting of data types. The solution we found to this problem is to use unformatted files. In unformatted files, the information is stored in machine readable form as opposed to human readable form. The information is transferred in large blocks as opposed to line by line. This allows us to read or write the data structure in table 3.4 much more efficiently. The biggest advantage, however, is that it allows us random access to the objects. Suppose we want to read object number n. We know the size of the data for an object is fixed at s=wlen(object). So we just reset the file pointer to (n-1)*s + 1 and read the object. Thus, getting the last object in the file is as fast as getting the first, while before it would require reading from the beginning.

The drawback to this method is that it makes debugging harder. To see state of the data at any given moment, the information must first be converted back to human readable form.

4 What BRUCE Can Do

In this section we will run through an example of a simple session using BRUCE. The session will demonstrate how a novice can use BRUCE to create complex scenes with little effort. Only the elementary commands listed in table 3.1 are used. It will also explain in more detail some of the more importants and how BRUCE implements them. The figures referred to are located at the end of this section and were drawn on a TEK4662 plotter. The commands typed by the user will be in boldface.

When BRUCE starts running it prints a welcome banner, clears the screen and prompts the user for a command. The first thing the user must do is to get some objects to draw. To do this the user types: **create cube my_cube**. This causes BRUCE to create a polyhedron named my_cube (the name is checked for being unique) that is a cube. Currently, BRUCE can create cubes, pyramids and tetrahedrons. The created object has unit length sides with one vertex at the corner and the other vertices in the positive x, y, z direction. If the keyword **centered** is given after the name, then the object is shifted so that its center is at the origin.

After creating the object the user wants to see it, so he uses the **Draw all** command to get figure 4.1 which draws all the lines in the scene. If the user had typed **Draw visible** instead, he would have gotten figure 4.2 which only draws the faces on the front of the objects. The visible option forces the data to be processed through two passes. The **Draw** command without any argument draws the data at the current level a processing. To clear the screen before drawing, the user types the command **Clear**.

The first pass in processing the data computes geometric information.

Each vertex gets mapped to screen coordinates, the depths of the vertice, faces and object is computed and the bounding boxes for each face and object is determined. The second pass removes all faces which are on the back side of the polyhedron taking advantage of the object coherence of the polyhedron. Since the objects are solid, the faces on the back will be obscured by the object so they cannot be seen. Typically this halves the number of faces that have to be checked. The faces which remain have to be check further to determine if they are visible.

Now the user wants to make a more complex object. He can do this by combining more elementary polyhedron together. Suppose the user wants to put a point on the right side of the cube. This is accomplished by putting a pyramid with the correct orientation flush with the right side of the cube. First, he creates a pyramid with **Create pyramid my_pyr**. Next, the pyramid should be rotated so that it is pointing to the right by rotating it clockwise around the x axis 90 degrees. The command for this is **Rotate my_pyr x 90**. The user now types **Draw** to see what he has (figure 4.3). The pyramid has the correct orientation must be shifted into position. Since the cube has sides of length 1, we can tell from figure 4.3 that it should be shifted along the y axis 1 unit and along the z axis 1 unit and stay the same along the x axis. The user accomplishes this by typing **Shift my_pyr 0, 1, 1. Draw**ing it gives figure 4.4.

Now that the scene is composed correctly, the user wants to see it in final form without the hidden lines and surfaces. The **Hline** command draws the scene in a wire-frame outline with all obscured edges not drawn (figure 4.5). The **Surface** command fills in the visible faces with the color of the face (figure 4.6). Hline is described in more detail in section 4, and

Surface is described in section 5.

Suppose the user has to leave at this moment. Typing **Save fred** causes BRUCE to save the current scene in file fred. Fred is scratched before the information is written to it and will be created if it does not already.

When the user returns later, he types **Load fred** which will load in the data stored in fred. While he was away, he decided that he wanted the object pointing upwards, not to the right. He could accomplish this by using the **New** or the **Delete** command and starting over, or he could transform each indiviual object the appropriate amount. The easiest way, however, is to give the command **Rotate all x -90; Hline** to produce figure 4.7. By specifying all instead of an objects name, every object gets transformed.

Now the user wants to experiment with a more complex scene. He decides to place an object in front of what is already there. The command **Create tetrahedron my_tet; rotate my_tet z 50; shift my_tet 1 0 1.2; draw** produces figure 4.8. Using **Hline** yields figure 4.9.

The more complex figures produced by BRUCE were composed in this manner. Using simple polyhedron and the shift, scale, and rotate transformation, more complex scenes can be made. The more sophisticated commands allow more precise control over how the scene is drawn. For a complete list of the commands see section 3.3.

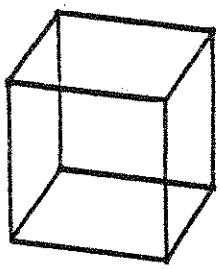


Figure 4.1

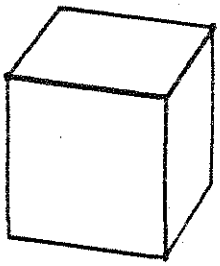


Figure 4.2

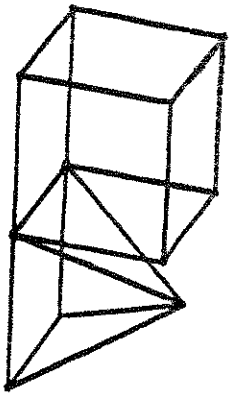


Figure 4.3

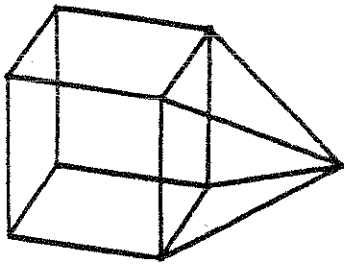


Figure 4.4

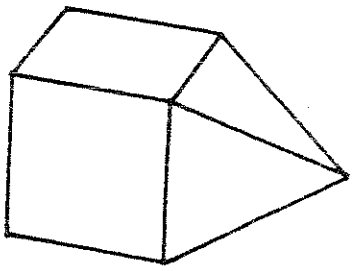


Figure 4.5

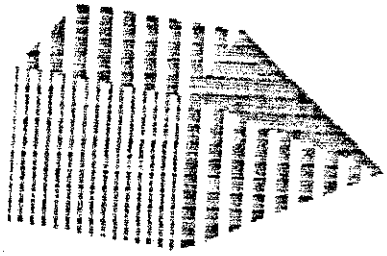


Figure 4.6

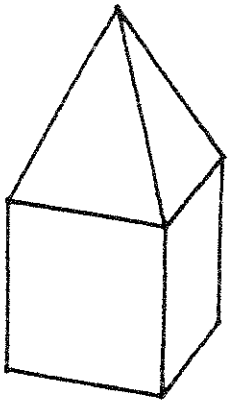


Figure 4.7

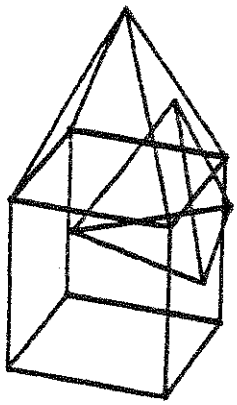


Figure 4.8

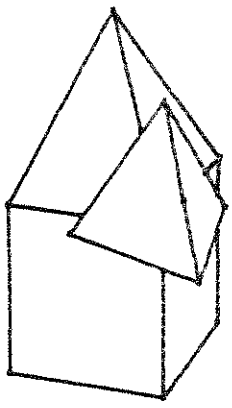


Figure 4.9

5. Hidden Line Removal

Hidden line removal algorithms draws the scene with only the edges or parts of edges that are visible. Edges that are totally or partially obscured must be eliminated. In the 1960's when most of the graphics terminals were vector devices, many different hidden line algorithms were developed. With the development of raster terminals, however, the emphasis has shifted toward hidden surface algorithms.

Hidden line algorithms are still important for two reasons. First, they are usually much faster than hidden surface algorithms. Because of this, many graphical systems use hidden line algorithms for editing and checking for correct data before drawing the final scene with the hidden surface algorithm (see Kunii, pp. 350-9). Second, hidden line algorithms can be done on any type of graphical terminal that is particularly important to us since Dartmouth does not have a graphics terminal to can draw surfaces well.

5.1 Existing Methods

The earliest algorithms worked by stepping a small increment along each edge and testing to see if anything intersected the line segment from the point to the eye position (Appel, pp. 287-293). With a small increment, this method is very precise but also extremely slow. The next improvement was by Loutrel who improved the method of detecting obscuring faces by using topographical properties of the solids.

Encarnacao's priority method (Giloi, pp. 168-172) differs from the earlier two in that it deals with clipped triangles instead of clipped edges.

Every face is decomposed into triangular segments by adding "auxiliary" edges. Then each triangle is checked against every other triangle to see if it is obscured. Encarnacao identified seven possible ways for two triangles to overlap which he reduced to four cases. The output of these tests is no longer triangles but the visible edges which are then drawn.

5.2 Our Method

Our method of removing hidden lines, which we will call Hline, is a generalization of the priority method. Instead of having to use triangles, you can use any convex polygon. Concave faces can be decomposed into convex faces by adding auxilliary edges. The reason for using convex polygons is that then one can guarantee that the edge will only intersect the polygon in at most two place.

Hline works on finding the visible edges of one polyhedron at a time. First, it creates a linked list of all the unique edges of the object. The information is stored in a structure shown in table 5.1. The structure contains room for information about the places of intersection with other faces. Next, for every edge in the list, it clips it against every face of every object including its own (since we allow concave polyhedra).

Because there may be a large number of objects and even more faces, we prune out most of those who do not interest us. For those faces which do not get pruned, we determine how the edge gets clipped. If the edge is totally obscured, we go do the next edge; if it is still totally or partially visible, we test it against the next face. After the edge has been tested against every face, the visible portion of it is drawn. When every edge of an object has been processed, we continue with the next object until we are

done.

```
dcl 1 Edge based,                      /* Information on an edge */
      3 Next pointer,                  /* Next in the linked list */
      3 Box (4) fixed,                 /* Bounding box */
      3 Depth (2) float,               /* Min/max */
      3 Which (2) fixed,               /* Which start and end vertex */
      3 Start_point like Object.vertex, /* Beginning */
      3 End_point like Object.vertex,  /* Ending */
      3 Intersect (2),                 /* Info on the intersections */
        5 Prop float,                  /* Where on the edge */
        5 Infront boolean,             /* Who's closer */
        5 Where like Object.vertex;    /* Exactly where it happened */
```

Object is shown in table 3.4.

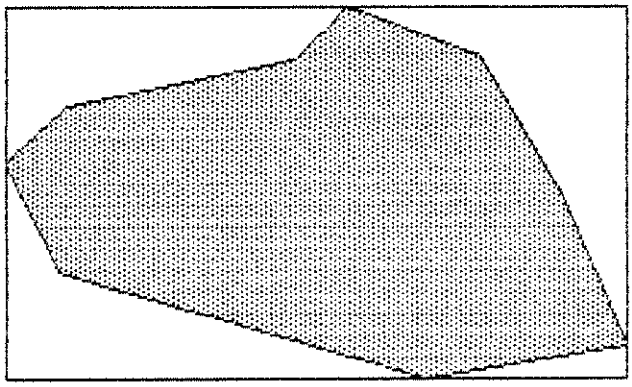
Table 5.1

5.3 Pruning

One way to greatly increase the speed of the hidden line algorithm is to prune out objects before clipping which one can tell quickly and easily will not affect an object. If you know that an polyhedon does not obscure a given edge, then there is no need to test the edge against all the faces of the polyhedron. Hline uses two methods of pruning both faces and objects which we will call an "element." The first is called Deeplap, and uses the depths of the edge and element. If we know that the maximum depth of the edge is less than the nearest part of the element, then the edge is strictly in front of the element and cannot be obscured by it.

The second method is called Overlap, and uses the screen position of the edge and the element. We compute the bounding boxes for the edge

and the element. The bounding box is the smallest rectangle which encloses the mapped image. Figure 5.1(a) shows the bounding box for the image of a face. The bounding box is computed by finding the screen coordinates of the extreme left, right, bottom and top of the image. If bounding box of the edge does not overlap the bounding box of the element then the element cannot obscure the edge. If the two boxes do overlap then the element may obscure the edge but further testing is need to be certain.



Bounding box

Figure 5.1

5.4 Clipping

The clipping routine determines a face obscures and edge. It first computes where and how often the edge intersects the face. For every edge of the face, we see if it intersect the other edge. The computation is done in the image space which turns the problem into determining if two two-dimensional line segments intersect. For each point of intersection we compute the three dimensional position both on the edge and on the face. The depths of these two points is compared to determine who is in front.

After that has been done we can have one of three cases: zero intersecting points, one intersecting point and two intersecting points. The routines zero, one, and two are then called as appropriate.

5.4.1 Zero Intersections

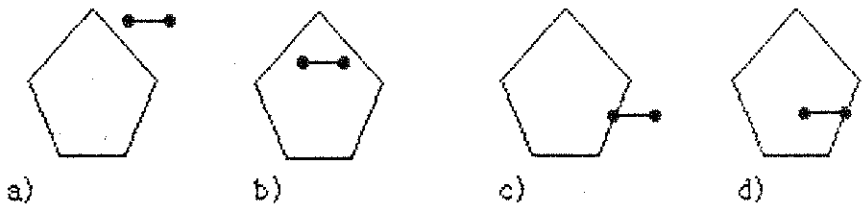


Figure 5.2 Zero Intersections

The case with zero intersections is the most complicated with four cases shown in figure 5.2. In cases a) and c) the edge can be seen but in cases b) and d) it hidden by the face. We proceed by testing whether an end point of the edge is inside the polygon. If it is then we can see the edge. If it is not then the edge is either completely in front of the face or completely behind. We determine who is in front by finding the angle between a vector from the face to the edge and the face normal. If the angle is positive then we the edge is in front and we can see it.

Case c) and d) occur when one end point is on the edge of the polygon. In this case, we repeat the test described above on the other end point to determine if we can see the edge.

5.4.2 One Intersection

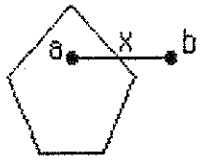


Figure 5.3 One Intersection

The one intersection case shown in figure 5.3 is the easiest one. First, we see if at point **x**, the edge is in front of the face. If it is then the edge is unaffected by the face. Otherwise the segment **ax** must be clipped. We find which end of the edge corresponds to point **a** and replace it with point **x**. All further testing is now done on the shorter edge **xb**.

5.4.3 Two Intersections

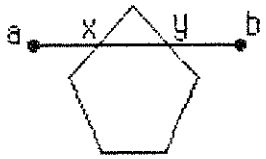


Figure 5.4 Two Intersections

The two intersection case is shown in figure 5.4. The first test you make is to see if at point **x** the edge is in front of the face. If it is then the edge is unaffected by the face. Otherwise, the edge gets split into two pieces: **ax** and **yb**. The current edge becomes **ax**, and the segment **yb** gets put onto the list of edges to be processed later.

5.5 Problems

The biggest problem with this method is rounding errors. We get places where two points should map to the same screen coordinates but is occasionally off by one pixel.

The most common error occurs in determining the point on the edge in three dimensions where it intersects the face. To compute this we map the end points from problem coordinates to screen coordinates, determine how far along the edge in the image space the intersection occurs, and go the corresponding distance along the edge in the problem space. Going from problem coordinates to screen coordinates back to problem coordinates magnifies the error. The effect of this error is to draw some edges one or two pixels to far.

Another rounding error occurs in the case of zero intersections in trying to determine who is in front. The method we use involves cross products to find angles. Occasionally it will err in thinking that the edge is behind the face instead in front of or on the face. This results in whole edges not being drawn.

There are several ways to alleviate rounding errors. A temporary fix is to change testing **x = 0** with **abs(x) < epsilon**. We have implemented this but the problem comes in deciding what epsilon should be. If epsilon is too small then you let some errors through. On the other hand, if epsilon is too big then you fail some good tests. The dangerous situation occurs if you have an object with dimensions smaller than epsilon. Thus epsilon has an upper bound proportional to the size of the smallest object.

Another possible fix is to increase the precision of the numbers. The extreme case would be to make every number float decimal (59) which

gives you accuracy to 59 decimal places. This would fix the problem but slow down the computations.

A final solution is to change the methods of computation to use only numbers it knows accurately. One should try to avoid using numbers derived by computation from other numbers. The numbers we consider accurate are the coordinates of the vertices. Unfortunately the coordinates may even be inaccurate. When an object gets rotated, the coordinates of the vertices get multiplied by sines and cosines of the angle of rotation. Thus, a cube rotated 100 degrees around the z axis then back 100 degrees may not be in the same position as when it started.

6. Hidden Surface Elimination.

6.1 Tactical Approaches

To change the picture from figure 6.1 to figure6.2, we use a hidden surface algorithm.

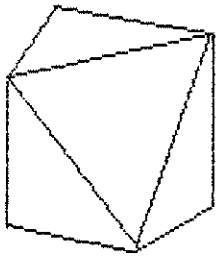


Figure 6.1

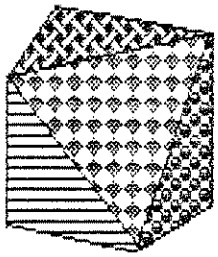


Figure 6.2

The computer mimics a person's vision by not displaying those planar faces that, in real life, would simply not reflect light to the eye. The method of hidden surface removal will be different from the method of hidden line removal, since the nature of the problem is different. In hidden surface removal, a decision must be made to select the color or intensity of each pixel bounded by a visible surface. Notice that because a hidden surface algorithm considers all the possible pixels involved with a visible scene, the hidden surface package can easily be expanded to allow object shading.

While the goal is clear cut; the method is not. Newman and Sproull maintain that there is no best answer to the hidden surface question, only choices to be made based on different constraints such as output requirement and internal data structures. (p 367). One thing that all the algorithms we examined capitalized on was what Partick Henry Wilson

calls "good representation" of the problem at hand. (p. 22) These methods expose natural constraints in the spacial geometric relationships to orchestrate the computation which solves the problem. For example, some hidden surface algorithms use geometric sorting to create their displays. Other algorithms use what Newman and Sproull refer to as "area coherance" of the scene to draw it.

In general, the methods we looked at either work in the object space or the image space. Object space algoritms work where the objects live: in three dimensions. Newman and Sproull and others point out that these algoritms make geometric calculations with as much precision as the floating point precision of the computer hardware will allow. In most cases the precision available is greater than the resolution of the display device (p368). The advantage of such precision is that the picture can be enlarged many times without loss of detail. If the scene is not to be enlarged, then working with the two dimension projection of the picture or the image space provides all the resolution necessary to present the highest quality picture possible. Wylie, Romney Evans and Erdhal (Watkins pp. 3) maintained that the the hidden line problem need only be solved at the discreet resolution points of the output device. Their reason: ease and performance. Why spend time computing something that you will never see.

In devising our hidden surface algorithm, the various techniques available today must be considered. The constraints of implementing a graphics system on D1 call for an algorithm that can be used on many different terminals in the Dartmouth College system. Another constraint is

upon the objects in the pictures themselves. Beside the spacial geometric relations given in any picture of objects, the objects in our graphic package will have to obey the restraint of limiting themselves to existing only as faceted objects with convex planar faces. With this in mind we will look at what's around.

6.2 The Methods

6.2.1 Depth Buffer

A depth buffer is an array which holds the current depth from the eyespot and the intensity of every pixel to be displayed on the screen. Each polygon or object face is processed only once, and the program will compare the eye depth of each screen coordinate bounded by that face with the corresponding depth buffer location. The closest points to the eyespot and their intensities will be maintianed. At the end of this processing the visible surfaces will be contained in this pixel array.

Depth buffer algorithms work in the image space. All verticies of an object must be transformed into screen coordinates before work can begin. According to Newman and Sproull (p 369) the computation time of an algorithm that works in the image space will grow with the complexity of the visible image space. This is an advantage over an object space method whose complexity grows with number of faces in the scene, visible or not.

The problem with the depth buffer is that it is not always practical. There is a problem with a variable depth buffer if the algorithm is to be implemented on D1. A GIGI computer terminal with a raster of479 X 767 would require 734,786 storage locations for the depth and intensity array. On the techtronix 4107 with 4095 X 4095 pixels the storage locations for

the display array would be a staggering 33,538,050 memory locations. Another related problem, is holding the memory for the depth buffer. This could cause problems, especially if the hidden surface program was part of an ever growing graphics package.

The way to solve this problem is to divide the depth buffer into smaller spans. The subdivisions could be either smaller squares or into horiziontal or vertical bands. Using depth bands, the process time for each segment is a constant number of pixels, and there can be as many segments as necessary for the type of display you are using. The advantage is saved space. Problems occur because, faces can overlap then bands and these faces will have to be called several times before the display is resolved. However, Newman and spoull point out (p 370) that the subdividing of the screen doesn't always hurt the execution time, it can be a help reduce the work needed to compute the entire image somethimes. This is due to the coherance between small pieces of the display screen, especially in the case of the horizontal or vertical bands.

6.2.2 Recursive Windows

Warnock, at the University of Utah, tried a more direct approach. His idea is to give more attention to areas in the scene which are more complicated to resolve visually. The program attempts to "solve" the given figure at the current screen size location. Polygonal faces relevant to a given window are grouped into one of three groups:

Disjoint Polygons: polygons that do not overlap any part of the curent screen window.
Intersector polygons: polygons that lie completely or partially inside the current screen window.

Surrounder polygons: polygons that completely surround the current screen window.

Warnocks algorithm will look at a window. If the window contains polygons that the program can solve the picture, baised on the three polygonal types, then Warnocks algorithm will draw this section of the picture. If the particular window seems unsolvable, then the program will break up the window into four small windows and recursively follow the above procedure. For example, in the first case, the screen size location is the whole screen. If warnocks algorithm cannot solve the problem and draw the figure in this window; if it is too complicated, then the program breaks up the picture into smaller subpictures and then tries to solve them.

Warnock experimented with the crieteria needed for a window to be solvable. The earliest tries would draw the window if there was a polygon in the subscene that belonged to group three and covered all other polygons in the area or if no polygons would be seen in that section of the viewing screen. The recursion using this crieteria would go quite deep, ofter terminating in a single pixel window. Smarter algorithms would recurse less, since they could accurately draw more complicated pictures, but the decision time for each recursion would take more time than the simple algorithms. Warnock tried stopping recursion if the above conditions held and, in addition, only one polygonal bound line split the current window. Since this addition was not computationally time consuming, gains in speed were achieved. Other efficiency increases are quite possible.

Warnock's algorithm makes time savings by taking advantage of the

natural constraints of the objects. By recursing the picture into smaller pictures, the whole picture is generated just like a perfix order tree traversal (pp. 4), with each parent node spawning four children, if recursion is necessary. Warnock here uses what he terms as "inherited information" to make gains in time. The gain occurs in the case

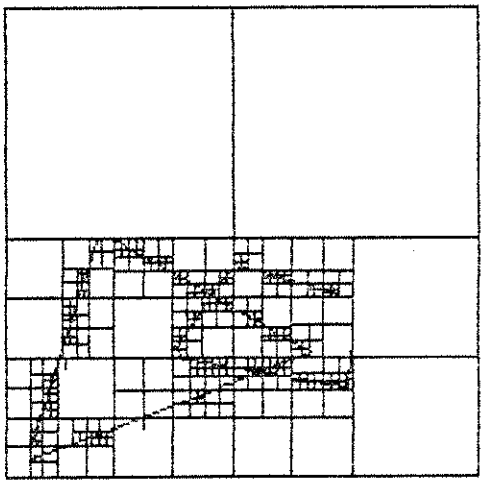


figure 6.3 the method of warnock

of a polygon that does not intersect a window at all. That polygon can be marked so that if the window is recursed into smaller windows the marked polygon will not have to be re-examined for intersections with the smaller windows.

There are some problems with Warnocks algorithm. For one, the "inherited information" seems artificial in its construction and a place must be maintained in the data structure as a permanant fixture. Because we will be holding our planar faces in object files, time will be spent reading the objects into memory from these files. Since a ploygon relevant to a window (i.e. a polygon that intersects or bounds that window), time will be spent computing window intersections for each recursion level. Other methods will need to access each face fewer times in determinig the final picture. Also, Newman and Sproull (pp.378) point out that the recursion into fixed sized picture windows is an source of inefficiency. They suggest having variable partitioning of the windows, based upon the bounds of the objects themselves.

6.2.3 Scan Lines.

A new algorithm can be formed out of the Depth Buffer by splicing the buffer into an area one pixel wide stretching accross the screen. A good way to think of the scan line is to project a plane from the viewers eyespot through the scan line on the screen. Intersections of the object faces with the scan line plane will form line segments. Since we maintain a planar restriction on the objects faces, there will be no curves in this intersection plane. Figure6.4 shows the possible intersections looking

perpendicular to the scan line plane.

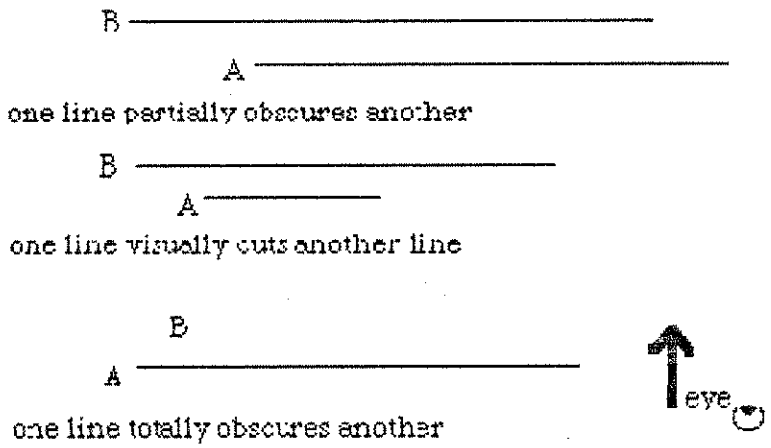


Figure 6.4 the possible scan line intersections

The goal after obtaining the segments is sorting by depth and printing the result.

This method seems to have a simplicity and an elegance that the other algorithms did not have. It seems less clunky than the original depth buffer and more straight forward than Warnock's algorithm. There is a set number of scan lines that will be drawn, namely the number of pixels in the either the length or width of the screen for each display terminal's resolution. The facts that you are only drawing say 480, if we are using a GIGI terminal, seems to be an advantage over the indeterminent recursing of Warnock's algorithm. Still, there will be the problem of looking up each object face relevant to a scan line for as long as scan lines are passing through the face. As Newman and Sproull said earlier, it's all tradeoffs.

6.2.3b Watkins Algorithm

One algorithm of this type that we looked at was a thesis by Gary Watkins. His interesting improvement to the problem was in his data

structure for the objects themselves, as in figure6.5.

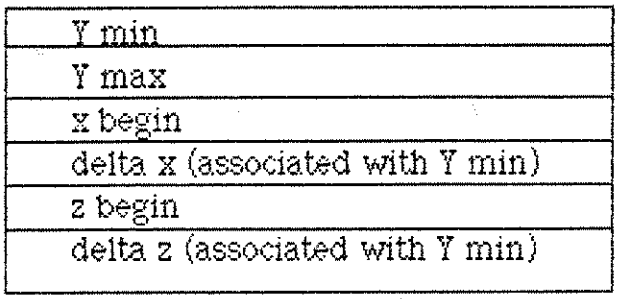


Figure 6.5 Watkin's Data Structure.

Notice how he keeps a firm Ystart and Yend, and only keeping a firm Xstart and Zstart. Watkins chooses to use incremental delta X and delta Z that will step the X and Z values for each Y value. Using this data structure, an active object face can just incrementally step through the X intersection values while the scan line plane falls between the start and ending Y values.

Once the segments of the scan line are generated, they need to be sorted in some fashion to arrive at the display. Watkins implements a complicated way of parsing the line segments into blocks of area conflict. This algorithm goes through every object in a scene, one line at a time. Watkins noticed that the execution time of his scan line algorithm will grow as the complexity of the visible faces grows. This is true because the actual depth computation takes place after the segments are found to be visible. This is area of the complex scan line sorting routine.

6.2.3.c Notes on Romney

Gordon Romney, in his thesis, also explored the possibilities of the scan line algorithm. One of his ideas was the "speedy" scan line segment. Suppose you have already drawn a scan line like the segments of the picture in figure6.6.

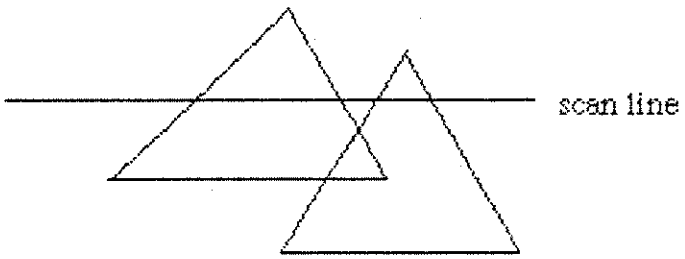


Figure 6.6: Romey's speedy segment.

Since the triangle depths in the figure will not change, relative to the figures at hand the depths will not need to be re-computed. (Romney p. 56). Here only the incremental changes of the segment endpoints need adjusted and the output can be printed. This also takes advantage of "area coherance."

6.2.4 Painters Algorithm

One final algorithm that we looked at was the painters algorithm, desctibed in Newman and Sproull. This algorithm catagorizes the objects by priority number. The objects furthest from the viewer will be given the lowest priority and the closest objects will be given the highest priority. The surfaces will be drawn in increasing prioity order, and the scene will emerge. There are some problems with the painters algorithm, too. For example, figure6.7 depicts a small box visually behind a larger box whose

depth is decresing.

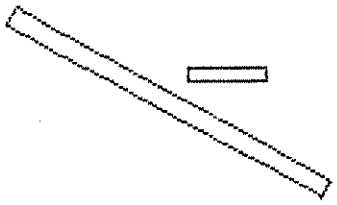


Figure 6.7: The Priority Problem

Since the prioities are given out on the basis of something like the greatest depth of each object. the smaller box would be seen where is should be invisible. Also there are two situations for which no priority exists as in figure6.8.

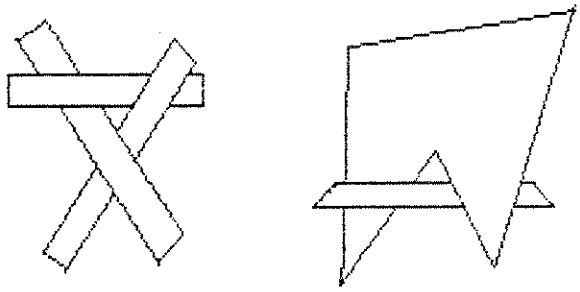


figure 6.7 no priority exists for these configurations.
(from Newman and Sproull p.382)

In the case of our thesis, one of the cases would not matter since we do not allow anything but convex planar faces. But we would like to find an algorithm that is general enough so that in time it could be expanded to allow for more complex polyhedra.

6.3 what we did.

We understand from the algorithms, there are some desirable properties that we look for when designing a hiden surface removal method. One of them is the watchword of Warnocks solution: **give more attention to the more complicated areas.** Another property is to capitalize on properties of the objects themselves; obtain area coherance. The scan line algorithm would be most beneficial. It has the straight forwardness that would allow for a good representation of the problem and later modifications. It would allow for area coherance modifications later and it would also fit well with the object data structure.

The scan line algorithm we implemented starts from where the image space coordinates of the objects have been translated into screen coordinates and the hidden line surfaces of the objects themselves have been removed. Only hidden surface decisions between objects have to be made.

6.3.1The Method: Saving Time on Easy Things

The processing starts from the either the top on the visually highest object or the top of the screen, which ever comes first. For each scan line, we search through the objects to discover if any of these object faces will intersect the scan line. A time consuming feature of our data structure is the fact that each time a objects is needed to be examined it must be called from memory. One speed gain would be to limit the needless object calls and only call up relevant objects. When the program executes Pass1, the screen coordinates of the objects are calculated along with other relevant

data on the objects like the object and facial bounds. For each object, an array is packed with the object number and the top and bottom bounds of the object. Since we are already looking at the objects at this pass it seems useful to fill the buffer at this point because the user will probably use the hidden surface removal to create a meaningful color picture from the object data. Filling the array saves a needless re-search of all the objects when the the hidden surface algorithms is called. Once the buffer is established, for any given scan line, a list relevan objects will be examined to find intersection points. This will save time looking at fruitless objects.

6.3.3 Finding the intersection segments

For each object, the program, now, looks for and intersection with a face. To do this the program is going to "walk " around each segment on a face of an object, looking for an intersection with the scan line. The goal here is to find two intersection points, since each segment must have a start and an end point, and the faces are going to be convex polygons. The algorithm only looks at the faces that are relevant to the current scan line by bypassing the object faces who are out of bounds to the scan line and also thse faces who have been marked as invisible by the PASS2 routine.

The algorithm will check for each segment on a relevant face the y max and y min bounds, and if the scan line lies between, but not including the first point (i.e *ay* in the program) and the second point (i.e. *by* in the program) which is included in the comparison, then an intersection occurs. The reason why the first point is not included is the aviod the situation in

figure6.9. The problem here is that the program has just looked at segment A on the face and the program has generated and interction point at the end point *by*. Since the verticies in the object data structure are arranged in a clockwise ordering, the next segment the program generates looking for the end point to this scan line segment is face segment B. Here too, the scan line intersects the facial segment and the program feels that is has determined that the segment of the scan line for this face; the program is wrong. So, to combat this problem the segment are examined in a one side open ended fashion, as in figure6.10. The algorithm is sure to catch any intersection that occurs in a vertex because each vertex is part of two segments, and regular clockwise ordering of the verticies allows each vertex once to be *ay* and once to be *by*.

This lead to a another special case problem. Suppose the scan line is crosses a single vertex on an face, as in figure11. This problem will occur at either the top or the bottom on the object face. By the above method, the routine to generate segments will only come up with one point. However a line segment has a startling point the an end point, even if they are one in the same. If the program has only generated one intersection point then this part will allow the starting point to be the end point. This problem occured only because we solved the other problem of generating the incorrect segment end points.

When an intersection with a scan line does occur it can occur in one of three cases:

Vertical line intersection. The slope of the line segment intersecting the scan line is 0. This intersection x screen value will be either of the endpoints of the segments and need not be computed. This case must be looked for because of the division by zero.

Horizontal line intersection. Since the program has gotten past all the restrictive cases the intersection segment will be the actual line segent you are currently at.

Diagonal line intersection. These cases have nonzero slopes, either positive or negative. The intersection point here must be computed. This is done by taking the slope and solving the equasion of the line y= mx + b, using *ay* to solve for b and the scan line to solve for the final x.

When a relevant object face is about to be examined, the program will allocate an x_box from based storage. The intersections for each relevant face are written into the x_box as the intersection are generated. At that point, the color of the intersection point and the depth of that point on the line segment are places into the x_box. Later modifications would include placing some information about the intensity of the color for shading purposes. .

Now, to draw a meaningful picture, these segments of objects that have intersected with the scan line must be ordered in sorted with the only the visible parts showing in the final display. To accomplish this, we first make each individual segment box with the smaller x intersection point in x_box.x_int.(1) along with the data corresponding to that point in the corresponding first places in each x_box. This process is easily handled. It is important to get some kind of uniform order to the segments. Also, a

change in depth, delta d, accross the segment is computed for every segment. This is used to compare depths when segments overlap in a later algorithm. Once the information in the boxes is organized, each box is placed into a doubly linked list by the subroutine called, insert. The list is maintained in an incresing x_box.x_int(1) order. The important thing again is to establish an ordering of the segment to work with in the next routine. After all the segments for a scan line are generated and inserted into the linked list, the list is drawn.

The final output is determined by the Draw_scan routine. The method here is straight forward in approach. Rather than scaling through the line iteratively or by recursivly chopping the scan line segments into smaller pieces every time there is a conflict, the goal is to take each segment and draw as much of it as we can. We begin by taking the first segment on the list and set the out put color to be that segments color. For as long as the segment is visible, the program will compare it to other relevant segments looking for an end point that will allow the routine to draw the segment . To find an end point the routine must compare depths at the points of possible overlap. Here is the list of comparison possibliities.

Case I: the current segment completely covers the comparison segment. Because the comparisoon segment will be invisible in the final output, it will be marked as invisible by the Covered flag in x_box.

Case II: the comparison segment either completely or partially covers the current segment. The subroutine which creates the list of scan line segments for draw scan makes no comparisons for segments with equal staring x value. A comparison segment of smaller depth than the current segment could cover the current segment , if it had the same starting point and spanned a longer space. If the current segment is totally covered, the routine simply goes on to the next segment from the list. If the current segment is partially, the algorithm will do the following: Draw the part of the segment before the start of the interruption by the comparison segment, if there is any. Now form a new current segment, starting from the end point of the intruding comparison segment. A depth at that new start is computed. And the comparisons will continue until the current segment is fully and correctly displayed.

case III. the comparison segment covers the end of the current segment. No more comparisions need be made for that segment. The program will draw the visible portion of the current segment and then go on to the next segment.

case IV. The current segment partially obscures a comparison segment. The end of the current segment will fall somewhere in the middle of the comparison segment. Since, the algorithm can't mark the comparison segment as partially covered, it will create a new segment from the remains of the old segment and re-insert the new segment back onto the x_box list.. The x_box containing the values for the partially obscured segment will be marked as covered, so it does not get in the way.

Decisions for each of the cases are baised upon depth comparisons at equal x and y screen coordinate values. The depth values are floating point

decimal values that have been iterpolated, sometimes several times before arriving at the depth to be compared. The over computation will lead to some inaccuracies in computing the depth. Since are figures are very small, a little miscalculation can lead to the wrong surfaces being exposed due to round off error. A few special cases must be identified and delt with, because of the inaccuracies of the machine.

The problem occurs when the depth of the two segments at the comparison points are very close and the algorithm distinguishes between the two in an almost arbirtrary fashion. At this point in the program, since the depths are equal, the final output decision is baisied upon a comparison of the depths at the endpoint of the shorter segment. This method capitalizes on the constraint of our objects being nonintersecting.

Depending on the position and the eye and the rounding errors involved, the program can't judge the comparison points and determine which segment is in front of which. Like there was a ghost in the machine, the program will print the incorrect segments. Our solution is to establish a buffer zone in which these segments can be tested further before a final output decision is made. All segments which lie very close together can be recompared by taking a comparison depth at the closest overlapping endpoint. Here, we are taking advantage of the fact that segments cannot overlap. If a scan line segments position is nonapparent to a comparision segment, then if the next overlapping endpoint can resolve the question the answer from this comparison will also be the answer to the original comparison point. Notice that simply comparing the endpoints will not

give the correct result. It is necessary to have some kind of overlap in order to employ the geometric coherance of the segments.

There is a case where this method will not work. Suppose there are two segments which are almost parellel and separated by very little distance. If the depth to the eye between the segments at a comparison point falls inside of the buffer zone, we look to the nearest overlappinf end point for resolution. If the overlapping end point depths fall into this buffer zone, then it will offer us no help in making the final decision. The result here is that there is no way to tell if one segment lies in front of another. Understand that this case seems to be quite rare and that even moving the figure just slightly out of the buffer zone will create a perfect rendering, as shown in the BRUCE portfolio.

Our method of hidden surface removal by scan line then is a very good, clean algorithm. It does not produce output in real time. Our megalithic speed stems form the problem of setting up consistent patterns and drawing them rather than the actual scan line decision making process.

7.1 Improvements to the Scan Line Algorithm.

In their October, 1983 article, A.Requicha and H.B. Voelcker noted that the current research in creating speedier hidden surface algorithms was to expand upon the area coherence of the objects being depicted (IEEE Oct,83 p.29). The one dictum that would improve the running time of the hidden surface algorithm is the lazy man's approach: **don't go back to the object file unless you have to; stretch your information to the fullest.** This is the essence of area coherance. The scan line below the one just displayed is going to probably look very similar to the previous scan line. In fact the odds of a new object being introduced on the next scan line will be (Number of objects/number of scan lines.) It seems that the new method will be advantageous to scenes with very few objects and terminals with very high resolutions. In any event our scan line algorithm can be improved later by taking advantage of these principals.

The area coherance that we can use is modified from Romey's "speedy segment" concept. The way to accomplish these improvements is to employ a more event driven procedure rather than scan line or clock driven algorithm than we use now.

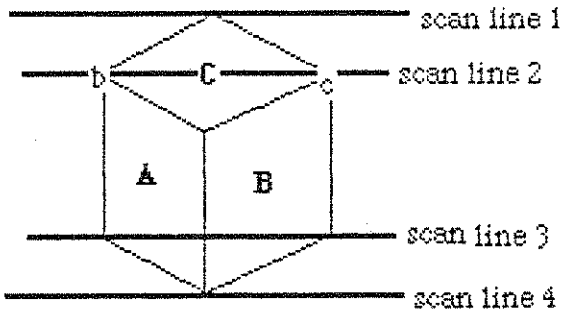


Figure 7.1

Take the example in figure7.1. We can draw this figures surfaces with only 4 accesses to the objects file for data. However, each time we collect the scan line intersection points from the object file we need to collect more infromation in x_box than just the start and end x segment values. We also need now need the incremental depth for each of the start and end values. From scan line 1 to scan line 2 we will increment the x_values and draw the segments. The next event is signaled by the change of the incremental x values at the verticies B and C. The big savings here is in the needless lookups and resetting of the object file, and also the time spent looking at each face of each object in determing the intersection point for each scan line will be saved.

The method looks good for the simple case of single objects and objects which do not visually overlap. The problem gets a little sticky in the area of object conflict. Notice that the introduction of a new object is not a problem. A new object is a new event and new x_boxes will have to be established for the scene at the point. The problem is when an object face "grows into" or "grows away from" another object. The two cases are

represented in figure7.2.

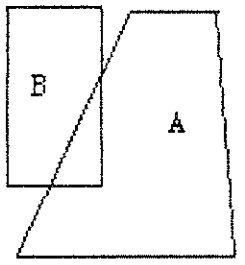


Figure 7.2a

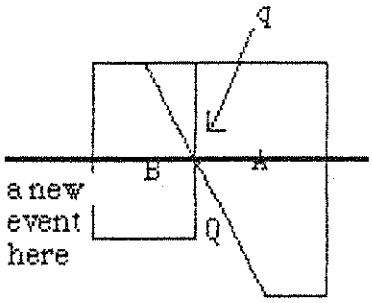


Figure 7.2b

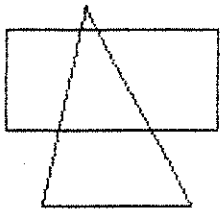


Figure 7.2c

Handling the first case is easier than the second. A point of conflict will develope between the figures A and B as A grows into B in figure7.2a. At this point, the program will resolve the conflict by comparing the depths as discussed above. Say that face A is visually in front of B. When the confilct is resolved, instead of simply chopping the segment, the proposed improvement would replace the incremental x value for the end point of object B's scan line segment with the incremental x value for the starting point of object A's scan line segment. Because we do not allow intersecting faces, A will continue to obscure B at least until the the next event. If the face of A will continues to grow and, at some scan line, completely obscures face B, the program can still correctly draw the scene if we add this mandate to the routine which increments the group of x_boxes:

> if the segments starting x value < the segments ending x value, then the segment is finished. Either make the box as covered or remove it from the list.

Briefly, if face B visulally covers face A in our example, the incremental x value of B's endpoint will become the incremental x value of the starting point of face A's scan line segment. The scene will have no

changes until the next event which happens under face B.

The second case slightly more difficult in that the program will have to create a new event on the fly. Say we have the situation in figure7.2b. when the scan line is x, there is a conflict between faces A and B. If A is visually infront of B then the conflict will be resolved in favor of face A by the method discussed above. Leaving the processing unchecked, the face of B would be drawn without the free space Q, as the correct picture should look. Using the next event signal, this case must check to see if there will be an intersection of the edges of these faces before that next event. If that is true, as in the point Q of figure7.2b, then the y value at that intersection point should signal the next event. Notice that the figure7.2c will employ a combination of these two case to resolve the conflict.

There are probably other methods to speed up the scan line algorithm that we have not examined, This one improves upon the methods that we examined in the following way. At a new event, after just computing the new segment boxes and resolving the overlap conflicts for that new scan line, no new comparisons will need to be made, only incrementations of the segment endpoints. This is a true example of area coherance.

7.2 Shading

We developed BRUCE with the intention of making it handle shading. Ultimately, however, the lack of a good graphics terminal prevented us from implementing it. Nevertheless, we design of the hidden surface algorithm makes it easy to incorporate the method of shading described below.

Shading is used to further increase the realism of a computer generated display. If a curved surface was approximated with many flat faces, shading can restore the smoothness. Two important characteristics are used in determining what color and intensity to draw. First is the properties of the face including color, reflectivity and transparency. The second is the properties of the illumination including the number and sources of illumination and the amount of ambient light. The shading contribution from a source of light will vary as the orientation changes according to Lambert's law, which states that the energy falling on a surface varies as the cosine of the angle of incident light. The problem becomes one of finding the normal to the surface at a given point and then getting the angle between the normal and the ray hitting the face.

Our hidden surface algorithm could easily implement a shading technigue called Gouraud shading. The normal is determined by double linear interpolation along scan lines.

First, the normal at the vertex is approximated by averaging the normals of the faces that meet at the corner. This is used to get the precise shading value at the vertex. Shade inside the face is interpolated from the shades at the vertices. Now consider a scan line intersecting a face at two

points a, b. The shade at these points is interpolated from the end vertices of the edge each point is on. The shade for point p inside the face on the scan line is interpolated from the shade values of points a and b.

There are several drawbacks to this method, however. First is the large number of interpolation calculations needed. This can be reduced somewhat by using incremental values which need to be computed less often. Second, anomalies in shading can arise because of the averaging of normals at the vertices. Adding small surfaces near the vertices can reduce this problem. Another drawback to Gouraud shading is that the interpolation can introduce the Mach band effect, a phenomenom of the human visual system due to discontinuities of the rate of shading (Newman and Sproull, p. 400).

8. Conclusion

We were inspired to write this thesis because we wanted to know how computer generated pictures were generated. We wanted to develop a system to allow anyone at Dartmouth College to do the same. As a result, we developed BRUCE, a very powerful graphic system that solves many of the problems of three dimensional graphics.

BRUCE is designed to be easily expanded. It provides a good basis and suitable environment for further research into the relatively new field of three dimensional graphics.

The last line though, is that now we know how those pretty pictures were drawn, and we supply a portfolio of our own pictures.

Bibliography

Appel, A.: "The Notion of Quantitative Invisibility and the Machine Rendering of Solids," *Proc. ACM Nat. Conf.*, Thompson Books, Washington, D.C., 1967, p. 387.

Barnhill, R. E. and W. Boehm (eds): "Surfaces in Computer Aided Geometric Design," North Holland Publishing Co., Amsterdam, 1983.

Clark: "Designing Surfaces in 3-D," *Communications of the ACM*, **19**(8), August, 1976, pp. 454-60.

Giloi, W. K.: "Interactive Computer Graphics Data Structures, Algorithms, Languages," Prentice-Hall, Englewood Cliffs, N.J., 1978.

Gouraud, H.: "Computer Display of Curved Surfaces," *Univ. Utah Computer Science Dept.* UTEC-CSc-71-113, June 1971; NTIS AD-762 018.

Kubert, B. R.: "A Computer Method For Perspective Representation of Curves and Surfaces," *Aerospace Corporation*, San Bernardino, CA, December 1968.

Kunii, T. L.(ed): "Computer Graphics Theory and Applications," *Proc. of InterGraphics '83*, Springer-Verlag, Tokyo, 1983.

Loutrel, P. P.: "A Solution to the Hidden-Line Problem for Computer Drawn Polyhedra," *Phd Thesis, Electrical Engineering, New York Univ*, New York, September, 1967 (also in *IEEE Trans.* **EC-19**(3):205 March 1970.

Newman, W. M. and R. F. Sproull,: "Principles of Interactive Computer Graphics," McGraw-Hill, New York, 1979.

Raloff, J.: "New Process for Computer Images," *The Science News*, Febuary, 1984, p.86.

Randi, R.: "Hidden Surface Elimination the Easy Way," *Creative Computing* **10**(2), Febuary, 1984, pp. 189-192.

Requicha, A. G. and H. B. Vaelcker: "Solid Modeling; Current Graphics and Applications," *IEEE Computer Society*, October, 1983, p.25-30.

Rogers, D. F.: "Mathematical Elements for Computer Graphics," McGraw-Hill, New York, 1976.

Romney, G. W.: "Computer Assisted Assembly and Rendering of Solids," *Univ. Utah Computer Science Dept.*, TR 4-20, 1970. NTIS AD-753 673.

van Dam, A.: "Compuer Software for Graphics," *Scientific American*, **253**(3): 146-159, September, 1984.

Warnock, J. E.: "A Hidden Surface Algorithm for Computer Generated Halftone Pictures," *Univ. Utah Computer Science Dept.*, TR 4-15, 1969. NTIS AD-753 671.

Watkins, G. S.: "A Real Time Visible Surface Algorithm," *Univ. Utah Computer Science Dept*, UTEC-CSc-70-101, June 1970. NTIS AD-762 004.

Winston, P. H.: "Artificial Intelligence," Addison-Wesley, Reading, MA, 1984.

Wylie, E. et al.: "Half Tone Perspective Drawings by Computer," *AFIPS Proc.* FJEC 31, 49, November, 1967.

Yamaguchi, K. K. et al.: "Computer Integrated Manufacturing of Surfaces Using Octree Encoding," *Computer Graphics and Apps.*, January, 1984, pp. 60-70.