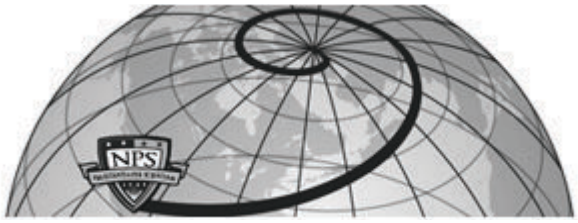


NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

AN INEXPENSIVE REAL–TIME INTERACTIVE THREE–DIMENSIONAL FLIGHT SIMULATION SYSTEM

by

Douglas Bernard Smith
Dale Gerard Streyle

June 1987

Thesis Advisor: M. J. Zyda

Approved for public release; distribution is unlimited.

unclassified
CURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

REPORT SECURITY CLASSIFICATION: unclassified	1b RESTRICTIVE MARKINGS
SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution is unlimited.
DECLASSIFICATION/DOWNGRADING SCHEDULE	
PERFORMING ORGANIZATION REPORT NUMBER(S)	5 MONITORING ORGANIZATION REPORT NUMBER(S)
NAME OF PERFORMING ORGANIZATION: ival Postgraduate School; 6b OFFICE SYMBOL (If applicable): 52	7a NAME OF MONITORING ORGANIZATION: Naval Postgraduate School
ADDRESS (City, State, and ZIP Code): Monterey, California 93943-5000	7b ADDRESS (City, State, and ZIP Code): Monterey, California 93943-5000
NAME OF FUNDING/SPONSORING ORGANIZATION; 8b OFFICE SYMBOL (If applicable)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
ADDRESS (City, State, and ZIP Code)	10 SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO; PROJECT NO; TASK NO; WORK UNIT ACCESSION NO

TITLE (Include Security Classification): AN INEXPENSIVE REAL-TIME INTERACTIVE THREE-DIMENSIONAL FLIGHT SIMULATION SYSTEM

PERSONAL AUTHOR(S): Smith, Douglas Bernard and Streyle, Dale Gerard

TYPE OF REPORT	13b TIME COVERED FROM ____ TO ____	14 DATE OF REPORT (Year, Month Day)	15 PAGE COUNT
ister's Thesis		1987 June	237

SUPPLEMENTARY NOTATION

COSATI CODES: FIELD	GROUP	SUB-GROUP

18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number): flight simulation; DMA terrain data, computer graphic terrain display

ABSTRACT (Continue on reverse if necessary and identify by block number)

A prototype flight simulator for the Fiber-Optically Guided Missile (FOG-M) is presented. This prototype demonstrates the practicability and feasibility of using low-cost graphics hardware to produce acceptable simulation of flight over terrain generated from Defense Mapping Agency (DMA) digital elevation data. The flight simulator displays a dynamic, three-dimensional, out-the-window view of the terrain in real-time while responding to operator control inputs. The total system cost (software and hardware) of the simulator is an order of magnitude less than most flight simulation systems in current use.

DISTRIBUTION/AVAILABILITY OF ABSTRACT: ☐ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION: unclassified
NAME OF RESPONSIBLE INDIVIDUAL: Prof. Michael J. Zyda	22b TELEPHONE (Include Area Code): (408) 646-2305; 22c OFFICE SYMBOL: Code 52Zk

FORM 1473, 84 MAR — 83 APR edition may be used until exhausted; All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE: unclassified

Approved for public release; distribution is unlimited.

An Inexpensive Real–Time Interactive Three–Dimensional Flight Simulation System

by

Douglas Bernard Smith
Captain, United States Marine Corps
B. S., Duke University, 1981

and

Dale Gerard Streyle
Lieutenant, United States Coast Guard
B. S., United States Coast Guard Academy, 1980

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL

June 1987

ABSTRACT

A prototype flight simulator for the Fiber-Optically Guided Missile (FOG-M) is presented. This prototype demonstrates the practicability and feasibility of using low-cost graphics hardware to produce acceptable simulation of flight over terrain generated from Defense Mapping Agency (DMA) digital elevation data. The flight simulator displays a dynamic, three-dimensional, out-the-window view of the terrain in real-time while responding to operator control inputs. The total system cost (software and hardware) of the simulator is an order of magnitude less than most flight simulation systems in current use.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

The authors wish to express their gratitude to a number of people who supported this study. To our advisor, **Dr. Michael Zyda**, who provided us with the knowledge and insight necessary to complete the project, and then stepped back, allowing us the freedom to learn through exploration.

To the following people who provided programs and subroutines which were incorporated into the project:

- **MAJ Ron Ross**, USA, for the original versions of the *prelaunch* & *make–database–e* routines.
- **LCDR Mike Oliver**, USN, for enhancements to the tank image.
- **Dr. Michael Zyda** for the original version of the *gammaramp* routine.
- **CAPT Gary W. Taylor**, USMC, for the original version of the *lightorient* routine.
- **LCDR James Manley**, USN, for the *netV* networking routines.

The authors would also like to note that the order of the names on the cover is alphabetic, and has no other significance.

LT Streyle would like to personally thank his wife, Robin, for the tremendous amount of patience and support provided during all phases of the project. By taking care of the myriad of details involved in running a home with two children and shuffling her and the family's schedule around the times I absolutely had to work, she provided me the time necessary to fully pursue the project. I would also like to thank my lovely daughter Sarah and son Timothy, who both let me know when I had worked enough to "earn" another trip to the park to play.

CAPT Smith would like to thank his wife, Becky, and son, Timothy, for the generous amounts of time and pleasures foregone in their support of this work. Thanks also to my friend and co-author, who made this and many other projects much easier and more enjoyable than they would otherwise have been.

I. INTRODUCTION

Flight simulation has been an important computer graphics application, embracing a range of systems from a $32.00 program for a personal computer [Ref. 1] to special purpose machines costing millions of dollars [Ref. 2]. The capabilities of these systems are spread across a range nearly as wide as their costs, with great variances in speed (frames displayed per second), realism, flexibility, and area of flight. We present here a system that is relatively inexpensive, yet still fast enough to present a real-time three-dimensional view of digitized terrain. We built this system on a commercially available, high-performance graphics workstation, the Silicon Graphics, Incorporated IRIS-2400 Turbo. The IRIS system was selected because of its local availability and its performance capabilities. The flight simulator presented here does not use the natural color and shape of individual terrain elements (in order to achieve real-time performance), but it is sufficiently realistic to provide the feeling of flight and allow identification of the displayed terrain and targets.

A. FOG-M

1. Background

The project presented here was built in response to the United States Army Combat Developments Experimentation Center's need to simulate the

operation of the Fiber-Optically Guided Missile (FOG-M) [Ref. 3], but this missile is also being considered for use by the United States Marine Corps [Ref. 4]. Simulation is necessary in order to test and evaluate the tactics, doctrine and training requirements associated with the missile without the expense and danger of actual firings during simulated combat field trials. The FOG-M is a generic family of remotely-piloted, video-guided munitions, but for the purpose of this prototype simulator, the weapons are all logically equivalent, and the entire family is referred to as "the missile." In order to avoid security constraints, the parameters and operational characteristics used in this work were not taken from exact FOG-M specifications. The parameters and technical specifications are all estimates, based on reasonableness and consistency with general, unclassified descriptions of the FOG-M.

2. Description

The actual FOG-M missile is six inches in diameter, five and one-half feet high, weighs eighty-three pounds, and costs about $20,000 [Ref. 4]. It has a video camera mounted in its nose, which transmits a black-and-white picture back to the operator's console (which consists of a television screen, a computer, and a joystick) over the fiber-optic link. (The simulator display offers the user the choice of either color or black-and-white; color is the default for the simulator despite the operator view of the missile being black-and-white. The color compensates for some of the loss in realism and identifiability inherent in a polygonal representation of natural objects). Before launch, in normal operation, the missile

is given a general direction to a target and the altitude of the highest point within its range. The simulator allows values in excess of FOG-M operational capabilities for speed, range, and altitude above ground level (AGL), but the default values of two hundred knots, ten kilometers, and one thousand meters are characteristic of this type of missile. As soon as the missile is in position, it begins transmitting video images. When launched, the missile rises to approximately two hundred feet above the highest terrain point, and then levels off in horizontal flight in the targeted direction. The operator controls the pan and tilt angle of the camera with the joystick, and can dial in changes to the heading and altitude of the missile. The operator also has the capability to zoom the camera's field of view from eight degrees to fifty-five degrees, and to *designate* ("lock-on" to) a target for automatic homing by the missile.

B. ASPECTS OF FLIGHT SIMULATION

There are many aspects to flight simulation. Modern commercial simulators provide sophisticated mock-ups of cockpits and controls and highly detailed out the window views. By mounting the simulator on a moving platform, a true sense of the physical feelings of acceleration and roll can be achieved. These systems also cost millions of dollars.

One of the first decisions that must be made when designing a flight simulator is, "For what purpose will the simulator be used?" The answer to this question drives most of the design decisions that have to be made. Since the purpose of

this project is to provide a simulation of the FOG-M missile as viewed from its operator's console, it is felt that the most important items to model are the simulated video display of the terrain and the actual operator controls. The terrain should appear realistic enough that its major features are recognizable to a person familiar with the area. The controls should allow for the same functionality as the actual console. The simulator must, of course, also provide a feeling that the missile is in motion over the terrain. The effectiveness of the feeling of motion provided by a flight simulator can be largely measured by two criteria: the realism of the displayed scene and the update rate of the display.

1. Realism

Many factors contribute to the perceived realism of a displayed natural scene. Early attempts to quantitatively measure realism consisted of counting the number of "edges" or lines that a scene contained. This later gave way to counting the number of "faces" or polygons in a scene. Since each polygon was colored in a single shade, it was felt that each polygon represented a single "bit" of information in the scene. Therefore, the more polygons the scene contained, the more "realistic" it was felt to be [Ref. 5:pp. 27-28].

The latest advances in computer graphics have also made this measure of effectiveness obsolete. With the introduction of systems that are able to fill polygons with textured patterns, a single polygon can now contain thousands of "bits" of information. As a result, a scene drawn with a few textured polygons can appear more realistic than one with an order of magnitude more untextured

ones. Early textures consisted of superimposing things such as mathematical noise functions or stripes on the polygons. More recent advances have allowed the texture to be derived from digital photographs of a similar scene. For example, polygons representing a part of terrain covering by meadow could be filled with a digital texture derived from an aerial photograph of a meadow [Ref. 5: p. 28].

Since most currently available graphics workstations do not support the use of texture filled polygons, the use of texture was deemed to be outside the scope of the current project. Rather, the project's work concentrated on determining how realistically a scene could be rendered in real-time incorporating only the use of lighting and shading models along with terrain hidden-surface algorithms. These topics are covered in more detail in Chapter V.

2. Frame Update Speed

Another important aspect of a flight simulator's performance is the speed at which it is capable of displaying successive frames in a scene. The faster the update rate, the more continuous the motion appears. As a reference, standard motion picture film is projected at a rate of twenty-four frames per second. Although the IRIS workstation is capable of displaying up to sixty frames per second, the amount of computation that must be done between successive frames in the simulation has limited the update rate to an average of three frames per second. While this presents a less than smooth motion, it is felt to be adequate for the purposes of the prototype.

C. ORGANIZATION

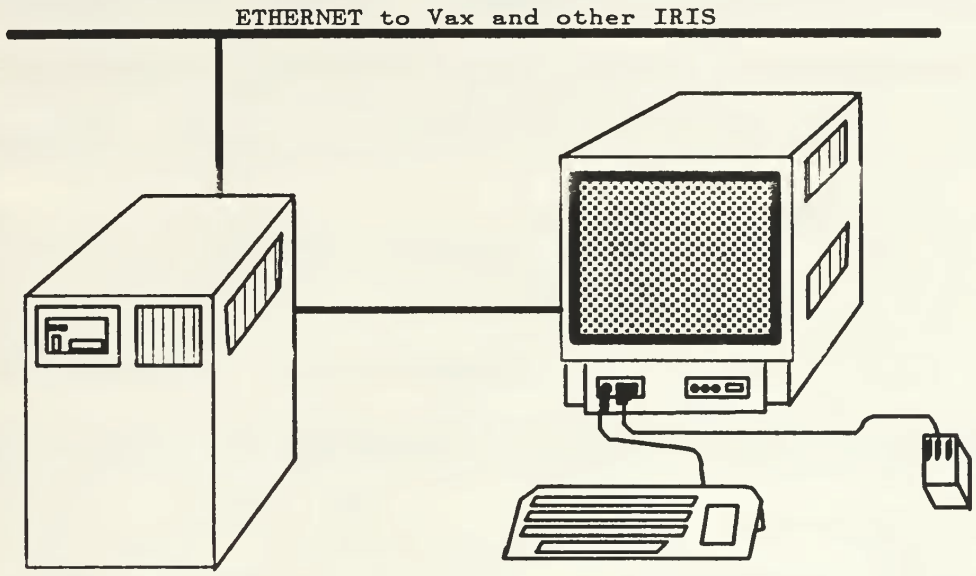
The above sections of this chapter have provided background on flight simulation in general, and the missile whose flight is specifically being simulated. Chapter II provides an overview of the hardware used in running the simulation. The structure and content of the Defense Mapping Agency (DMA) Digital Terrain Elevation Data (DTED) are discussed in Chapter III. Chapter IV discusses the motivation behind and creation of the two-dimensional contour map displays. Chapter V covers the storage and use of the DMA DTED to produce a lighted and shaded three-dimensional polygonal terrain display. The mathematics and process involved in simulating flight over the terrain are detailed in Chapter VI. Chapter VII discusses the creation, insertion, animation, and designation of targets. Chapter VIII covers the creation and drawing of cultural features. Chapter IX contains a user's guide for operation of the FOG-M simulator. Chapter X concludes with a discussion of limitations, future extensions and research topics, and summarizes the research conducted. Narrative descriptions of the modules and listings of the program source code for each of the modules are included as Appendices A and B respectively.

II. COMPUTER SYSTEM

As discussed in Chapter I, flight simulators are nothing new. The significance of this work lies in the speed with which it was developed, the display rate achieved, and the realism and fidelity of the display in comparison to the cost of the system that supports it. This project was technically feasible only because of the capabilities and high performance of the IRIS (Integrated Raster Imaging System) Turbo 2400 Graphics Workstation, manufactured by Silicon Graphics, Incorporated. Others have also used the IRIS as a base on which to build flight simulators [Ref. 6]. This low-cost ($50,000 to $100,00) three-dimensional display system is summarized in Figure 2.1 and is discussed more fully below.

A. HARDWARE AND SYSTEM OVERVIEW

The IRIS has a conventional Von Neumann type computer architecture but adds custom-built special purpose VLSI circuits and a pipelined design to provide the graphics functions that are implemented in software on other comparably-priced workstations. Conceptually, there three pipelined components in the IRIS hardware: the applications/graphics processor, the Geometry Pipeline, and the raster subsystem [Ref. 7:p. 1-1]. The applications/graphics processor is a conventional Motorola MC68020 processor running at 16.7 MHz. This processor runs the applications program(s) within a UNIX System V operating system.



- 32 bit 16.7 MHz Motorola MC68020 CPU
- 6 Megabytes of CPU Memory
- 32 1024x768 bitplanes of Display Memory
- Hardware matrix multiplier & floating point accelerator
- Hardware Gouraud shading, depth cueing & backface polygon removal
- 12 pipelined custom VLSI Geometry Engines™
- 16-bit Z-buffer for Hidden Surface Elimination
- 2 72 Megabyte Winchester Disk Drives
- 60 Hz non-interlaced 19" RGB high resolution monitor
- 83 key up-down encoded keyboard
- 3 button mouse
- 32-button and 8-dial valuator boxes
- Unix System V
- Ethernet to VAX's
- IRIS Graphics Library

Features of the IRIS Turbo 2400 Graphics Workstation

Figure 2.1

Applications either issue graphics commands in *immediate mode*, in which case they are sent through the Geometry Pipeline immediately as individual graphics primitives, or compile graphics commands into *graphical objects*, in which case they are sent through the Geometry Pipeline as a single geometric entity when explicitly called at some later point in time.

The Geometry Pipeline takes commands in terms of the user's world coordinates, performs specified matrix transformations on them using the matrix multiplier and floating point accelerator built into the hardware, and then clips and scales the transformed coordinates into screen coordinates. The raster subsystem takes the screen coordinates output by the Geometry Pipeline and updates the *bitplanes* (display memory) to contain the lines, polygons, or characters specified by the input coordinates. The raster subsystem also performs polygon fill, shading, depth-cueing, and hidden surface removal. A conventional video controller uses the values in the bitplanes and the *color table* to produce an image on the monitor.

B. SOFTWARE

The C programming language is native to UNIX and is the language used for all of the IRIS system software. The IRIS Graphics Library, which provides both high- and low-level graphics and utility commands, can be called in C, FORTRAN, Pascal, or LISP. However, due to the built-in bias of UNIX and the IRIS, plus the local pool of knowledge, the C programming language was the

pro forma choice for programming all parts of the FOG-M simulator. The IRIS User's Guide [Ref. 7] breaks the Graphics Library commands into the following twelve categories:

- *Global State* commands initialize the hardware and control global variables, and are used mostly in FOG-M's *init_iris* routine.
- *Drawing Primitives* are used throughout FOG-M. They create points, lines, polygons, circles, arcs, and text strings.
- *Coordinate Transformations* specify mappings within and between user-defined world coordinates and screen coordinates. These are used to move targets and to simulate flight.
- *Drawing Attribute* commands specify textures and fonts. Although texture would greatly improve the appearance of the terrain, the IRIS provided textures are applied in the screen coordinate system, so they do not scale or tilt to conform to the terrain, and produce a very artificial result.
- *Display Mode and Color* commands determine how the bitplanes are used and what colors appear on the screen. These include the commands that set double-buffering, establish writemasks, and define the color table.
- *Input/Output* commands initialize and read the dials and mouse.
- *Object Creation and Editing* commands allow manipulation of complex displays as a single entity. They are used in all FOG-M displays.
- *Picking and Selecting* commands are not used in FOG-M.
- *Geometry Pipeline Feedback* commands are not used in FOG-M.
- *Curve and Surface* commands draw complex curves and smooth surfaces. Experiments with these produced more realistic terrain images, but not even close to real-time, making flight animation impossible.
- *Shading and Depth−cueing* commands provide Gouraud shading of polygons and intensities that vary with distance from the viewer.
- *Textport* commands define an area of the screen for text. They are not used in FOG-M.

Also available on the system, and used by FOG-M, are the math library with sine, cosine, arctangent, hypotenuse, and exponentiation functions, and routines that access the system clock in order to determine elapsed time.

III. DIGITAL ELEVATION TERRAIN DATA

A. INTRODUCTION

Unlike other flight simulation systems, which may rely on manual creation of the terrain [Ref. 8], the source data for the terrain in the FOG-M simulation is a Defense Mapping Agency (DMA) digital terrain elevation database (DTED) for Fort Hunter-Liggett, California. The database is not Level 1 DTED, but rather a DMA special product produced about 1980 at a higher resolution than normal Level 1 DTED [Ref. 9]. Level 1 DMA data contains elevation points spaced at three arc-second intervals, or approximately every one hundred meters. The Fort Hunter-Liggett special data contains points at twelve and one-half meter spacing, which is eight times the resolution of Level 1 data.

B. COVERAGE

The area covered by the database is thirty-six kilometers wide and thirty-five kilometers high, with 6400 data points per square kilometer. This area includes most of Fort Hunter-Liggett plus some surrounding land, and is bounded by latitudes $36^{\circ}\ 05'\ 00''$ (to the north) and $35^{\circ}\ 50'\ 00''$ (south) and longitudes $121^{\circ}\ 04'\ 30''$ (east) and $121^{\circ}\ 20'\ 30''$ (west). In terms of Universal Transverse Mercator (UTM) coordinates, the area has easting (X) of 10SFQ41000 to 10SFQ77000 and northing (Y) of 10SFQ60000 to 10SFQ95000. The database

appears to be based on DMA forty foot interval contour map products, because peaks tend to have flattened tops. This was confirmed both by a comparison of surveyed instrumentation sites on or near peaks with their digital terrain values [Ref. 10: pp. 1-2], and by a Bezier surface patch image of the data created locally.

C. STRUCTURE

The data is stored in an unformatted sequential file that is organized as a stream of integers. Each integer (sixteen bits) represents both the vegetation code and bald terrain elevation in feet at one sampling point, as illustrated in Figure 3.1 below.

	Veg. Code			Bald Terrain Elevation												
bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Figure 3.1 DTED Data Encoding

The thirteen low-order (rightmost) bits contain the elevation, allowing a range from zero to 8191 feet, although the highest point in the database is 3744 feet. The three high-order (leftmost) bits specify one of eight vegetation codes, which are given in Table 3.1 below. Vegetation codes are only available for points within the boundaries of Fort Hunter-Liggett proper. The file is written one

TABLE 3.1 DTED VEGETATION CODES

Code	Description
0	Less than one meter
1	One to four meters
2	Four to eight meters
3	Eight to twelve meters
4	Twelve to twenty meters
5	Greater than twenty meters
6	No data available
7	Unused

square kilometer at a time, beginning with the lower left one kilometer grid square (41,60), proceeding up the column to the upper left grid square (41,94), then doing the next column from bottom to top (42,60 to 42,94) and so on; the upper right one kilometer grid square (76,94) is the last one written. Within each one kilometer grid square, the individual data points are written in the same pattern, beginning with the lower left, doing each column from bottom to top, and doing the columns from left to right. This file layout is summarized in Figure 3.2. The position in the file of the elevation for a point expressed in five digit local UTM X and Y coordinates is found as shown in Equation 3.1.

$$position = 35 * (integer(X/1000) - 41) + (integer(Y/1000) - 59) \quad (3.1)$$

D. LOCATION

The complete DTED file occupies 16,128,000 bytes of storage. Due to a local shortage of available disk space, this file must permanently reside on the UNIX VAX 11/785 system rather than on the IRIS system. The FOG-M simulator

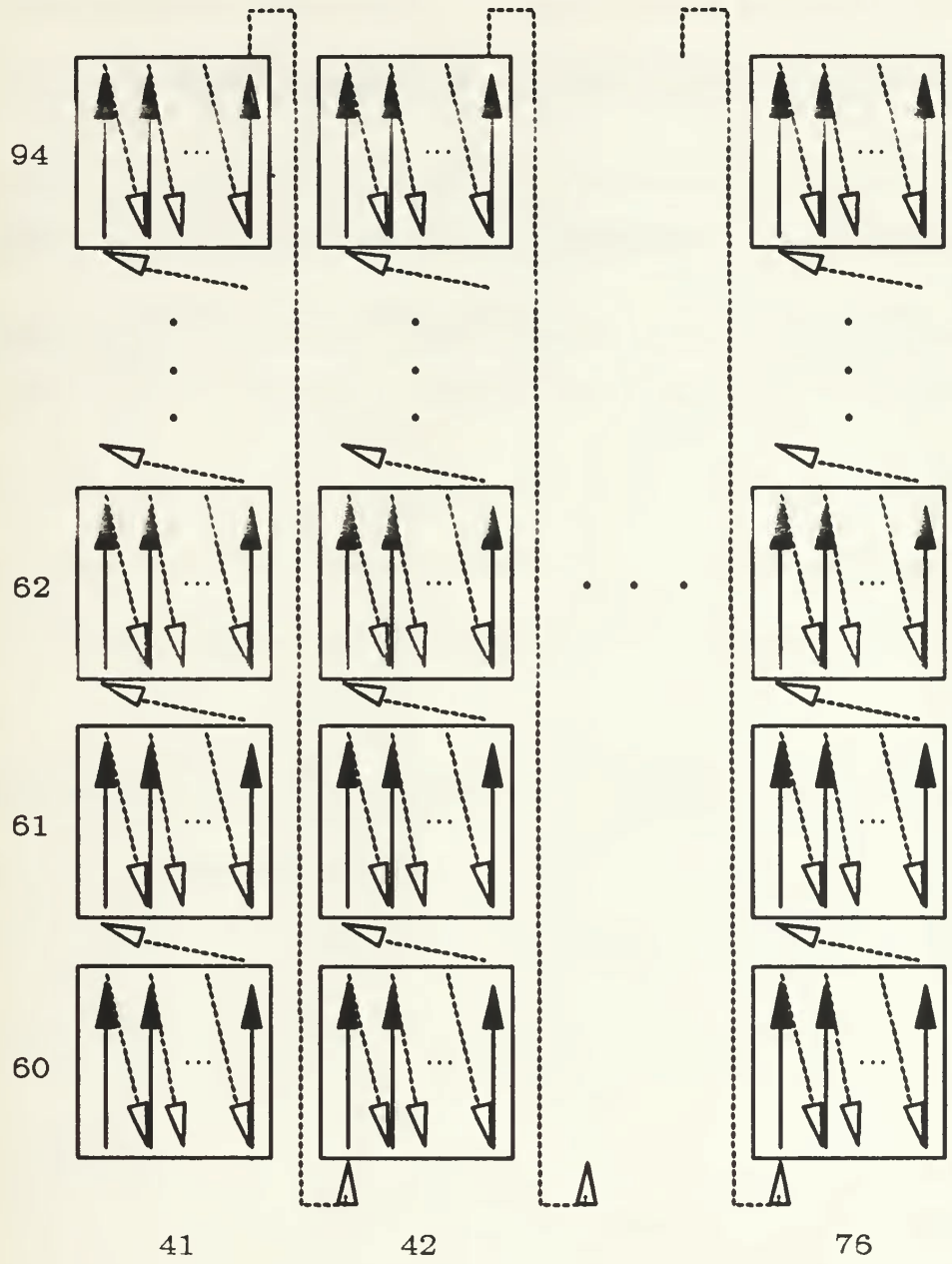


Figure 3.2 DTED File Layout

presently operates on a ten kilometer square extract from this database. A program on the VAX called **make−database−e** allows interactive specification of the area and resolution desired, and produces an extract. This extract is sent over the Ethernet to the IRIS to serve as the input for a FOG-M run. However, if the data is sent directly, it is received with each pair of bytes *swapped*, so another program, **swapdma**, is run on the VAX before transmittal. This program swaps the low- and high-order bytes of each integer so that the swapping during transmission is cancelled.

IV. TWO-DIMENSIONAL TERRAIN MAP PORTRAYAL

The two-dimensional representation of the terrain was begun as the first graphics portion of the system, in order to gain familiarity with the IRIS graphics workstation and the Defense Mapping Agency (DMA) digital terrain elevation data (DTED). Contour maps are the traditional approach to two-dimensional terrain portrayal, and thus were the basis for the two-dimensional images of the terrain generated here (Figure 4.1). Although these two-dimensional images are not true contour maps, they are still referred to as such in this study because of their close relation and common origin. The algorithms for determining and drawing the forty foot contour lines found on a normal contour map seemed non-trivial, so a simpler alternative was chosen. Each elevation datum is represented by a *tile*, with the implicit X and Z (easting and northing, respectively) coordinates of the elevation datum being the center of the tile.

A. COLORS

The color of a tile is determined by its vegetation code, and its intensity (or *shading*) by its elevation. The intent was to use green for tiles with vegetation and brown for tiles without vegetation. However, the DTED vegetation codes lump together both "no vegetation" and "vegetation less than one meter high." Brown tiles thus include both unvegetated areas (e.g. rock slabs, areas above the

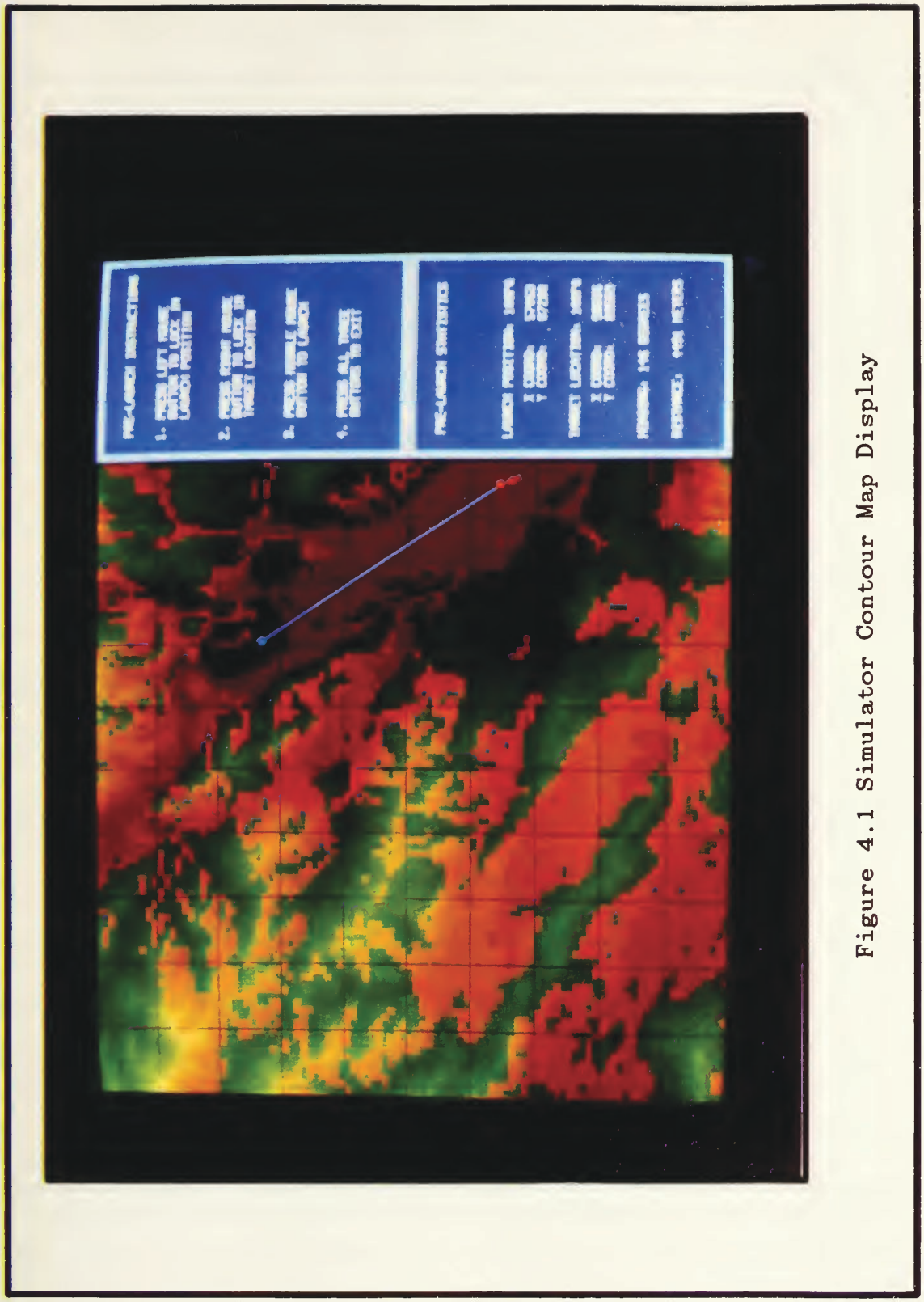


Figure 4.1 Simulator Contour Map Display

treeline) and grasslands or meadows. This is significant in the Fort Hunter-Liggett area, because most of the valleys are covered in grass, and all of the high ground is below the treeline. The result is a map with brown valleys and green ridgelines. While this was readily accepted as natural by most viewers, pilots with a background in low-level flight found it awkward, and contrary to their expectations (from flight charts) of green valleys and brown mountains. While this might be significant in other flight simulation applications (particularly those designed for pilots), the initial representation was deemed most appropriate for the target audience of the FOG-M simulator.

A similar initial, intuitive choice was made for the elevation-keyed shading. High intensity (light) colors were used for higher elevations, and low intensity (dark) colors for lower elevations. This was accepted as natural by almost all viewers. The optimum number of intensities (shadings) to use in the map was experimentally determined to be sixteen. A small power of two was desirable due to the nature of the writemasks used to improve display speed. A large number of colors provides greater elevation definition and prevents large masses of the same color in areas where elevations change gradually. However, having too many colors destroys the contour-map effect, since adjacent colors are so close that no boundary is distinguishable between them. Eight shades each of green and brown were used initially. The shift to sixteen shades of each produces a better looking map. Due to the RGB (red, green, blue) nature of color creation on the IRIS, the greens were still barely differentiable at thirty-two shades, but the browns (a

combination of mostly red, some green, and, in some shades, a trace of blue) began to blend together.

To determine the elevations at which color shades should change (in order to use the full range of shades), the maximum and minimum elevations of the terrain section in use must be known. Rather than preprocess the data before each run, these values are coded as constants in a header file. The equation for which color index to use is straightforward (see Equation 4.1) but takes significant time when repeated ten thousand times.

$$index = base\_index + \frac{elevation - MIN}{MAX - MIN} * \#\_of\_shades \qquad (4.1)$$

Therefore, the fifteen points at which the shade changes are precalculated and stored in an array so that no calculations are needed at each point, just an array lookup.

B. DRAWING

The map can then be produced by determining the color and shade for each tile, and drawing it as a filled square. However, an increase in speed can be gained by exploiting the structure of the data and the line drawing hardware of the IRIS. The data is still processed a point at a time within each one kilometer column, but no drawing is done until an elevation/shading breakpoint is reached. Then a single line of one tile's width is drawn to color all tiles since the previous elevation breakpoint.

C. WRITEMASKS

A more significant speed improvement (on the order of fifty per cent more frames per second) was achieved with writemasks. Writemasks are a relatively low-level hardware feature that can be used for many purposes. In the FOG-M simulator, they are used to prevent the contour map from being overwritten. This allows the map to be drawn only once into the bitplanes, and have it remain on the screen without being re-drawn during each frame update. In order to understand how writemasks work, one must understand the layout and use of the IRIS's color table and bitplanes.

1. Color Table

The color table associates a particular binary number with a color. When the display system asks what color some number is, the color table replies with the intensities for the red, green and blue color guns that will produce the color defined for the input number. This input number is referred to as a *colorindex*. Thus the color displayed on the screen depends on the colorindex associated with a given pixel, and the color associated with that colorindex in the color table. Table 4.1 gives the color table entries that are the defaults on the IRIS workstation.

2. Bitplanes

The colorindex that is associated with each pixel is stored in the display memory, which is composed of *bitplanes*. Each bitplane has one bit for each pixel on the display screen, so a bitplane is 1024 bits wide, 768 bits high and one bit

TABLE 4.1 IRIS DEFAULT COLORINDEX DEFINITIONS

Color	Colorindex	
	Decimal	Binary
Black	0	0000000000000000
Red	1	0000000000000001
Green	2	0000000000000010
Yellow	3	0000000000000011
Blue	4	0000000000000100
Magenta	5	0000000000000101
Cyan	6	0000000000000110
White	7	0000000000000111

deep. When used in *double−buffer* mode (as in FOG-M), the IRIS uses sixteen bitplanes (numbered 0 to 15) for each buffer. The *frontbuffer* is the one whose binary contents define the image being displayed. While the frontbuffer is being displayed, the next image is created by issuing drawing commands which affect only the *backbuffer*. Once a new image is completed in the backbuffer, the buffers are *swapped*, so the backbuffer becomes the frontbuffer and is displayed. The old frontbuffer becomes the backbuffer, and is then available for update.

3. Writemask Example

Consider the pixel at location (0,0) – the lower left corner of the screen. The colorindex of that pixel is determined by sixteen bits: one from the lower left corner of each bitplane. The display system reads those sixteen bits as a binary number (the colorindex), and uses the color table to determine what color to make that pixel. For example, using the default colors defined in Table 4.1 above, that pixel will be colored black if all sixteen bitplanes have zeroes in their lower-

left corners, since the value of the sixteen bit binary number 0000000000000000_2 is zero. If the current color is set to magenta (color five, whose binary value has ones in bits zero and two) and a drawing command is issued, bitplanes zero and two are set to one, and all other bitplanes are set to zero, for every pixel covered by the drawing command. These pixels will now be displayed as magenta, because the colorindex constructed from the sixteen bitplanes will be 0000000000000101_2 (5_{10}), and the color table tells the display system that color 5_{10} is magenta.

The previous example showed that a drawing command works by placing ones in certain bitplanes, and zeroes in all of the rest, with the current color specifying which bitplanes get which. A writemask tells each bitplane to either allow or ignore the changes a drawing command says to make. In normal double-buffered usage, the writemask is 1111111111111111_2, meaning all sixteen bitplanes should allow updates. Now suppose there is an image on the screen which uses just the default eight colors. Bitplanes three through fifteen are all zeroes, because all of the colors have colorindices with three or less binary digits, which will be in bitplanes zero, one, and two. If the writemask is changed to 1111111111111000_2 after drawing the image, those lower three bitplanes are "frozen" and will not be changed by any drawing command. Setting the color to black and clearing the screen *will not change anything*. The upper bitplanes will be set to all zeroes, which they already were. The lower three bitplanes will be told to reset to zero, but will not do it because they are protected by the writemask.

Now suppose you want to draw a grey line on top of the image. The line only needs one color, so it can be drawn in one bitplane. (Two bitplanes will allow three more colors on top of the map, three bitplanes allow seven, etc.) The first "free" bitplane is number three. Turning on a bit in this plane (and not turning on any bits in higher planes) requires a colorindex in the range 1000_2 to 1111_2 (8_{10} to 15_{10}). Defining color eight in the color table as grey, making color eight the current color, and then drawing the line is sufficient to get the image into the bitplanes, but the display will not show the desired effect. If the image underneath the line is black (i.e. bitplanes zero through two are all zeroes form some pixels), the line will appear grey, as intended, for those pixels. However, if the image underneath the line is red (i.e. the lower bitplanes contain 001_2), the composite colorindex retrieved by the display system is 0000000000001001_2 or 9_{10}) and since color nine is not defined in the color table, it appears as black. Thus every colorindex that has bit three (because the line is in bitplane 3) set to one (i.e. colorindices 1000_2 to 1111_2, or 8_{10} to 15_{10}) must be defined as grey in order to produce the desired image.

4. Writemasks in FOG-M

The map image used in FOG-M is stored in the first six bitplanes (numbered 0 through 5) of both buffers, which means sixty-four colors are available: eight are the IRIS defaults, sixteen are shades of brown, sixteen are shades of green, and twenty-four are unused. The writemask defined as SAVEMAP ($C0_{16}$ or 0000000011000000_2) allows things to be drawn on top of the

map in bitplanes six and seven. Colorindices 64 through 127 are all defined as blue in the color table, so anything drawn in bitplane six appears on top of the map in blue. Similarly, bitplane seven is used for red, with colorindices 128 through 255 all correspondingly defined to be red.

The speed improvement due to writemasks in FOG-M comes from not having to re-draw the map each time the screen is updated. The cost is the use of many more indices in the color table, which limits the number of colors available for use on top of the map. For our simulation system, with only two colors needed on top of the map, there is plenty of room in the color table. Therefore, the gain in speed comes at no real cost.

V. THREE-DIMENSIONAL TERRAIN CONSTRUCTION

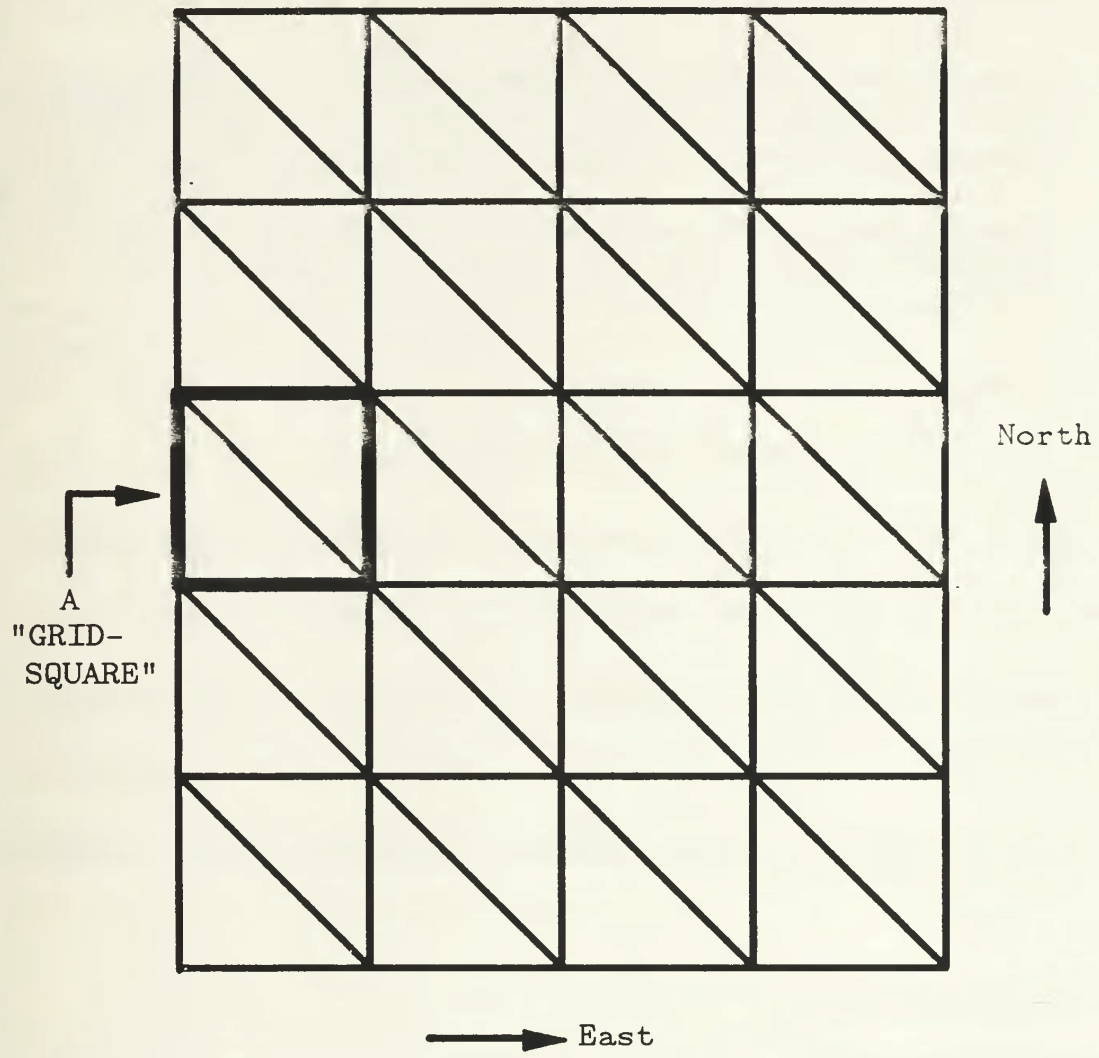
A. REPRESENTATION DECISIONS

1. Polygons versus Patches

Early experiments in the study involved attempting to display the terrain using *parametric bi-cubic surface patches*. A surface patch is simply a smooth curved surface fitted to a set of data points. A discussion of the theory and use of surface patches can be found in the IRIS User's Guide [Ref. 7:sec. 11-3] and Hearn and Baker [Ref. 11:pp. 193-205]. It was quickly determined that it would not be possible to use surface patches to represent the terrain and still maintain a real-time update of the terrain during flight.

An alternate method of displaying a three-dimensional object is through the use of a set of planar polygon surfaces that join at common edges to form the terrain object. This method has the advantage of being much simpler, and therefore faster, to generate and display. For this reason it was chosen for use in the project.

Figure 5.1 shows the method of constructing the terrain surface as a set of triangles. The term *gridsquare* is used in the remainder of the chapter to refer to a set of two triangles with a common hypotenuse that form a square of the terrain grid.



View from above looking down on the terrain.

-Terrain elevation points are connected to form triangular polygons with common edges.

Figure 5.1 Polygonal Terrain Construction

2. Resolution

The special DMA data file used in this project contains elevation data that is spaced at a twelve and one-half meter interval. One of the first questions which had to be answered concerning the three-dimensional portrayal of this data was, "In how fine a resolution can the data be displayed, while still allowing for a sufficient frame update speed?" Early test runs showed that using the full twelve and one-half meter resolution would be much too slow, although it provided an excellent representation of the terrain. An adequate frame update rate (approximately three to four frames per second) was achieved with a seventy-five meter resolution or every sixth data point. Since this was an early test, displaying terrain without any targets or cultural features, a one hundred meter resolution was decided upon for use in the remainder of the project. This allowed for an adequate "cushion" of processing time to complete the additional computations that would be needed in the final product, while still providing an adequate degree of resolution.

3. Elevation Scaling

After viewing the early representations of the terrain, it appeared that the hills did not give an appropriate appearance of height. Although this was a subjective judgement, it was shared by most people who viewed the display and compared it to aerial photographs of the area. Because of this, it was decided to scale the elevations of the displayed points upward. Two approaches, linear scaling and exponential scaling, were examined.

In the linear scaling approach, each elevation point was simply multiplied by a scale factor as shown in Equation 5.1.

$$Elev_{new} = \sigma * elev_{old} \tag{5.1}$$

Using this approach, it appeared that a scaling factor between 1.5 and 2.0 was necessary to achieve the desired effect.

In the exponential approach, the elevation of each point was raised to a fixed power as shown in Equation 5.2.

$$Elev_{new} = Elev_{old}^{\sigma} \tag{5.2}$$

This approach has the effect of exaggerating the higher elevations to a greater degree than the lower ones. It was chosen as the approach for use in the project based on subjective observations of the displays produced by the two methods. The scaling factor, σ, was chosen as 1.05. Using this factor produces the equivalent of a linear scaling of 1.5 for the maximum elevation and 1.4 for the minimum elevation contained in our area of interest.

Subsequent to the decision to use an exaggerated elevation scale, research results were discovered which supported it. In a study conducted by the U.S. Army Research Institute for the Behavioral and Social Sciences, observers were asked to pick a computer generated line drawing that best matched actual terrain. The line drawings had different exaggerations of the vertical (elevation) scale. The overall ratios chosen by the four observers ranged from 1.25:1 to

1.50:1. The drawings presented to the observers had exaggeration ratios ranging from 1:1 to 1.75:1. [Ref. 12]

4. Shading and Texturing

As explained above, each one hundred meter square of the terrain, a "gridsquare," is represented by two triangles in three-space that share a common diagonal edge. The process of applying colors to these polygons, *shading*, was the next area of research in the project.

a. Elevation Based Shading

Three different shading algorithms were investigated. The first was a simple algorithm where the shade of a polygon was a function of its elevation. Higher elevations are shaded in lighter shades of green while lower elevations receive darker shades. Equation 5.3 represents the assignment of a shade from the color map.

$$color\_index = base\_index + \frac{elev - Min\_Elev}{Max\_Elev - Min\_Elev} * \#\_of\_shades \quad (5.3)$$

The darkest green is stored in the *base_index* color map location and the lightest green in the *baseindex* + *#_of_shades* location. Although this approach works well for two-dimensional contour maps (see Chapter IV), and is currently used in another "low cost" simulator [Ref. 6], it did not appear to present a realistic view of the terrain. An advantage of this approach, however, is that the calculation of the color index is simple enough to be done with no preprocessing.

b. Lambert's Cosine Law Shading

The second method of determining the shade for a polygon involved the use of a point light source and Lambert's cosine law [Ref. 11:p. 278]. Let $\vec{N}$ be a unit normal vector to the polygon, and $\vec{L}$ be a unit vector in the direction of the light source. The angle between $\vec{N}$ and $\vec{L}$, Φ, is the *angle of incidence*. Lambert's Law states that the intensity of the light reflected from the polygon is proportional to cos Φ (Equation 5.4).

$$I \alpha \cos \Phi \tag{5.4}$$

In order to use this law, the normal vector ($\vec{N}$), the light source vector ($\vec{L}$), and the angle between them (Φ) must be known. $\vec{N}$ can be determined by taking the cross product of $\vec{v1}$ and $\vec{v2}$, where $\vec{v1}$ is a unit vector in the direction from vertex B to vertex C of the polygon, and $\vec{v2}$ is a unit vector in the direction from vertex B to vertex A of the polygon (Equation 5.5 and Figure 5.2).

$$\vec{N} = \vec{v1} \times \vec{v2} \tag{5.5}$$

With $\vec{N}$ and $\vec{L}$ available, cos Φ can be computed as their dot product (Equation 5.7).

$$\cos \Phi = \vec{N} \cdot \vec{L} \tag{5.7}$$

Since the intensity is proportional to cos Φ, the appropriate color index to use can be computed as

$$color\_index = min\_index + (\#\_shades * \cos \Phi) \tag{5.8}$$

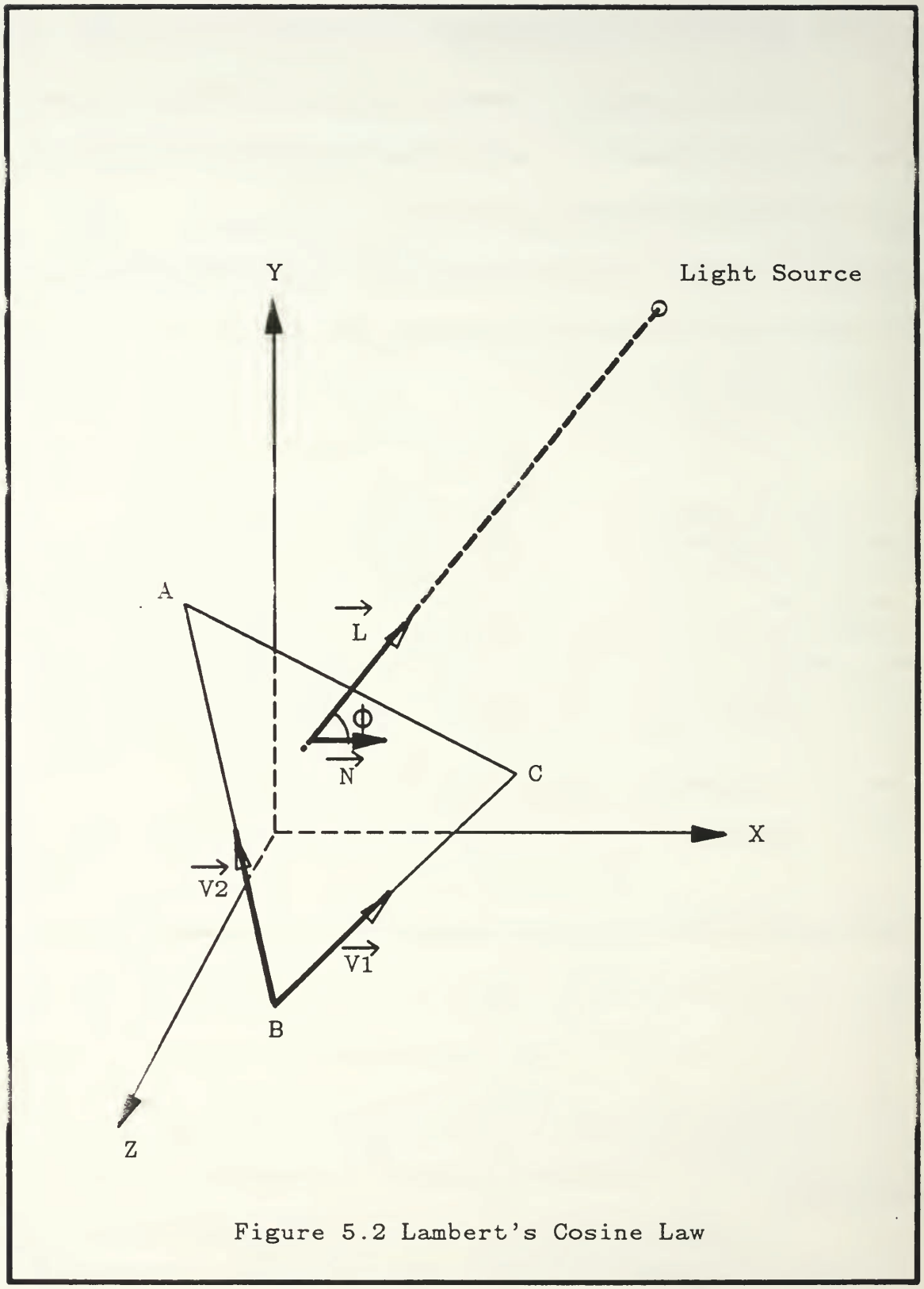


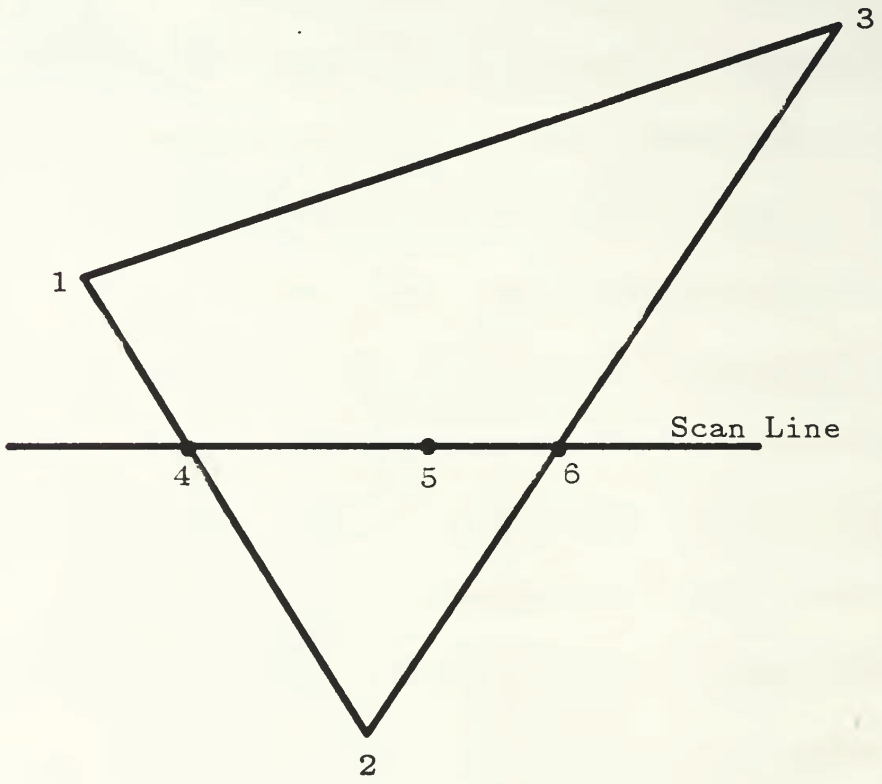
Figure 5.2 Lambert's Cosine Law

where *min_index* is the color index of the lowest intensity green and *min_index* + #_*shades* is the color index of the highest intensity green.

c. Gouraud Shading

The final shading model investigated involved the use of *Gouraud shading*. The purpose of Gouraud shading is to provide a continuous transition of shades across a polygon so that the shades at the edges of adjoining polygons match. This in effect eliminates the visible boundary between polygons and provides a smooth continuous surface. The Gouraud algorithm involves interpolating to determine the intensity to be used at each pixel along a scan line, and is illustrated in Figure 5.3 as reproduced from Hearn and Baker [Ref. 11:p. 290]. To use the algorithm, intensity values for each vertex of the polygon must be known. In the project's implementation, the intensity at each vertex was computed as the average of the intensity values for all the polygons meeting at that vertex, where the individual polygon's intensity values were calculated using Lambert's cosine law.

The use of this model posed two problems. First, even though the IRIS supports Gouraud shading in its graphics library, its use increased the time between frames to an unacceptable rate (approximately one and one-half to three seconds between frames). Second, the smoothing of the algorithm worked too well, resulting in terrain displays that lacked the necessary position cues to detect motion. This second problem could be alleviated by adding artificial texture to the terrain but in light of the speed problem it was not pursued further.



For interpolated shading, the intensity value at point 4 is determined from intensity values at points 1 and 2, intensity at point 6 is determined from values at points 2 and 3, and intensities at other points (such as 5) along the scan line are interpolated between the values at points 4 and 6.

Figure 5.3 The Gouraud Shading Algorithm

d. Adding Texture

Lambert's cosine law was chosen as the shading model for use in the project, providing the most realistic display within the allowed computation time constraints. However, a problem with its use is that the flat valleys, with little variance in the surface normals of their polygons, produce large geographic areas having a near constant shade. This results in a lack of motion cues in these areas similar to that experienced with the Gouraud shading model. To remedy this situation, a simple artificial texture, in the form of a checker board, was imposed on the terrain. The checker board effect was implemented as follows. First, the shades for the two triangles in each gridsquare were averaged, and this average shade was used for both of them. This of course causes the visible boundary between the triangles to disappear leaving a square shaded in a single color. Second, two slightly offset color ramps were used with adjacent grid squares using different ramps to compute their shades. One ramp is composed of green intensities ranging from 255 to 50, while the other uses intensities ranging from 245 to 40.* This causes the shades for two adjacent gridsquares with identical surface normals to vary, providing the necessary texturing.

*A value of 255 is the highest intensity green obtainable, a value of zero indicates the absence of the color green.

B. INTERNAL DATA STRUCTURES

Two global arrays are maintained which store the information necessary to display the terrain. The first is a five-dimensional array, *savetriangle*, that stores the values of the coordinates for each triangle making up the terrain structure. The second is a two-dimensional array *savecolor* that stores the color map indices for each of the terrain's grid squares. The purpose and range of each of *savetriangle*'s indices is shown in Table 5.1. For example, *savetriangle*[3][5][1][1][2] would contain the value of the Y coordinate (fifth dimension = 2), of the second vertex (fourth dimension = 1), of the northern triangle (third dimension = 1), of the grid square with X index five and Z index three (second dimension = five and first dimension = three).

TABLE 5.1 LAYOUT OF THE *SAVETRIANGLE* ARRAY

Dimension	Index Range		Purpose
	Start	End	
First	0	98	Grid square index in the Z direction. 0 is the southern most square, 98 is the northern most.
Second	0	98	Grid square index in the X direction. 0 is the western most, 98 is the eastern most.
Third	0	1	Triangle identifier within a grid square. 0 is the southern triangle. 1 is the northern.
Fourth	0	2	Vertex number of the triangle. 0 is the first vertex, 2 is the last.
Fifth	0	2	Coordinate identifier of the vertex. 0 is the X coordinate, 1 the Y coordinate and 2 the Z coordinate.

Table 5.2 lists the purpose and ranges of each of *savecolor*'s indices. For example, *savecolor*[30][10] contains the color map index to be used for the grid square with a Z index of thirty and an X index of ten.

TABLE 5.2 LAYOUT OF THE *SAVECOLOR* ARRAY

Dimension	Index Range		Purpose
	Start	End	
First	0	98	Grid square index in the Z direction. 0 is the southern most square, 98 is the northern most.
Second	0	98	Grid square index in the X direction. 0 is the western most, 98 is the eastern most.

These two arrays contain all the information necessary to construct an image of the terrain. The following chapter provides the details of using their data to create a real-time, updated image of the terrain as it is seen from the FOG-M's camera.

VI. FLIGHT SIMULATION

A. OVERVIEW

The previous chapter discussed the methodology of constructing the three-dimensional terrain from the provided elevation data. This chapter's purpose is to explain the details of displaying this terrain in real time as it is seen through the missile's camera.

The high level pseudocode for the main program's terrain display loop is shown in Figure 6.1. Chapter VII explains the details of step two. The details of steps one and six are explained in Appendix B under the procedures *readcontrols* (for step one) and *edit_navbox* and *edit_indbox* (for step two). The remainder of this chapter discusses the details, considerations, and results of implementing steps three through five.

B. UPDATING THE MISSILE'S POSITION

Determining the missile's new position can be broken into two cases:

[1] the missile is under operator control and its new position is a function of the old position, the commanded direction of flight, the commanded altitude, and the commanded speed.

[2] the missile is locked onto a target and its new position is a function of its old position, the position of the desired target, and the commanded speed.

In both cases, a very large simplifying assumption is made to **ignore the dynamics of the missile's flight**. This means that the missile is able to

While missile is flying do

1) Read the values from the operator's controls

2) Determine new positions for all the targets

3) Determine the new position for the missile

4) Determine the position of where the camera is looking

5) Display the terrain as seen by the camera

6) Update the operator's control indicators

End while

Figure 6.1 Main Display Loop Pseudocode

instantaneously change heading, speed, and altitude. This assumption was made only because of development time constraints. It is felt that the computations necessary to more realistically model the dynamics of the flight can be done without a serious degradation of the simulator's performance.

1. Case 1 - Operator Control

Under this case the missile's X, Y, and Z coordinates are computed as shown below.

$$\Delta Dist = Speed^{*}\Delta Time \tag{6.1}$$

Where

- $\Delta Dist$ is the distance traveled over the ground since the last position was calculated.
- $Speed$ is the missile's speed in feet per second and
- $\Delta Time$ is the elapsed time since the last position was calculated

Having calculated the distance the missile must move during this frame, the missile's new coordinates (MX,MY,MZ) can be calculated as

$$MX_{new} = MX_{old} + [cos(Dir_{cmd}) * \Delta Dist] \quad (6.2)$$

$$MZ_{new} = MZ_{old} - [sin(Dir_{cmd}) * \Delta Dist] \quad (6.3)$$

$$MY_{new} = (Alt_{cmd})^{\sigma} \quad (6.4)$$

Where

- Dir_{cmd} is the commanded heading in radians
- Alt_{cmd} is the commanded altitude in feet
- σ is the altitude scaling factor (see Chapter V, Section A.3).

2. Case 2 - Locked Onto a Target

In the case where the missile is locked onto a target, the missile's new position is computed as follows. $\Delta Dist$ is computed as in Equation 6.1. Next the missile's heading is computed so as to steer it directly toward the target's position:

$$Dir_{tgt} = arctan2(-[TZ-MZ],[TX-MX]) \quad (6.5)$$

Where

- Dir_{tgt} is the direction from the missile's position to the target's position
- TX is the X coordinate of the target's position
- TZ is the Z coordinate of the target's position
- MX is the X coordinate of the missile's position
- MZ is the Z coordinate of the missile's position
- $arctan2(a,b)$ is a function which returns the $arctan\left(\frac{a}{b}\right)$ in the range 0 *to* 2Π, based on the sign of a and b.

Once Dir_{tgt} is known, the missile's new X and Z coordinates can be calculated as

$$MX_{new} = MX_{old} + [cos(Dir_{tgt}) * \Delta Dist] \tag{6.6}$$

$$MZ_{new} = MZ_{old} - [sin(Dir_{tgt}) * \Delta Dist] \tag{6.7}$$

Next the missile's altitude (MY) is adjusted a proportion of the total altitude difference between it and the target, based on the ratio of $\Delta Dist$ to the total distance (along the horizontal plane) to the target.

$$Dist_{tgt} = \sqrt{(TX - MX)^2 + (TZ - MZ)^2} \tag{6.8}$$

$$MY_{new} = MY_{old} - \left[-TY) * \frac{\Delta Dist}{Dist_{tgt}} \right] \tag{6.9}$$

Where

- $Dist_{tgt}$ is the distance to the target measured along a horizontal plane.
- MY and TY are the Y (altitude) coordinates of the missile and target, respectively.

C. DETERMINING THE LINE OF SIGHT

Once the new position of the missile has been calculated, the next step in displaying the terrain is to determine another point along the camera's line of sight: the **look-at** position. This calculation is also broken into two cases based on whether the missile is or is not locked onto a target (see Figure 6.2).

The case where the missile is locked on is trivial, the look-at position is simply set to the coordinates of the locked-on target.

$$LX = TX \tag{6.10}$$

$$LY = TY \tag{6.11}$$

$$LZ = TZ \tag{6.12}$$

Where LX, LY, and LZ are the X, Y, and Z coordinates of the look-at position. This centers the target in the displayed three-dimensional scene.

When the missile is not locked onto a target, the camera's look-at position is a function of the missile's position, the missile's heading, and the pan and tilt angles of the camera. It is determined as follows

$$Dir_{look} = Head_{msl} + Pan \tag{6.13}$$

$$LX = MX + [\cos(Dir_{look}) * Dist_{look}] \tag{6.14}$$

$$LZ = MZ - [\sin(Dir_{look}) * Dist_{look}] \tag{6.15}$$

$$LY = MY + [Dist_{look} * \tan(Tilt)] \tag{6.16}$$

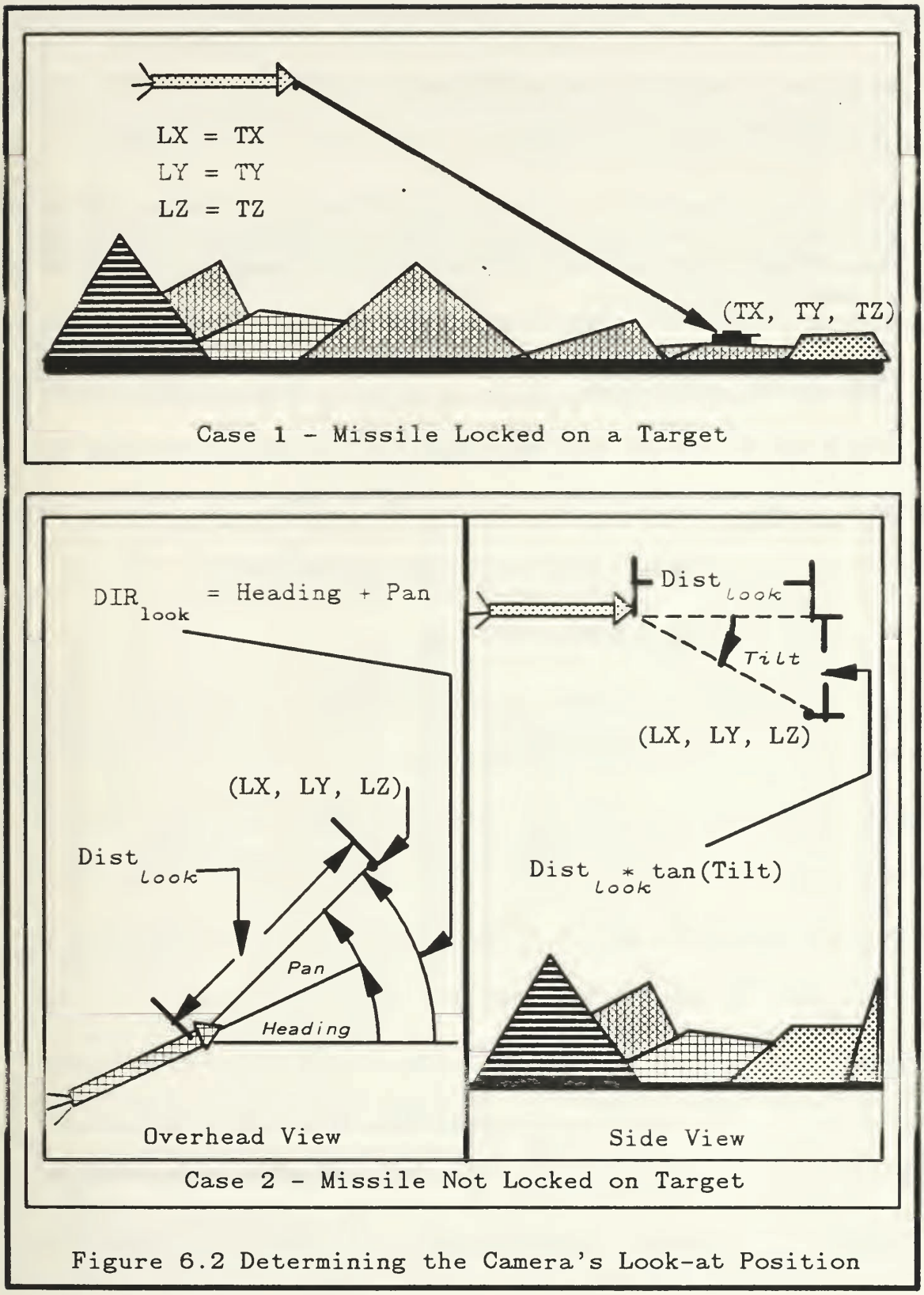


Figure 6.2 Determining the Camera's Look-at Position

Where

- Dir_{look} is the direction the camera is looking
- *Pan* is the pan angle of the camera
- *Tilt* is the tilt angle of the camera
- $Dist_{look}$ is an arbitrary distance over the ground that the camera looks ahead. Since the only purpose of LX, LY, and LZ is to determine a point along the camera's line of sight, any positive number will be acceptable. A value of five kilometers is currently used.

D. DISPLAYING THE SCENE

Once a line of sight has been determined, the next steps are to apply the appropriate viewing transformations, draw the filled polygons that make up the terrain, and add other items to the scene such as targets and roads.

1. Viewing Transformations

It is possible to project a three-dimensional object onto a two dimensional viewing surface in two basic ways. In one method, **the parallel projection** all the points of the object are projected along parallel lines. This has the advantage of preserving the relative dimensions and angles within an object and is used when accurate views of various sides of an object are needed such as in architectural drawings. In the other method, **the perspective projection**, all the points of an object are projected along lines that converge at a single point called the *Center of Projection*. In this method, relative dimensions are not preserved. Lines closer to the projection plane appear larger than those that are more distant. The perspective projection provides a view of three-dimensional

objects that is more realistic, similar to that provided by the human eye or a camera. Both these projections are illustrated in Figure 6.3. [Ref. 11:pp. 235-241]

Because of its more realistic presentation of the scene, a perspective projection was used for the project's three-dimensional scenes. The IRIS's graphics library provides a procedure called *perspective* which constructs the necessary transformation matrix* to obtain a perspective projection. The matrix is defined as [Ref. 7:p. C-2]

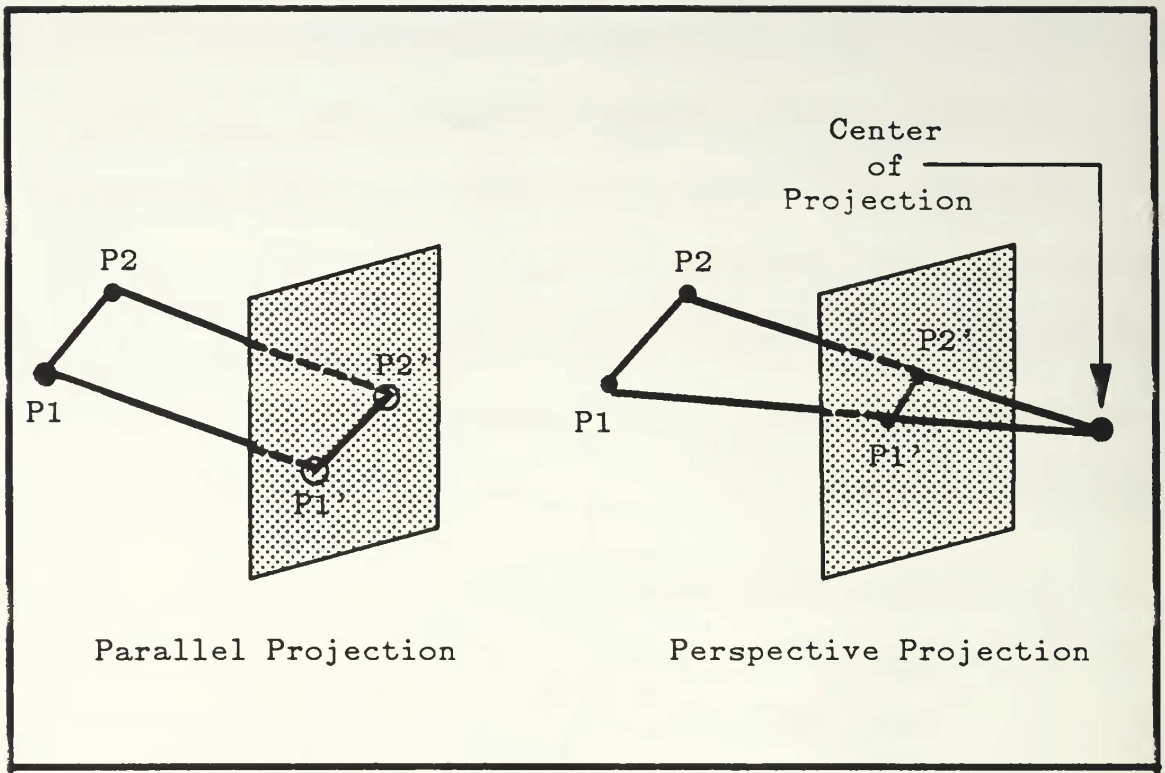
$$Perspective(fovy, aspect, near, far) =$$

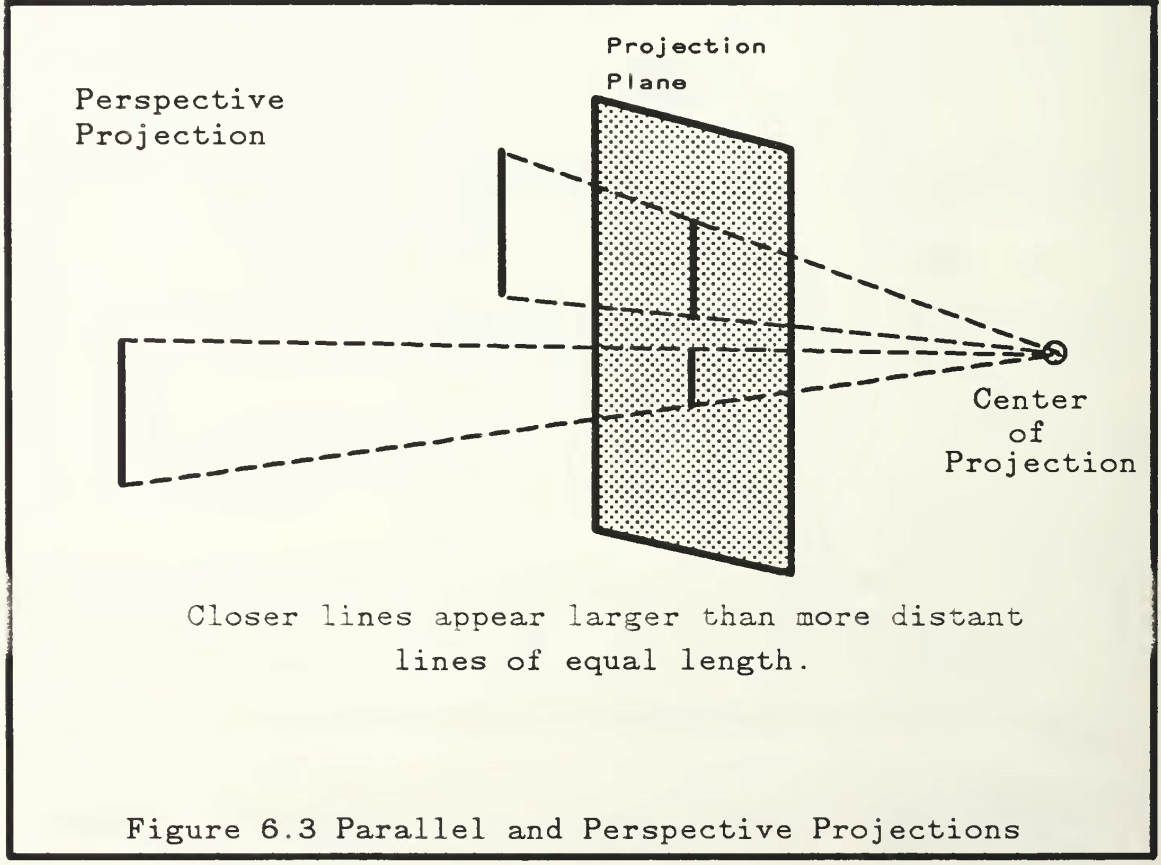
$$\begin{vmatrix} \frac{\cot(\frac{fovy}{2})}{aspect} & 0 & 0 & 0 \\ 0 & \cot(\frac{fovy}{2}) & 0 & 0 \\ 0 & 0 & -\frac{far+near}{far-near} & -1 \\ 0 & 0 & -\frac{2\times far\times near}{far-near} & 0 \end{vmatrix} \qquad (6.17)$$

Where

- *fovy* is the field of view angle
- *aspect* is the aspect ratio, a ratio of the distance a viewer sees in the X direction to the distance he sees in the Y direction. It is generally set to be the same as the ratio of the width to the height of the viewport.
- *near* and *far* are the distances from the viewer to the near and far clipping planes.

*A knowledge of using transformation matrices to perform graphical operations is assumed here. Hearn and Baker [Ref. 11:chaps. 11-12] provides excellent coverage of the subject.





Closer lines appear larger than more distant lines of equal length.

Figure 6.3 Parallel and Perspective Projections

The perspective projection forms a view frustum as shown in Figure 6.4. Any object within the frustum between the near and far **clipping planes** will be displayed in the scene. Objects outside this **view volume** are **clipped** and discarded.

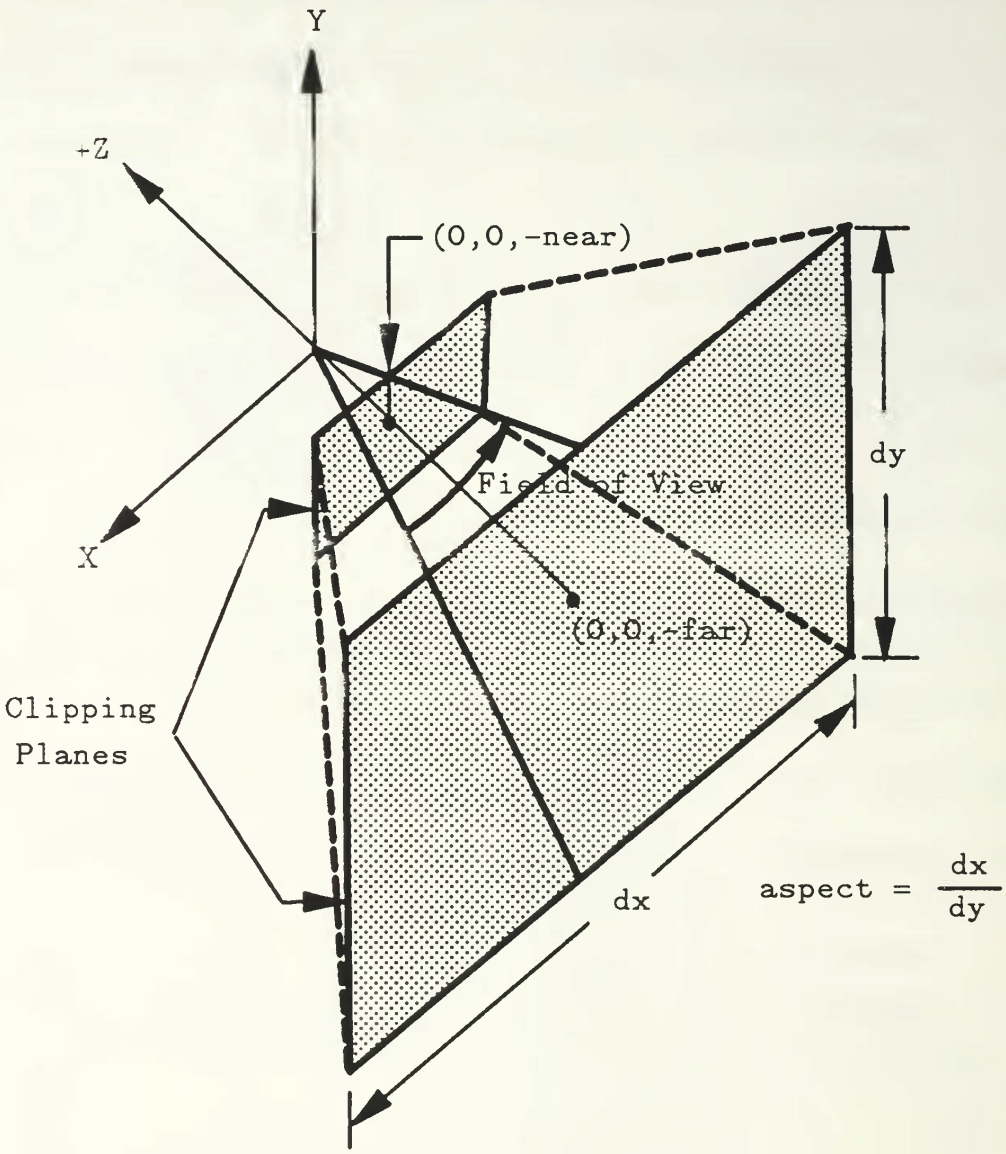
Next, the frustum formed by the perspective projection must be positioned along the camera's line of sight. This is accomplished by another transformation matrix constructed via a graphics library procedure named *lookat*. The *lookat* procedure takes the following inputs:

- V_x, V_y, and V_z: the *X, Y, and Z* coordinates of the center of projection.
- P_x, P_y, and P_z: the *X, Y, and Z* coordinates of the look-at position.
- *Twist*, a right handed rotation of the scene about the line of sight.

The transformation matrix formed by *lookat* is actually the result of multiplying four other transformation matrices [Ref. 7:p. C-2]

$$\begin{aligned} &Lookat(V_x, V_y, V_z, P_x, P_y, P_z, Twist) = \\ &Trans(-V_x, -V_y, -V_z) \times Rot_y(\Theta) \times Rot_x(\Phi) \times Rot_z(-Twist) \end{aligned} \tag{6.18}$$

$$Where\ Trans(-V_x, -V_y - V_z) = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -V_x & -V_y & -V_z & 1 \end{vmatrix} \tag{6.19}$$



The perspective command defines a near and far clipping plane, a field of view, and an aspect ratio.

Figure 6.4 The Perspective Command

$$Rot_y(\Theta) = \begin{vmatrix} cos(\Theta) & 0 & -sin(\Theta) & 0 \\ 0 & 1 & 0 & 0 \\ sin(\Theta) & 0 & cos(\Theta) & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} \tag{6.20}$$

$$Rot_x(\Phi) = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & cos(\Phi) & sin(\Phi) & 0 \\ 0 & -sin(\Phi) & cos(\Phi) & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} \tag{6.21}$$

$$Rot_z(-Twist) = \begin{vmatrix} cos(-Twist) & sin(-Twist) & 0 & 0 \\ -sin(-Twist) & cos(-Twist) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} \tag{6.22}$$

$$And \qquad \Theta = \sin^{-1}\left(\frac{P_x - V_x}{\sqrt{(P_x - V_x)^2 + (P_z - V_z)^2}} \right) \tag{6.23}$$

$$\Phi = \sin^{-1}\left(\frac{V_y - P_y}{\sqrt{(P_x - V_x)^2 + (P_y - V_y)^2 + (P_z - V_z)^2}} \right) \tag{6.24}$$

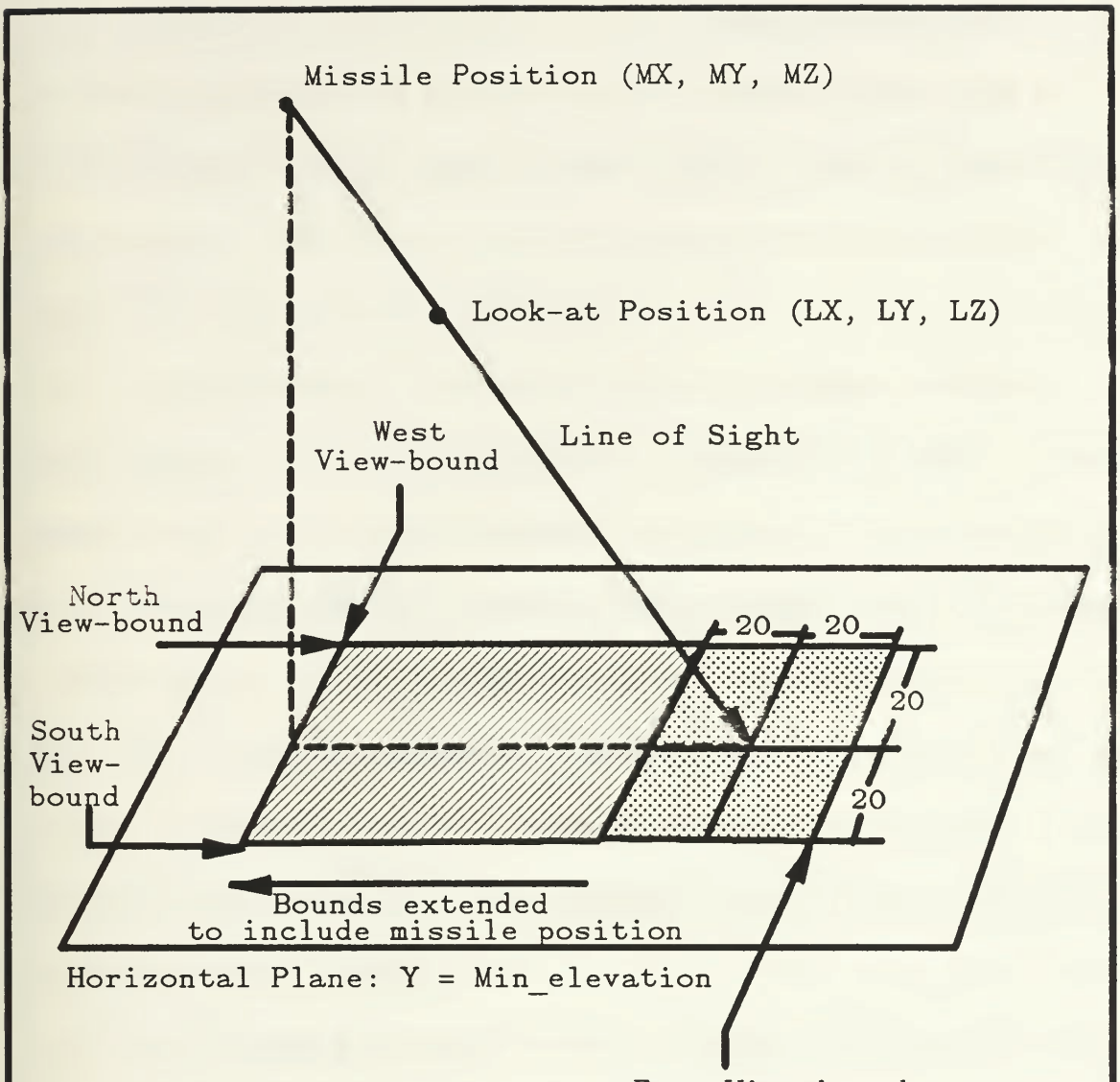
As can be seen, this transformation simply translates the center of projection to the origin, then rotates the view frustum about X and Y axes to align with the line of sight. Finally the twist angle is added with a rotation about the Z axis.

In the flight simulation, the twist angle is analogous to the "roll" angle of an aircraft or missile. A value of zero is currently used, but other values could be used if the roll of the missile during flight was added to the model.

2. Determining Which Polygons to Draw

After the correct viewing transformations have been applied, the polygons that comprise the scene must be drawn. Although the IRIS will "clip" polygons which lie outside the perspective projection's view volume, an increase in frame update speed can be achieved by not attempting to draw those that obviously lie outside. This is discussed further in the following section on simulator performance.

The term *view−bound* is used to describe a north-south oriented bounding box around those parts of the scene that are sent to the graphics pipeline. The view-bound is described by the index of the northernmost, southernmost, easternmost, and westernmost *gridsquare* to be drawn. It is calculated by extending (if necessary) the line-of-sight vector until it intersects the horizontal plane $Y = Min\_elev$, where Min_elev is the minimum elevation value of the terrain. The *view−bound* is calculated as being 20 *gridsquares* to the north, south, east, and west of this intersection point. If the missile's X and Z coordinates are not within the calculated *view−bound*, the bounds are extended to include them. Figure 6.5 illustrates this construction.



1) Line of sight vector is extended down to intersect the minimum elevation plane.
2) View bound extends 20 gridsquares north, south, east and west of the intersection.
3) Bound is extended, if necessary to include the missile's position.

Figure 6.5 Construction of the View-bound

3. Hidden Surface Removal

A final detail that must be taken care of is the removal of **hidden surfaces** from the scene. A hidden surface is simply a part of the scene that is obscured by some object in the foreground, such as a valley that it hidden behind a large hill.

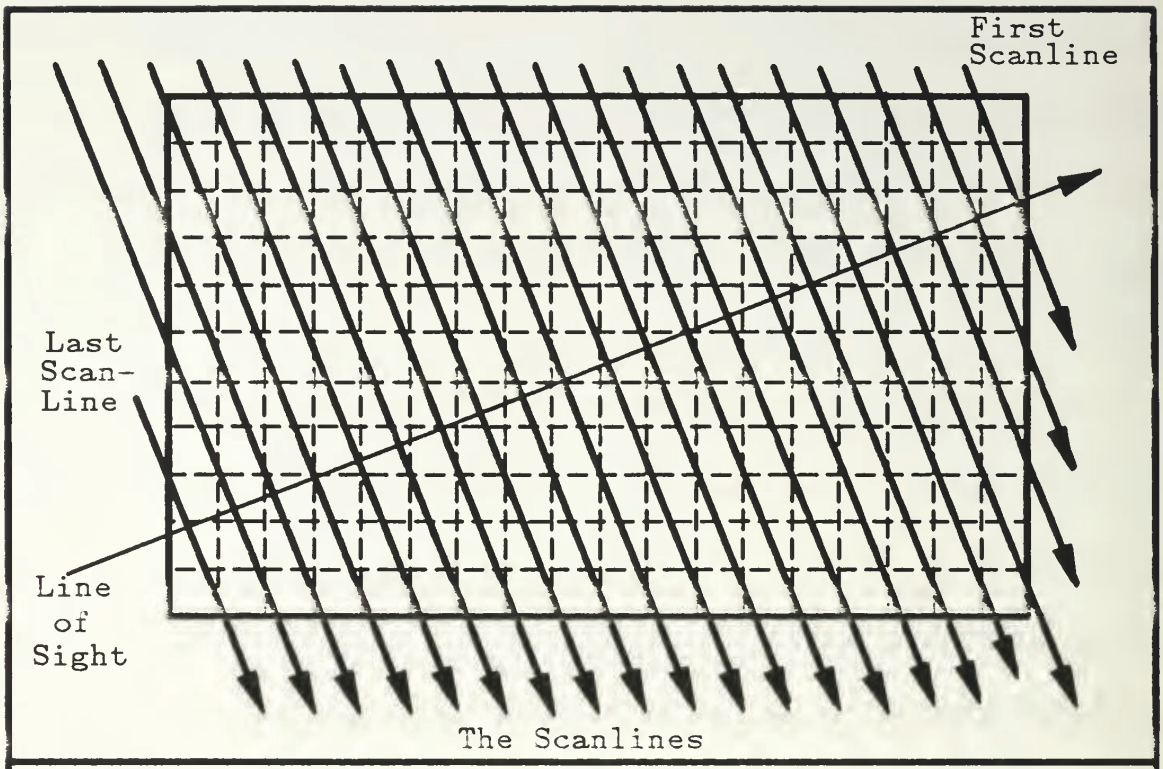
The IRIS supports a method in hardware called *Z-Buffering*. In this method, a buffer is maintained for each pixel position on the monitor and contains the "depth" (transformed Z coordinate) of the part of the scene that generated that pixel. Before drawing is started, the buffer is initialized to the maximum depth value (the value of the far clipping plane) for each pixel position. Before each new pixel is drawn, its depth is compared to the depth stored in the buffer. If its depth is greater than the stored depth it is not drawn. If it is less than the stored depth, it is drawn and its depth value replaces the value in the buffer. This method could not be used in the project for two reasons. First, with comparisons having to be made on a pixel-by-pixel basis, it slows down the frame update rate to an unacceptable level. Second, the IRIS does not allow the use of Z-buffering and double-buffering at the same time. Double-buffering is necessary to implement the animation of the scenes.

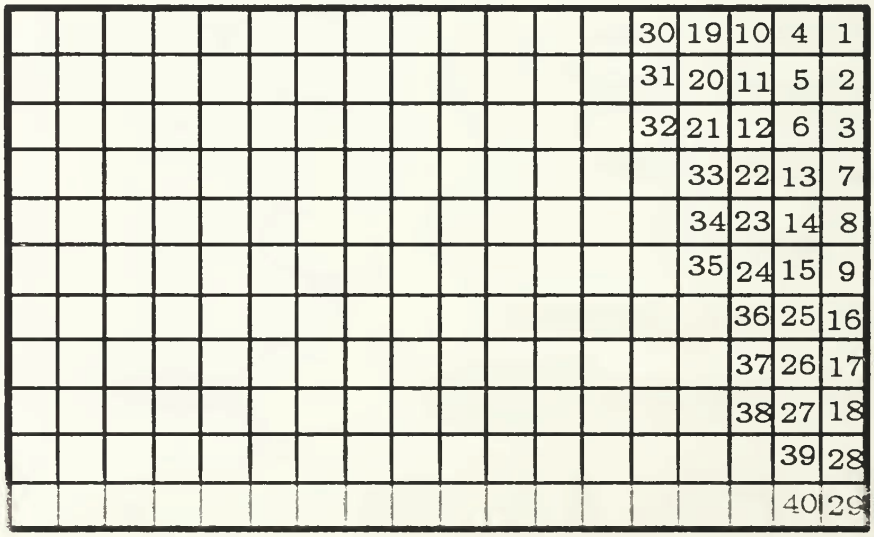
Another common method of hidden surface removal is the *painter's algorithm*. It derives its name from the way a painter would draw a scene on canvas, drawing in all the background and then adding foreground objects by painting over the background objects they obscure. Implementing this algorithm

in computer graphics means drawing the scene in an ordered fashion, such that the most distant objects from the viewer are drawn first and those closest to the viewer are drawn last. Since the gridsquares comprising the terrain form well defined rows and columns, an efficient implementation of this algorithm is possible. That implementation is described below.

The implementation can be thought of on a conceptual level as follows. A line, perpendicular to the line-of-sight, is constructed to serve as a pseudo-scanline. Gridsquares within the view-bound are drawn as they are intersected by this scanline. The scanline is first positioned along the line-of-sight vector so that it intersects the far corner gridsquare of the view-bound. After all the gridsquares along the scanline have been drawn, it is moved one gridsquare closer to the view position, along the line-of-sight vector, and the process is repeated. This continues until all the gridsquares within the view-bound have been drawn. Figure 6.6 illustrates this process.

From Figure 6.6, notice that each scanline passes through three gridsquares in a column, shifts over a column, then passes through three gridsquares in the next column. The number of gridsquares drawn in a column (or row) before advancing to the next column (or row) can be determined by computing the tangent of the scanline's direction. If the magnitude of the tangent is greater than 1.0, scanlines will run and shift along columns of gridsquares. If it is less than 1.0, scanlines will run and shift along rows of gridsquares. The term *threshold* is used in the remainder of the algorithm to





Drawing Order of the Gridsquares From
the First 5 Scanlines

Figure 6.6 The Scanline Hidden Surface Algorithm

describe the number of gridsquares drawn before a shift of column (or row) takes place. It is computed as

$$threshold = \begin{cases} nearest\_integer\left|\tan(Dir_{scan})\right| , & \text{if } \left|\tan(Dir_{scan})\right| \geq 1.0 \\ nearest\_integer\left|(\tan(Dir_{scan}))^{-1}\right| , & \text{if } \left|\tan(Dir_{scan})\right| < 1.0 \end{cases} \quad (6.25)$$

The pseudocode for implementing the algorithm is shown in Figure 6.7. The case shown is for a line-of-sight direction that is in the first octant (between 0 and $\frac{\Pi}{4}$ radians). The algorithm for the other seven octants is similar, the difference being the direction the scan line advances, and the direction it shifts when the threshold is reached. Table 6.1 summarizes these parameters for all eight octants.

TABLE 6.1 VARYING PARAMETERS FOR THE SCANLINE ALGORITHM BASED ON THE OCTANT OF THE LOOK DIRECTION

Octant	Look Directions		Scan Line Advances		When Threshold is Reached
	From	To	From	To	
1	0	Π/4	North	South	Shift one column East
2	Π/4	Π/2	East	West	Shift one row North
3	Π/2	3Π/2	West	East	Shift one row North
4	3Π/2	Π	North	South	Shift one column West
5	Π	5Π/4	South	North	Shift one column West
6	5Π/4	3Π/2	West	East	Shift one row South
7	3Π/2	7Π/4	East	West	Shift one row South
8	7Π/4	2Π	South	North	Shift one column East

Notice the step *draw gridsquare[z_index][x_index]* in the algorithm. Since a gridsquare contains terrain, and can also contain roads and targets, an

```
Calculate the threshold value

count ← 0

start_x_index ← west_view_bound
start_z_index ← north_view_bound

While start_z_index > south_view_bound do
  z_index ← start_z_index
  x_index ← start_x_index

  while (x_index ⩽ east_view_bound) and (z_index ⩾ south_view_bound) do

    { traverse a scanline }
    draw gridsquare[z_index][x_index]
    z_index ← z_index - 1  {move it one gridsquare south}
    count ← count + 1

    if count = threshold then
      x_index ← x_index + 1  {move it one gridsquare east}
      count ← 0          { reset count}
    endif

  end while

  {move on to next scanline: start it one gridsquare to the west}
  start_x ← start_x - 1
  count ← 0

  if (start_x < west_view_bound) then
    start_x ← west_view_bound
    start_z ← start_z - threshold
  endif

endwhile
```

Figure 6.7 Pseudocode for the First Octant Scanline Algorithm

ordering of these parts of the gridsquare must also take place. The two triangles forming the terrain are drawn first, next any roads are drawn, and finally any

targets are drawn. The details of integrating the targets and roads into the scene are covered in the following two chapters.

The resulting scene is shown in Figure 6.8, a photograph of the IRIS monitor during the flight simulation. Note how the hidden surface removal allows the foreground hills to naturally obscure the valleys behind them. Also note the effect of the lighting model and texturing described in Chapter V.

E. SIMULATOR PERFORMANCE

Data collected while running the simulator shows that the average frame update rate is approximately four frames per second. The Unix *profile* utility was used to determine which procedures accounted for the majority of the simulator's time usage. Table 6.2 shows the results for the top four routines.

TABLE 6.2 FOG-M ROUTINES USING THE MOST CPU TIME

% CPU Time	Routine Name	Purpose
16.9	polf	Iris graphics library filled polygon routine.
13.7	display_terrain	Output 3-D scene with hidden surface removal.
8.7	malloc	C language built in routine for dynamic memory allocation.
4.5	gl_findhash	Low level Iris graphics library routine, used for the hash tables associated with graphical objects (Not user accessible).

The top two entries in Table 6.2 are directly involved with outputting polygons to build the terrain image. It is therefore reasonable to believe that the frame update rate depends heavily on the number of polygons that are passed to the geometry engines.

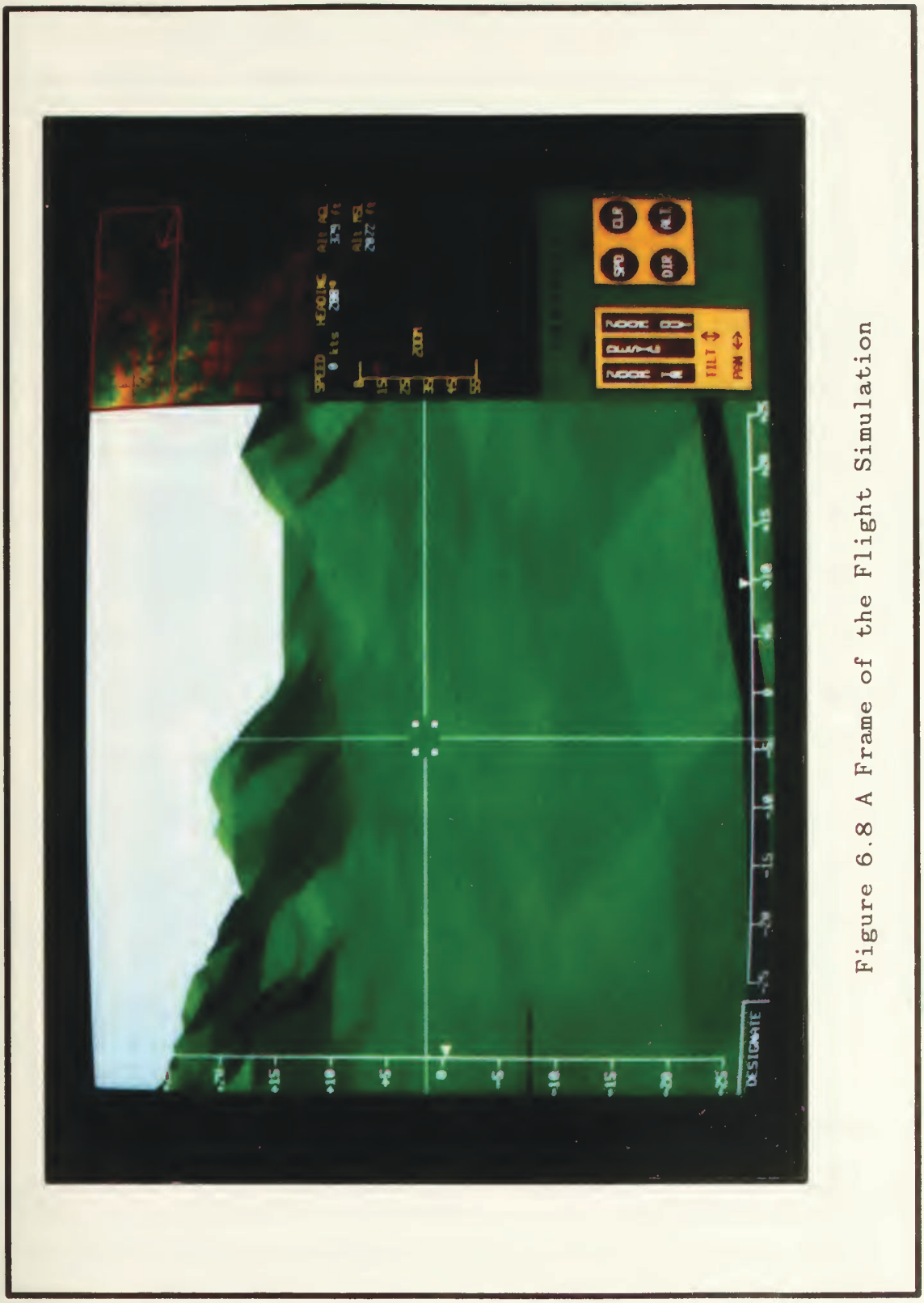


Figure 6.8 A Frame of the Flight Simulation

Figure 6.9 is a scatterplot showing the frame update speed achieved when various numbers of polygons were attempted to be drawn. The data was generated by reading the system clock before each frame update and calculating the number of polygons based on the *view−bound* that was used during that frame. The graph clearly shows the effect the view-bound has on the frame update rate. The next two entries, *malloc* and *gl_findhash*, are traceable to the making and deleting of the graphical objects that store the targets (this process is explained in Chapter VII). As an experiment, the construction and deletion of the targets' objects was removed from the simulation and the targets were simply displayed in stationary positions. The profile results from the simulator run in this configuration is shown in Table 6.3. Figure 6.10 is another scatterplot, generated in the same manner as Figure 6.9, except that the simulator was run in the stationary target configuration. Eliminating the dynamic memory management associated with the target's graphical objects increased the average frame update rate from 2.99 to 3.90 frames per second. Also, the maximum frame update rate achieved doubled from 7.5 to 15.0 frames per second. This would suggest that an area for further research is an improved algorithm for target updating that does not involve dynamically allocating memory.

The fact that the frame update rate is so heavily dependent on the number of polygons passed to the geometry engine suggests that a more sophisticated method of determining the view-bound may pay off in increased performance. For example, the present method does not take into account the field of view

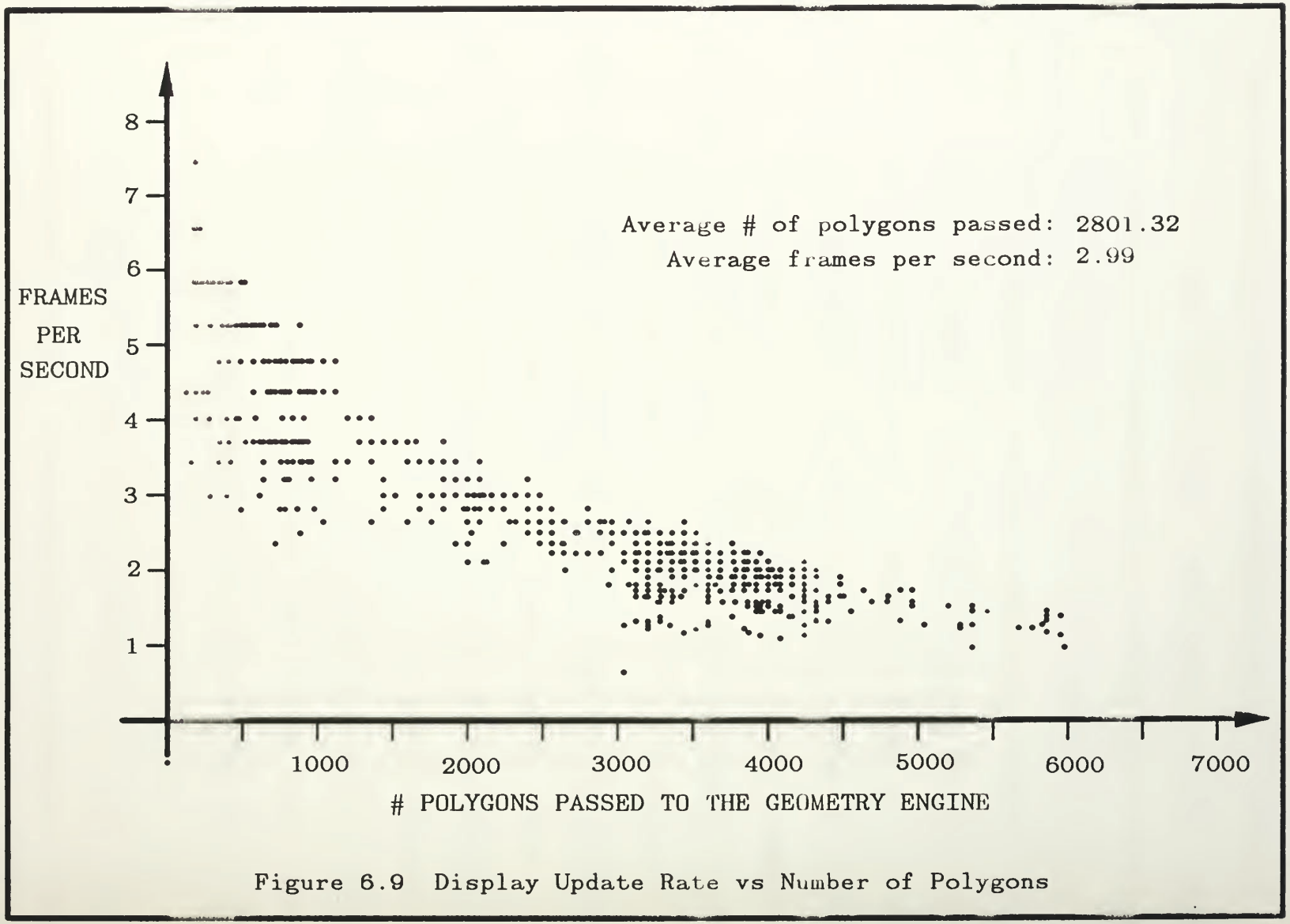


Figure 6.9 Display Update Rate vs Number of Polygons

angle. It should be possible to bound the line-of-sight intersection point with less than twenty grid squares when the field of view angle is small. However, any new algorithm developed can not be so sophisticated that it negates the performance increase by requiring intensive computations.

TABLE 6.3 FOG-M ROUTINES USING THE MOST CPU TIME (WITH STATIONARY TARGETS)

% CPU Time	Routine Name	Purpose
23.9	polf	Iris graphics library filled polygon routine.
22.3	display_terrain	Output 3-D scene with hidden surface removal.
5.5	color	Iris graphics library routine which sets the current drawing color.
4.1	line_intersect2	Finds the intersection point of two lines. This routine is used exclusively during the road building process and therefore is not used at all in the display loop.
4.0	poly	Iris graphics library unfilled polygon routine. Used during the display loop to outline the road segments.

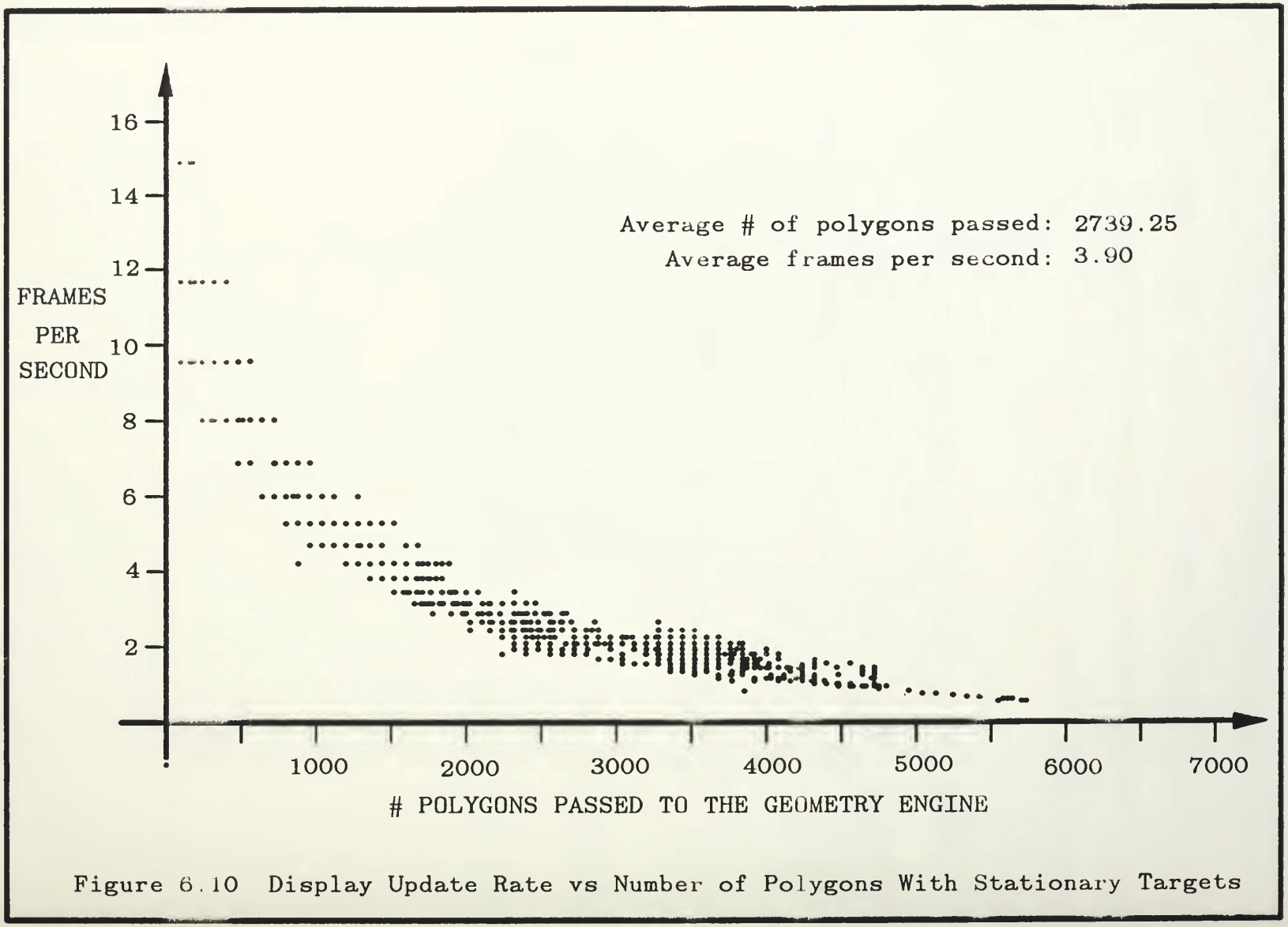


Figure 6.10 Display Update Rate vs Number of Polygons With Stationary Targets

VII. TARGET INTEGRATION

A. GENERAL

The primary targets of a FOG-M missile are tanks, helicopters, and reinforced ground installations. The simulator is designed to handle many types of targets, including various tanks and helicopters, but only a single type of tank is currently implemented. The prototype simulator provides an Ethernet networking capability to allow the input of actual target positions in real-time. This simulates the input that would be received by a production simulator during computerized mock combat field experiments. In its networking mode, the simulator receives target position and orientation data from an interactive program running on a different IRIS workstation. The target program, still in testing and not detailed in this study, provides the capability to dynamically insert and delete targets at any location, and to modify their speed and direction. In the simulator's stand-alone mode, there are ten tanks defined by default that criss-cross the ten kilometer square terrain area. These tank targets move at a constant speed of fifteen knots and reverse direction when they reach one of the edges of the ten kilometer terrain square. No automated path planning is presently performed in either mode, so the tanks blithely traverse even the

steepest terrain. The default targets minimize this problem by traveling the length of the valleys for the most part.

B. TARGET CREATION

Target creation is simplified through the use of *graphical objects*. The actual image of a tank is defined initially by the tedious specification of the three coordinates of each vertex of each of the polygons that comprise the tank (Figure 7.1). Using objects, this need only be done once, placed in an object, and then referred to by a single name within each target object. Thus each target is described by an object (the tank object) within another object (the target object). In addition to the tank object, the target object also contains the transformation commands that move the tank from the origin to its location on the terrain (a *translation*), and face it in the direction it is moving (a *rotation*).

1. The System Matrix

The rotation and translation commands work by modifying the *system matrix*. The system matrix is a global data structure that is used to transform coordinates from the three-dimensional world space into the two-dimensional screen space. Each transformation can be performed as a series of computations on individual X, Y, and Z coordinates, but the transformations can also be accomplished with a single matrix multiplication. The IRIS has a matrix multiplier built into its hardware, so matrix operations are very efficient. At least three transformations must be applied to every endpoint on the tank: a coordinate

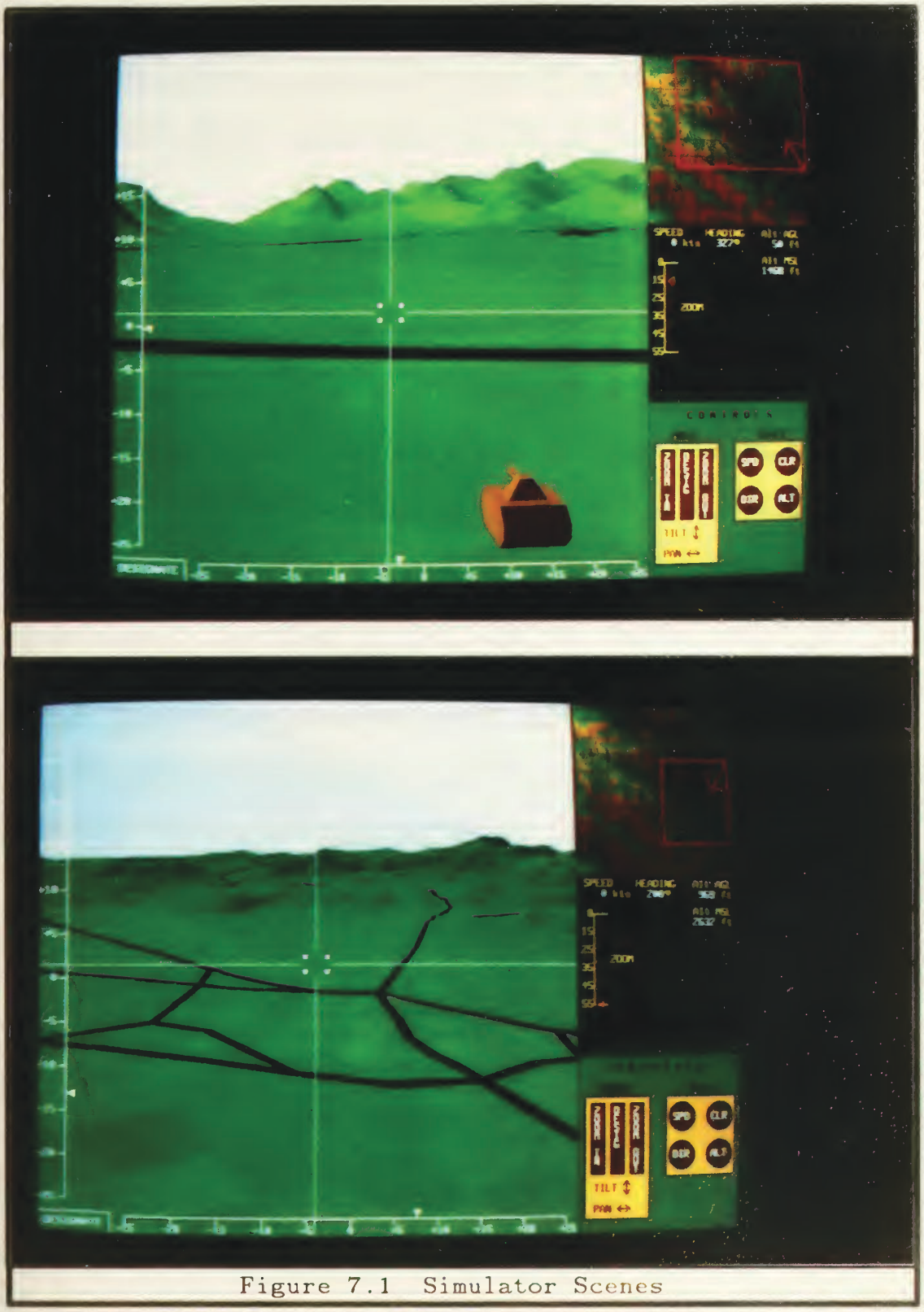


Figure 7.1 Simulator Scenes

scaling, a translation, and a rotation. Rather than do three separate matrix multiplications, the three transformation matrices can be combined, so that all of the transformations are accomplished in a single matrix multiplication. The matrices are combined by applying each of them to the system matrix. Each point is now completely transformed through a single multiplication with the system matrix. When a new transformation is needed, the system matrix must be reset by applying the inverses of the old transformations, or by copying the original contents back into the system matrix. Two commands are provided with the IRIS to support the latter method. *Pushmatrix* takes a copy of the system matrix's current contents and saves it on the *system stack*. After the transformations have been applied, and the drawing that used those transformations has been completed, the system matrix is reset by calling *popmatrix*, which retrieves the copy placed on the stack by pushmatrix and restores the contents of the system matrix to the previously saved values.

2. Target Transformations

The tank is initially defined with its center interior at the origin (coordinates (0,0,0)). While it is not important which point on or in the tank is placed at the origin, it is crucial that the tank be defined somewhere around the origin in order for the rotation command to have the desired effect. The original direction of the tank is significant only to the extent that it must be known in order to calculate the appropriate rotation to achieve a specified heading. The tank in FOG-M faces to the right (zero radians mathematically, or a compass

heading of ninety degrees) initially. During target creation, dummy (zero valued) rotation and translation commands are placed in the target object, to be updated for display by a later *editing* of the object. Since all rotation and translation commands affect the system matrix (as previously described) and are cumulative, each target object must apply its transformations, be drawn, and then remove those transformations so that latter drawing commands are not distorted. Within each target object, the contents of the system matrix are saved with a pushmatrix call, the appropriate rotation and translation commands are applied to the system matrix (in reverse order, due to the nature of matrix multiplication), the target is drawn by calling the tank object, and then popmatrix is called to reset the system matrix.

C. ANIMATION

Animation of the targets is accomplished using the objects and transformations described above. The targets must be moved slightly before being redrawn in the next frame. This requires new (X, Y, Z) coordinates, from the network or from local calculations. Then a global data structure is updated to indicate when in the display algorithm the target should be drawn, and the translation command in the target object is edited to provide the new coordinates. As each frame is displayed, targets appear in slightly shifted positions, and give the appearance of animated motion.

The calculation of new coordinates requires the maintenance of position, speed, and direction data for each target. The total distance traveled between screen updates is the product of the elapsed time (obtained from the IRIS's real-time clock) and the target's speed, scaled so the units match. In the networking version of the simulator this is done remotely; in the stand-alone version everything must be maintained locally. The target's direction of travel is stored in radians, and is measured using the standard mathematical convention as opposed to a compass heading (Figure 7.2). This allows calculation of the the appropriate east/west (ΔX) and north/south (ΔZ) movement as follows:

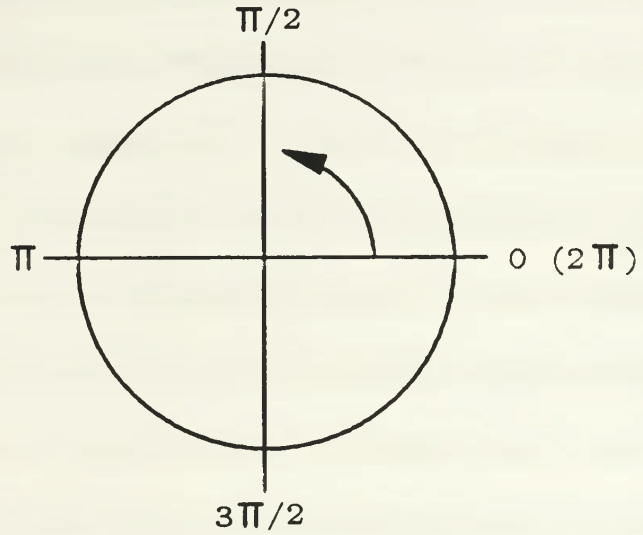
$$\Delta X = \cos(direction) * time * speed * scale\_factor \tag{7.1}$$

$$\Delta Z = -\sin(direction) * time * speed * scale\_factor \tag{7.2}$$

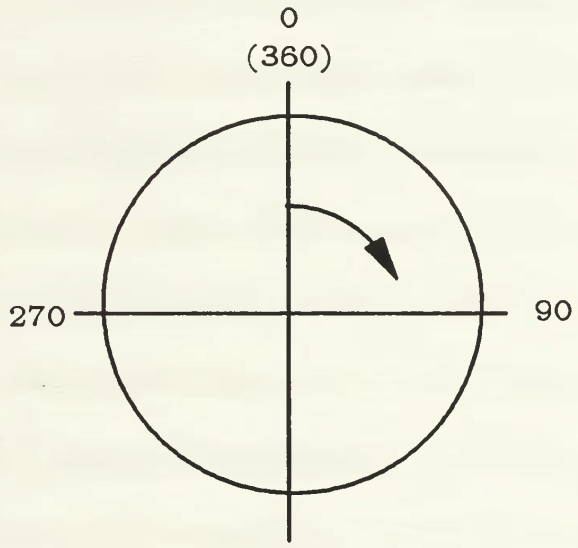
The new target (X,Z) position is the sum of the old position and the offsets (ΔX,ΔZ) from Equations 7.1 and 7.2. Since all of the current targets are tanks, their Y coordinates (altitude) should be taken from the height of the terrain underneath the tank. This is obtained from the DTED interpolation routine gnd_level, which is called with the new (X,Z) coordinates as input parameters.

D. DISPLAY

Chapter Five explained the exploitation of the structure of the data and the use of the painter's algorithm to solve the polygon ordering problem without resorting to slower or more complicated schemes like Z-buffering or Binary Space Partitioning [Ref. 13]. Targets cannot merely be drawn after the terrain because



Mathematical Convention
(Radians)



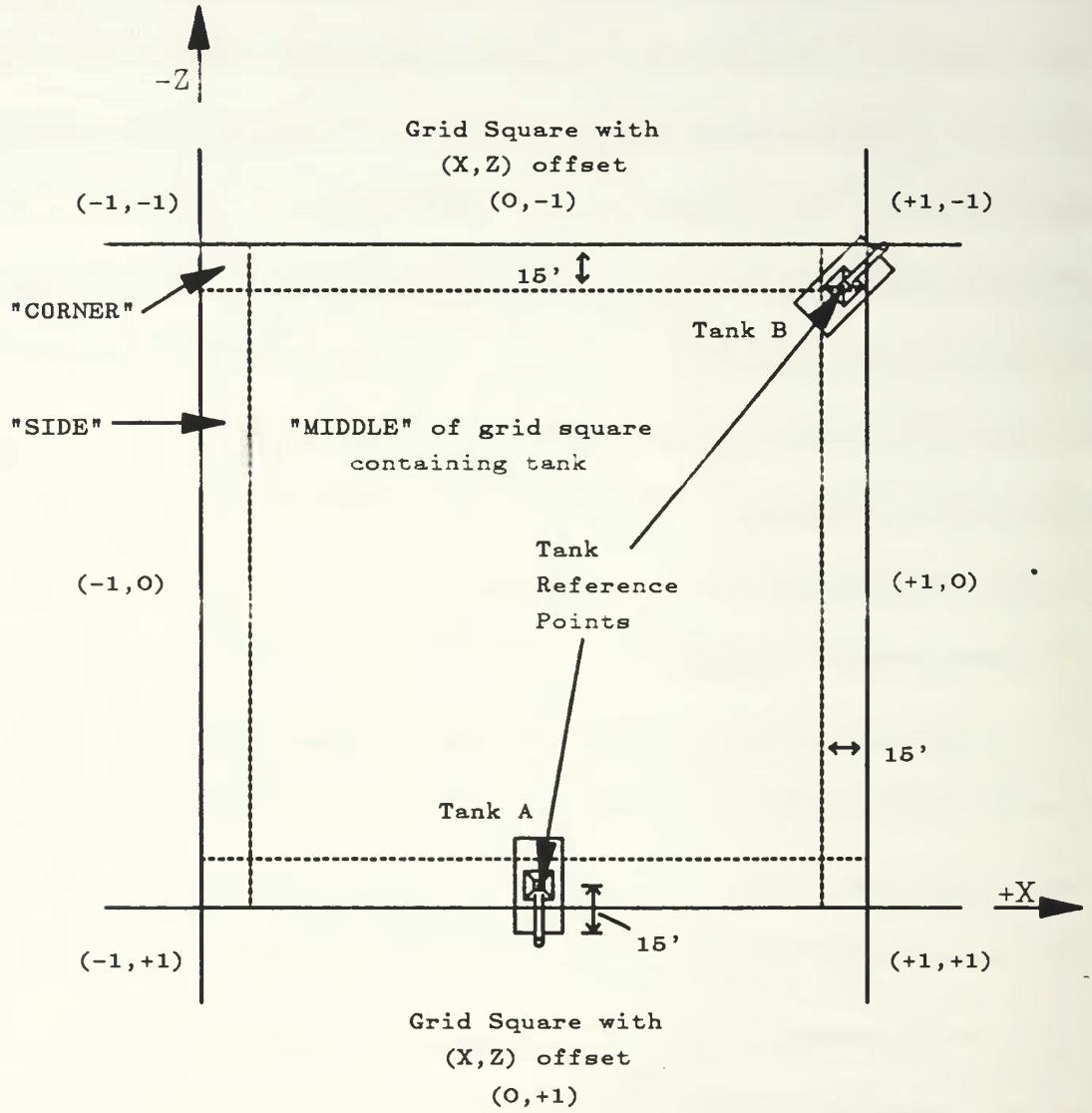
Compass Convention
(Degrees)

Figure 7.2 Direction Conventions

of the same ordering problem. Otherwise, targets appear in front of everything, and it is impossible to simulate a target moving out of sight into the distance or behind some terrain feature. The implementation of the target display algorithm is greatly facilitated by the use of objects. Objects allow the grouping of drawing commands into a subroutine-like package, which can be edited (effectively allowing parameterization) and then displayed with a single command. A two-dimensional array of object "names" (the *object−name−array*) is initialized so each element of the array represents the target object to be drawn in the one hundred meter square of terrain with the same indices. Since the C programming language recognizes the value integer zero as FALSE, and anything else as TRUE, this array does double duty as an array of booleans indicating the presence or absence of a target object in a particular one hundred meter grid square. (No target objects are given the "name" zero, which would indicate FALSE.) A list of targets is used to reset this array to all zeroes before each screen update (i.e. only those elements that contained targets need to be zeroed) so maintenance overhead of the array is minimized. The new target positions are received over the network, or are calculated, based on each target's position, speed, and direction, plus the elapsed real-time since the last update. The appropriate object-name-array indices are calculated from the new target position and the object-name-array is updated. If this is the first (or only) target in the designated one hundred meter grid square, the update is accomplished by making a new object, and setting the object-name-array element equal to the new object's integer "name."

If the array shows that some other target is already in that particular piece of terrain (i.e. the object-name-array element is non-zero), the current target is just added to the object specified by the "name" in the array. Once this has been done for each target, this array is available for the *display_terrain* module. *Display_terrain* checks the array as it draws each square of the terrain to see if any targets should be drawn. If so, it calls the indicated target object just after it has drawn the one hundred meter grid square on which the target(s) rests. Note that this causes the target(s) to be drawn at the correct *time* for the painter's algorithm. The correct *place* to draw the target still must be specified by the transformation commands within the target object.

In some cases it is necessary to draw a target more than once. Targets that straddle a one hundred meter grid square boundary must be drawn on top of both (or possibly all four) grid squares in order to avoid being partially obscured by whichever grid square is drawn last. (The target must be drawn immediately after the grid square on which it rests to ensure that the target *will* be obscured when it should be, by terrain drawn in the foreground.) Since the calculation of boundary intersection involves several trigonometric functions and an allowance for the distance between the center of the tank and its boundaries (which varies with the direction of the tank), a simplifying algorithm is used. If the tank is close enough to a boundary that the most distant part of the tank might cross the boundary (see tanks A and B in Figure 7.3), the target object is also drawn after the adjoining grid square(s).



(Not drawn to scale)

Figure 7.3 - Boundary Conditions

The one hundred meter grid square is essentially divided into three areas: the middle, its sides, and its corners. In the middle, the tank cannot overlap any other grid square. On the sides, the tank *may* overlap one adjoining grid square, and in the corners, the tank *may* overlap three adjoining grid squares. The reference point on the tank (the position the X, Y, and Z coordinates refer to) is located at the very center of the tank. The tank is thirty feet long, so the most distant parts of the tank are within a fifteen foot radius of the tank's reference point. The lines that mark the side and corner areas are thus fifteen feet inside the borders of the grid square. Once the tank's reference point is within these areas, it is potentially obscured by the later drawing of the adjacent grid square(s). It might not be obscured if it is paralleling a side, for example, but the overhead of drawing it twice (or even four times) when it does not need to be is smaller than the overhead of the calculations to determine if the position and direction of the tank have it actually crossing one or more edges.

The repeated drawing is accomplished by adding a "new" target to the array of target objects. The "new" target object is drawn at the exact same location in the three-dimensional terrain, but it is drawn after a different one hundred meter grid square, so it will have different target object array indices, and be in a separate target object, even though the two (or four) targets drawn will overwrite each other and produce a single image.

VIII. CULTURAL FEATURE INTEGRATION

The addition of cultural features add much to the realism of the displayed scene. They also provide valuable landmarks from which a person observing the scene can geographically orient himself. This chapter covers the addition of one type of cultural feature, roads, to the FOG-M simulation. Roads were chosen as the first feature to add because of the special problems associated with their implementation, the ease of extracting their locations from contour maps, and the visual impact added to all parts of the scene due to their wide-ranging locations. Three areas will be discussed: (1) the format of the external data file that contains the road's locations, (2) the process of mapping the roads onto the existing terrain, and (3) the integration of the roads into the terrain display loop.

A. EXTERNAL DATA FILE FORMAT

The data being used in the simulation was obtained by manually extracting the roads' positions from a DMA Topographic Center (DMATC) contour map of the area. Although this data is available in the DMA's Digital Feature Analysis Data (DFAD) file, the software necessary to access it was not available. The road data file's format is such that the DFAD data can be easily used when the access software is developed.

Figure 8.1 shows a segment of the file containing data for two roads along with a diagram showing their locations within the terrain. Each road entry is composed of three parts. The first part is the width of the road in feet. Next is an integer N, where N is the number of data points used to digitize the road. Third is a set N coordinate pairs, where each pair represents the location of a digitized point along the road's centerline. The first coordinate of the pair is the east-west location of the point. It is measured in feet from the western terrain boundary. The second coordinate of the pair is the north-south location of the point, measured in feet from the southern terrain boundary. All the data is stored as ASCII text, which facilitates editing of the data using any text editor. The DFAD data file also contains road width information (in meters) and stores roads as a series of digitized points. The major difference is that DFAD's points are stored as latitudes and longitudes, which need to be converted before they can be used in the simulation. [Ref. 9]

B. CONSTRUCTION OF THE ROAD POLYGONS

Knowing the width and centerline locations for the road, the next step is to construct the polygons which represent it. Although, this seems like a simple procedure, it is complicated by the fact that the road must follow the rise and fall of the terrain. Also, in order for hidden surface elimination to occur, the road must be divided at the gridsquare boundaries so that each piece can be drawn along with its corresponding gridsquare. The result is that the road must be

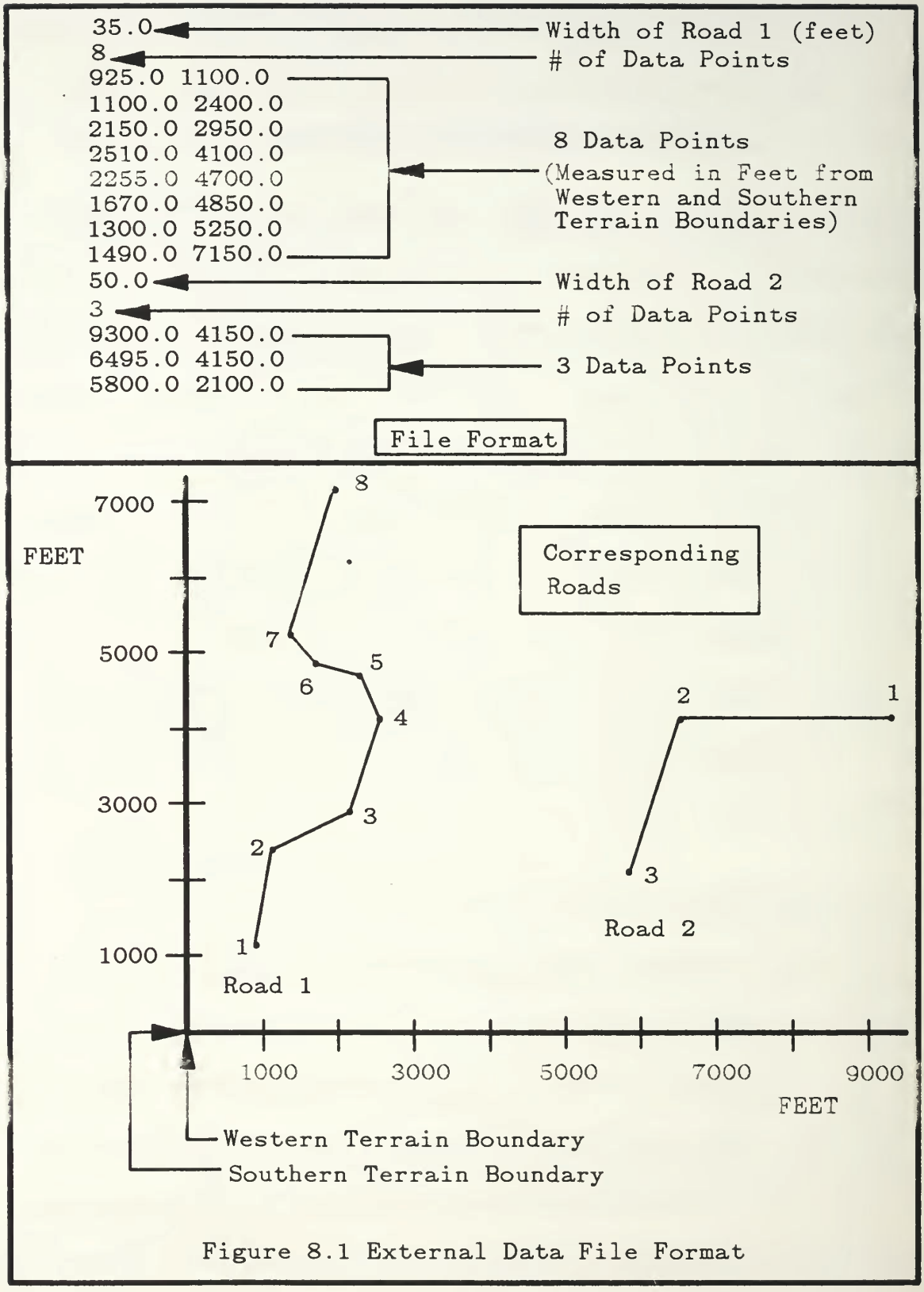


Figure 8.1 External Data File Format

broken into many planar polygons, where each polygon is a portion of the road that overlays one of the terrain triangles within a gridsquare. Figure 8.2 illustrates this division and defines some of the terms used in the description that follows. The high level pseudocode for processing the road data and constructing the planar polygons is shown in Figure 8.3. As the pseudocode shows, each road is processed a segment at a time. For each segment

- The end points of the segment's left and right side are calculated. A look-ahead to the next road segment is done, allowing the ends of adjacent segments to be calculated so that they meet cleanly.
- A bounding box, which contains all the gridsquares intersected by the segment, is constructed.

Next, for each gridsquare in the bounding box, the road segment is divided into the road-polygons at the gridtriangle boundaries. Note that all the vertices of the road-polygons fall into one of five types:

- The intersection of a segment's left side with the side of a gridtriangle.
- The intersection of a segment's right side with the side of a gridtriangle.
- A gridsquare's cornerpoint that is contained within the road segment.
- An endpoint of the left side of the road.
- An endpoint of the right side of the road.

The road polygon is constructed by finding all the above vertices which exist, and ordering them counterclockwise. The counterclockwise ordering is necessary for backface polygon removal to take place. The intersections only define the X and Z coordinates or the vertices. The Y (elevation) coordinate is found by interpolating between the terrain's elevation at the three corners of the corresponding gridtriangle.

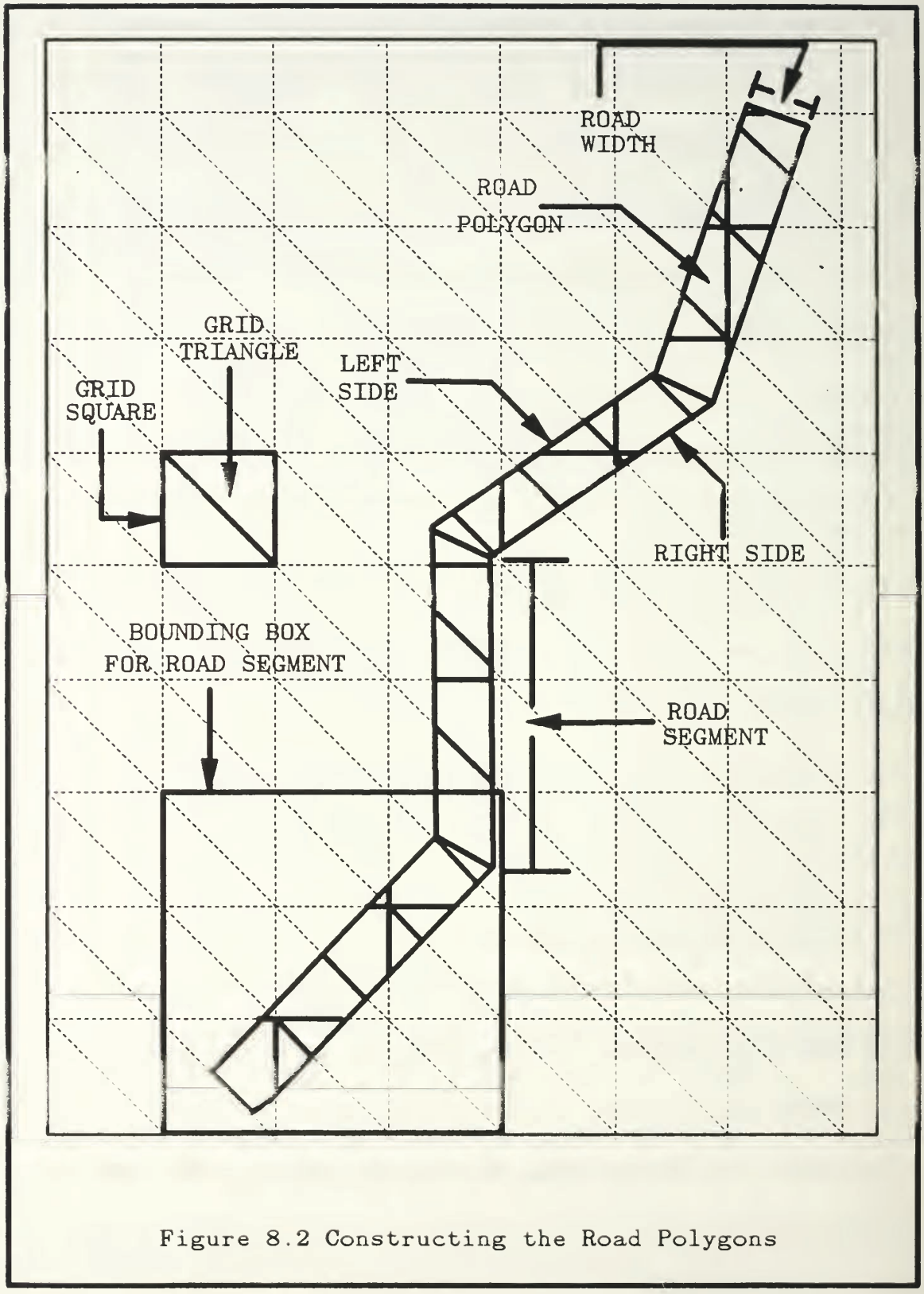


Figure 8.2 Constructing the Road Polygons

```
While more data in the road data file do
  read width_of_road
  read number_of_points
  read segment's start coordinate pair (seg_start)
  read segment's end coordinate pair (seg_end)

  for i = 3 to number_of_points + 1 do

  if i ⩽ number_of_points then
    read the next segment's end coordinate pair (next_seg_end)
  else
    next_seg_end_x ← seg_end_x
    next_seg_end_z ← seg_end_z
  endif

  calculate the start and end points for the segment's left and right side
  (left_start, left_end, right_start, right_end)

  calculate a bounding box around the road segment

  for each gridsquare within the bounding box do

    Construct the polygon which overlays the gridsquare's northern triangle
    Add the polygon to the road object associated with this gridsquare

    Construct the polygon which overlays the gridsquare's southern triangle
    Add the polygon to the road object associated with this gridsquare
  right_start ← right_end

endwhile
```

Figure 8.3 Pseudocode for Constructing Road Polygons

C. INTERNAL ROAD-POLYGON STORAGE

A global, two-dimensional array of *graphicalobjects*, named *road*, is used to store the road polygons. Each entry in the array corresponds to the pieces of road that lie within a gridsquare. An object is created when the first road-polygon is constructed for a gridsquare, with subsequent road-polygons being inserted into the already existing object. Since the roads are static in nature, the use of objects

does not present the dynamic memory allocation problems associated with their use in storing targets (see the Simulator Performance Section of Chapter VI). As each gridsquare of the terrain is drawn, a check is made to see if a *road* object exists for that square. If one does exist, the associated road-polygons are drawn immediately after the terrain. This insures that hidden surface elimination occurs for the roads as well as the terrain. A photograph of terrain which includes some sections of roads can be seen in Chapter VII, Figure 7.1).

IX. FOG-M SIMULATOR USER'S GUIDE

A. OVERVIEW

This section of the report is a user's guide to running the FOG-M simulator. The simulator was built to be largely self documenting. Instructions are clearly displayed on the screen, including diagrams which serve as a reminder of the functions of the various controls. A knowledge of the logon procedure for the IRIS workstation and the basic commands of the UNIX operating system is assumed.

B. STARTING THE SIMULATION

To start the simulation, logon to the IRIS workstation and use the UNIX **cd** command to change to the directory containing the simulation. Currently the simulation is in the directory **/work/terrain**. Therefore issue the command:

cd /work/terrain

Next, start execution of the simulation by typing the command **fogm**. A welcome screen will appear on the display as shown in Figure 9.1. Pressing **all three of the mouse buttons** simultaneously will stop the program and return to the UNIX command level. This option of pressing all three buttons to exit is available at any time during the execution of the program. Pressing the middle mouse button advances the display to the next screen of instructions. When the

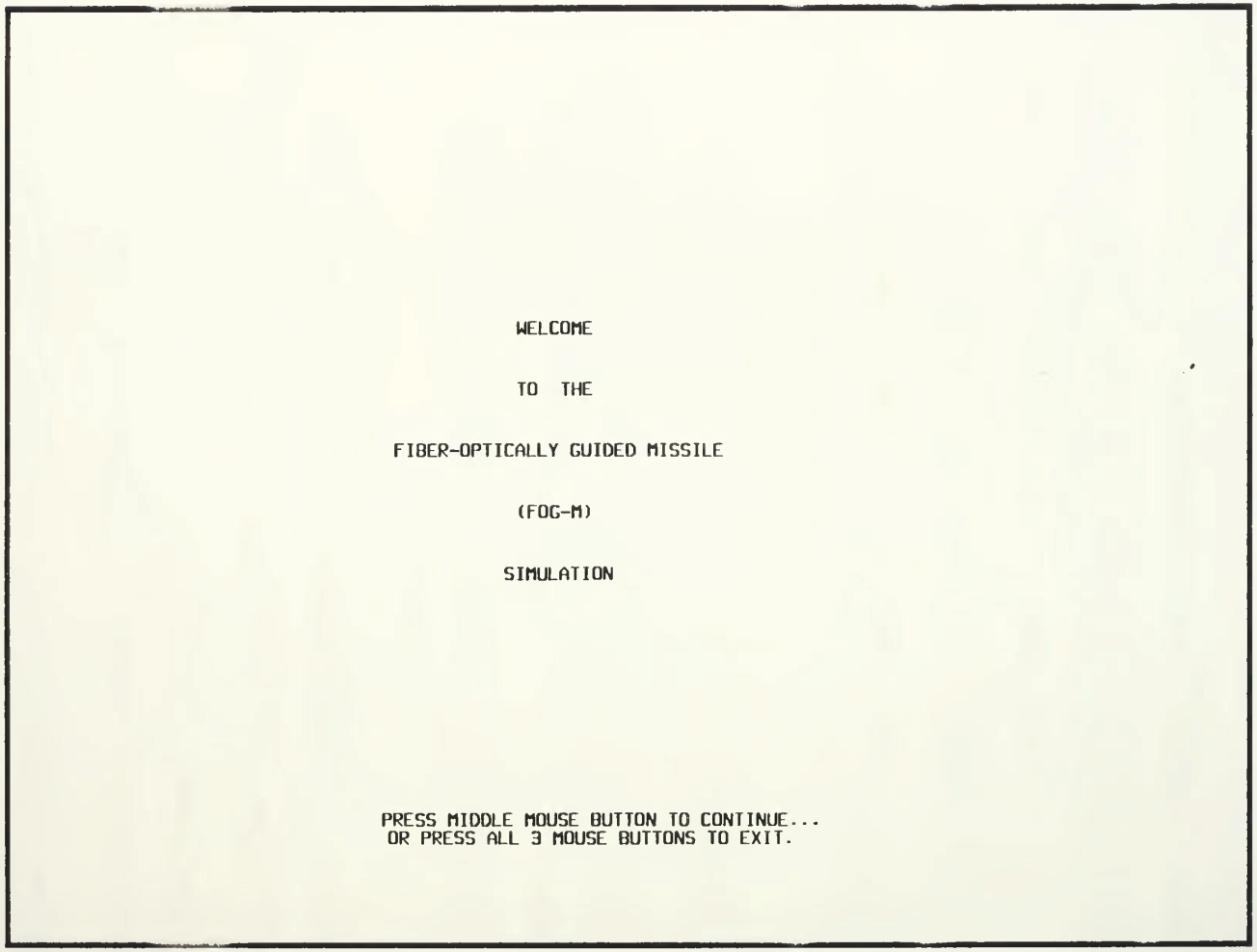


Figure 9.1 The Welcome Screen

user has advanced through the welcome screen and the two instruction screens (Figures 9.2 and 9.3) he is presented with a display showing a two-dimensional contour map. This is the prelaunch phase of the simulation.

C. PRELAUNCH CONTROLS

The purpose of the prelaunch phase is to allow the user to designate a missile launch position and a suspected target location position. In effect, the user describes an initial flight path for the missile.

1. The Prelaunch Display

The prelaunch display is divided into three sections as shown in Figure 9.4. The upper right corner of the display contains an instruction box which summarizes the functions of the mouse buttons for this phase. The lower right corner contains a prelaunch statistics box. The meanings of the various items within the statistics box are explained below. The majority of the display is occupied by a two-dimensional contour map. Each of the square grids on the contour map represents a one square kilometer area. The colors on the map can be interpreted as follows. Green areas indicate terrain that is covered with vegetation that is greater than one meter high. Brown areas indicate terrain where the vegetation is less than one meter high. Within each of the color categories, the elevation of the terrain is indicated by the intensity of the color, with the brighter colors representing the higher elevations.

Figure 9.2 The First Instruction Screen

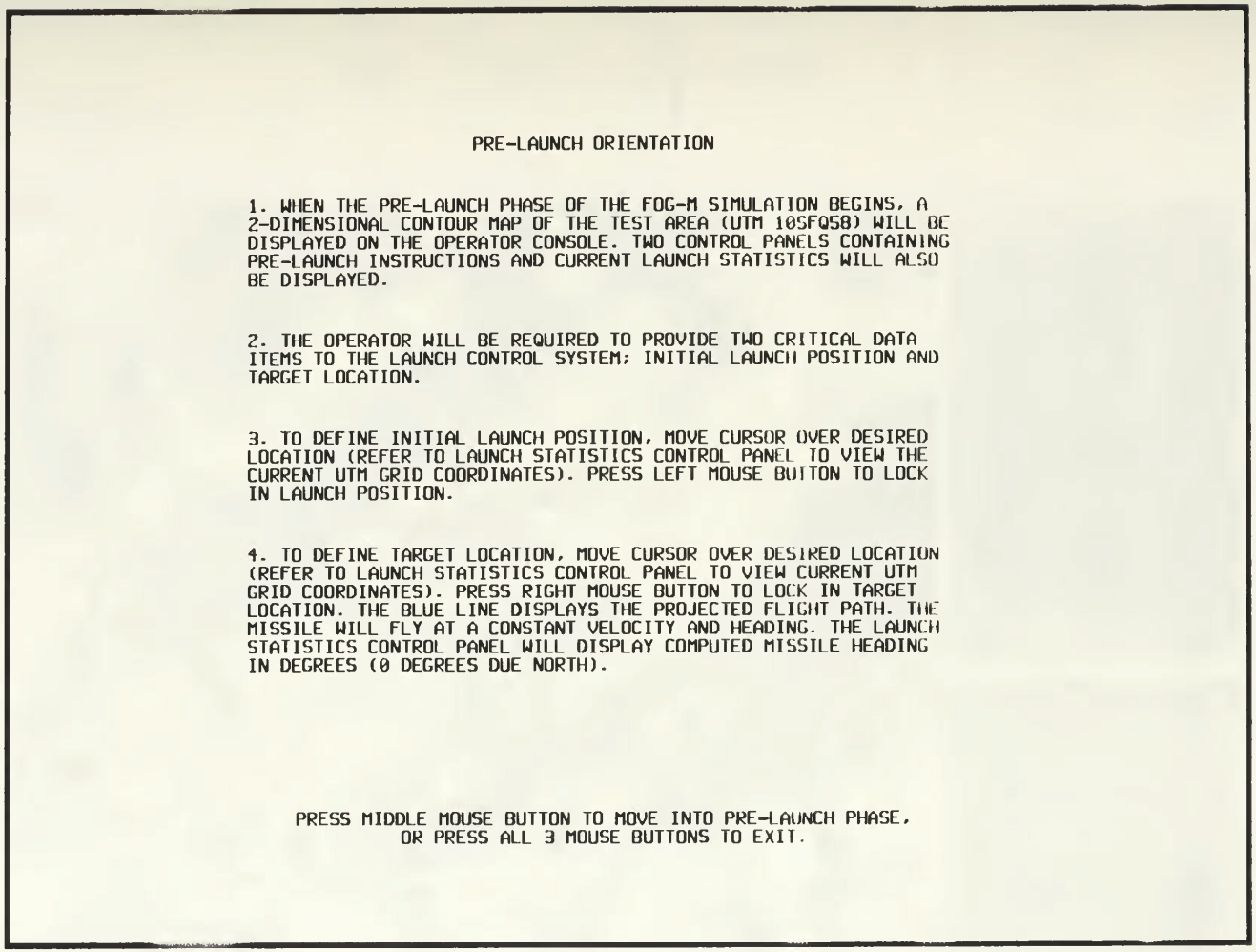


Figure 9.3 The Second Instruction Screen

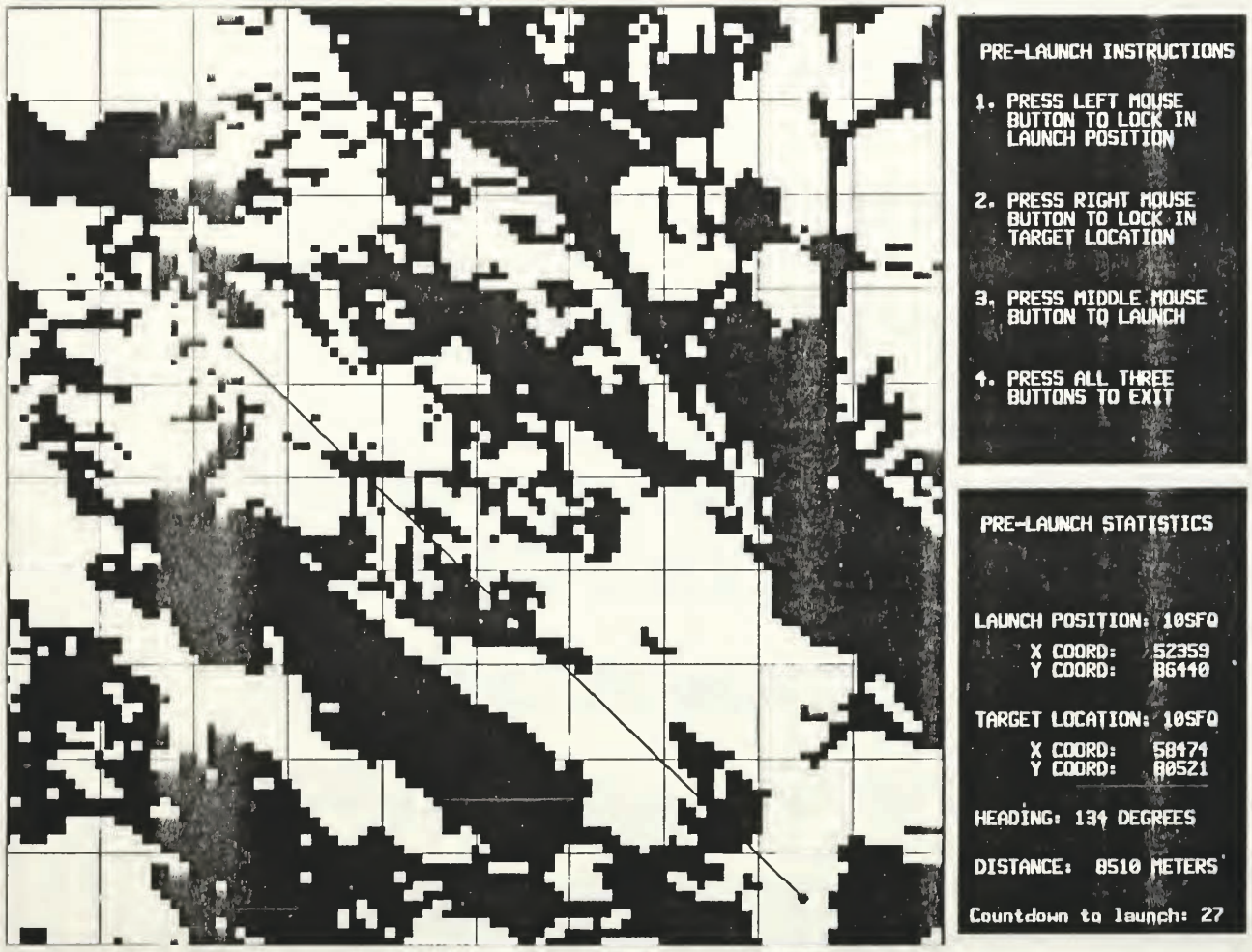


Figure 9.4 The Prelaunch Display

2. Selecting the Launch Position

The launch position must be selected first. To select the launch position, use the mouse to move the red arrow cursor to the desired location on the contour map. As the cursor is moved, the UTM coordinates of the current cursor location are shown in the Launch Position field of the statistics box. These coordinates can be used when a more accurate selection of the launch position is required than is obtainable from the contour map alone. When the cursor is in the desired position, press the **left mouse button** to lock in that position. A blue circle will appear on the contour map showing the position selected and the workstation will "beep," confirming the selection. The launch position can be changed any time before the launching of the missile by simply moving to the new desired location and pressing the **left mouse button.**

3. Selecting the Target Position

The target position can only be selected after a launch position has been set. After the launch position has been selected, moving the cursor over the contour map produces the following effects:

- The UTM coordinates of the current cursor position are shown in the Target Location field of the statistics box.
- A "rubber band" line is drawn on the contour map from the launch position to the current cursor location. This line represents the flight path the missile would take if the current cursor position was selected as the target location.
- The direction and length of the flight path represented by the above line are displayed in the statistics box in the Heading and Distance fields respectively.

Once the cursor is at the desired target location, press the **right mouse button**

to lock in the position. A red circle will appear on the contour map showing the selected location and the workstation will "beep," confirming the selection.

The missile is now ready for launch. The target location can be changed any time before launch by simply moving the cursor to the desired new location and pressing the **right mouse button.**

4. Launching the Missile

Launching can not take place until both a launch and target location have been selected. If the launch and target locations selected are acceptable, the missile is "launched" by pressing the **middle mouse button.**

If this is the initial launch of this execution of the program, a several (three to four) minute delay will follow during which calculations are done to construct the upcoming three-dimensional scenes. Again, this delay only occurs during the first launch of any execution. Subsequent launches proceed with no delay. During this delay, a countdown will appear in the bottom of the statistics box. Launch occurs when the countdown reaches zero.

D. IN-FLIGHT CONTROLS

1. The In-Flight Display

After the missile is launched, the display changes to the in-flight display shown in Figure 9.5. The left side of the display contains:

- A three-dimensional view of the terrain as seen from the missile's camera.
- A slider bar scale along the bottom edge indicating the camera pan angle.

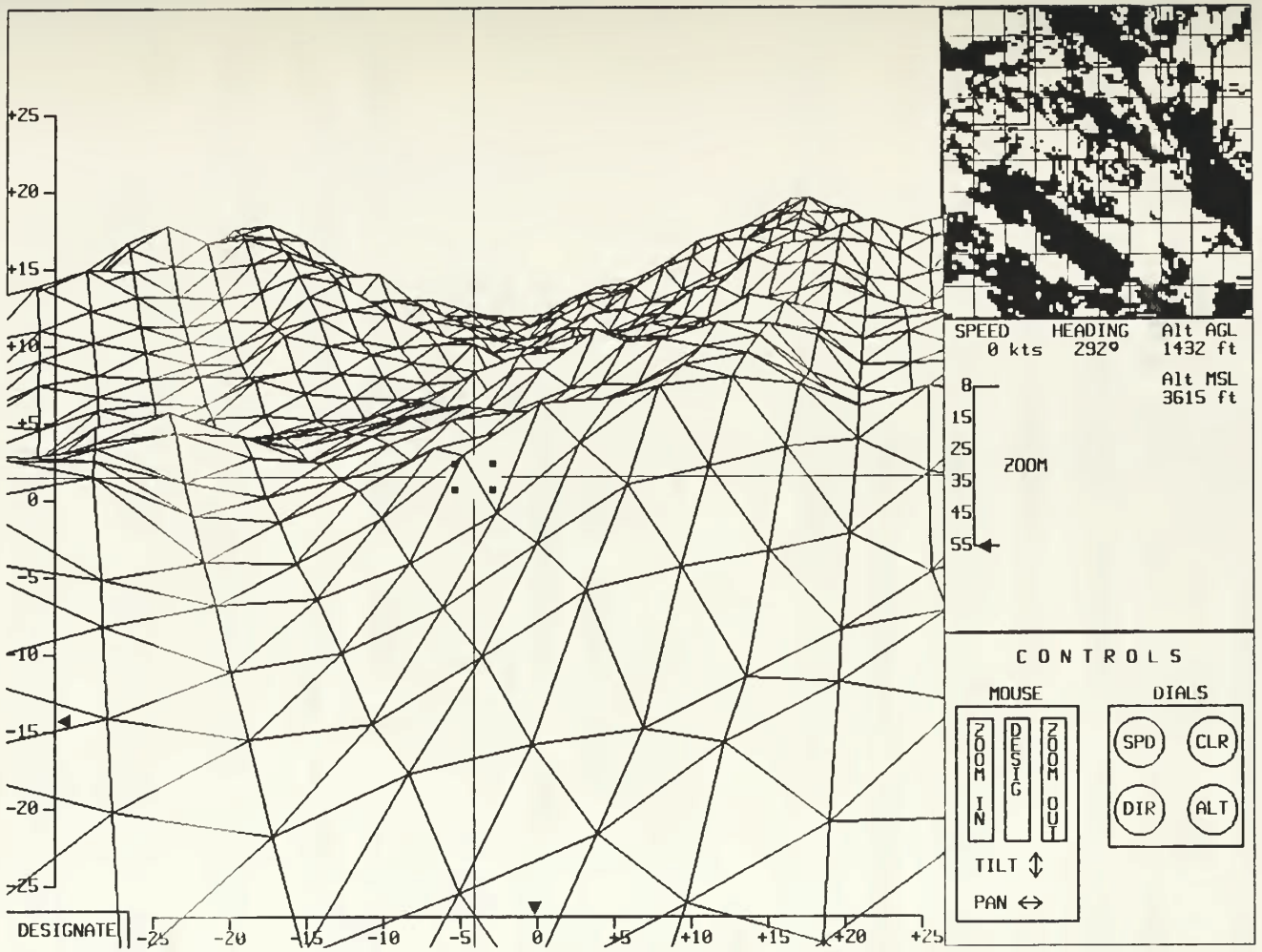


Figure 9.5 The In-Flight Display

- A slider bar scale along the left hand edge indicating the camera tilt angle.
- A box in the lower left corner containing either the word DESIGNATE or REJECT. The word DESIGNATE in this box indicates that the missile is **not** locked on to a target and is waiting for a command to designate one. The word REJECT indicates that the missile is locked on to a target and is waiting for a command to reject that target.
- Cross hairs used to sight the camera onto a target.

The upper right corner of the display contains a scaled copy of the contour map seen in the prelaunch phase. The red arrow superimposed on the contour map shows the missile's current position (the tail of the arrow) and its direction of flight. The red rectangle on the map indicates that area of the terrain that is currently being shown in the three-dimensional display.

The middle right section of the display contains four indicators which show the following:

- The speed of the missile in knots.
- The direction the missile is traveling in degrees.
- The height of the missile above ground level (AGL) in feet.
- The height of the missile above mean sea level (MSL) in feet.
- A slider bar indicating the zoom setting of the camera in degrees.

The lower right section of the display contains a summary of the functions performed by the mouse and dials. These are explained further below. The in-flight phase continues until the missile impacts a designated target or **all three mouse buttons** are pressed simultaneously (to stop the execution of the simulation).

2. Controlling the Camera

The ranges and initial values of the camera's functions are shown in Table 9.1. All of the camera's functions are controlled with the mouse.

- To pan the camera, move the mouse left or right as needed.
- To tilt the camera, move the mouse up or down as needed.
- To zoom in to a tighter field of view, press the left mouse button.
- To zoom out to a wider field of view, press the right mouse button.

3. Controlling the Missile Flight

The missile can be controlled by changing its direction, speed, and altitude. The ranges and initial values of each of the flight parameters is shown in Table 9.2. The missile flight parameters are controlled by using the dials on the IRIS's button/dial box (see Figure 9.6). Dial zero (lower left) controls the missile's direction, dial one (lower right) controls the missile's altitude, and dial two (above dial zero) controls the missile's speed. Refer to the display's control

TABLE 9.1 CAMERA CONTROL RANGES AND INITIAL VALUES

Control	Range		Initial Value
	Maximum	Minimum	
Pan	25 degrees right	25 degrees left	0 degrees
Tilt	25 degrees down	15 degrees up	15 degrees down
Zoom	55 degrees	8 degrees	55 degrees

TABLE 9.2 MISSILE CONTROL RANGES AND INITIAL VALUES

Control	Range		Initial Value
	Maximum	Minimum	
Altitude	10,000 MSL	200 AGL	200 AGL
Speed	400 kts	0 kts	200 kts
Direction	359.9 degrees	0 degrees	From prelaunch

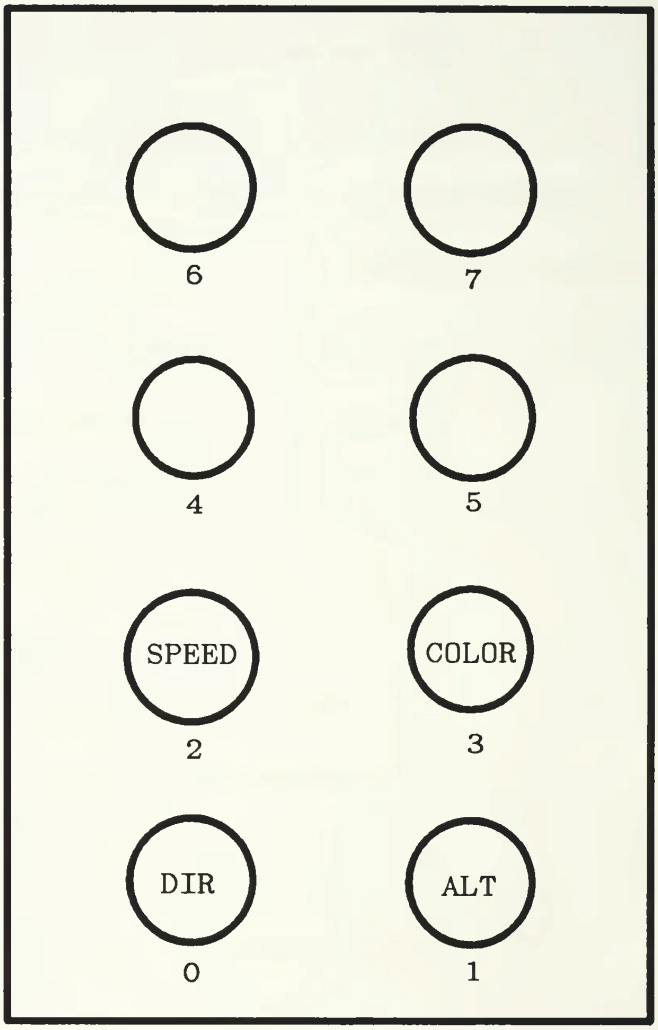


Figure 9.6 IRIS Dial Box Fuctions

summary box for a reminder of each dial's purpose and location during flight. The controls are used as follows:

- **Direction of flight -** Turning dial zero clockwise turns the missile to the right. Turning it counterclockwise turns it to the left. The missile will move freely through the 360 degree mark so that, for example, turning the missile right two degrees from a heading of 359 degrees will produce a heading of 001.
- **Altitude -** Turning dial one clockwise increases the missile's altitude up to the maximum of 10,000 feet MSL. Turning the dial counterclockwise decreases the missile's altitude. The simulator will not allow an altitude to be selected that is less than 200 feet above ground level.
- **Speed -** Turning dial two clockwise increases the missile's speed, while counterclockwise decreases the speed.

4. Designating and Rejecting Targets

The **middle mouse button** is used to designate (lock on to) and reject (release the lock on) targets. When the missile is not locked on to a target the word DESIGNATE will appear in the lower left corner of the display. To designate a target, center the target within the cross hairs and press the **middle mouse button**. In order for the missile to lock on, some portion of the target *must* be in the center of the cross hairs. If the designation is successful, the workstation will "beep" and word REJECT will appear in place of the word DESIGNATE on the display. Once a target is designated the missile will automatically adjust its heading and altitude to home in on the selected target. An explosion is displayed after impact with the target occurs. The user is then returned to the prelaunch phase of the simulation to begin another launch.

A locked on target can be rejected and missile flight control returned to the user by pressing the **middle mouse button** any time before impact with the target occurs. The workstation will respond with a "beep" and the reject/designate box will again show the word DESIGNATE. The missile is now ready to accept the designation of a new target.

X. CONCLUSIONS AND RECOMMENDATIONS

A. LIMITATIONS

There are several limitations to the flight simulator presented in this study. First, a trade-off had to be made between resolution and frame update (display) speed. Even though data was available at a resolution of twelve and one-half meters, the simulator uses one hundred meter resolution in order to achieve an acceptable frame update rate.

Second, the simulator's flight is confined to a ten kilometer square area. Any ten kilometer square area of the DTED file can be used during a run of the simulation, but the simulator must be exited before switching to a new area. This limitation is not too restrictive for the current range of the FOG-M, but may be inadequate if the range of the missile is increased as planned.

Third, road data is available in a format usable by the simulator for only one 10 kilometer square area. Since access routines were not developed for the DFAD data file, roads must be digitized by hand.

Fourth, the simulator does not model any of the missile's flight dynamics. As stated earlier, this limitation was imposed only because of development time constraints. It is felt that the dynamics can be acceptably modeled without adversely affecting the performance.

B. FUTURE RESEARCH AREAS

A follow-on to this project, which will provide more realistic targets and allow viewing of the scene as seen from inside any of them, is currently underway at the Naval Postgraduate School. The project's plans are to use the Ethernet to allow several workstations to take part in the simulation simultaneously. Each workstation will control one weapon (a target or the missile) and its monitor will display the scene as viewed from that weapon.

Work is also underway at the Naval Postgraduate School in the use of digitized photographic images on the IRIS. This work could possibly be incorporated into the FOG-M project through the use of digitized target images, digitized cultural features, or digitized textures for the terrains.

Another possible research area is the addition of various environmental effects into the simulation. These include clouds, smoke, and rain, which affect the camera's view by reducing visibility, and also dust, which aids the missile operator in acquiring moving targets.

Much work could be done in the area of the missile's flight dynamics. The goal would be to provide an acceptably accurate model without too much of a sacrifice in speed.

C. SUMMARY AND CONCLUSIONS

The project has proven the practicality and feasibility of building a low-cost flight simulator with commercial, off-the-shelf hardware. With a relatively small

investment of time and funds, a simulator with significant capabilities was developed. As the speed and power of graphics hardware increases, even more realistic displays at faster update rates will be possible.

APPENDIX A – MODULE DESCRIPTIONS

BUILD_ROAD.C

Input: None.

Output: None.

Side Effects: Modifies the global array *road*, an array of graphical objects, where each object contains the polygons representing the road in a particular gridsquare.

Description: *Build_road* reads the file road width and centerline information from the file *Road.data* and constructs polygons which represent the road. The polygons are stored in the array of graphical objects *road*. A more detailed discussion of building the roads is contained in Chapter VIII.

BUILDTERRAIN.C

Input: None.

Output: None.

Side Effects: *Buildterrain* modifies the global arrays *savetriangle* and *gridcolor*.

Description: *Buildterrain* reads terrain height information from the global array *gridpixel* and constructs the terrain as a set of planar triangles. The details of constructing the triangles and the format of the *savetriangle* and *gridcolor* arrays can be found in Chapter VI.

COLORRAMP.C

Input: The inputs to *colorramp* are two booleans, *greyscale* and *init*. If *greyscale* is TRUE, the terrain, sky, and target colortable entries are defined in shades of grey to produce a black-and-white image. If *greyscale* is FALSE, the terrain colors are green, the sky is blue, and targets are brown. *Init* is set to TRUE when this routine is initially called, so that every entry in the colortable is defined, including those for terrain, sky, targets, and writemasked lines on top of the contour maps. Should the display be switched between color and black-and-white, only the terrain, sky, and target entries need to be redefined, which is what happens when *init* is FALSE.

Output: None.

Side Effects: *Colorramp* changes the system's colortable, and thus determines the colors that appear on the display for the images drawn by other routines.

Description: *Colorramp* is called by the main program **fogm** as part of the initialization that takes place before the flying loop is entered. At that point, *greyscale* is set to its default value (usually FALSE, indicating color images) and *init* is TRUE. The *readcontrols* routine also calls *colorramp* to toggle the display image between color and black-and-white, based on the position of one of the dials. This call is made with the desired value for *greyscale* and with *init* FALSE. *Colorramp* uses the IRIS routine *mapcolor* to directly update the colortable for the contour map colors, and calls the user written routine *gammaramp* to define appropriately shaded ranges of the greens and browns (or greys) used for the terrain and targets.

COMPASS.C

Input: *Compass* takes as input a float, *direction*, which is an angle in radians.

Output: *Compass* returns a float which is the compass direction in degrees corresponding to the input *direction*.

Side Effects: None.

Description: The function *Compass* converts an radian angle measured using the standard mathematical convention, and converts it to a degree angle measured using the standard navigational convention.

DISP_TERRAIN.C

Input: *Display_terrain* takes eleven inputs: the X, Y, and Z, coordinates of the missile position VX, VY, and VZ; the X, Y, and Z coordinates of the camera's look-at position PX, PY, PZ; the field of view angle (camera zoom value), $FOVY$; and the X and Z ranges of gridsquares to be displayed, $FIRST\_X$, $FIRST\_Z$, $LAST\_X$ and $LAST\_Z$.

Output: None.

Side Effects: None.

Description: *Disp_terrain* outputs a frame of the terrain scene to the monitor using a hidden surface algorithm. The scene contains terrain, roads, and targets. Details of the hidden surface algorithm can be found in Chapter VI.

DIST_TO_LOS.C

Input: *Dist_to_los* takes seven inputs: the X, Y, and Z coordinates of the start of a line segment; the X, Y, and Z coordinates of the end point of a line segment; and three dimensional array, *pt*, which contains the coordinates of a point.

Output: *Dist_to_los* returns a float which is the perpendicular distance from the input point, *pt*, to the input line.

Side Effects: None.

Description: Function which computes the perpendicular distance from a point to a line in three-space.

DO_BOUNDARY.C

Input: *Do_boundary* takes the following inputs:

- An integer *Bound_type* which is interpreted as:
  - 0 - a diagonal boundary
  - 1 - a horizontal boundary
  - 2 - a vertical boundary
- An integer *which_triangle* that is interpreted as:
  - 0 - the lower triangle of the gridsquare.
  - 1 - the upper triangle of the gridsquare.
- The indices, *xgrid* and *zgrid*, of the gridsquare for which the road is being constructed.
- The coordinates of the start point of the boundary stored in a three dimensional array, *bound_start*.
- The coordinates of the end point of the boundary stored in a three dimensional array, *bound_end*.
- The coordinates of the start point of the left side of the road stored in a three dimensional array, *left_start*.
- The coordinates of the end point of left side of the road stored in a three dimensional array, *left_end*.
- The coordinates of the start point of the right side of the road stored in a three dimensional array, *right_start*.
- The coordinates of the end point of right side of the road stored in a three dimensional array, *right_end*.
- A boolean, *start_corner_flag*, which is TRUE if the gridsquare corner at the boundary's start is ALREADY in the road polygon array, FALSE otherwise.
- A boolean, *end_corner_flag*, which is TRUE if the gridsquare corner at the boundary's end is ALREADY in the road polygon array, FALSE otherwise.
- The partially complete road polygon array, *road_poly*.

- An integer, *vertex_cnt*, that is the number of vertices currently in the *road_poly* array.

Output: *Do_boundary* outputs the following:
- *start_corner_flag* (see Inputs for a description)
- *end_corner_flag* (see Inputs for a description)
- *road_poly*, the road polygon array with the vertices along this boundary added.
- *vertex_cnt* (see Inputs for a description)

Side Effects: None.

Description: *Do_boundary*'s purpose is to find all the intersections of the road's left and right sides with the input boundary of a gridtriangle. As an intersection is found the point is put into a temporary array. After all the intersections are found for the boundary the points in the temporary array are sorted then added to the existing *road_poly* array. The order of the sorting is such that the resulting *road_poly* array will be ordered counterclockwise. See Chapter VIII for a detailed description of building the roads.

EDIT_INDBOX.C

Input: The inputs to *edit_indbox* are the name of the indicator object, the tags within that object for each of the indicators, and current values for the following missile parameters: X, Y, and Z position coordinates, pan, tilt, and zoom angles, and designate/reject status.

Output: None.

Side Effects: Since *edit_indbox* changes the indicator object, it has the side effect of changing the display when the indicator object is next called and displayed.

Description: The indicator object is edited between each display frame so that the heads-up display and the indicator box indicators show the current values for the missile's speed, heading, altitude, camera pan angle, camera tilt angle, camera field of view (zoom), and designate/reject status. The input speed, heading, and MSL altitude (Y position coordinate) are converted to strings for display. AGL altitude is calculated as the difference between MSL altitude and the elevation of the ground directly below the missile as obtained from *gnd_level* with the X and Z position coordinates as input. The boolean *designate* determines whether "DESIGNATE" or "REJECT" is printed in the lower left corner of the terrain display. Finally, the positions of the tilt, pan, and

zoom indicators are calculated from the missile parameters. The equations in the code have been simplified to avoid excess computation; the derivations are given below.

The x screen coordinate of the zoom (field of view, or fov) indicator is fixed. The y screen coordinate varies from 200 (at 8° fov) to 70 (at 55° fov). The input missile parameter *zoom* is in tenths of degrees, and thus ranges from 80 to 550. The y coordinate is determined from Equation A.1.

$$y = 200 - \left[\left(\frac{zoom}{10} - 8 \right) * \frac{200 - 70}{55 - 8} \right] \tag{A.1}$$

$$= zoom * -0.2766 + 222.128$$

Likewise, the screen x coordinate of the tilt indicator is fixed, while the y coordinate varies from 680 (at +25° tilt) to 50 (at −25° tilt). The input missile parameter *tilt* is in radians, and is converted to degrees by multiplying it with the RTOD (**R**adians **TO** **D**egrees) constant from the header file *fogm.h*. The y coordinate of the tilt indicator is calculated as shown in Equation A.2.

$$y = 50 + \left\{ \left[(tilt * DTOR) + 25 \right] * \frac{680 - 50}{25 - -25} \right\} \tag{A.2}$$

$$= tilt * 721.92682 + 365$$

The pan slider bar is horizontal, so the y coordinate is fixed, and the x coordinate ranges from 120 (at −25° pan) to 750 (at +25° pan). Like tilt, the pan value is in radians and must be converted to degrees. The pan indicator x coordinate is given by Equation A.3.

$$x = 750 - \left\{ \left[(pan * DTOR) + 25 \right] * \frac{750 - 120}{25 - -25} \right\} \tag{A.3}$$

$$= pan * -721.92682 + 435$$

EXPLOSION.C

Input: None.

Output: None.

Side Effects: None.

Description: The *explosion* routine simulates the effect of a missile destroying a target by rapidly flashing a succession of red, black, and yellow screens. One buffer is kept black to pronounce the flash effect, and the other buffer is alternately cleared to red, yellow, red, yellow, and red. A short pause with a cleared, black screen is provided before the routine exits.

FOGM.C

Input: *Fogm* is the name given to the main program in the simulator. It has no parameters, but gets data from its header files and through the *readdata* routine. Interactive input is also received vial the *readcontrols* routine.

Output: None.

Side Effects: None.

Description: The **fogm** program consists of global variable declarations, local variable declarations, system initializations, an *active* loop, and some exit housekeeping. The initialization portion includes reading in the DMA elevation data, making network connection (if in use), setting the IRIS display configuration, defining the color table entries, building all of the graphical objects used in the displays, and computing the lighting and position of the polygons used to produce the terrain image. Within the active loop is some additional initializations and the *flying* loop. In the active loop initializations, the dial and mouse controls are reset to their initial defaults, and the display buffers are loaded with the images that remain unchanged during flight simulation (the contour map and the legend/instruction box). Control is then passed to the flying loop, which produces the flight simulation images until either a target is hit or the simulation exit command is received. If a target was hit, an explosion is displayed and the pre-launch phase of designating launch and target positions is re-entered. If all three mouse buttons have been pressed, the display is cleared and various system parameters are reset to provide a graceful exit from the simulator.

The flying loop contains the subroutine calls that produce the simulation of flight. First, the mouse and dials are checked for control input. Then the targets', missile's, and lookat reference point's positions are all updated based on the elapsed time since the previous frame and the appropriate speeds. *View_bounds* is called to determine which one kilometer grid squares are in view, and then the indicators are all updated to show the new control values, missile statistics, and view area. The main display routine then draws the appropriate sections of the terrain, plus cultural features and targets where appropriate. Finally, the updated indicator objects are drawn, and the display buffers are swapped to display the newly created image.

GAMMARAMP.C

Input: The inputs to *gammaramp* are a correction factor, a color table starting index, the number of color table entries (shades) to be defined, red, green, and blue intensities for the brightest color to be defined, and finally, red, green, and blue intensities for the darkest color to be defined.

Output: None.

Side Effects: *Gammaramp* has the side effect of defining entries in the system color table.

Description: Displayed colors do not correspond linearly to the numeric red, green, and blue intensity values that are used to produce them. If a range of colors (0 .. #colors-1) is defined in the straightforward way with a uniform increment, the intensity of the n^{th} color (I_n) is given by Equation A.4, and the bright colors will appear more widely spaced than the dark colors.

$$I_n = n * \frac{MaxI - MinI}{\#colors} + MinI \qquad \text{(A.4)}$$

Gammaramp avoids this by using a power function to increase spacing between the dark colors' intensity values and to decrease the intensity increment as the colors get brighter. The strength of the correction is determined by a value γ, which is constant for a given range, but must be experimentally determined for each range that differs in color or number of colors. FOG-M uses a γ value of 1.5. The intensity of the n^{th} color in a *gammaramp* created table is given by Equation A.5.

$$I_n = \left(\frac{n}{\#colors - 1} \right)^{\frac{1}{\gamma}} * (MaxI - MinI) + MinI \qquad \text{(A.5)}$$

GET_TGT_POS.C

Input: The input to *get_tgt_pos* is a socket number for Ethernet communication (if in use), a boolean indicating designate/reject status, the index of the currently designated target, and the "name" of the tank object.

Output: Output is the new X,Y,Z position coordinates of the currently designated target.

Side Effects: *Get_tgt_pos* updates several global data structures. It sets the number of target images, updates the target position arrays, and updates the array of target object names.

Description: The primary purpose of *get_tgt_pos* is to move the targets in the simulation. If the networking capability is in use, the target positions for the next frame are received over the network. When networking is not in use, targets are moved at a set speed of fifteen knots, and reverse course when they reach the boundaries of the ten kilometer square terrain area. As explained in Chapter VII, an array of graphical objects is defined to match one object per one hundred meter square of terrain, and this array is also used as booleans to indicate the presence or absence of targets in the one hundred meter grid square. *Get_tgt_pos* begins by removing each target from this array. New target positions are calculated or received over the network. If one of the targets has been "locked-onto," its new position is returned to be used as the current aim point for the missile. This is easily determined if networking is off because the designated target's index remains the same and the new position can be directly accessed. The index correspondence is not guaranteed when networking, so the index of the new target whose coordinates are closest to the old targeted point is used.

Targets that straddle a one hundred meter grid square boundary must be drawn on top of both (or possibly all four) grid squares in order to avoid being partially obscured by whichever square is drawn last. (The target must be drawn immediately after the grid square on which it rests to ensure that the target will be obscured when it should be by terrain drawn in the foreground.) Since the calculation of boundary intersection requires several trigonometric functions plus an allowance for the distance between the center of

the tank and its boundaries (which varies with the direction of the tank), a simplifying algorithm is used. If the tank is close enough to a boundary that the most distant part of the tank might cross the boundary, the target is also drawn after the adjoining grid square(s) (see Figure 7.3). This is done by adding a "new" target to the array of target objects. The "new" target object is drawn at the exact same location in the three-dimensional terrain, but it is drawn after a different one hundred meter grid square, so it will have different target object array indices, and be in a separate target object.

After all of the targets (originals and boundary copies) have updated positions and target object array indices, objects are added to the target object array as described in Chapter VII. This array is then used by the terrain display routine to actually draw the targets.

GND_LEVEL.C

Input: *Gnd_level* takes as inputs the X and Z coordinates of the point for which the elevation is desired.

Output: *Gnd_level* returns a float which is the elevation at point X and Z.

Side Effects: None.

Description: *Gnd_level* computes, through interpolation, the scaled elevation of any point within the terrain boundaries. A calculation is done to determine which gridtriangle contains the point. Then, using the known elevations at the vertices of the triangle, the elevation of the point is found.

IN_THIS_POLY.C

Input: *In_this_poly* takes the following inputs:
- An array of points, *polygon*, which define a polygon. (Note: only the X and Z coordinates of the points are used, the Y value is ignored).
- An integer, *num_vertex*, that is the number of vertices in *polygon*.
- A point, *pnt*, that is to be tested. (Note: only the X and Z coordinates of the point is used, the Y value is ignored).

Output: *In_this_poly* returns a boolean which is TRUE if *pnt* is inside the polygon defined by *polygon*, FALSE otherwise.

Side Effects: None.

Description: *In_this_poly* is a function which tests whether a point is inside a given polygon, where both the point and the polygon are in the *XZ* plane. The algorithm used constructs a bounding box around the polygon. If the point lies outside the bounding it obviously can not be inside the polygon. If the point lies inside the bounding box a further test is made. A line is constructed from a point outside the bounding box to the point to be tested. Each of the edges of the polygons are then tested to see if they intersect the constructed line and a count is kept of the number that do intersect. The point lies inside the polygon if and only if the constructed line intersects an odd number of the polygon's edges.

INIT_CTRLS.C

Input: *Init_ctrls* takes as inputs the initial *altitude* of the missile, in feet; the initial *heading* of the missile in degrees; and a boolean, *greyscale*, which is TRUE if greyscaled terrain is to be displayed and FALSE if color terrain is to be displayed.

Output: *Init_ctrls* has as outputs the initial *pan* angle of the camera in radians; the initial *tilt* angle of the camera in radians, and the initial zoom setting of the camera in tenths of a degree.

Side Effects: The *MOUSEX*, *MOUSEY*, *DIAL*0, *DIAL*1, *DIAL*2, and *DIAL*3 valuators are set as a result of calling this routine.

Description: *Init_ctrls*'s purpose is to initialize the mouse and dial valuators used for the operator controls. The initial altitude, heading, and greyscale valuator settings are passed in as inputs. The pan, tilt, and field of view settings are read from an "include" file and their values passed back as outputs.

INIT_IRIS.C

Input: None.

Output: None.

Side Effects: Calling this routine sets the Iris attributes and configures the Iris.

Description: *Init_iris* accomplishes the following: it puts the Iris into doublebuffer mode, sets the chunksize (the minimum memory increment used in objects), sets the monitor type to either NTSC or HZ60, and enables backface polygon removal.

INIT_TGTS.C

Input: None.

Output: None.

Side Effects: *Init_tgts* always initializes the global target object array to all zeros. If target data is not being received over the network, *init_tgts* also defines ten targets by setting initial values in the global target counter, target position array, and target direction array. An auxiliary function *init_tgt* is used to perform the actual update of the global arrays.

INTERP_ELEV.C

Input: *Interp_elev* takes three inputs, each an array of X, Y, and Z coordinates, representing a point. One array is the start point of a line, the second array is the end point of a line, and the third array is a point along the line.

Output: *Interp_elev* returns a float that is the elevation value of the point along the line.

Side Effects: None.

Description: *Interp_elev* returns a float which is the linear interpolation of the Y (elevation) coordinate of the point along the line, based on the elevations at the start and end points of the line.

LIGHT_ORIENT.C

Input: *Light_orient* takes as inputs the following:
- An array of coordinates for the polygon.
- An integer, *num_coords*, the number of coordinates in the polygon.
- The X, Y, and Z coordinates of a point that is "behind" the polygon (an interior point).
- The X, Y, and Z coordinates of a light source.
- The minimum and maximum color map indices to be used for this polygon.

Output: *Light_orient* returns the color map index of the color to use in lighting this polygon. It also reorders the polygon array (if necessary) so that the points are ordered counterclockwise.

Side Effects: None.

Description: *Light_orient* computes a lighting for a polygon based on Lambert's cosine law, which states that the intensity of the light reflected

from an object is proportional to the $cos(\Phi)$, where Φ is the angle of incidence of the light ray. (see Figure 5.2). *Light_orient* also orders the vertices of the polygon in a counterclockwise fashion so that backface polygon removal can take place (see the module description for *npoly_orient*).

LINE_INTER2.C

Input: *Line_inter2* takes the following inputs:
- An array containing the X and Z coordinates of the start point of line_one is ignored.)
- An array containing the X and Z coordinates of the end of line_one. (Note: a three element array is used, but the second, Y coordinate, element is ignored.)
- An array containing the X, Y, and Z coordinates of the start of line_two. (Note: a three element array is used, but the second, Y coordinate, element is ignored.)
- An array containing the X, Y, and Z coordinates of the end of line_two. (Note: a three element array is used, but the second, Y coordinate, element is ignored.)

Output: *Line_inter2* returns as outputs:
- An array containing the X and Z coordinates of the intersection of line_one and line_two. If the lines do not intersect these values are undefined not considered in the calculation).
- An integer which can be interpreted as follows:

  0 - the lines do not intersect.

  1 - the lines intersect, but the intersection uses an extension of at least one of the lines past its start or end points.

  2 - the lines intersect, and the intersection occurs between the input start and end points of both lines.

Side Effects: None.

Description: *Line_inter2* computes the point of intersection between two lines in the XZ plane. The type of intersection, as explained above in "Output" is also determined. Throughout the routine, three element arrays are used for compatibility with other routines. The second, Y, coordinate is not considered in any of the calculations.

MAKEINDBOX.C

Input: None.

Output: *Makeindbox* returns a graphical object "name," tags for editing the speed, direction, altitude, and designate/reject readouts, and tags for editing the zoom, pan, and tilt indicators.

Side Effects: None.

Description: *Makeindbox* generates a graphical object that contains both the indicator box in the middle of the displays on the right side of the screen and the "heads-up" display that is superimposed on the terrain image (Figure 6.8). The object consists almost entirely of straightforward line and character string drawing commands, but there are two interesting points. First, within a single object, there are two different coordinate systems: one for the indicators superimposed on the terrain, and another for the separate indicator box. This is accomplished with an *ortho*2 call for each coordinate system, and by bracketing each *ortho*2 with *pushmatrix* and *popmatrix* commands. Note that the heads-up display is truly superimposed; it is specified in two-dimensional screen coordinates as opposed to the three-dimensional terrain coordinates.

The second interesting aspect is the movement of the slider bar indicators. Drawing the indicators as polygons would require a sequence of *pushmatrix*, *translate*, and *popmatrix* calls for each indicator, with movement achieved by editing the translate call. To avoid all of this matrix movement and multiplication, the "triangle" of the indicator is actually an overlapped line that "fills" the triangle by spiraling inwards. The line is drawn relative to the indicated point, with each segment of the line specified as offsets from that initial point, rather than as absolute coordinates (Figure A.1). Movement of an indicator triangle defined in this way is achieved by editing the parameters of a *move*2 call in the object, which sets the current graphics drawing position to the indicated point on the slider bar scale. *Makeindbox* is called once by **fogm** before the flying loop is entered, and then the object is edited (to update the indicator values) and called (to display it) every frame.

MAKEINSTRBOX.C

Input: None.

Output: *Makeinstrbox* returns the name of an object to **fogm**.

Side Effects: None.

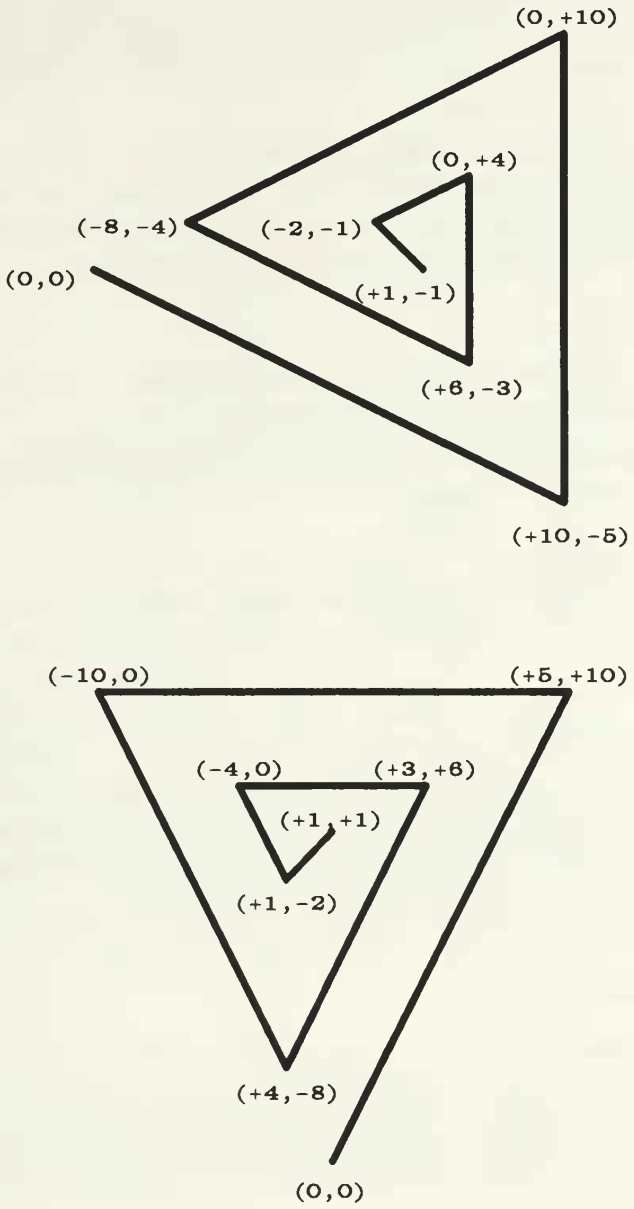


Figure A.1

Indicator Fill using Line Segments

Description: *Makeinstrbox* creates the object that produces the display in the lower right of the screen (Figure 6.8) during flight simulation. This display contains the legend for the FOG-M controls and the flight parameters they affect. *Makeinstrbox* is called once by **fogm** to create the object, and then the object is called twice per flight to put the image into each buffer. Note that writemasks are not necessary as they are with *makemap* and *makenavbox*, because nothing else writes to the instruction box portion of the screen during flight. The image thus remains undisturbed in the bitplanes despite the changes in other screen areas.

MAKEMAP.C

Input: The input to *makemap* is the globally defined array of elevation and vegetation values, *gridpixel*.

Output: The output from *makemap* is a graphical object "name," which is returned to **fogm**.

Side Effects: None.

Description: *Makemap* generates the object containing the contour map and grid that appear full screen during the pre-launch phase, and appear in the upper right corner of the screen during flight simulation (Figure 4.1). The map is produced using the methodology described in Chapter IV. **Fogm** calls the object returned by *drawcontour* twice, in order to place the map image in both buffers. The image is then protected from overwrite by a writemask. **Fogm** also passes the object name to *prelaunch*, which uses it in much the same way as **fogm**.

MAKESCREENS.C

Input: None.

Output: *Makescreens* returns an array of objects: instruction panel, statistics box, flight path between launch and target endpoints, and the three welcome screens, plus tags to update the statistics and flight path.

Side Effects: None.

Description: *Makescreens* builds all of the objects (mostly screens of text) that are used by *prelaunch*.

MAKETANK.C

Input: None.

Output: *Maketank* returns the name of an object containing a single tank, drawn around the origin.

Side Effects: None.

Description: *Maketank* builds a object that consists solely of the drawing commands to produce a single tank. The tank is thirty-two feet long, ten feet high, and ten feet wide. Its center bottom is at the origin (coordinates 0,0,0), with its left side on the plane Z = -5, its back on the plane X = -15, its bottom on the plane Y = 0, and it faces to the right along the positive x axis. For each of the twenty polygon faces that make the tank, the X, Y, and Z coordinates of each polygon vertex are stored in an array, passed to *lightorient*, and then drawn with *polf*, the filled polygon drawing command. *Lightorient* ensures the vertices are ordered counter-clockwise in the array (with respect to an interior point) for backface polygon removal, and then calculates the appropriate color for the polygon using the same lighting model that is used for the terrain (see Chapter V).

NEAREST_TGT.C

Input: *Nearest_tgt* takes as inputs the X, Y, and Z coordinates of the missile position, and the X, Y, and Z coordinates of the camera's look-at position. (The end points of the line of sight vector).

Output: *Nearest_tgt* returns as output an integer, *tgt_idx*, which is the target index of the target that is closest to the line of sight vector.

Side Effects: None.

Description: For each of the existing targets, *nearest_tgt* computes the distance between the target and the line of sight vector. It returns the index of the target that was found to be closest. In the case of two targets which are the same distance apart, the highest index value will be returned.

NPOLY_ORIENT.C

Input: *Npoly_orient* takes as input:
- An integer, *num_coords*, that is the number of vertices in the polygon.
- An array containing the coordinates of the polygon.
- The X, Y, and Z coordinates of a point that is "behind" the

polygon (an "interior" point).

Output: *Npoly_orient* returns as output an integer which is interpreted as:
1 - the vertices of the polygon are ordered clockwise.
2 - the vertices of the polygon are ordered counterclockwise.

Side Effects: None.

Description: *Npoly_orient* determines if the polygon is ordered clockwise or counterclockwise by computing two points: one along the normal vector and the other, the same distance from the polygon, but along the vector in the direction opposite the normal. Next the distance between these points and the "interior" point is computed. If the "interior" point is closer to the point along the normal vector, the polygon is ordered clockwise, otherwise the polygon is ordered counterclockwise.

PRELAUNCH.C

Input: The input to *prelaunch* is two arrays. The first contains objects, and the second contains tags for editing those objects.

Output: *Prelaunch* returns the X, Y, and Z coordinates of the missile's designated launch position, and the initial direction of flight for the missile. This direction is returned in both radians and compass degrees (Figure 7.1).

Side Effects: None.

Description: *Prelaunch* first provides three screens of introductory information. Each screen is an object defined by *makescreens*. After those, the user is presented with a full screen contour map of the ten kilometer by ten kilometer area available for overflight. Mouse-selected points define the missile's initial position and direction of flight, and are displayed on top of the map. The map is writemask protected, so it is only drawn twice (once for each buffer) even though the flight path is repeatedly drawn and erased on top of the map. The flight path is made to act like a rubber band between the launch and cursor positions by repeatedly editing of the positions in the object containing the flight path line drawing commands. Once the flight path is confirmed, the launch position and heading are returned to the **fogm** program.

RANDNUM.C

Input: *Randnum* uses the global random number seed.

Output: *Randnum* returns a floating point random number.

Side Effects: The global seed value used by *randnum* is updated during every invocation.

Description: *Randnum* is a linear congruential pseudo-random number generator. The algorithm is a modified version of the one given by Sedgewick [Ref. 13]. It uses a a special piecewise multiplication routine *mult* to preserve the low-order digits of the newly generated seed even in case of overflow. The value returned is the new seed, scaled to fall between zero and one, inclusive. The random numbers are used in **fogm** to vary the point on the tank that the missile aims for. This simulates the variance in impact point that results from the optical homing of the real missile.

RANDSEED.C

Input: *Randseed* takes a long integer as input.

Output: None.

Side Effects: *Randseed* updates the global random number seed value.

Description: The pseudo-random number generator implemented in *randnum* always returns the same string of numbers when it starts with a given seed value. *Randseed* provides the means to change that initial seed value so that different program runs will have different strings of "random" numbers.

READCONTROLS.C

Input: The inputs to *readcontrols* are the global X, Y, and Z random offset values for the aim point on the target, the current designate/reject status, and the black-and-white versus color boolean *greyscale*.

Output: All of the user-commanded control values are output from *readcontrols*: missile speed, heading and altitude, camera pan, tilt, and zoom angles, plus designate/reject status, *greyscale* status. *Readcontrols* also returns values for the booleans that control the *active* and *flying* loops.

Side Effects: When a target is first designated, *readcontrols* calls *randnum* and updates the global target aim offsets *randx*, *randy*, and *randz*.

Description: *Readcontrols* checks the status of all of the valuators that provide input to the FOG-M simulator, and performs scaling, units conversion, and immediate processing, as appropriate. It determines whether to accept or reject a "designate" command, based on the color index of the pixel at the center of the screen. (If a tank is in the crosshairs, the color index will be from the tank's color ramp, and a designate command will be accepted. Otherwise, a designate command will be ignored.)

READDATA.C

Input: None.

Output: None.

Side Effects: *Readdata* fills the global array *gridpixel*.

Description: *Readdata* opens and reads the values from the terrain elevation data file and stores the values in the *gridpixel* array. Note that the elevation data file is arranged in a format as discussed in Chapter III. The *gridpixel* array is arranged in straight rows and columns analogous to the geographic positions of the data.

ROAD_BOUNDS.C

Input: *Road_bounds* takes as input the following:
- Three arrays ($pt1$, $pt2$ and $pt3$) containing the X and Z coordinates of three points along the centerline of the road. The line segment from $pt1$ to $pt2$ defines the first segment of the road. The segment from $pt2$ to $pt3$ defines the next segment of the road.
- A float, *width*, which is the width of the road in feet.

Output: *Road_bounds* returns the following as outputs: - Four arrays (*left_pt*1, *right_pt*1, *left_pt*2, and *right_pt*2) which contain the X and Z coordinates of the first segment's left and right sides. The left side runs from *left_pt*1 to *left_pt*2 and the right side runs from *right_pt*1 to *right_pt*2.
- Four integers, *first_xgrid*, *first_zgrid*, *last_xgrid* and *last_zgrid*, which are the indices of the bounding box surrounding the first road segment (see Figure 8.2).

Side Effects: None.

Description: Given three points along the center line of the road, and the road's width, *road_bounds* computes the start and end coordinates for the first segment's left and right sides. The end coordinates are computed as the intersection of the first segment's left (or right)

side with the second segment's left (or right) side. This insures that adjoining segments will meet cleanly. The second function of *road_bounds* is to compute a bounding box around the first road segment. This box is defined as the row indices of the northern and southern most gridsquares that the road segment intersects, and the column indices of the eastern and western most gridsquares that the road segment intersects (See Chapter VIII for a more detailed discussion).

SORT_ARRAY.C

Input: *Sort_array* takes as inputs:
- An array of points, *pnts*.
- An integer that is the number of entries in the *pnts* array.
- A boolean, which is TRUE if the array should be sorted in descending order, FALSE if the array should be sorted in ascending order.
- The index number of the coordinate that is the sort key: 0 for the X coordinate, 1 for the Y coordinate, and 2 for the Z coordinate.

Output: *Sort_array* returns the array *pnts* with the points sorted according to the input parameters.

Side Effects: None.

Description: *Sort_array* performs a simple "bubble-sort" of the input points according to the input parameters.

UP_LOOK_POS.C

Input: *Up_look_pos* takes the following as inputs:
- The heading of the missile in radians.
- The pan angle of the camera in radians.
- The tilt angle of the camera in radians.
- The X, Y, and Z coordinates of the missile's position.
- The X, Y, and Z coordinates of the locked-on target (if any).
- A boolean which is TRUE if the missile is locked-on a target, FALSE otherwise.

Output: *Up_look_pos* returns as outputs the X, Y, and Z coordinates of the camera's look-at position.

Side Effects: None.

Description: *Up_look_position* computes a point along the camera's line of sight. If the missile is locked on a target, the look-at position is the locked-on target's position. Otherwise it is any point along the

camera's line of sight. See Chapter VI and Figure 6.2 for a more detailed discussion.

UP_MSL_POSIT.C

Input: *Up_msl_posit* takes as inputs:
- The heading of the missile in radians.
- The speed of the missile in knots.
- The X, Y, and Z coordinates of the missile's position.
- The X, Y, and Z coordinates of the locked-on target (if any).
- A boolean which is TRUE if the missile is locked-on a target, FALSE otherwise.

Output: *Up_msl_posit* returns as outputs:
- The new heading of the missile in radians, if it was changed to track a locked-on target.
- The new heading of the missile in degrees measured in the compass convention.
- A boolean which is TRUE if the missile is still flying (has not hit a target), and FALSE if the missile has hit the target.

Side Effects: None.

Description: *Up_msl_posit* calculates a new missile position for the next frame. The new position is either based on the commanded direction, speed, and altitude (when the missile is NOT locked onto a target), or the commanded speed and the direction to the target (if the missile is locked onto a target). For a detailed discussion of the routine, see Chapter VI.

VIEW_BOUNDS.C

Input: *View_bounds* takes as inputs the X, Y, and Z coordinates of the missile's position; the X, Y, and Z coordinates of the camera's look-at position; and the field of view (zoom) value.

Output: *View_bounds* returns as outputs the row indices of the northern and southern most gridsquares to be drawn, and the column indices of the western and eastern most gridsquares to be drawn.

Side Effects: None.

Description: The purpose of *view_bounds* is to construct a bounding box around the gridsquares which are to be drawn. The box is constructed by extending the line of sight vector down until it intersects the minimum elevation plane. The view bounds extends 20 gridsquares north, south, east, and west of this intersection point.

If the missile's position is not within the bounds, the bounds are extended to include the missile's position. For a more detailed discussion, see Chapter VI and Figure 6.5

APPENDIX B – SOURCE LISTINGS

BUILD_ROAD

```
#include "stdio.h"
#include "fogm.h"
#include "files.h"
#include "gl.h"
#include "math.h"

#define X    0
#define Y    1
#define Z    2

#define DIAGONAL     0
#define HORIZONTAL   1
#define VERTICAL     2

#define LOWER        0
#define UPPER        1

build_road()
{
      extern Object road[99][99];
      extern short gridpixel[100][100];
      FILE *fp, *fopen();
      float road_width;                    /* road width if feet */
      int num_pts;                         /* number of data points
                                           for the road seqment */
      int segnum = 0;
      char temp[100];
      int cnt, i, j;
      int vertex_cnt, num_duplicates;
      float gnd_level();
      float elev;
      float pt1[3], pt2[3], pt3[3];
      float nw_corner[3], ne_corner[3], sw_corner[3], se_corner[3];
      float right_pt1[3], right_pt2[3];
      float left_pt1[3], left_pt2[3];
      float north_bound, south_bound, east_bound, west_bound;
      float delta_x, delta_z;
      float seg_dir;
      int ne_flag, nw_flag, se_flag, sw_flag;
      int xgrid, zgrid;
      int first_xgrid, last_xgrid, first_zgrid, last_zgrid;
      float poly1[10][3];

      frontbuffer(TRUE);
      fp = fopen(ROAD_FILE,"r");
```

```
while (fscanf(fp, "%e", &road_width) != EOF) {
      fscanf(fp, "%d", &num_pts);
      fscanf(fp, "%e %e", &pt1[X], &pt1[Z]);
      fscanf(fp, "%e %e", &pt2[X], &pt2[Z]);

      delta_x = pt2[X] - pt1[X];
      delta_z = pt2[Z] - pt1[Z];
      seg_dir = atan2(delta_z, delta_x);
      left_pt1[X] = pt1[X] + (cos(seg_dir + HALFPI)*road_width/2.0);
      right_pt1[X] = pt1[X] + (cos(seg_dir - HALFPI)*road_width/2.0);
      left_pt1[Z] = pt1[Z] + (sin(seg_dir + HALFPI)*road_width/2.0);
      right_pt1[Z] = pt1[Z] + (sin(seg_dir - HALFPI) * road_width/2.0);
      for (cnt = 3; cnt <= num_pts + 1; ++cnt) {
            if (cnt <= num_pts) {
                  fscanf(fp, "%e %e", &pt3[X], &pt3[Z]);
            }
            else {
                  pt3[X] = pt2[X];
                  pt3[Z] = pt2[Z];
            }
            /* print new road segment number on title screen */
            segnum += 1;
            pushmatrix();
            ortho2(0.0, 1023.0, 0.0, 767.0);
            viewport(0,1023,0,767);
            sprintf(temp, "Building road segment: %d%", segnum);
            color(BLUE);
            rectf(780.0, 20.0, 1010.0, 30.0);
            color(CYAN);
            cmov2i(780, 20);
            charstr(temp);
            popmatrix();
            /* determine the boundaries of this road segment */
            road_bounds(pt1, pt2, pt3, road_width, left_pt1, right_pt1,
            left_pt2, right_pt2, &first_xgrid,
            &first_zgrid, &last_xgrid, &last_zgrid);
            for (xgrid = first_xgrid; xgrid <= last_xgrid; ++xgrid){
                  for (zgrid = first_zgrid; zgrid <= last_zgrid; ++zgrid){
                        ne_flag = FALSE;
                        nw_flag = FALSE;
                        sw_flag = FALSE;
                        se_flag = FALSE;
                        vertex_cnt = -1;
                        east_bound = (float)(xgrid + 1) * FT_100M;
                        west_bound = (float)(xgrid) * FT_100M;
                        north_bound = (float)(zgrid + 1) * FT_100M;
                        south_bound = (float)(zgrid) * FT_100M;

                        sw_corner[X] = west_bound;
                        sw_corner[Z] = south_bound;
                        elev = gridpixel[zgrid][xgrid] & elev_mask;
                        sw_corner[Y] = pow(elev, ALTSCALE);
```

```
se_corner[X] = east_bound;
se_corner[Z] = south_bound;
elev = gridpixel[zgrid][xgrid+1] & elev_mask;
se_corner[Y] = pow(elev,ALTSCALE);

nw_corner[X] = west_bound;
nw_corner[Z] = north_bound;
elev = gridpixel[zgrid+1][xgrid] & elev_mask;
nw_corner[Y] = pow(elev,ALTSCALE);

ne_corner[X] = east_bound;
ne_corner[Z] = north_bound;
elev = gridpixel[zgrid+1][xgrid+1] & elev_mask;
ne_corner[Y] = pow(elev, ALTSCALE);


/* determine points of intersection between the left and
            right sides of the road and the eastern grid boundary
            and add these points to the polygon vertex array */

do_boundary(VERTICAL, UPPER, xgrid, zgrid, se_corner, ne_corner,
left_pt1, left_pt2, right_pt1, right_pt2, &se_flag,
&ne_flag, poly1, &vertex_cnt);

/* determine points of intersection between the left and
            right sides of the road and the northern grid boundary
            and insert these points into the polygon vertex array */
do_boundary(HORIZONTAL, UPPER, xgrid, zgrid, ne_corner,
nw_corner, left_pt1, left_pt2, right_pt1,
right_pt2, &ne_flag, &nw_flag, poly1, &vertex_cnt);

/* determine points of intersection between the left and
            right sides of the road and the diagonal and
            insert these pointsinto the polygon vertex array */

do_boundary(DIAGONAL, UPPER, xgrid, zgrid, nw_corner, se_corner,
left_pt1, left_pt2, right_pt1, right_pt2, &nw_flag,
&se_flag, poly1, &vertex_cnt);
/* remove duplicate entries from the polygon array */
num_duplicates = 0;
for (i = 1; i <= vertex_cnt; ++i) {
      if ((poly1[i][0] == poly1[i-1][0]) &&
         (poly1[i][2] == poly1[i-1][2])) {
            for (j = i; j < vertex_cnt - num_duplicates; ++j) {
                  poly1[j][0] = poly1[j+1][0];
                  poly1[j][1] = poly1[j+1][1];
                  poly1[j][2] = poly1[j+1][2];
            }
            num_duplicates += 1;
      }
}
vertex_cnt -= num_duplicates;
```

```
if (vertex_cnt > 0) {  /* add polygon to grid_object */
      if (road[zgrid][xgrid] != 0) {
            editobj(road[zgrid][xgrid]);
      }
      else {
            road[zgrid][xgrid] = genobj();
            makeobj(road[zgrid][xgrid]);
      }
      color(ROADGREY);
      polf(vertex_cnt +1, &poly1[0][0]);
      linewidth(3);
      poly(vertex_cnt + 1, &poly1[0][0]);
      closeobj();
}
vertex_cnt = -1;
ne_flag = FALSE;
nw_flag = FALSE;
sw_flag = FALSE;
se_flag = FALSE;

/* determine points of intersection between the left and
            right sides of the road and the southern grid boundary
            and insert these points into the polgon vertex array */
do_boundary(HORIZONTAL, LOWER, xgrid, zgrid, sw_corner,
se_corner, left_pt1, left_pt2, right_pt1,
right_pt2, &sw_flag, &se_flag, poly1, &vertex_cnt);

/* determine points of intersection between the left and
            right sides of the road and the diagonal and
            add these points to the polygon vertex array */

do_boundary(DIAGONAL, LOWER, xgrid, zgrid, se_corner, nw_corner,
left_pt1, left_pt2, right_pt1, right_pt2, &se_flag,
&nw_flag, poly1, &vertex_cnt);

/* determine points of intersection between the left and
            right sides of the road and the western grid bound
            and add these points to the polygon vertex array */

do_boundary(VERTICAL, LOWER, xgrid, zgrid, nw_corner, sw_corner,
left_pt1, left_pt2, right_pt1, right_pt2, &nw_flag,
&sw_flag, poly1, &vertex_cnt);

/* remove duplicate entries from the polygon array */

num_duplicates = 0;
for (i = 1; i <= vertex_cnt; ++i) {
      if ((poly1[i][0] == poly1[i-1][0]) &&
         (poly1[i][2] == poly1[i-1][2])) {
            for (j = i; j < vertex_cnt - num_duplicates; ++j) {
                  poly1[j][0] = poly1[j+1][0];
                  poly1[j][1] = poly1[j+1][1];
```

```
                                        poly1[j][2] = poly1[j+1][2];
                                    }
                                    num_duplicates += 1;
                                }
                            }
                            vertex_cnt -= num_duplicates;
                            if (vertex_cnt > 0) {  /* add polygon to grid_object */
                                if (road[zgrid][xgrid] != 0) {
                                    editobj(road[zgrid][xgrid]);
                                }
                                else {
                                    road[zgrid][xgrid] = genobj();
                                    makeobj(road[zgrid][xgrid]);
                                }
                                color(ROADGREY);
                                polf(vertex_cnt +1, &poly1[0][0]);
                                linewidth(3);
                                poly(vertex_cnt + 1, &poly1[0][0]);
                                closeobj();
                            }
                        }
                    }
                    right_pt1[X] = right_pt2[X];
                    right_pt1[Z] = right_pt2[Z];
                    left_pt1[X] = left_pt2[X];
                    left_pt1[Z] = left_pt2[Z];
                    pt1[X] = pt2[X];
                    pt1[Z] = pt2[Z];
                    pt2[X] = pt3[X];
                    pt2[Z] = pt3[Z];
                }
            }
            fclose(fp);
            frontbuffer(FALSE);
        }
```

BUILDTERRAIN

```
/*  buildterrain.c - this function builds objects representing 1km grid squares
    in 3-D, with each grid square generating 4 objects, identical except for
    order of drawing                                                    */

#include "gl.h"       /* get the graphics defs        */
#include "device.h"   /* get the graphics device defs */
#include "fogm.h"     /* default constants            */
#include "math.h"     /* math function declarations   */

buildterrain()

{
      /* array of data points to build the terrain */
      extern short gridpixel[100][100];

      extern float savetriangle[99][99][2][3][3];

      extern long gridcolor[99][99];

      extern Object target[99][99];

      extern float ground_plane[4][3];

      extern long gnd_plane_color;

      float gnd_plane_ht;

      Coord  triangle1[3][3], triangle2[3][3];      /* polygon coordinates */

      short xgrid, zgrid;           /* indexes into the grid object array */

      short endrow, endcol;         /* miscellaneous indexes etc  */
      int row, col;

      float ax,ay,az;   /* interior point for use in the lightpoly function */

      float lx,ly,lz;   /* position of light source in lightpoly function */

      /* min and max colormap indexes for lighting the poly */
      long colormin, colormax;

      /* color index to use returned by the lightpoly function */
      long colortouse, color1, color2;

      char temp[50];    /* character string for countdown */

      float x,y;
      float gammacorr;
      long  rampamax, rampamin, rampbmax, rampbmin;
```

```
int  startrow, startcol, coordidx, vertex;


lx = 500 * FT_100M;     /* direction of light source */
ly = 100000 * FT_100M;
lz = ly;

frontbuffer(TRUE);      /* write to front buffer */

/* compute color for ground_plane polygon */
gnd_plane_ht = pow((float)MIN, ALTSCALE);
ground_plane[0][0] = -NUMXGRIDS * FEETPERGRID;
ground_plane[0][1] = gnd_plane_ht;
ground_plane[0][2] = NUMZGRIDS * FEETPERGRID;

ground_plane[1][0] = 2.0 * NUMXGRIDS * FEETPERGRID;
ground_plane[1][1] = gnd_plane_ht;
ground_plane[1][2] = NUMZGRIDS * FEETPERGRID;

ground_plane[2][0] = 2.0 * NUMXGRIDS * FEETPERGRID;
ground_plane[2][1] = gnd_plane_ht;
ground_plane[2][2] = -2.0 * NUMZGRIDS * FEETPERGRID;

ground_plane[3][0] = -NUMXGRIDS * FEETPERGRID;
ground_plane[3][1] = gnd_plane_ht;
ground_plane[3][2] = -2.0 * NUMZGRIDS * FEETPERGRID;

lightorient(ground_plane,4,0.0,0.0,0.0,lx,ly,lz,256,461, &gnd_plane_color);

/* compute coordinates and colors for triangles and store in global
    variable savetriangle for later display          */

for (col = 0; col < 99; ++col) {
      /* print new countdown number on title screen */
      pushmatrix();
      ortho2(0.0, 1023.0, 0.0, 767.0);
      viewport(0,1023,0,767);
      sprintf(temp, "Countdown to launch: %d%", 98 - col);
      color(BLUE);
      rectf(780.0, 15.0, 1010.0, 30.0);
      color(CYAN);
      cmov2i(788, 20);
      charstr(temp);
      popmatrix();

      for (row = 0; row < 99; ++row) {

            /* choose which color ramp to use so that a checker board
                  effect is acheived */
            if ((row+col)%2){
                  colormin = 256;
                  colormax = 461;
```

```
}
else {
      colormin = 462;
      colormax = 667;
}

/* build the polygon */
triangle1[0][2] = (float)row * (-41.01) * 8.0;
triangle1[0][0] = (float)col * 41.01 * 8.0;
triangle1[0][1] = pow((float)(gridpixel[row][col]&elev_mask)
   , ALTSCALE);

triangle1[1][2] = (float)row * (-41.01) * 8.0;
triangle1[1][0] = (float)(col+1) * 41.01 * 8.0;
triangle1[1][1] = pow((float)(gridpixel[row][col+1]&elev_mask)
   ,ALTSCALE);

triangle1[2][2] = (float)(row+1) * (-41.01) * 8.0;
triangle1[2][0] = (float)col * 41.01 * 8.0;
triangle1[2][1] = pow((float)(gridpixel[row+1][col]&elev_mask)
   ,ALTSCALE);

/* copy common vertex values for opposing triangle of grid */
for (vertex = 1; vertex < 3; ++vertex) {
      triangle2[vertex][0] = triangle1[vertex][0];
      triangle2[vertex][1] = triangle1[vertex][1];
      triangle2[vertex][2] = triangle1[vertex][2];
}

/* change corner coordinate to form opposing triangle of grid */
triangle2[0][2] = (float)(row+1) * (-41.01) * 8.0;
triangle2[0][0] = (float)(col+1) * 41.01 * 8.0;
triangle2[0][1] = pow((float)(gridpixel[row+1][col+1]&elev_mask)
   , ALTSCALE) ;

/* compute an interior point for triangle1  */
ax = triangle1[0][0] + 15.0;
ay = -10.0;
az = triangle1[0][2] -15.0;

/* light and orient triangle1 */
lightorient(triangle1,3,ax,ay,az,lx,ly,lz,colormin, colormax, &color1);

/* compute interior point for triangle2  */
ax = triangle2[0][0] - 15.0;
ay = -10.0;
az = triangle2[0][2] +15.0;

/* compute the light for and orient triangle2 */
lightorient(triangle2,3,ax,ay,az,lx,ly,lz,colormin,colormax, &color2);

/* compute average color for the square */
```

```
                colortouse = (color1 + color2) / 2;


                /* save this triangles color and orientation */
                for (vertex = 0; vertex < 3; ++vertex)
                    for (coordidx = 0; coordidx < 3; ++coordidx) {
                        savetriangle[row][col][0][vertex][coordidx] =
                            triangle1[vertex][coordidx];
                        savetriangle[row][col][1][vertex][coordidx] =
                            triangle2[vertex][coordidx];
                    }
                gridcolor[row][col] = colortouse;

            }
        }

        frontbuffer(FALSE);
}
```

COLORRAMP

```
/*  constructs the color ramps to be used for displaying the terrain.
    If greyscale is true, constructs greyscale ramps, else it
    constructs green ramps.      */

#include "fogm.h"                       /* fogm constants */

colorramp(greyscale,init)
int greyscale, init;

{
      int i;

      /* build two gamma corrected color ramps with slightly offset colors */
      if (greyscale) {
            gammaramp(1.5,256,205,255,255,255,50,50,50);  /* even terrain ramp */
            gammaramp(1.5,462,205,245,245,245,40,40,40);  /* odd terrain ramp */
            gammaramp(1.5,668,180,235,235,235,30,30,30);  /* tank ramp */
            mapcolor(SKYBLUE,230,230,230);              /* sky color */
            mapcolor(ROADGREY,35,35,35);
      }
      else {
            gammaramp(1.5,256,205,0,255,0,0,50,0);        /* even terrain ramp */
            gammaramp(1.5,462,205,0,245,0,0,40,0);        /* odd terrain ramp */
            gammaramp(1.5,668,180,255,165,55,75,55,0);    /* tank ramp */
            mapcolor(SKYBLUE,200,200,255);              /* sky color */
            mapcolor(ROADGREY,35,35,35);
      }

      if (init) {
            mapcolor(16,0,70,0);          /* set up colors for contour map */
            mapcolor(17,0,80,0);
            mapcolor(18,0,90,0);
            mapcolor(19,0,100,0);
            mapcolor(20,0,110,0);
            mapcolor(21,0,120,0);
            mapcolor(22,0,130,0);
            mapcolor(23,0,140,0);
            mapcolor(24,0,150,0);
            mapcolor(25,0,165,0);
            mapcolor(26,0,180,0);
            mapcolor(27,0,190,0);
            mapcolor(28,0,210,0);
            mapcolor(29,0,225,0);
            mapcolor(30,0,240,0);
            mapcolor(31,0,255,0);
            mapcolor(32,75,55,0);
            mapcolor(33,95,60,0);
            mapcolor(34,115,70,0);
            mapcolor(35,125,76,0);
```