ABSTRACT

DEPARTMENT OF MATHEMATICAL AND COMPUTER SCIENCES

WALKER, REGINALD LOUIS B.S., MORRIS BROWN COLLEGE, 1981

THE IMPLEMENTATION OF A GRAPHICS PACKAGE IN ADA

Advisor: Dr. Benjamin Martin

Thesis dated July 1986

The motivation for this thesis was the need for an inexpensive graphics package that could be used to support courses in computer graphics and computer vision in the Mathematical and Computer Sciences Department of Atlanta University. The implemented graphics package used a portion of the CORE Graphics System and the hardware used consisted of Zenith Z-100 micro-computers in the Micro-computer Laboratory of Atlanta University. This graphics system was initially implemented in the Microsoft Pascal programming language. Due to limitations inherent in Pascal, the initial graphics package did not represent the best design practices. The graphics package was converted and expanded using the Ada programming language. The Ada programming language had the ability to satisfy all of the objectives of this project which were: to create a graphics package that

was portable, expandable, represented the best software design practices, and able to support computer courses at Atlanta University. Discussed in this thesis are the basic features of the extended graphics system in Ada, the general principles, an operation guide, and problems encountered using the CORE Graphics System.

THE IMPLEMENTATION OF A GRAPHICS PACKAGE IN ADA

A THESIS

SUBMITTED TO THE FACULTY OF ATLANTA UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR

THE DEGREE OF MASTER OF SCIENCE

BY

REGINALD LOUIS WALKER

SCHOOL OF ARTS AND SCIENCES

ATLANTA, GEORGIA 30314

JULY 1986

ACKNOWLEDGMENTS

I would like to express my thanks to Dr. Benjamin Martin, Dr. C. Bennett Setzer, and Dr. Robert Bozeman for their encouragement, instruction, and examples during my pursuit of this degree. I would like to thank my family and friends for their encouragement. Special thanks is expressed to my mother for her inspiration and patience.

TABLE OF CONTENTS

INTRODUCTION

The motivation for this project was the need for an inexpensive graphics package that could be used to support courses in computer graphics and computer vision in the Mathematical and Computer Sciences Department of Atlanta University. The word "package" is used loosely here to mean a group of routines. Graphics packages that were available were too expensive or hosted by unsuitable languages, such as BASIC. Since Pascal was the major language used throughout the computer science curriculum at Atlanta University, Pascal was chosen to be the host language for the CORE Graphics System.

The CORE system creates pictures by drawing lines from one point to another, positioning an imaginary pen at a given point, and changing pen colors. Points on the medium (CRT, paper, etc.) are referenced in normalized coordinates so that the system is as device independent as possible. In normalized coordinates, the medium measures 1 unit in height and 1 unit in width. The lower left corner is the origin and has coordinates (0.0,0.0); the upper right corner has coordinate (1.0,1.0); and the middle has coordinates (0.5,0.5). The commands, documented fully in the chapter entitled THE SYSTEM COMMANDS, utilize these concepts.

Graphics become especially powerful through the ability to create complicated pictures from previously created pictures altered to fit into higher-level pictures. This implementation of the CORE Graphics System utilizes the three most commonly used geometric transformations: translation, rotation, and scaling. Individual pictures are stored in groups of basic graphics instructions called segments. Each segment can be either visible or invisible. Each segment can be translated, rotated or scaled individually to create an infinite number of variations on any given theme.

In implementing this system, a working knowledge of computer graphics was developed and the potential power that a graphics package can give the user was demonstrated.

CHAPTER 1

THE EXTENDED GRAPHICS SYSTEM IN ADA

The CORE system was derived from a proposal for a standard graphics system developed by the Graphics Standards Planning Committee of the Association for Computing Machinery. This system was designed to be device independent and contains basic graphics primitives from which more complex or special-purpose graphics routines may be built. The idea is that a program written for the CORE system can be run on any installation's CORE system. The reference used to implement the CORE system was: COMPUTER GRAPHICS:A Programming Approach. This book, which was written by Steven Harrington, contained a very detailed description of a CORE subset, including pseudo-code routines. An advantage of developing the system locally was that the actual algorithms used would be available for study in courses. The initial implementation was carried out by G. Payne of Atlanta University [Payne]. The Ada implementation is a translation and extension of this initial implementation in Pascal.

After the successful implementation of the CORE system in Pascal, two issues arose that prompted further

development of this graphics package. The first issue was due to the limitations inherent in Pascal, in that the package did not represent the best software design practices. The second issue concerned research efforts in computer graphics and computer vision that needed a dynamic and hierarchical concept of a graphics image. Before these issues arose, several faculty members in the Mathematical and Computer Sciences Department became involved with programming in Ada, especially Dr. Benjamin Martin and Dr. Bennett Setzer. Also, some of the faculty proposed that Ada become the basic language used throughout the computer science curriculum. This led to the implementation of the CORE graphics system in Ada.

The CORE Graphics System was implemented on the Zenith Z-100 micro-computers using Z-DOS (Zenith's version of Microsoft's MS-DOS Operating System). The initial implementation was performed in Microsoft Pascal. The initial subset of the CORE system supported included 2 dimensional pictures, segments, filled polygons, and transformations [PAYNE]. No windowing or clipping and mapping to viewports were supported by the initial implementation. This was the starting point for the graphics system in Ada. The initial Ada version was implemented by converting the Pascal routines to Ada. The

Ada version was implemented in JANUS Ada, a subset implementation for the Ada programming language. JANUS Ada is produced by R&R Software for the Z80 and 8088 based machines. The initial Ada implementation has been extended to include windowing and clipping, mapping to viewports, and 3 dimensional pictures.

CHAPTER 2

THE GLOBAL DATA STRUCTURES, VARIABLES, AND CONSTANTS

The CORE Graphics System uses eight basic data structures to store instructions, alter instructions and display the drawings. These data structures are the CLIPPING ARRAY, the CLIPPING POLYGON ARRAY, the DISPLAY FILE ARRAY, the 2D POLYGON ARRAY, the 3D POLYGON ARRAY, the SEGMENT ARRAY, the 2D TRANSFORMATION MATRIX ARRAY, and the 3D TRANSFORMATION MATRIX ARRAY.

The CLIPPING ARRAY stores information that sets up clipping plane test conditions for the "old" endpoint of the next line segment to be clipped. Each entry is composed of an X coordinate (XS) for a vertex, a Y coordinate (YS) for a vertex, a Z coordinate (ZS) for a vertex, and the calculated test conditions for the old endpoints of the four window clipping planes (OPTTE). The array holds six XS, YS, and ZS component entries and four OPTTE entries.

The CLIPPING POLYGON ARRAY stores the polygon side instructions. Each instruction is composed of an operation code (IT), an X coordinate (XT), a Y coordinate (YT), and a Z coodinate (ZT). This array holds 32 entries.

The DISPLAY FILE ARRAY stores all graphic instruction. Each instruction is composed of an operation code (DF_OP), an X coordinate (DF_X), and a Y coordinate (DF_Y). This array can hold a maximum of 2000 instructions.

The 2 dimensional POLYGON ARRAY stores information pertaining to polygons. Each entry is composed of an X coordinate (XARRAY) for a vertex, a Y coordinate (YARRAY) for a vertex, a maximum Y value for the side (YMAX), a minimum Y value for the side (YMIN), the X value corresponding to the maximum Y value for the side (XA), and the inverse slope of the side (DXARRAY). This array can hold a maximum of 31 entries.

The 3 dimensional POLYGON ARRAY stores information pertaining to polygons. Each entry is composed of an X coordinate (XARRAY) for a vertex, a Y coordinate (YARRAY) for a vertex, a Z coordinate (ZARRAY) for a vertex, a maximum Y value for the side (YMAX), a minimum Y value for the side (YMIN), the X value corresponding to the maximum Y value for the side (XA), and the inverse slope of the side (DXARRAY). This array can hold a maximum of 31 entries.

The SEGMENT ARRAY stores the information needed to identify a segment's collection of instructions in the DISPLAY FILE ARRAY. Each entry is composed of a pointer to the first instruction in the DISPLAY FILE ARRAY associated

with the segment (SEG_START), the number of instructions associated with the segment (SEG_SIZE), an indicator for whether the segment is to be visible when the picture is to be drawn (VISIBLE), the number of radians the segment is to be rotated (ANGLE), the X scaling factor for the segment (SCALE_X), the Y scaling factor for the segment (SCALE_Y), the X translation for the segment (TRANS_X), and the Y translation for the segment (TRANS_Y). This array can hold a maximum of 2000 entries.

The 2 dimensional TRANSFORMATION MATRIX ARRAY (H) stores the information needed to alter or transform a point. This array is a 3 X 2 matrix.

The 3 dimensional TRANSFORMATION MATRIX ARRAY (TMAT) stores information pertaining to translation and rotation about an axis to create a viewing transformation. This array is a 4 X 3 matrix.

The following pages list a brief description of each global constant and variable used in the system.

ANGLE is an array in which each entry indicates the number of radians a segment is to be rotated about the point indicated by the array entries of TRANS_X and TRANS_Y.

BACK represents the back plane position.

BAC_Z represents the back plane position in view plane coordinates.

B_FLAG indicates whether back clipping is used or not.

COUNT_IN represents the number of polygon sides remaining to be processed.

COUNT_OUT represents a counter for the number of sides on a clipped polygon.

DFSIZE represents the maximum number of entries that can reside in the DISPLAY FILE ARRAY.

DF_OP is an array in which each entry represents the operation code for a graphics instruction.

DF_PEN_X represents the X coordinate of the current pen position in normalized coordinates.

DF_PEN_Y represents the Y coordinate of the current pen position in normalized coordinates.

DF_PEN_Z represents the Z coordinate of the current pen position in normalized coordinates.

DF_X is an array in which each entry represents the X coordinate of a graphics instruction.

DF_Y is an array in which each entry represents the Y coordinate of a graphics instruction.

DX represents the change in the X coordinate.

DXARRAY is an array in which each entry represents the inverse slope of a polygon side.

DXN represents the X coordinate of the view plane normal.

DXP represents the X coordinate of the direction of projection.

DXUP represents the X coordinate of the view-up direction.

DY represents the change in the Y coordinate.

DYN represents the Y coordinate of the view plane normal.

DYP represents the Y coordinate of the direction of projection.

DYUP represents the Y coordinate of the view-up direction.

DZ represents the change in the Z coordinate.

DZN represents the Z coordinate of the view plane normal.

DZP represents the Z coordinate of the direction of projection.

DZUP represents the Z coordinate of the view-up direction.

D_VIS represents the distance between the view reference point and the view plane.

E_FLAG indicates whether the screen is to be erased or not.

FILLER represents the color of the pen when filling in a polygon.

FREE represents the index of the next free DISPLAY FILE entry.

FRONT represents the front plane position.

FRO_Z represents the front plane position in view plane coordinates.

F_FLAG indicates whether front clipping is used or not.

F_PEN_X represents the X coordinate of the pen position in screen coordinates.

F_PEN_Y represents the Y coordinate of the pen position in screen coordinates.

H is an array which represents the 2 dimensional TRANSFORMATION ARRAY.

HEIGHT represents the difference between the starting vertical pixel and the ending vertical pixel position.

H_END represents the ending vertical pixel position on the screen.

H_START represents the starting vertical pixel position on the screen.

IT is an array in which each entry represents the operation code for a graphics instruction.

LINECHR represents the color of the pen.

NUM_SEGS represents the maximum number of segments that can be described in the SEGMENT FILE ARRAY.

NOW_OPEN indicates the segment which is currently open.

ON indicates whether polygons are to be filled.

OPTTE is an array which represents the test conditions for the four window clipping planes.

PERS_FLAG indicates the type of projection (parallel or perspective).

P_FLAG indicates if a polygon is being clipped.

RNDOFF represents a constant value for some small number greater than any round-off error.

SCALE_X is an array in which each entry indicates the X scaling factor to be used when drawing a segment.

SCALE_Y is an array in which each entry indicates the Y scaling factor to be used when drawing a segment.

SCAN_DEC represents the amount of line spacing to be used in the filling of a polygon. If the value is 1.0, the filling will be single spaced. If the value is 2.0, the filling will be double spaced.

SEG_SIZE is an array in which each entry represents the number of instructions in the DISPLAY FILE ARRAY defining a segment.

SEG_START is an array in which each entry represents the location in the DISPLAY FILE ARRAY of the first instruction defining a segment.

SOLID indicates whether the polygon is to be filled or not.

SXP represents the parallel projection vector ratio for the X coordinate.

SYP represents the parallel projection vector ratio for the Y coordinate.

TMAT is an array in which each entry represents the translation or rotation applied to a 3 dimensional image about an axis.

TRANS_X is an array in which each entry indicates the X translation to be applied to a segment.

TRANS_Y is an array in which each entry indicates the Y translation to be applied to a segment.

VISIBLE is an array in which each entry indicates whether a segment is visible or not.

VXH represents the right viewport clipping parameter.

VXH_HOLD represents the right viewport boundary.

VXL represents the left viewport clipping parameter.

VXL_HOLD represents the left viewport boundary.

VXP represents the X coordinate for the direction of projection in view planes coordinates.

VYH represents the top viewport clipping parameter.

VYH_HOLD represents the top viewport boundary.

VYL represents the bottom viewport clipping parameter.

VYL_HOLD represents the bottom viewport boundary.

VYP represents the Y coordinate for the direction of projection in view planes coordinates.

VZP represents the Z coordinate for the direction of projection in view planes coordinates.

V_LAR represents a constant value for a very large number approximating infinity.

WIDTH represents the difference between the starting horizontal pixel position and the ending horizontal pixel position.

WSX represents the window to viewport scale factor for the X coordinate.

WSY represents the window to viewport scale factor for the Y coordinate.

WXH represents the right window clipping parameter.

WXH_HOLD represents the right window boundary.

WXL represents the left window clipping parameter.

WXL_HOLD represents the left window boundary.

WYH represents the top clipping parameter.

WYH_HOLD represents the top window boundary.

WYL represents the bottom clipping parameter.

WYL_HOLD represents the bottom window boundary.

W_END represents the ending horizontal pixel position on the screen.

W_START represents the starting horizontal pixel position on the screen.

XA is an array in which each entry represents the X coordinate corresponding to the maximum Y coordinate of a polygon side.

XARRAY is an array in which each entry represents the X coordinate of a vertex of a polygon.

XC represents the X coordinate for the center of projection in view plane coordinates.

XHM represents the right window clipping plane slope.

XLM represents the left window clipping plane slope.

XPCNTR represents the X coordinate of the center of projection.

XR represents the X coordinate for the reference point.

XS is an array in which each entry represents the X coordinate of the point where a line segment intersects the window boundary.

XT is an array in which each entry represents the X coordinate of a vertex of a polygon.

YARRAY is an array in which each entry represents the Y coordinate of a vertex of a polygon.

YC represents the Y coordinate for the center of projection in view plane coordinates.

YHM represents the top window clipping plane slope.

YLM represents the bottom window clipping plane slope.

YMAX is an array in which each entry represents the maximum Y coordinate of a polygon side.

YMIN is an array in which each entry represents the minimum Y coordinate of a polygon side.

YPCNTR represents the Y coordinate of the center of projection.

YR represents the Y coordinate for the reference point.

YS is an array in which each entry represents the Y coordinate of the point where a line segment intersects the window boundary.

YT is an array in which each entry represents the Y coordinate of a vertex of a polygon.

ZARRAY is an array in which each entry represents the Z coordinate of a vertex of a polygon.

ZC represents the Z coordinate for the center of projection in view plane coordinates.

ZPCNTR represents the Z coordinate of the center of projection.

ZR represents the Z coordinate for the reference point.

ZS is an array in which each entry represents the Z

coordinate of the point where a line segment intersects the window boundary.

ZT is an array in which each entry represents the Z coordinate of a vertex of a polygon.

The global data structures used in this implementation of the CORE Graphics System are all coded in the package bodies. These packages must be 'withed' and 'used' as the first line in the user's program. The global constants and variables are defined in a library package for each package body.

CHAPTER 3

THE SYSTEM COMMANDS

CREA_SEG(N)

ARGUMENTS N:INTEGER

This command defines a segment composed of the proceeding instructions. The value N is an integer used to identify the segment and must range in value from 1 to 2000. A corresponding CLOS_SEG command must be used after all instructions that make up the segment N are coded. For example, suppose the command CREA_SEG(2) was issued. This would identify the next group of instructions as being those used to create segment number 2.

CLOS_SEG

This command indicates that the preceeding instruction is the last one in the segment N. This command must be used before another segment can be opened. For example, suppose the segment created is segment number 2, CREA_SEG(2). This segment is closed so that every instruction between CREA_SEG(2) and CLOS_SEG will be used to draw segment number 2.

DLET_SEG(N)

ARGUMENTS N:INTEGER

This command indicates that a segment that has been created is now to be deleted. The value N is an integer used to identify the segment to be deleted and must range in value from 1 to 2000. For example, suppose the command DLET_SEG(2) was issued. This command would delete the segment identified as segment number 2.

E_FLAG:=A

ARGUMENTS A:BOOLEAN

This command controls the erasure of the screen. The value A must be either TRUE or FALSE. If A is TRUE, the screen will be erased. If A is FALSE, the screen will not be erased. For example, suppose the command E_FLAG:=TRUE is issued. This will result in the screen being cleared before another drawing is made.

INIT

This command initializes all variables used internally by the graphics system. It must be the first graphics command used in a program. It sets the initial pen position to (0.0,0.0), the initial pen color to blue, the initial polygon fill color to blue, the initial fill option to unfilled, the initial window and viewport coordinates to (0.0,1.0,0.0,1.0), and the polygon, front and back clipping flags to false. Also, it sets the view depth to (0.0,0.0), reference point to (0.0,0.0,0.0), view plane normal vector

to (0.0,0.0,-1.0), view distance to 0.0, view up point to (0.0,0.0,0.0), and parallel projection vector to (0.0,0.0,1.0).

LIN_ABS2(X,Y)

ARGUMENTS X:FLOAT

Y:FLOAT

This command draws a line from the current pen position (Xo,Yo) to the point (X,Y). X and Y must be real values representing the point to which the pen is to be moved in normalized coordinates ($-1.0 \leq X \leq 1.0$ and $-1.0 \leq Y \leq 1.0$). For example, suppose the current pen point is (0.1,0.1), and the command LIN_ABS2(0.2,0.3) was issued. This command would cause a line to be drawn from (0.1,0.1) to (0.2,0.3).

LIN_ABS3(X,Y,Z)

ARGUMENTS X:FLOAT

Y:FLOAT

Z:FLOAT

This command draws a line from the current pen position (Xo,Yo,Zo) to the point (X,Y,Z). X, Y and Z must be real values representing the point to which the pen is to be moved in normalized coordinates ($-1.0 \leq X \leq 1.0$, $-1.0 \leq Y \leq 1.0$ and $-1.0 \leq Z \leq 1.0$). For example, suppose the current pen point is (0.1,0.1,0.1), and the command LIN_ABS3(0.2,0.2,0.3) was issued. This command would cause a line to be drawn from

(0.1,0.1,0.1) to (0.2,0.2,0.3).

LIN_REL2(X,Y)

ARGUMENTS X:FLOAT

Y:FLOAT

This command draws a line from the current pen position (Xo,Yo) to the point X units horizontally and Y units vertically. X and Y must be real values ($-1.0 \leq X \leq 1.0$ and $-1.0 \leq Y \leq 1.0$). For example, suppose the current pen point is (0.1,0.1), and the command LIN_REL2(0.2,0.3) was issued. This command would cause a line to be drawn from (0.1,0.1) to (0.3,0.4).

LIN_REL3(X,Y,Z)

ARGUMENTS X:FLOAT

Y:FLOAT

Z:FLOAT

This command draws a line from the current pen position (Xo,Yo,Zo) to the point X units horizontally, Y units vertically, and Z units diagonally. X, Y and Z must be real values ($-1.0 \leq X \leq 1.0$, $-1.0 \leq Y \leq 1.0$, and $-1.0 \leq Z \leq 1.0$). For example, suppose the current pen point is (0.1,0.1,0.1), and the command LIN_REL3(0.2,0.2,0.3) was issued. This command would cause a line to be drawn from (0.1,0.1,0.1) to (0.3,0.3,0.4).

MOV_ABS2(X,Y)

ARGUMENTS X:FLOAT

Y:FLOAT

This command moves the pen point from the current pen position (Xo,Yo) to the point (X,Y). X and Y must be real values representing the point to which the pen is to be moved in normalized coordinates ($-1.0 \leq X \leq 1.0$ and $-1.0 \leq Y \leq 1.0$). For example, suppose the current pen point is (0.1,0.1), and the command MOV_ABS2(0.2,0.3) was issued. This command would cause the pen point to be moved from (0.1,0.1) to (0.2,0.3).

MOV_ABS3(X,Y,Z)

ARGUMENTS X:FLOAT

Y:FLOAT

Z:FLOAT

This command moves the pen point from the current pen position (Xo,Yo,Zo) to the point (X,Y,Z). X, Y, and Z must be real values representing the point to which the pen is to be moved in normalized coordinates ($-1.0 \leq X \leq 1.0$, $-1.0 \leq Y \leq 1.0$, and $-1.0 \leq Z \leq 1.0$). For example, suppose the current pen position is (0.1,0.1,0.1), and the command MOV_ABS2(0.2,0.2,0.3) was issued. This command would cause the pen point to be moved from (0.1,0.1,0.1) to (0.2,0.2,0.3).

MOV_REL2(X,Y)

ARGUMENTS X:FLOAT

Y:FLOAT

This command moves the pen point from the current pen position (Xo,Yo) to the point X units horizontally and Y units vertically. X and Y must be real values ($-1.0 \leq X \leq 1.0$ and $-1.0 \leq Y \leq 1.0$). For example, suppose the current pen point is (0.1,0.1), and the command MOV_REL2(0.2,0.3) was issued. This command would cause the pen point to be moved from (0.1,0.1) to (0.3,0.4).

MOV_REL3(X,Y,Z)

ARGUMENTS X:FLOAT

Y:FLOAT

Z:FLOAT

This command moves the pen point from the current pen position (Xo,Yo,Zo) to the point X units horizontally, Y units vertically, and Z units diagonally. X, Y and Z must be real values ($-1.0 \leq X \leq 1.0$, $-1.0 \leq y \leq 1.0$, and $-1.0 \leq Z \leq 1.0$). For example, suppose the current pen point is (0.1,0.1,0.1), and the command MOV_REL3(0.2,0.2,0.3) was issued. This command would cause the pen point to be moved from (0.1,0.1,0.1) to (0.3,0.3,0.4).

M_PIC_CU

This command displays the drawing created by the previous graphics commands.

POL_ABS2(XARRAY,YARRAY,N)

ARGUMENTS XARRAY:ARRAY(1..31) OF FLOAT

YARRAY:ARRAY(1..31) OF FLOAT

N:INTEGER

This command draws a polygon using the real coordinates contained in the arrays XARRAY and YARRAY as its vertices in the polygon. The value N is an integer representing the number of polygon sides. For example, suppose the command POL_ABS2(XARRAY,YARRAY,4) is issued. This command would result in a four(4) sided polygon being created using the contents of XARRAY and YARRAY as the vertices.

POL_ABS3(XARRAY,YARRAY,ZARRAY,N)

ARGUMENTS XARRAY:ARRAY(1..31) OF FLOAT

YARRAY:ARRAY(1..31) OF FLOAT

ZARRAY:ARRAY(1..31) OF FLOAT

N:INTEGER

This command draws a polygon using the real coordinates contained in the arrays XARRAY, YARRAY, and ZARRAY as its vertices in the polygon. The value N is an integer representing the number of polygon sides. For example, suppose the command POL_ABS3(XARRAY,YARRAY,ZARRAY,4) was issued. This command would result in a four(4) sided polygon to be created using the contents of XARRAY, YARRAY, and ZARRAY as its vertices.

POL_REL2(XARRAY,YARRAY,N)

ARGUMENTS XARRAY:ARRAY(1..31) OF FLOAT

YARRAY:ARRAY(1..31) OF FLOAT

N:INTEGER

This command draws a polygon using the real relative coordinates contained in the arrays XARRAY and YARRAY as its vertices. The value N is an integer representing the number of polygon sides. For example, suppose the command POL_REL2(XARRAY,YARRAY,4) was issued. This command would create a four(4) sided polygon using the relative coordinates contained in XARRAY and YARRAY as its vertices.

POL_REL3(XARRAY,YARRAY,ZARRAY,N)

ARGUMENTS XARRAY:ARRAY(1..31) OF FLOAT

YARRAY:ARRAY(1..31) OF FLOAT

ZARRAY:ARRAY(1..31) OF FLOAT

N:INTEGER

This command draws a polygon using the real relative coordinates contained in the arrays XARRAY, YARRAY, and ZARRAY as its vertices. The value N is an integer representing the number of polygon sides. For example, suppose the command POL_REL3(XARRAY,YARRAY,ZARRAY,4) was issued. This command would create a four(4) sided polygon using the relative coordinates contained in XARRAY, YARRAY, and ZARRAY as its vertices.

RENA_SEG(No,Nn)

ARGUMENTS No:INTEGER

Nn:INTEGER

This command indicates that a segment that has been created is now to be renamed. The value No is an integer used to identify the segment to be renamed and the value Nn is an integer used to identify the new name to be given to the segment. No and Nn must range in value from 1 to 2000. For example, suppose the command RENA_SEG(1,2) was issued. This command would rename the segment number 1 as segment number 2.

SET_BAC_CLIP(ON_OFF)

ARGUMENTS ON_OFF:BOOLEAN

This command turns the back clipping flag on or off. If ON_OFF is TRUE, the back plane will be clipped. If ON_OFF is FALSE, the back plane will not be clipped. For example, suppose the command SET_BAC_CLIP(TRUE) was issued. This command would cause the back plane to be clipped.

SET_FRO_CLIP(ON_OFF)

ARGUMENTS ON_OFF:BOOLEAN

This command turns the front clipping flag on or off. If ON_OFF is TRUE, the front plane will be clipped. If ON_OFF is FALSE, the front plane will not be clipped. For example, suppose the command SET_FRO_CLIP(TRUE) was issued.

This command would cause the front plane to be clipped.

SET_PARA(DX,DY,DZ)

ARGUMENTS DX:FLOAT

DY:FLOAT

DZ:FLOAT

This command sets the direction of the parallel projection vector. The values DX, DY, and DZ are the real values that represent the X, Y, and Z coordinates respectively. For example, suppose the current parallel projection vector is (0.1,0.1,0.1), and the command SET_PARA(0.2,0.2,0.2) was issued. This command would cause the parallel projection vector to be moved from (0.1,0.1,0.1) to (0.2,0.2,0.2).

SET_PERS(X,Y,Z)

ARGUMENTS DX:FLOAT

DY:FLOAT

DZ:FLOAT

This command indicates a perspective projection and saves the center of projection. The values X, Y, and Z are the real values that represent the X, Y, and Z coordinates respectively. For example, suppose the current center of projection is (0.1,0.1,0.1), and the command SET_PERS(0.2,0.2,0.2) was issued. This command would cause the center of projection to be moved from (0.1,0.1,0.1) to

(0.2,0.2,0.2).

SET_VIEW(XL,XH,YL,YH)

ARGUMENTS XL:FLOAT

XH:FLOAT

YL:FLOAT

YH:FLOAT

This command sets the left, right, bottom, and top viewport boundaries. The values XL, XH, YL, and YH are the real values that represent the left, right, bottom, and top viewport boundaries respectively. For example, suppose the command SET_VIEW(0.2,0.8,0.2,0.8) was issued. This command would create a viewport where the left boundary is 0.2, the right boundary is 0.8, the bottom boundary is 0.2, and the top boundary is 0.8.

SET_VIEWDEP(F_DIS,B_DIS)

ARGUMENTS F_DIS:FLOAT

B_DIS:FLOAT

This command sets the position of the front and back clipping planes. The values F_DIS and B_DIS are the real values that represent the front clipping plane distance and back clipping plane distance respectively. For example, suppose the command SET_VIEWDEP(0.2,0.2) was issued. This command would cause the position of front and back clipping plane distances to be moved from (0.1,0.1) to (0.2,0.2).

SET_VIEWDIS(D)

ARGUMENTS D:FLOAT

This command sets the distance between the view reference point and the view plane. The value D is a real value that represents the new distance. For example, suppose the current distance is (0.1) and the command SET_VIEWDIS(0.2) was issued. This command would cause the distance to be moved from (0.1) to (0.2).

SET_VIEWPLANOR(DX,DY,DZ)

ARGUMENTS DX:FLOAT

DY:FLOAT

DZ:FLOAT

This command sets the view plane normal vector coordinates. The values DX, DY, and DZ are the real values that represent the X, Y, and Z coordinates respectively. For example, suppose the current view plane normal vector coordinates are (0.1,0.1,0.1), and the command SET_VIEWPLANOR(0.2,0.2,0.2) was issued. This command would cause the view plane normal vector coordinates to be moved from (0.1,0.1,0.1) to (0.2,0.2,0.2).

SET_VIEWREFPT(X,Y,Z)

ARGUMENTS X:FLOAT

Y:FLOAT

Z:FLOAT

This command sets the view reference point coordinates. The values X, Y, and Z are the real values that represent the X, Y, and Z coordinates respectively. For example, suppose the current view reference point is (0.1,0.1,0.1), and the command SET_VIEWREFPT(0.2,0.2,0.2) was issued. This command would cause the view reference point to be moved from (0.1,0.1,0.1) to (0.2,0.2,0.2).

SET_VIEWUP(DX,DY,DZ)

ARGUMENTS DX:FLOAT
DY:FLOAT
DZ:FLOAT

This command sets the direction that will be vertical on the image. The values DX, DY, and DZ are the real values that represent the X, Y, and Z coordinates respectively. For example, suppose the current view-up direction is (0.1,0.1,0.1), and the command SET_VIEWUP(0.2,0.2,0.2) was issued. This command would cause the view-up direction to be moved from (0.1,0.1,0.1) to (0.2,0.2,0.2).

SET_VIS(N,A)

ARGUMENTS N:FLOAT
A:BOOLEAN

This command turns the visibility of a given segment on or off. The value N is an integer used to identify the segment ranging in value from 1 to 2000. The value A must

be either TRUE or FALSE. If A is TRUE, the segment will be visible. If A is FALSE, the segment will not be visible. For example, suppose the command SET_VIS(2,TRUE) was issued. This command would cause segment number 2 to be visible on the screen.

SET_WINDOW(XL,XH,YL,YH)

ARGUMENTS XL:FLOAT
XH:FLOAT
YL:FLOAT
YH:FLOAT

This command sets the left, right, bottom, and top window boundaries. The values XL, XH, YL, and YH are the real values that represent the left, right, bottom, and top window boundaries respectively. For example, suppose the command SET_WINDOW(0.2,0.8,0.2,0.8) was issued. This command would create a window where the left boundary is 0.2, the right boundary is 0.8, the bottom boundary is 0.2, and the top boundary is 0.8.

S_FILL(A)

ARGUMENTS A:BOOLEAN

This command turns the fill option of the polygon drawing routine on or off. The value A must be either TRUE or FALSE. If A is TRUE, the polygon will be filled. If FALSE, the polygon will not be filled. For example, suppose

the command S_FILL(TRUE) was issued. This command would cause segment number 2 to be filled.

S_FILSTY(N)

ARGUMENTS N:INTEGER

This command sets the color to be used to fill the proceeding polygons. The value N is an integer ranging from 1 to 3. If N is 1, then the polygons will be filled in blue. If N is 2, then the polygons will be filled in red. If N is 3, then the polygons will be filled in green. For example, suppose the command S_FILSTY(2) was issued. This command would cause polygons to be filled in red.

S_LINSTY(N)

ARGUMENTS N:INTEGER

This command sets the color of the pen to draw the proceeding lines. The value N is an integer ranging from 1 to 3. If N is 1, then the pen color is blue. If N is 2, then the pen color is red. If N is 3, then the pen color is green. For example, suppose the command S_LINSTY(2) was issued. This command would change the color of the pen to red.

S_TRANSF(N,Sx,Sy,A,Tx,Ty)

ARGUMENTS N:INTEGER

Sx:FLOAT

Sy:FLOAT

A:FLOAT

Tx:FLOAT

Ty:FLOAT

This command scales, rotates, or translates a given segment. The value N is an integer used to identify the segment to be transformed and must range in value from 1 to 2000. The values Sx and Sy are the real values that represent the X and Y scaling factors respectively. The value A is a real number representing the angle in radians the segment is to be rotated. The value Tx and Ty are real values that represent the X and Y translating factors respectively. When A is not 0.0, Tx and Ty represent the point about which the segment will be rotated. For example, suppose the command S_TRANSF(2,0.5,0.5,1.0,0.5,0.5) was issued. This command would result in segment number 2 being scaled in both the X and Y directions by 0.5 and rotated 1.0 radians about the point (0.5,0.5).

CHAPTER 4

ERROR MESSAGES

There are several error messages generated by the CORE System due to user oversight of the systems limitations. The error messages and a brief description of their causes are listed below:

NO SEGMENT OPEN -

This message is issued whenever an attempt is made to close a segment using the CLOS_SEG command and segment N has not been opened using the CREA_SEG(N) command.

SEGMENT STILL OPEN -

This message is issued whenever an attempt is made to create a segment using the CREA_SEG(N) command or delete a segment using DLET_SEG(N) command before a CLOS_SEG command has been issued for another segment.

INVALID SEGMENT NAME -

This message is issued whenever an attempt is made to create a segment using the CREA_SEG(N) command, delete a segment using the DLET_SEG(N) command, transform a segment

using the S_TRANF(N,Sx,Sy,A,Tx,Ty), or set the visibility of a segment using the SET_VIS(N,Z) command and N does not fall in the range of 1 to 2000.

SEGMENT ALREADY OPEN -

This message is issued whenever an attempt is made to create a segment using the CREA_SEG(N) command and segment N already exists.

POLYGON SIZE ERROR -

This message is issued whenever the POL_ABS2(XARRAY,YARRAY,N), POL_ABS3(XARRAY,YARRAY,ZARRAY,N), POL_REL2(XARRAY,YARRAY,N), or POL_REL3(XARRAY,YARRAY,ZARRAY,N) commands are used and N does not fall in the range of 3 and 31.

DISPLAY FILE FULL -

This message is issued whenever more than 2000 graphics instructions are generated.

BAD VIEWPORT -

This message is issued whenever the left viewport boundary is greater than the right viewport boundary, or the bottom viewport boundary is greater than the top viewport

boundary using the SET_VIEW(XL,XH,YL,YH) command.

BAD WINDOW -

This message is issued whenever the left window boundary is greater than the right window boundary, or the bottom window boundary is greater than the top window boundary using the SET_WINDOW(XL,XH,YL,YH) command.

CLIPPED POLYGON TOO BIG -

This message is issued whenever the number of polygon sides stored, COUNT_OUT, is greater than 32 using the POL_CLIP command.

INVALID VIEW PLANE NORMAL -

This message is issued whenever the length of the user's specification vector, D, is less than some small number greater than any round-off error, RNDOFF, using the SET_VIEWPLANOR(DX,DY,DZ) command.

NO SET-VIEW-UP DIRECTION -

This message is issued whenever the sum of the absolute values of DX, DY, and DZ is less than some small number greater than any round-off error, RNDOFF, using the SET_VIEWUP(DX,DY,DZ) command.

NO DIRECTION OF PROJECTION -

This message is issued whenever the sum of the absolute values of DX, DY, and DZ is less than some small number greater than any round-off error, RNDOFF, using the SET_PARA(DX,DY,DZ) command.

SET-VIEW-UP ALONG VIEWPLANE NORMAL -

This message is issued whenever the square root of the sum of the squares of each view-up direction coordinate is less than some small number greater than any round-off error, RNDOFF, using the SET_VIEWPLATRANS command.

CENTER OF PROJECTION BEHIND VIEW PLANE -

This message is issued whenever the z coordinate of the center of projection in view plane coordinates is less than zero using the M_PERS_TRANS command.

PROJECTION PARALLEL VIEW PLANE -

This message is issued whenever the z coordinate of the direction of projection in view plane coordinates is less than some small number greater than any round-off error, RNDOFF, using the M_PARA_TRANS command.

FRONT PLANE BEHIND BACK PLANE -

This message is issued whenever the front plane distance from the view reference point is greater than the back plane distance from the view reference point along the view plane normal using the SET_VIEWDEP(F_DIS,B_DIS) command.

CHAPTER 5

LINE GENERATION

Computer graphics images are created by setting the intensity (that is, brightness) and color of the pixels which compose the screen. The method used in this project to create graphical images is called vector graphics. In this case, a vector is a line segment that has a single direction and a length. The line segments are built from a finite number of points. Since these points must have some size, they are not really points but instead pixels (short for picture elements). Each pixel has a x, y, and intensity value [HARRINGTON]. The type of algorithm used to calculate the x and y values for each pixel is called an incremental alogorithm. The name comes from the fact that at each step, incremental calculations are based on the preceeding step. The incremental algorithm that was used after extending the graphics system is called Bresenham's Line Algorithm. Bresenham's algorithm was attractive because it uses only integer arithmetic. No real variables are used and hence rounding is not needed [FOLEY].

The algorithm uses a decision variable Di which at each step is proportional to the difference between variables s

and t. The variable s represents the distance between the actual slope and the calculated slope, S(i), that lies below the actual slope. The variable t represents the distance between the actual slope and the calculated slope, Ti, that lies above the actual slope. At the ith step, the pixel P(i-1) has been determined to be closest to the actual line being drawn and the calculations must be performed to determine the next pixel, P(i), to be set, T(i) or S(i). If s < t, then S(i) is closer to the desired line and should be set; else T(i) is closer and should be set. In other words, S(i) is chosen if s - t < 0, otherwise T(i) is chosen [FOLEY].

Parametric equations are used to represent the equation of a line. In parametric equations, the x and y values are given in terms of a parameter, in this case u (u=1). To generate the line segment between two points (x1,y1) and (x2,y2), it is necessary for the x coordinate to go uniformly from x1 to x2 and the y coordinate to go uniformly from y1 to y2. This may be expressed by the general parametric equations

$$x = x1 + (x2 - x1) * u$$

and

$$y = y1 + (y2 - y1) * u.$$

The starting point used is, x is x1 and y is y1. As the sum

increases by 1, x moves uniformly to x2 and y moves uniformly to y2. These two equations together describe a straight line. The idea is to start at (x1,y1) and increment x1 and y1 by unit steps until they reach the point (x2,y2). This is useful for drawing a line. At each step, the intensity is set for the pixel which contains the point (x,y). This process of "turning on" the pixels for a line segment is called vector generation [HARRINGTON].

CHAPTER 6

PROBLEMS ENCOUNTERED USING THE CORE GRAPHICS SYSTEM

The conversion of the Pascal routines into Ada had been completed without any problems. Some problems occurred, however, after the conversion was completed. The problems encountered dealt with speed and the limitations of the initial implementation. The word speed is used loosely here to mean the rate at which a line segment is drawn.

The major problem with the graphics package is speed. Vector generators are judged on linearity, speed, brightness, uniformity, and endpoint matching [FOLEY]. The extended graphics package possessed the qualities previously mentioned except the speed factor. Speed is typically either a constant or some function of vector length. In the latter case, the time for short vectors is constant because the set-up time for calculating and addressing a particular pixel becomes the dominating factor. The actual time that it takes to draw a vector is proportional to the change in x or y, whichever is larger.

There are several additional factors which slow the speed of the vector generator. In implementing a system

that uses vector graphics, more instructions are added to the memory of the display file. The display file contains all of the input commands used by the vector generator to generate a particular image. These commands are examined and lines are drawn using the vector-generating routine. In order to present a steady image, the image must be drawn repeatedly. This means that the vector generator must be applied to all of the lines of the image before a flicker is noticeable (more than 30 times a second). The implemented graphics package also contains numerous subroutines and uses floating point calculations for vector generation.

One solution to the above problem is to perform the calculations needed by the vector generator and store the calculated values in a separate file. This file would contain all of the instructions required by the vector generator for the construction of a particular image. This type of file would be used with all of the created images. To display a chosen file, the display file processor reads this file and the vector generator generates the image. Due to the lack of time, this solution to the problem will be further explored at a later date. One possible constraint that would be encountered using this approach would be due to the lack of available computer memory on some micro-computers. The lack of memory may be a problem using

a larger computer.

Another solution to the above mentioned problem is to reduce the number of subroutines. Even though each subroutine performs different tasks, some subroutines can be merged together into a larger subroutine. This solution has been implemented and it does increase the rate of drawing a line segment, but it does not drastically increase the rate of the vector generator. Also, the vector generator used in the initial implementation used a mixture of integer and floating point calculations involving rounding. The vector generator implemented after extending the graphics package used only integer calculations and hence rounding is not needed with integer calculations. The line segments are generated faster because it takes the computer longer to perform floating point calculations. Even though the initial Ada implementation used a subroutine that accelerated floating point calculations, but due to the large number of calculations, the rate was still slow. This process of using only integer calculations has been implemented and the speed of the vector generator has drastically increased. However, there is still more research needed to explore different algorithm for increasing the speed of the vector generator.

A second problem was due the limitations of the initial

graphics packages. The initial implementation only included 2 dimensional pictures with no circles, windowing or clipping and mapping to viewports. This problem was solved by extending the graphics package. The extended version includes circles, windowing, clipping, mapping to viewports, and 3 dimensional images. With the ability to draw circles, the circle routine has the capability to draw ellipses.

Parametric equations are used to represent the equation of a circle. In the parametric equations for the circle, the x and y values are calculated in terms of the equation

$$x = Z + A \cos(U)$$

and

$$y = Z + B \sin(U).$$

The radius of the circle is represented by the parameters A and B. The Z parameter is the distance that the center of the circle is away from the origin. The screen has normalized coordinates and the measurements are 1 unit wide and 1 unit high. The lower-left corner of the screen is the origin and the upper-right corner is the point (1,1). When A=B, the output is a circle. When A≠B, the output is an ellipse. The circle is generated by drawing line segments along the boundary of the circle.

The technique used for windowing is a method for selecting and enlarging or shrinking portions of a drawing.

This gives the effect of looking at the image through a window. Clipping is the technique for not showing the parts of a drawing that is not of interest for viewing. A window can be considered as a box that contains a portion of an object or the entire object. A viewport can be considered as a box on the screen that the object is confined to. The window box would in turn be viewed through the viewport box. Now the image inside the box has been mapped to a viewport. All of the images created by the extended version are viewed through a window and mapped to a viewport. The initial coordinates window and viewport are (0.0,1.0,0.0,1.0). The coordinates are changed by using window and viewport routines.

CONCLUSION

The CORE Graphics System serves as a foundation for more complex and special purpose routines. Even though the graphics package has been extended, it is desirable to continue extending the graphics package so that the implemented graphics package represents the best software design practices. This can be accomplished by implementing and testing new graphic techniques. This will lead to a graphics package that can be suited to the needs of a specific group of individuals for a specific purpose.

The Ada implementation had fewer limitations than the Pascal implementation due to the standardization of the language. Also, the Ada implementation can be expanded with greater ease than the Pascal implementation. The Ada programming language strengthens one of the goals of the Graphics Standards Planning Committee of the ACM by being a standardized language. This will allow the implemented graphics package to be device independent and portable by simply implementing the CORE Graphics System in Ada.

BIBLIOGRAPHY

[FOLEY] Foley, J. D. and Dam, A. Van. Fundamentals of Interactive Computer Graphics. Addison-Wesley, Reading, Ma., 1983.

[HARRINGTON] Harrington, Steven. Computer Graphics: A Programming Approach. McGraw Hill, New York, 1983.

[HOROWITZ] Horowitz, Ellis. Programming Languages: A Grand Tour. Computer Science Press, Rockville, Md., 1983.

[MARTIN] Martin, Benjamin J., Setzer, Bennett, and Walker, Reginald. Implementation of a Graphics Package in Ada. In Proceedings of the 4th Annual National Conference on Ada Technology, Department of Commerce, Springfield, Va., 1986, pp. 100-103.

[PAYNE] Payne, Gregory. "Implementation of the CORE Graphics System on the Zenith Z-100". Thesis, Atlanta University, Atlanta, Ga., 1985.

APPENDIX A

A BRIEF OVERVIEW OF ADA

The programming language, Ada, originated in the early 1970s from a proposal by the United States Department of Defense to find a suitable language that would enable DoD to cut the rising cost of software. Ada was designed for so-called embedded computer systems, systems which must reside in aircraft or ships. The embedded systems sector embraces applications such as tactical weapon systems, communications, command and control and so on. Nevertheless the language which was developed is very broad in scope and will likely find itself most suited for large-scale software development on a large computer. Ada is the first practical language of the second revolution and embodies the fruits of research of the last decade [HOROWITZ].

APPENDIX B

SOURCE CODE

```
function CLIP_XORY(A1,B1,Z1,A2,B2,Z2,A_CLIP,
                   Z_CLIP:float) return float is
-- This function calculates the third coordinate of the
-- intersection of a line with the clipping plane
--    (A1,B1,Z1) and (A2,B2,Z2) are the coordinates of the
--       endpoints of the line segment being clipped
--    B1 and B2 correspond to the unknown coordinate
--    Z_CLIP is the z coordinate of the intersection point
--    A_CLIP is the other known coordinate of the
--       intersection point
--    RNDOFF is some small number greater than any
--       round-off error
begin
   if abs(Z2-Z1)>RNDOFF then
      return ((B2-B1)/(Z2-Z1)*(Z_CLIP-Z1)+B1);
   elsif abs(A2-A1)>RNDOFF then
      return ((B2-B1)/(A2-A1)*(A_CLIP-A1)+B1);
   else
      return B1;
   end if;
end CLIP_XORY;


function CLIP_Z(A1,Z1,A2,Z2,M,B:float) return float is
-- This function calculates the z value of the intersection
-- of a line with the clipping plane
--    A1 and Z1 are the coordinates of one endpoint of the
--       line segment
--    A2 and Z2 are the coordinates of other endpoint
--    M is the slope of the clipping plane
--    B is the intercept in the clipping plane equation
begin
   return ((A2-A1)*Z1+(B-A1)*(Z2-Z1))/(A2-A1-M*(Z2-Z1));
end CLIP_Z;


function INT(E:float) return integer is
-- This function will take the integer value of a
-- floating point number
R:integer;
F:float;
begin
```

```
   R:=intege (E);
   F:=E;
   if E<0.0 then
      return(-(INT(abs(E))));
   elsif abs(float(R)-E) ≥ 0.09 and float(R)-E < 0.5 then
      F:=F-0.5;
      return(integer(F));
   else
      R:= integer(E);
      return(R);
   end if;
end INT;


procedure BRESENHAM(X1,Y1,X2,Y2:float;INTENSE:integer) is
-- Algorithm taken from Fund. of Computer Graphics
-- Replaces DDA
-- This procedure caluates the pixel values of the frame
-- buffer along a line segment
--   X1 and Y1 are the coordinates of the starting point
--   X2 and Y2 are the coordinates of the ending point
--   INTENSE is the intensity setting to be used for the
--     vector
--   DX is the change in the X value
--   DY is the change in the Y value
--   XX1 and YY1 represent the integer values of X1 and Y1,
--     respectively
--   XX2 and YY2 represent the integer values of X2 and Y2
--   INCR1 is the constant used for incrementing D if D < 0
--   INCR2 is the constant used for incrementing D if D ≥ 0
--   D is a decision variable which is proportional to the
--     difference in DX and DY for each step
--   X and Y are the points along the line segment
--   XEND equals the X2 coordinate of the end point
--   YEND equals the Y2 coordinate of the end point
   DX,DY:integer;
   XX1,XX2,YY1,YY2,INCR1,INCR2:integer;
   D,X,Y,XEND,YEND:integer;
begin
   XX1:=INT(X1);
   XX2:=INT(X2);
   YY1:=INT(Y1);
   YY2:=INT(Y2);
   DX:=abs(XX2-XX1);
   DY:=abs(YY2-YY1);
   D:=2*DY-DX;
   INCR1:=2*DY;
```

```
    INCR2:=2*(DY-DX);
    if XX1 > XX2 then
       X:=XX2;
       Y:=YY2;
       XEND:=XX1;
       YEND:=YY1;
    else
       X:=XX1;
       Y:=YY1;
       XEND:=XX2;
       YEND:=YY2;
    end if;
    SET_PIX(X,Y,INTENSE);
    if DX<DY then
       while Y<YEND loop
          Y:=Y+1;
          if D<0 then
             D:=D+INCR1;
          elsif (XX1<XX2 and YY1>YY2)
                or (XX1>XX2 and YY1<YY2) then
             X:=X-1;
             D:=D+INCR2;
          else
             X:=X+1;
             D:=D+INCR2;
          end if;
          SET_PIX(X,Y,INTENSE);
       end loop;
    else
       while X<XEND loop
          X:=X+1;
          if D< 0 then
             D:=D+INCR1;
          elsif (XX1<XX2 and YY1>YY2)
                or (XX1>XX2 and YY1<YY2) then
             Y:=Y-1;
             D:=D+INCR2;
          else
             Y:=Y+1;
             D:=D+INCR2;
          end if;
          SET_PIX(X,Y,INTENSE);
       end loop;
    end if;
end BRESENHAM;
```

```
procedure BUILDTRN(SEG_NAME:integer) is
-- This procedure builds the image transformation matrix
--     SEG_NAME is the segment which is being transformed
--     SCALE_X, SCALE_Y, ANGLE, TRANS_X and TRANS_Y are the
--        arrays for the attribute part of the segment
--        table
--     H is the transformation array of 3 X 2 elements
--     I is for stepping through the array
begin
   IDEN_MAT;
   MULNSCAL;
   if ANGLE(SEG_NAME) /= 0.0 then
      MULNTRAN(0.0-TRANS_X(SEG_NAME),0.0-TRANS_Y(SEG_NAME));
      MULNROTA(ANGLE(SEG_NAME));
      MULNTRAN(TRANS_X(SEG_NAME),TRANS_Y(SEG_NAME));
   else
      MULNROTA(ANGLE(SEG_NAME));
      MULNTRAN(TRANS_X(SEG_NAME),TRANS_Y(SEG_NAME));
   end if;
end BUILDTRN;


procedure CIRCLE(D:in character;C:in integer;A,B,Z:in float)
is
E,X,Y:float;
begin
   INIT;
   if D='Y' or D='y' then
      s_fill(true);
   end if;
   E:=1.5707963;
   S_LINSTY(C);
   FOR M IN 0..360 LOOP
   IF D='Y' OR D='y' THEN
     CREA_SEG(M);
   END IF;
   X:=Z+A*COS(E);
   Y:=Z+B*SIN(E);
   IF M=0 THEN
      MOV_ABS2(X,Y);
   ELSE
      LIN_ABS2(X,Y);
   END IF;
   E:=E+0.0174533;
   IF D='Y' or D='y' THEN
      CLOS_SEG;
   END IF;
```

```
   END LOOP;
   e_flag:=false;
   m_pic_cu;
end CIRCLE;


procedure CLIP(OP:integer;X,Y,Z:float) is
-- This procedure is a top-level clipping routine.  It
-- decides between handling polygons and handling other
-- graphics primitives.
--    OP, X, Y and Z are the instruction being clipped
--    P_FLAG indicates that a polygon is being processed
--    COUNT_IN is the number of polygon sides still to be
--       input
--    COUNT_OUT is the number of clipped polygon sides
--       stored
--    XS, YS and ZS are arrays for saving the last point
--       drawn
begin
   if PFLAG then
      POL_CLIP(OP,X,Y,Z);
   elsif OP>2 and OP<31 then
      PFLAG:=TRUE;
      COUNT_IN:=OP;
      COUNT_OUT:=0;
      for I in 1..6 loop
         XS(I):=X;
         YS(I):=Y;
         ZS(I):=Z;
      end loop;
      M_CLIP_TEST;
   else
      CLIP_LEFT(OP,X,Y,Z);
   end if;
end CLIP;


procedure CLIP_BACK(OP:integer;X,Y,Z:float) is
-- This procedure clips against the back plane
--    OP, X, Y and Z are the instructions for the new
--       endpoint
--    BAC_Z is the position of the back clipping plane
--    B_FLAG is the back clipping flag
--    XS(5), YS(5) and ZS(5) are the coordinates of the line
--       segment's old endpoint
--    X_CLIP and Y_CLIP are the coordinates of the clipped
--       point
```

```
--    Z_CHAN is the fractional change in the z coordinate
--       due to clipping
   X_CLIP,Y_CLIP,Z_CHAN:float;
begin
   if B_FLAG then
      if (Z>=BAC_Z and ZS(5)<BAC_Z)
        or (Z<=BAC_Z and ZS(5)>BAC_Z) then
         Z_CHAN:=(BAC_Z-Z)/(Z-ZS(5));
         X_CLIP:=(X-XS(5))*Z_CHAN+X;
         Y_CLIP:=(Y-YS(5))*Z_CHAN+Y;
         if ZS(5)<BAC_Z or OP>31 then
            CLIP_FRONT(1,X_CLIP,Y_CLIP,BAC_Z);
         else
            CLIP_FRONT(OP,X_CLIP,Y_CLIP,BAC_Z);
         end if;
      end if;
      XS(5):=X;
      YS(5):=Y;
      ZS(5):=Z;
      if Z>=BAC_Z then
         CLIP_FRONT(OP,X,Y,Z);
      end if;
   else
      CLIP_FRONT(OP,X,Y,Z);
   end if;
end CLIP_BACK;


procedure CLIP_BOTTOM(OP:integer;X,Y,Z:float) is
-- This procedure clips against the lower boundary
--    OP, X, Y and Z are the instructions for the new
--       endpoint
--    WYL is the lower window boundary
--    YLM is the slope of the lower clipping plane
--    XS(3), YS(3) and ZS(3) are the coordinates of the old
--       endpoint
--    OPTTE(3) the test condition for the old endpoint
--    NPTTE is the test condition for the new endpoint
--    X_CLIP, Y_CLIP and Z_CLIP are the coordinates of the
--       clipped point
   NPTTE,X_CLIP,Y_CLIP,Z_CLIP:float;
begin
   NPTTE:=YLM*Z+WYL;
   if (Y>=NPTTE and YS(3)<OPTTE(3))
     or (Y<=NPTTE and YS(3)>OPTTE(3)) then
      -- Crosses plane so find the intersection point
      Z_CLIP:=CLIP_Z(Y,Z,YS(3),ZS(3),YLM,WYL);
```

```
      Y_CLIP:=YLM*Z_CLIP+WYL;
      X_CLIP:=CLIP_XORY(Y,X,Z,YS(3),XS(3),ZS(3),Y_CLIP,
                        Z_CLIP);
      if YS(3)<OPTTE(3) or OP>31 then
         -- Case of outside going in
         CLIP_TOP(1,X_CLIP,Y_CLIP,Z_CLIP);
      else
         -- Case of inside going out
         CLIP_TOP(OP,X_CLIP,Y_CLIP,Z_CLIP);
      end if;
   end if;
   -- Remember point to serve as endpoint for the next line
   -- segment
   XS(3):=X;
   YS(3):=Y;
   ZS(3):=Z;
   -- Remeber the test condition for the point too
   OPTTE(3):=NPTTE;
   -- Case of point inside
   if Y>NPTTE then
      CLIP_TOP(OP,X,Y,Z);
   end if;
end CLIP_BOTTOM;


procedure CLIP_FRONT(OP:integer;X,Y,Z:float) is
-- This procedure clips against the front plane
--    OP, X, Y and Z are the instructions for the new
--       endpoint
--    FRO_Z is the position of the back clipping plane
--    F_FLAG is the back clipping flag
--    XS(6), YS(6) and ZS(6) are the coordinates of the line
--       segment's old endpoint
--    X_CLIP and Y_CLIP are the coordinates of the clipped
--       point
--    Z_CHAN is the fractional change in the z coordinate
--       due to clipping
   X_CLIP,Y_CLIP,Z_CHAN:float;
begin
   if F_FLAG then
      if (Z<FRO_Z and ZS(6)>FRO_Z)
        or (Z>FRO_Z and ZS(6)<FRO_Z) then
         Z_CHAN:=(FRO_Z-Z)/(Z-ZS(6));
         X_CLIP:=(X-XS(6))*Z_CHAN+X;
         Y_CLIP:=(Y-YS(6))*Z_CHAN+Y;
         if ZS(6)>FRO_Z or OP>31 then
            S_CLIP_PT(1,X_CLIP,Y_CLIP,FRO_Z);
```

```
            else
               S_CLIP_PT(OP,X_CLIP,Y_CLIP,FRO_Z);
            end if;
         end if;
         XS(6):=X;
         YS(6):=Y;
         ZS(6):=Z;
         if Z<FRO_Z then
            S_CLIP_PT(OP,X,Y,Z);
         end if;
      else
         S_CLIP_PT(OP,X,Y,Z);
      end if;
end CLIP_FRONT;


procedure CLIP_LEFT(OP:integer;X,Y,Z:float) is
-- This procedure clips against the left boundary
--     OP, X, Y and Z are a display file instruction
--     WXL is the left window boundary
--     XLM is the slope of the left clipping plane
--     XS(1), YS(1) and ZS(1) are the coordinates of the old
--        endpoint
--     OPTTE(1) is the test condition for the old endpoint
--     NPTTE is the test condition for the new endpoint
--     X_CLIP, Y_CLIP and Z_CLIP are the coordinates of the
--        clipped point
   NPTTE,X_CLIP,Y_CLIP,Z_CLIP:float;
begin
   NPTTE:=XLM*Z+WXL;
   if (X>NPTTE and XS(1)<OPTTE(1))
     or (X<NPTTE and XS(1)>OPTTE(1)) then
      -- Crosses plane so find the intersection point
      Z_CLIP:=CLIP_Z(X,Z,XS(1),ZS(1),XLM,WXL);
      X_CLIP:=XLM*Z_CLIP+WXL;
      Y_CLIP:=CLIP_XORY(X,Y,Z,XS(1),YS(1),ZS(1),X_CLIP,
                        Z_CLIP);
      if XS(1)<OPTTE(1) or OP>31 then
         -- Case of outside going in
         CLIP_RIGHT(1,X_CLIP,Y_CLIP,Z_CLIP);
      else
         -- Case of inside going out
         CLIP_RIGHT(OP,X_CLIP,Y_CLIP,Z_CLIP);
      end if;
   end if;
   -- Remember point to serve as endpoint for the next line
   -- segment
```

```
   XS(1):=X;
   YS(1):=Y;
   ZS(1):=Z;
   -- Remember the test condition for the point too
   OPTTE(1):=NPTTE;
   -- Case of point inside
   if X>NPTTE then
      CLIP_RIGHT(OP,X,Y,Z);
   end if;
end CLIP_LEFT;


procedure CLIP_RIGHT(OP:integer;X,Y,Z:float) is
-- This procedure clips against the right boundary
--    OP, X, Y and Z are the instructions for the new
--       endpoint
--    WXH is the right window boundary
--    XHM is the slope of the right clipping plane
--    XS(2), YS(2) and ZS(2) are the coordinates of the old
--       endpoint
--    OPTTE(2) the test condition for the old endpoint
--    NPTTE is the test condition for the new endpoint
--    X_CLIP, Y_CLIP and Z_CLIP are the coordinates of the
--       clipped point
   NPTTE,X_CLIP,Y_CLIP,Z_CLIP:float;
begin
   NPTTE:=XHM*Z+WXH;
   if (X<NPTTE and XS(2)<OPTTE(2))
     or (X>NPTTE and XS(2)>OPTTE(2)) then
      -- Crosses plane so find the intersection point
      Z_CLIP:=CLIP_Z(X,Z,XS(2),ZS(2),XHM,WXH);
      X_CLIP:=XHM*Z_CLIP+WXH;
      Y_CLIP:=CLIP_XORY(X,Y,Z,XS(2),YS(2),ZS(2),X_CLIP,
                        Z_CLIP);
      if XS(2)>OPTTE(2) or OP>31 then
         -- Case of outside going in
         CLIP_BOTTOM(1,X_CLIP,Y_CLIP,Z_CLIP);
      else
         -- Case of inside going out
         CLIP_BOTTOM(OP,X_CLIP,Y_CLIP,Z_CLIP);
      end if;
   end if;
   -- Remember point to serve as endpoint for the next line
   -- segment
   XS(2):=X;
   YS(2):=Y;
   ZS(2):=Z;
```

```
   -- Remember the test condition for the point too
   OPTTE(2):=NPTTE;
   -- Case of point inside
   if X<=NPTTE then
      CLIP_BOTTOM(OP,X,Y,Z);
   end if;
end CLIP_RIGHT;


procedure CLIP_TOP(OP:integer;X,Y,Z:float) is
-- This procedure clips against the upper boundary
--    OP, X, Y and Z are the instructions for the new
--       endpoint
--    WYH is the upper window boundary
--    YHM is the slope of the upper clipping plane
--    XS(4), YS(4) and ZS(4) are the coordinates of the old
--       endpoint
--    OPTTE(4) the test condition for the old endpoint
--    NPTTE is the test condition for the new endpoint
--    X_CLIP, Y_CLIP and Z_CLIP are the coordinates of the
--       clipped point
   NPTTE,X_CLIP,Y_CLIP,Z_CLIP:float;
begin
   NPTTE:=YHM*Z+WYH;
   if (Y<=NPTTE and YS(4)>OPTTE(4))
     or (Y>=NPTTE and YS(4)<OPTTE(4)) then
      -- Crosses plane so find the intersection point
      Z_CLIP:=CLIP_Z(Y,Z,YS(4),ZS(4),YHM,WYH);
      Y_CLIP:=YHM*Z_CLIP+WYH;
      X_CLIP:=CLIP_XORY(Y,X,Z,YS(4),XS(4),ZS(4),Y_CLIP,
                        Z_CLIP);
      if YS(4)>OPTTE(4) or OP>31 then
         -- Case of outside going in
         CLIP_BACK(1,X_CLIP,Y_CLIP,Z_CLIP);
      else
         -- Case of inside going out
         CLIP_BACK(OP,X_CLIP,Y_CLIP,Z_CLIP);
      end if;
   end if;
   -- Remember point to serve as endpoint for the next line
   -- segment
   XS(4):=X;
   YS(4):=Y;
   ZS(4):=Z;
   -- Remember the test condition for the point too
   OPTTE(4):=NPTTE;
   -- Case of point inside
```

```
   if Y<NPTTE then
      CLIP_BACK(OP,X,Y,Z);
   end if;
end CLIP_TOP;


procedure CLOS_SEG is
-- This procedure closes a named segment
--    NOW_OPEN is the name of the currently open segment
--    FREE is the index of the next free display file cell
--    SEG_START and SEG_SIZE are the start and size of the
--       segments
begin
   if NOW_OPEN = 0 then
      put("NO SEGMENT OPEN");
      new_line;
   else
      DLET_SEG(0);
      SEG_START(0):=FREE;
      SEG_SIZE(0):=0;
      NOW_OPEN:=0;
   end if;
end CLOS_SEG;


procedure CREA_SEG(SEG_NAME:integer) is
-- This procedure creates a named segment
--    SEG_NAME is the segment name
--    NOW_OPEN is the segment currently open
--    FREE is the index of the next free display file cell
--    SEG_START, SEG_SIZE, ANGLE, SCALE_X, SCALE_Y, TRANS_X,
--       TRANS_Y and VISIBLE together make up the segment
--       table
--    NUM_SEGS is the size of the segment table
begin
   if NOW_OPEN > 0 then
      put("SEGMENT STILL OPEN");
      new_line;
   elsif (SEG_NAME < 1) or (SEG_NAME > NUM_SEGS) then
      put("INVALID SEGMENT NAME");
      new_line;
   elsif SEG_SIZE(SEG_NAME) > 0 then
      put("SEGMENT ALREADY EXIST");
      new_line;
   else
      NEW_VIEW3;
      SEG_START(SEG_NAME):=FREE;
```

```
      SEG_SIZE(SEG_NAME):=0;
      VISIBLE(SEG_NAME):=VISIBLE(0);
      ANGLE(SEG_NAME):=ANGLE(0);
      SCALE_X(SEG_NAME):=SCALE_X(0);
      SCALE_Y(SEG_NAME):=SCALE_Y(0);
      TRANS_X(SEG_NAME):=TRANS_X(0);
      TRANS_Y(SEG_NAME):=TRANS_Y(0);
      NOW_OPEN:=SEG_NAME;
   end if;
end CREA_SEG;


procedure DFE(OP:integer) is
-- This procedure combines operation and position to form an
-- instruction and save it in the display file
--    OP is the operation to be entered
--    DF_PEN_X and DF_PEN_Y are the coordinates of the
--       current pen position
--    PERS_FLAG is the perspective vs. parallel projection
--       flag
--    X, Y and Z hold the coordinates of the point that is
--       transformed
   X,Y,Z:float;
begin
   if OP<0 then
      PUT_PNT(OP,0.0,0.0);
   else
      X:=DF_PEN_X;
      Y:=DF_PEN_Y;
      Z:=DF_PEN_Z;
      V_PLA_TRANS(X,Y,Z);
      CLIP(OP,X,Y,Z);
   end if;
end DFE;


procedure DLET_ALL is
-- This procedure was used to delete all segments
begin
   for I in 0..NUM_SEGS loop
      SEG_START(I):=1;
      SEG_SIZE(I):=0;
   end loop;
   NOW_OPEN:=0;
   FREE:=1;
end DLET_ALL;
```

```
procedure DLET_SEG(SEG_NAME:integer) is
-- This procedure is used to delete a segment
--    SEG_NAME is the segment name
--    NOW_OPEN is the currently open segment
--    FREE is the index of the next free display file cell
--    SEG_START and SEG_SIZE are part of the segment table
--    NUM_SEGS is the size of the segment table
--    GEET is the location of an instruction to be moved
--    PUUT is the location to which an instruction should
--       be moved
--    SIZE is the size of the deleted segment
--    I is a variable for stepping through the segment
--       table
   GEET,PUUT,SIZE,OP:integer;
   X,Y:float;
begin
   if (SEG_NAME < 0) or (SEG_NAME > NUM_SEGS) then
      put("INVALID SEGMENT NAME");
      new_line;
   elsif (SEG_NAME = NOW_OPEN) AND (SEG_NAME /= 0) then
      put("SEGMENT STILL OPEN");
      new_line;
   elsif SEG_SIZE(SEG_NAME) /= 0 then
      PUUT:=SEG_START(SEG_NAME);
      SIZE:=SEG_SIZE(SEG_NAME);
      GEET:=PUUT+SIZE;
      -- Shift the display file elements
      while GEET < FREE loop
         GET_PNT(GEET,OP,X,Y);
         SET_PNT(PUUT,OP,X,Y);
         PUUT:=PUUT+1;
         GEET:=GEET+1;
      end loop;
      -- Recover the deleted storage
      FREE:=PUUT;
      -- Update the segment table
      for i in 0..NUM_SEGS loop
         if SEG_START(I) > SEG_START(SEG_NAME) then
            SEG_START(I):=SEG_START(I)-SIZE;
         end if;
      end loop;
      SEG_SIZE(SEG_NAME):=0;
      if VISIBLE(SEG_NAME) then
         NEW_FRAM;
      end if;
   end if;
end DLET_SEG;
```

```
procedure DOLINE(X,Y:float) is
-- This procedure draws a line
--    X and Y are the coordinates of the point to which to
--       draw the line (normalized coordinates)
--    F_PEN_X and F_PEN_Y are the coordinates of the pen
--       position (actual screen coordinates)
--    WIDTH and HEIGHT are the frame dimensions (screen
--        dimensions)
--    W_START and H_START are the coordinates of the
--       lower-left corner
--    W_END and H_END are the coordinates of the upper-right
--       corner
--    LINECHR is the style of the line
--    X1 and Y1 are the old end point coordinates of the
--       line segment
--    MIN is the minimum value
   X1,Y1,MIN:float;
begin
   X1:=F_PEN_X;
   Y1:=F_PEN_Y;
   if float(W_END) <= (X*float(WIDTH)+float(W_START)) then
      MIN:=float(W_END);
   else
      MIN:=X*float(WIDTH)+float(W_START);
   end if;
   if float(W_START) >= MIN then
      F_PEN_X:=float(W_START);
   else
      F_PEN_X:=float(MIN);
   end if;
   if float(H_END) <= (Y*float(HEIGHT)+float(H_START)) then
      MIN:=float(H_END);
   else
      MIN:=Y*float(HEIGHT)+float(H_START);
   end if;
   if float(H_START) >= MIN then
      F_PEN_Y:=float(H_START);
   else
      F_PEN_Y:=MIN;
   end if;
   BRESENHAM(X1,Y1,F_PEN_X,F_PEN_Y,LINECHR);
end DOLINE;


procedure DOMOVE(X,Y:float) is
-- This procedure performs a move of the pen
--    X and Y are the coordinates of the point to which to
```

```
--          the pen (normalized coordinates)
--       F_PEN_X and F_PEN_Y are the coordinates of the pen
--          position (actual screen coordinates)
--       WIDTH and HEIGHT are the frame dimensions (screen)
--       W_START and H_START are the coordinates of the
--          lower-left corner
--       W_END and H_END are the coordinates of the upper-right
--          corner
--       MIN represents the minimum value
      MIN:float;
begin
      if float(W_END) <= X*float(WIDTH)+float(W_START) then
         MIN:=float(W_END);
      else
         MIN:=X*float(WIDTH)+float(W_START);
      end if;
      if float(W_START) >= MIN then
         F_PEN_X:=float(W_START);
      else
         F_PEN_X:=MIN;
      end if;
      if float(H_END) <= Y*float(HEIGHT)+float(H_START) then
         MIN:=float(H_END);
      else
         MIN:=Y*float(HEIGHT)+float(H_START);
      end if;
      if float(H_START) >= MIN then
         F_PEN_Y:=float(H_START);
      else
         F_PEN_Y:=MIN;
      end if;
end DOMOVE;


procedure DOPOLY(OP:integer;
X,Y:float;NTH:integer) is
-- This procedure was used to process a polygon
-- command
      if SOLID then
         FILLPOLY(OP,X,Y,NTH);
      end if;
      DOMOVE(X,Y);
end DOPOLY;


procedure DOPROJ(OP:integer;
X,Y,Z:float) is
```

```
-- This procedure projects and saves a drawing command
--    OP, X, Y and Z are the instruction to be saved
--    PERS_FLAG is the type of projection flag
--    PX, PY and PZ are the projected coordinates
   PX,PY,PZ: float;
begin
   PX:=X;
   PY:=Y;
   PZ:=Z;
   if PERS_FLAG then
      PERS_TRANS(PX,PY,PZ);
   else
      PAR_TRANS(PX,PY,PZ);
   end if;
   VIEW_TRANS(OP,PX,PY);
end DOPROJ;


procedure DOSTYLE(OP:integer) is
-- This procedure inteprets the change of style commands
--    OP indicates the desired style
--    LINECHR is the line character used by BRESENHAM
--    FILLER is the polygon fill style
begin
   case OP is
      when -1 => LINECHR:=0;
      when -2 => LINECHR:=1;
      when -3 => LINECHR:=2;
      when -31 => FILLER:=0;
      when -32 => FILLER:=1;
      when -33 => FILLER:=2;
   end case;
end DOSTYLE;


procedure DO_TRAN is
-- This procedure was used to transform a point
begin
   TEMP:=X*H(1)(1)+Y*H(2)(1)+H(3)(1);
   Y:=X*H(1)(2)+Y*H(2)(2)+H(3)(2);
   X:=TEMP;
end DO_TRAN;


procedure ENABLE is
  VIDEO_PORT:integer;
  VALUE:byte;
```

```
begin
     VIDEO_PORT:=160D8;
     VALUE:=BYTE(1678);
     OUTPORT(VIDEO_PORT,VALUE);
end ENABLE;


procedure ERASE is
-- This procedure clears the frame buffer by assigning
-- every pixel a background value
begin
   put(character'val(27)); PUT('E');
   new_line;
end ERASE;


procedure FILLPOLY(OP:integer;X,Y:float;INDEX:integer) is
-- This procedure fills in a polygon
--    OP, X and Y are the polygon instructions
--    INDEX is the display file index of the instruction
--    YMAX is an array of the upper y coordinates for the
--       polygon sides
--    SCAN_DEC is the size of the scanline decrement
--    EDGES is the number of polygon sides considered
--    SCAN is the y value of the scanline
--    S_EDGE and E_EDGE indicate which polygon sides are
--       crossed by the scanline
--    XA is an array of edge intersection positions
--    NX is the number of line segments to be drawn
--    J is for stepping through the edges
--    K is for stepping through line segments
--    X1 is the starting x coordinate of the line segment
--    X2 is the ending x coordinate of the line segment
--    Y is the y coordinate of the line segment
--    FILLER is the polygon fill style
   EDGES,S_EDGE,E_EDGE:integer;
   SCAN:float;
   NX,J:integer;
   X1,X2,YY:float;
begin
   -- Load global arrays with the polygon vertex information
   LOADPOLY(OP,X,Y,INDEX,EDGES);
   -- Check number of sides to be considered
   if EDGES>2 then
      -- Set scanline
      SCAN:=float(INT(YMAX(1)-0.5));
      -- Initialize starting and ending index values for
```

```
      -- sides considered
      S_EDGE:=1;
      E_EDGE:=1;
      -- Fill in polygon and pick up any new sides to be
      -- included in this scan
      INCLUDE1(E_EDGE,EDGES,SCAN);
      -- Determine the side intersections for this scanline,
      -- removing any sides that have been passed
      UPDXVAL(E_EDGE,S_EDGE,SCAN);
      -- Repeat the filling until all sides have been passed
      while E_EDGE/=S_EDGE loop
         -- Fill in scanline
         FILLSCAN;
         SCAN:=SCAN-SCAN_DEC;
         INCLUDE1(E_EDGE,EDGES,SCAN);
         UPDXVAL(E_EDGE,S_EDGE,SCAN);
      end loop;
   end if;
end FILLPOLY;


procedure FILLSCAN is
-- This procedure was used to fill in the scanline
begin
   NX:=(E_EDGE-S_EDGE)/2;
   J:=S_EDGE;
   for K in 1..NX loop
      X1:=XA(J);
      YY:=SCAN;
      X2:=XA(J+1);
      BRESENHAM(X1,YY,X2,YY,FILLER);
      J:=J+2;
   end loop;
end FILLSCAN;


procedure GET_PNT(NTH:integer;OP:in out integer;
                  X,Y:in out float) is
-- This procedure retrieves the Nth instruction from the
-- display file
--    NTH is the number of the desired instruction
--    OP, X and Y form the instruction to be returned
--    DF_OP, DF_X and DF_Y are the three display file arrays
--       for holding instructions
begin
   OP:=DF_OP(NTH);
   X:=DF_X(NTH);
```

```
   Y:=DF_Y(NTH);
end GET_PNT;


procedure GETT_PNT(NTH,OP:in out integer;
                   X,Y:in out float) is
-- This procedure retrieves and transforms the Nth
-- instruction from the display file
--    NTH is the index of the desired instruction
--    OP, X and Y are the values of the instruction to be
--       returned
--    X and Y are the coordinates of the point to be
--       transformed
--    H is the 3 X 2 transformation matrix
--    TEMP is a temporary storage location for the new X
--       value
   TEMP:float;
begin
   GET_PNT(NTH,OP,X,Y);
   if OP>0 then
      DO_TRAN(X,Y);
   end if;
end GETT_PNT;


procedure IDEN_MAT is
-- This procedure creates the identity matrix
--    J is for stepping through the H array
begin
   for I in 1..3 loop
      for J in 1..2 loop
         if I=J then
            H(I)(J):=1.0;
         else
            H(I)(J):=0.0;
         end if;
      end loop;
   end loop;
end IDEN_MAT;


procedure IDEN_PAR is
-- This procedure was used to set transformation
-- parameters to the identity transformation
begin
   SCALE_X(0):=1.0;
   SCALE_Y(0):=1.0;
```

```
    ANGLE(0):=0.0;
    TRANS_X(0):=0.0;
    TRANS_Y(0):=0.0;
end IDEN_PAR;


procedure INCLUDE1(E_EDGE:in out integer;L_EDGE:integer;
                   SCAN:float) is
-- This procedure includes any new edges that intersect the
-- scanline
--    E_EDGE is the index of the last element of the
--       current side list
--    L_EDGE is the index of the last side
--    SCAN is the position of the current scanline
--    YMAX, XA and DX are arrays of edge information
--    SCAN_DEC is the size of a scanline decrement
begin
   while (E_EDGE<L_EDGE) and (YMAX(E_EDGE)>SCAN) loop
      -- Set starting point back to the last scanline
      XA(E_EDGE):=XA(E_EDGE)+(DXARRAY(E_EDGE)
                  *(SCAN_DEC+SCAN-YMAX(E_EDGE)));
      -- Save the change in x value per scan
      DXARRAY(E_EDGE):=DXARRAY(E_EDGE)*(-SCAN_DEC);
      E_EDGE:=E_EDGE+1;
   end loop;
end INCLUDE1;


procedure INIT is
-- This procedure is a combination of the procedures used
-- in the extended implementation to initialize all of the
-- necessary values for the graphics system
--    PFLAG is the polygon processing flag
--    XS, YS and ZS are arrays for saving the old endpoints
--       of a line segment
--    I is used for the initialization of the four clipping
--       routines
begin
   -- INIT6
   INITIAL;
   SET_VIEW(0.0,1.0,0.0,1.0);
   SET_WINDOW(0.0,1.0,0.0,1.0);
      FOR I IN 1..4 LOOP
      XS(I):=0.0;
      YS(I):=0.0;
   END LOOP;
   PFLAG:=FALSE;
```

```
    -- INIT7 omitted

    -- INIT9
    SET_VIEWDEP(0.0,0.0);
       -- INIT8
       SET_VIEWREFPT(0.0,0.0,0.0);
       SET_VIEWPLANOR(0.0,0.0,-1.0);
       SET_VIEWDIS(0.0);
       SET_VIEWUP(0.0,1.0,0.0);
       SET_PARA(0.0,0.0,1.0);
       NEW_VIEW3;

    SET_FRONCLIP(false);
    SET_BACCLIP(false);
    for I in 1..6 loop
       XS(I):=0.0;
       YS(I):=0.0;
       ZS(I):=0.0;
    end loop;
end INIT;


procedure INITIAL is
-- This procedure is a combination of the procedures used
-- in the initial implementation to initialize all of the
-- necessary values for the graphics system
--    W_START and H_START are the coordinates of the
--       lower-left corner
--    W_END and H_END are the coordinates of the upper-right
--       corner
--    HEIGHT is the height of the frame
--    WIDTH is the width of the frame
--    FREE is the index of the next free display file cell
--    DF_PEN_X and DF_PEN_Y are the coordinates of the
--       display file pen position
--    LINECHR is the line style used by the procedure
--       BRESENHAM
--    FILLER is the polygon fill style
--    VISIBLE is the segment visiblity table
--    NOW_OPEN is the currently open segment
--    SCALE_X, SCALE_Y, ANGLE, TRANS_X, and TRANS_Y are the
--       transformation parameters
--    SEG_START is the segment starting index array
--    SEG_SIZE is the segment size array
--    NUM_SEGS is the size of the segment table
--    I is used to step through the segment table
begin
```

```
    ENABLE;

    -- INIT1
    W_START:=106;
    H_START:=10;
    W_END:=533;
    H_END:=224;
    HEIGHT:=H_END-H_START;
    WIDTH:=W_END-W_START;

    -- INIT2A

    FREE:=1;
    DF_PEN_X:=0.0;
    DF_PEN_Y:=0.0;
    NEW_FRAM;

    -- INIT2
       DOSTYLE(-1);

    -- INIT3

       DOSTYLE(-31);

       S_FILL(false);

    -- INIT4
       IDEN_PAR;

    -- INIT5
       DLET_ALL;
       NEW_FRAM;
       VISIBLE(0):=true;
end INITIAL;


procedure INTRPRET(START,COUNT:integer) is
-- This procedure scans the display file performing the
-- instructions
--    START is the starting index of the display file scan
--    COUNT is the number of instructions to be interpreted
--    NTH is for stepping through the display file
--    NT is the the display file index
--    OP, X, and Y are the display file instructions
--    SOLID is a flag that indicates if the polgon should
--       be filled in
   NT,OP:integer;
```

```
   X,Y:float;
begin
   -- A loop to do all desired instructions
   for NTH in START..((START+COUNT)-1) loop
      NT:=NTH;
      GETT_PNT(NT,OP,X,Y);
      if OP<0 then
         DOSTYLE(OP);
      elsif OP=1 then
         DOMOVE(X,Y);
      elsif OP=2 then
         DOLINE(X,Y);
      elsif OP<32 then
         DOPOLY(OP,X,Y,NT);
      else
         put("OP-CODE ERROR");
         new_line;
         exit;
      end if;
   end loop;
end INTRPRET;


procedure LIN_ABS2(X,Y:float) is
-- This procedure saves a command to draw a line
--    X and Y are the coordinates of the point to which to
--       draw the line
--    DF_PEN_X and DF_PEN_Y are the coordinates of the
--       current pen position
begin
   DF_PEN_X:=X;
   DF_PEN_Y:=Y;
   DFE(2);
end LIN_ABS2;


procedure LIN_ABS3(X,Y,Z:float) is
-- This procedure is the 3D absolute line drawing routine
--    X, Y and Z are the  coordinates of the point to
--       draw the line to
--    DF_PEN_X, DF_PEN_Y and DF_PEN_Z are the coordinates
--       of the current pen position
begin
   DF_PEN_X:=X;
   DF_PEN_Y:=Y;
   DF_PEN_Z:=Z;
   DFE(2);
```

```
end LIN_ABS3;


procedure LIN_REL2(DX,DY:float) is
-- This procedure saves a command to draw a line
--    DX and DY are the changes over which draw the line
--    DF_PEN_X and DF_PEN_Y are the coordinates of the
--       current pen position
begin
   DF_PEN_X:=DF_PEN_X + DX;
   DF_PEN_Y:=DF_PEN_Y + DY;
   DFE(2);
end LIN_REL2;


procedure LIN_REL3(DX,DY,DZ:float) is
-- This procedure is the 3D relative line drawing routine
--    DX, DY and DZ are the displacements over which a line
--       is to be drawn
--    DF_PEN_X, DF_PEN_Y and DF_PEN_Z are the coordinates
--       of the current pen position
begin
   DF_PEN_X:=DF_PEN_X+DX;
   DF_PEN_Y:=DF_PEN_Y+DY;
   DF_PEN_Z:=DF_PEN_Z+DZ;
   DFE(2);
end LIN_REL3;


procedure LOADPOLY(OP:integer;X,Y:float;I:integer;
                   EDGES:in out integer) is
-- This procedure retrieves polygon side information from
-- the display file.  Positions are converted to actual
-- screen coordinates.
--    OP, X and Y are the polygon instruction
--    I is the display file index of the instruction
--    EDGES return the number of sides stored
--    W_START and H_START are the coordinates of the
--       lower-left corner
--    HEIGHT is the height in pixels of the actual screen
--    WIDTH is the width in pixels of the screen
--    X1, Y1, X2 and Y2 are the edge endpoints in actual
--       device coordinates
--    I1 is for stepping through the display file
--    K is for stepping through the polygon sides
--    DUMMY is a dummy argument
--    RNDOFF is a constant which is greater than any
```

```
--          round-off error
   X1,Y1,X2,Y2:float;
   I1,DUMMY:integer;
begin
   -- Set starting point for a side
   X1:=X*float(WIDTH)+float(W_START);
   -- Adjust y coordinate to lie between scanlines
   Y1:=float(INT(Y*float(HEIGHT)+float(H_START)))+0.5;
   -- Get index of first side command
   I1:=I+1;
   -- Inialize an index for storing side data
   EDGES:=1;
   -- A loop to get information about each side
   for K in 1..OP loop
      -- Get next vertex
      GETT_PNT(I1,DUMMY,X2,Y2);
      X2:=X2*float(WIDTH)+float(W_START);
      Y2:=float(INT(Y2*float(HEIGHT)+float(H_START)))+0.5;
      -- Check to see if the line is a horizontal line
      if abs(Y1-Y2)<RNDOFF then
         X1:=X2;
         I1:=I1+1;
      else
         -- Save data about side in order of largest y value
         POLYISRT(EDGES,X1,Y1,X2,Y2);
         -- Increment index for side data storage
         EDGES:=EDGES+1;
         -- Old point is reset
         Y1:=Y2;
         X1:=X2;
         I1:=I1+1;
      end if;
   end loop;
   -- Set EDGES to be a count of the edges stored
   EDGES:=EDGES-1;
end LOADPOLY;


procedure MOV_ABS2(X,Y:float) is
-- This procedure saves an instruction to move the pen
--    X and Y are the coordinates of the point to which
--       to move the pen
--    DF_PEN_X and DF_PEN_Y are the coordinates of the
--       current pen position
begin
   DF_PEN_X:=X;
   DF_PEN_Y:=Y;
```

```
   DFE(1);
end MOV_ABS2;


procedure MOV_ABS3(X,Y,Z:float) is
-- This procedure is the 3D absolute move
--    X, Y and Z are the world coordinates of the point to
--       move the pen
--    DF_PEN_X, DF_PEN_Y and DF_PEN_Z are the coordinates
--       of the current pen position
begin
   DF_PEN_X:=X;
   DF_PEN_Y:=Y;
   DF_PEN_Z:=Z;
   DFE(1);
end MOV_ABS3;


procedure MOV_REL2(DX,DY:float) is
-- This procedure saves a command to move the pen
--    DX and DY are the changes in the pen position
--    DF_PEN_X and DF_PEN_Y are the coordinates of the
--       current pen position
begin
   DF_PEN_X:=DF_PEN_X + DX;
   DF_PEN_Y:=DF_PEN_Y + DY;
   DFE(1);
end MOV_REL2;


procedure MOV_REL3(DX,DY,DZ:float) is
-- This procedure is the 3D relative move
--    DX, DY and DZ are the changes to be made to the pen
--       position
--    DF_PEN_X, DF_PEN_Y and DF_PEN_Z are the coordinates
--       of the current pen position
begin
   DF_PEN_X:=DF_PEN_X+DX;
   DF_PEN_Y:=DF_PEN_Y+DY;
   DF_PEN_Z:=DF_PEN_Z+DZ;
   DFE(1);
end MOV_REL3;


procedure MRK_ABS2(X,Y:float) is
begin
   MOV_ABS2(X,Y);
```

```
   LIN_REL2(0.0,0.0);
end MRK_ABS2;


procedure MRK_REL2(X,Y:float) is
begin
   MOV_REL2(X,Y);
   LIN_REL2(0.0,0.0);
end MRK_REL2;


procedure MULNROTA(A:float) is
-- This procedure multiplies the transformation matrix
-- by a rotation
--    A is the angle of counterclockwise rotation
--    C and S are the cosine and sine values
--    I is for stepping through the array
--    TEMP is for temporary storage of the first column  .b
--       of the transformation matrix
   TEMP,C,S:float;
begin
   C:=cos(A);
   S:=sin(A);
   for I in 1..3 loop
      TEMP:=H(I)(1)*C-H(I)(2)*S;
      H(I)(2):=H(I)(1)*S+H(I)(2)*C;
      H(II)(1):=TEMP;
   end loop;
end MULNROTA;


procedure MULNSCAL is
-- This procedure was used to multiply the transformation
-- matrix by a scale transformation
   for I in 1..3 loop
      H(I)(1):=H(I)(1)*SCALE_X(SEG_NAME);
      H(I)(2):=H(I)(2)*SCALE_Y(SEG_NAME);
   end loop;
end MULNSCAL;


procedure MULNTRAN(TX,TY:float) is
-- This procedure multiplies the transformation matrix by
-- a translation
--    TX is the translation in the x direction
--    TY is the translation in the y direction
begin
```

```
   H(3)(1):=H(3)(1)+TX;
   H(3)(2):=H(3)(2)+TY;
end MULNTRAN;


procedure M_CLIP_CONS is
-- This procedure calculates some of the 3D clipping
-- parameters
--    FRONT and BACK are the front and back plane distances
--       from the view reference point
--    V_DIS is the view plane distance from the view
--       reference point
--    FRO_Z and BAC_Z are the front and back plane positions
--       in view plane coordinates
--    PERS_FLAG is the type of projection flag
--    VXP, VYP and VZP are the parallel projection vector
--       coordinates
--    XC, YC and ZC are the perspective center of
--       projection coordinates
--    WXL, WXH, WYL, and WYH are the window boundary
--       specification
--    XLM, XHM, YLM and YHM are the window clipping plane
--       slopes
begin
   FRO_Z:=V_DIS-FRONT;
   BAC_Z:=V_DIS-BACK;
   if PERS_FLAG then
      XLM:=(XC-WXL)/ZC;
      XHM:=(XC-WXH)/ZC;
      YLM:=(YC-WYL)/ZC;
      YHM:=(YC-WYH)/ZC;
   else
      XLM:=VXP/VZP;
      XHM:=XLM;
      YLM:=VYP/VZP;
      YHM:=YLM;
          end if;
end M_CLIP_CONS;


procedure M_CLIP_TEST is
-- This procedure initializes test conditions for the
-- "old" endpoints
--    OPTTE is an array to hold the test conditions for the
--       four window clipping planes
--    XLM, XHM, YLM and YHM are the slopes of the clipping
--       planes
```

```
--    WXL, WXH, WYL and WYH are the window boundaries
--    ZS is an array containing the z coordinates of the
--       "old" endpoints
begin
   OPTTE(1):=XLM*ZS(1)+WXL;
   OPTTE(2):=XHM*ZS(2)+WXH;
   OPTTE(3):=YLM*ZS(3)+WYL;
   OPTTE(4):=YHM*ZS(4)+WYH;
end M_CLIP_TEST;


procedure M_PARA_TRANS is
-- This procedure calculates the direction of the projection
-- in view plane coordinates
--    TMAT is a 4 X 3 coordinate transformation matrix
--       array
--    DXP, DYP and DZP are the parallel projection vector
--       coordinates
--    VXP, VYP and VZP are view plane coordinates in the
--       direction of projection
--    SXP and SYP are the slopes of the projection relative
--       to the Z direction
--    RNDOFF is some small number greater than any round-off
--       error
begin
   VXP:=DXP*TMAT(1)(1)+DYP*TMAT(2)(1)+DZP*TMAT(3)(1);
   VYP:=DXP*TMAT(1)(2)+DYP*TMAT(2)(2)+DZP*TMAT(3)(2);
   VZP:=DXP*TMAT(1)(3)+DYP*TMAT(2)(3)+DZP*TMAT(3)(3);
   if abs(VZP)<RNDOFF then
      put("PROJECTION PARALLEL TO VIEW PLANE");
      new_line;
   end if;
   SXP:=VXP/VZP;
   SYP:=VYP/VZP;
end M_PARA_TRANS;


procedure M_PERS_TRANS is
-- This procedure converts the center of projection to view
-- plane coordinates
--    XPCNTR, YPCNTR and ZPCNTR are the center of projection
--       coordinates
--    XC, YC and ZC are the view plane coordinates for the
--       center of projection
begin
   XC:=XPCNTR;
   YC:=YPCNTR;
```

```
   ZC:=ZPCNTR;
   V_PLA_TRANS(XC,YC,ZC);
   if ZC<0.0 then
      put("CENTER OF PROJECTION IS BEHIND VIEW PLANE");
      new_line;
   end if;
end M_PERS_TRANS;

procedure M_PIC_CU is
-- This procedure shows the current display file
--    SEG_START, SEG_SIZE and VISIBLE together make up the
--       segment table
--    E_FLAG indicates whether the frames should be cleared
--    I is a variable used for stepping through the segment
--       table
--    NUM_SEGS is the size of the segment table
begin
   if E_FLAG then
      ERASE;
      E_FLAG:=false;
   end if;
   for I in 0..NUM_SEGS loop
      if (SEG_SIZE(I) /= 0) AND VISIBLE(I) then
         BUILDTRN(I);
         INTRPRET(SEG_START(I),SEG_SIZE(I));
      end if;
   end loop;
   DLET_SEG(0);
end M_PIC_CU;


procedure M_VIEW_PLA_TRANS is
-- This procedure makes the view-plane transformation
--    XR, YR and ZR are the view reference point
--       coordinates
--    DXN, DYN and DZN are the view plane normal
--        coordinates
--    DXUP, DYUP and DZUP are the view-up direction
--        coordinates
--    TMAT is a 4 X 3 transformation matrix array
--    PERS_FLAG is the perspective projection flag
--    V_DIS is the distance between the view reference
--       point and the view plane
--    V, XUP_VP, YUP_VP and RUP are storage variables for
--       partial results
--    RNDOFF is some small number greater than any
--       round-error
```

```
   V,XUP_VP,YUP_VP,RUP:float;
begin
   -- Start with the identity matrix
   N_TRANS3;
   -- Translate so that view plane center is new origin
   TRANSLAT3(-(XR+DXN*V_DIS),-(YR+DYN*V_DIS),
             -(ZR+DZN*V_DIS));
   -- Rotate so that view plane normal is z axis
   V:=sqrt(DYN**2+DZN**2);
   if V>RNDOFF then
      ROTATEX3(-DYN/V,-DZN/V);
   end if;
   ROTATEY3(DXN,V);
   -- Determine the view-up direction in these new
   -- coordinates
   XUP_VP:=DXUP*TMAT(1)(1)+DYUP*TMAT(2)(1)+DZUP*TMAT(3)(1);
   YUP_VP:=DXUP*TMAT(1)(2)+DYUP*TMAT(2)(2)+DZUP*TMAT(3)(2);
   -- Determine rotation needed to make view-up vertical
   RUP:=sqrt(XUP_VP**2+YUP_VP**2);
   if RUP<RNDOFF then
      put("SET-VIEW-UP ALONG VIEWPLANE NORMAL");
      new_line;
   end if;
   ROTATEZ3(XUP_VP/RUP,YUP_VP/RUP);
   if PERS_FLAG then
      M_PERS_TRANS;
   else
      M_PARA_TRANS;
   end if;
end M_VIEW_PLA_TRANS;


procedure NEW_FRAM is
-- This procedure was used to indicate when the frame
-- buffer should be cleared before showing the display
-- file
--    E_FLAG is a flag to indicate whether the frame
--       should be cleared
begin
   E_FLAG:=true;
end NEW_FRAM;


procedure NEW_VIEW2 is
-- This procedure was used to set the clipping and
-- viewing parameters from the current window and
-- viewport specifications
```

```
begin
   WXL:=WXL_HOLD;
   WYL:=WYL_HOLD;
   WXH:=WXH_HOLD;
   WYH:=WYH_HOLD;
   VXL:=VXL_HOLD;
   VYL:=VYL_HOLD;
   VXH:=VXH_HOLD;
   VYH:=VYH_HOLD;
   WSX:=(VXH-VXL)/(WXH-WXL);
   WSY:=(VYH-VYL)/(WYH-WYL);
end NEW_VIEW2;


procedure NEW_VIEW3 is
-- This procedure creates a new overall viewing
-- transformation
--    WXL_HOLD, WYL_HOLD, WXH_HOLD and WYH_HOLD are the
--       user's window parameters
--    VXL_HOLD, VYL_HOLD, VXH_HOLD and VYH_HOLD are the
--       user's viewport parameters
--    WXL, WYL, WXH, WYH, VXL, VYL, VXH and VYH are the
--       current clipping parameters
--    WSX and WSY are the window to viewport scale factors
begin
   NEW_VIEW2;
   M_VIEW_PLA_TRANS;
   M_CLIP_CONS;
   M_CLIP_TEST;
end NEW_VIEW3;


procedure N_TRANS3 is
-- This procedure initializes the viewing transformation
-- matrix to identity
--    TMAT is a 4 X 3 transformation matrix array
--    I and J are for stepping through the array elements
begin
   for I in 1..4 loop
      for J in 1..3 loop
         TMAT(I)(J):=0.0;
         if I/=4 then
            TMAT(I)(I):=1.0;
         end if;
      end loop;
   end loop;
end N_TRANS3;
```

```
procedure PAR_TRANS(X,Y,Z:in out float) is
-- This procedure performs the parallel projection of a
-- point
--    X, Y and Z are the coordinates of the point to be
--       projected
--    SXP and SYP are the parallel projection vector
--       ratios
begin
   X:=X-Z*SXP;
   Y:=Y-Z*SYP;
end PAR_TRANS;


procedure PERS_TRANS(X,Y,Z:in out float) is
-- This procedure performs the perspective projection of a
-- point
--    X, Y and Z are the view plane coordinates of the
--       point
--    XC, YC and ZC are the coordinates of the center of
--       projection
--    RNDOFF is some small number greater than any
--       round-off error
--    V_LAR is a very large number approximating infinity
begin
   if abs(ZC-Z)<RNDOFF then
      X:=(X-XC)*V_LAR;
      Y:=(Y-YC)*V_LAR;
   else
      X:=(X*ZC-XC*Z)/(ZC-Z);
      Y:=(Y*ZC-YC*Z)/(ZC-Z);
   end if;
end PERS_TRANS;


procedure POLYISRT(J:integer;X1,Y1,X2,Y2:float) is
-- This procedure performs the ordered insertion of polygon
-- edge information
--    J is the insertion index
--    X1, Y1, X2 and Y2 are the endpoints of the polygon
--       side
--    YMAX, YMIN, XA and DX are arrays that store polygon
--       edge information
--    J1 is for stepping through the stored edges
--    YM is the maximum y value of the new edge
   J1:integer;
   YM:float;
begin
```

```
    -- Insertion sort into global arrays on maximum y value
    J1:=J;
    -- Find the largest y value
    if Y1>=Y2 then
       YM:=Y1;
    else
       YM:=Y2;
    end if;
    -- Find the correct insertion point, moving items out of
    -- the way
    while (J1/=1) and then ((YMAX(J1-1))<YM) loop
       YMAX(J1):=YMAX(J1-1);
       YMIN(J1):=YMIN(J1-1);
       XA(J1):=XA(J1-1);
       DXARRAY(J1):=DXARRAY(J1-1);
       J1:=J1-1;
    end loop;
    -- Insert information about side
    YMAX(J1):=YM;
    DXARRAY(J1):=(X2-X1)/(Y2-Y1);
    if Y1>Y2 then
       YMIN(J1):=Y2;
       XA(J1):=X1;
    else
       YMIN(J1):=Y1;
       XA(J1):=X2;
    end if;
end POLYISRT;


procedure POL_ABS2(AX,AY:POLARRAY;N:integer) is
-- This procedure is used to enter an absolute polygon into
-- the display file
--    N is the number of polygon sides
--    AX and AY are arrays containing the vertices of the
--       polygon
--    DF_PEN_X and DF_PEN_Y are the coordinates of the
--       current pen position
--    I is used for stepping through the polygon sides
begin
   if (N>31) or (N<3) then
      put("POLYGON SIZE ERROR!!!!!");
      new_line;
   else
      -- Enter the polygon instruction
      DF_PEN_X:=AX(N);
      DF_PEN_Y:=AY(N);
```

```
      DFE(N);
      -- Enter the instructions for the sides
      for I in 1..N loop
         LIN_ABS2(AX(I),AY(I));
      end loop;
   end if;
end POL_ABS2;


procedure POL_ABS3(AX,AY,AZ:POLARRAY;N:integer) is
-- This procedure is used for 3D absolute polygon drawing
--    N is the number of polygon sides
--    AX, AY and AZ are arrays containing the coordinates
--    DF_PEN_X, DF_PEN_Y and DF_PEN_Z are the coordinates
--       of the current pen position
--    I is used for stepping through the polygon sides
begin
   if (N>31) or (N<3) then
      put("POLYGON SIZE ERROR!!!!!");
      new_line;
   else
      DF_PEN_X:=AX(N);
      DF_PEN_Y:=AY(N);
      DF_PEN_Z:=AZ(N);
      DFE(N);
      for I in 1..N loop
         LIN_ABS3(AX(I),AY(I),AZ(I));
      end loop;
   end if;
end POL_ABS3;


procedure POL_CLIP(OP:integer;X,Y,Z:float) is
-- This procedure is the polygon clipping routine
--    OP, X, Y and Z are a display file instruction
--    PFLAG indicates that a polygon is being drawn
--    COUNT_IN is the number of sides remaining to be
--       processed
--    COUNT_OUT is the number of sides to be entered in the
--       display file
--    IT, XT, YT and ZT are the temporary storage cells for
a
--       polygon
--    I is use for stepping through the polygon sides
begin
   COUNT_IN:=COUNT_IN-1;
   CLIP_LEFT(OP,X,Y,Z);
```

```
   if COUNT_IN=0 then
      -- Close the clipped polygon
      if COUNT_OUT>0 then
         CLIP_LEFT(2,XT(1),YT(1),ZT(1));
      end if;
      PFLAG:=false;
      -- Remove the extra side
      COUNT_OUT:=COUNT_OUT-1;
      if COUNT_OUT>3 then
         if COUNT_OUT<32 then
            -- Enter the polygon into the display file
            DOPROJ(COUNT_OUT,XT(COUNT_OUT),YT(COUNT_OUT),
                   ZT(COUNT_OUT));
            for I in 1..COUNT_OUT loop
               DOPROJ(IT(I),XT(I),YT(I),ZT(I));
            end loop;
         else
            put("CLIPPED POLYGON TOO BIG");
            new_line;
         end if;
      end if;
   end if;
end POL_CLIP;


procedure POL_REL2(AX,AY:POLARRAY;N:integer) is
-- This procedure is used for entering a relative polygon
-- into the display file
--    N is the number of polygon sides
--    AX and AY are arrays containing the vertices of the
--       polygon
--    DF_PEN_X and DF_PEN_Y are the coordinates of the
--       current pen position
--    I is used for stepping through the polygon sides
--    TMPX and TMPY are used to store the coordinates of
--       the point at which the polygon is closed
   TMPX,TMPY:float;
begin
   if (N>31) or (N<3) then
      put("POLYGON SIZE ERROR!!!!!");
      new_line;
   else
      DF_PEN_X:=DF_PEN_X+AX(1);
      DF_PEN_Y:=DF_PEN_Y+AY(1);
      -- save the starting point for closing the polygon
      TMPX:=DF_PEN_X;
      TMPY:=DF_PEN_Y;
```

```
      -- enter the polygon instruction
      DFE(N);
      -- enter the instructions for the sides
      for I in 2..N loop
         -- close the polygon
         LIN_REL2(AX(I),AY(I));
      end loop;
      LIN_ABS2(TMPX,TMPY);
   end if;
end POL_REL2;


procedure POL_REL3(AX,AY,AZ:POLARRAY;N:integer) is
-- This procedure is used for 3D relative polygon drawing
--    N is the number of polygon sides
--    AX, AY and AZ are arrays containing the displacements
--       for the polygon sides
--    DF_PEN_X, DF_PEN_Y and DF_PEN_Z are the
--       coordinates of the current pen position
--    I is used for stepping through the polygon sides
--    TMPX, TMPY and TMPZ are storage locations for the
--       point at which the polygon is closed
   TMPX,TMPY,TMPZ:float;
begin
   if (N>31) or (N<3) then
      put("POLYGON SIZE ERROR!!!!!");
      new_line;
  else.b       -- Move to starting vertex
      DF_PEN_X:=DF_PEN_X+AX(1);
      DF_PEN_Y:=DF_PEN_Y+AY(1);
      DF_PEN_Z:=DF_PEN_Z+AZ(1);
      -- Save vertex for closing the polygon
      TMPX:=DF_PEN_X;
      TMPY:=DF_PEN_Y;
      TMPZ:=DF_PEN_Z;
      DFE(N);
      -- ENTER THE POLYGON SIDES
      for I in 2..N loop
         LIN_REL3(AX(I),AY(I),AZ(I));
      end loop;
      -- Close the polyon
      LIN_ABS3(TMPX,TMPY,TMPZ);
   end if;
end POL_REL3;


procedure PUT_IN_T(OP:integer;X,Y,Z:float;
```

```
                    INDEX:integer) is
-- This procedure places the a polygon edge instruction in a
-- temporary storage buffer
--    OP, X, Y and Z are the instruction to be stored
--    INDEX is the position at which to store the
--       instruction
--    IT, XT, YT and ZT are the temporary storage arrays for
--       the polygon sides
begin
   IT(INDEX):=OP;
   XT(INDEX):=X;
   YT(INDEX):=Y;
   ZT(INDEX):=Z;
end PUT_IN_T;


procedure PUT_PNT(OP:integer;X,Y:float) is
-- This procedure places a full instruction into the display
-- file and updates the segment table
--    OP, X and Y form the instruction to be entered
--    NOW_OPEN is the segment currently open
--    SEG_SIZE is the segment size array
--    FREE is the position of the next free display file
cell
begin
   SEG_SIZE(NOW_OPEN):=SEG_SIZE(NOW_OPEN) + 1;
   SET_PNT(FREE,OP,X,Y);
   FREE:=FREE + 1;
end PUT_PNT;


procedure RENA_SEG(SEG_OLD,SEG_NEW:integer) is
-- This procedure renames a segment
--    SEG_OLD is the old name of the segment
--    SEG_NEW is the new name of the segment
--    SEG_START, SEG_SIZE, VISIBLE, ANGLE, SCALE_X, SCALE_Y,
--    TRAN_X and TRAN_Y together make up the segment table
--    NUM_SEGS is the size of the segment table
--    NOW_OPEN is the segment currently open
begin
   if (SEG_OLD<1) or (SEG_NEW<1) or (SEG_OLD>NUM_SEGS)
      or (SEG_NEW>NUM_SEGS) then
      put("INVALID SEGMENT NAME");
      new_line;
   elsif (SEG_OLD = NOW_OPEN) or (SEG_NEW = NOW_OPEN) then
      put("SEGMENT STILL OPEN");
      new_line;
```

```
   elsif SEG_SIZE(SEG_NEW) /= 0 then
      put("SEGMENT ALREADY EXIST");
      new_line;
   else
      -- Copy the old segment table entry into the new
      -- position
      SEG_START(SEG_NEW):=SEG_START(SEG_OLD);
      SEG_SIZE(SEG_NEW):=SEG_SIZE(SEG_OLD);
      VISIBLE(SEG_NEW):=VISIBLE(SEG_OLD);
      ANGLE(SEG_NEW):=ANGLE(SEG_OLD);
      SCALE_X(SEG_NEW):=SCALE_X(SEG_OLD);
      SCALE_Y(SEG_NEW):=SCALE_Y(SEG_OLD);
      TRANS_X(SEG_NEW):=TRANS_X(SEG_OLD);
      TRANS_Y(SEG_NEW):=TRANS_Y(SEG_OLD);
      -- Delete the old segment
      SEG_SIZE(SEG_OLD):=0;
   end if;
end RENA_SEG;


procedure ROTATE(A:float) is
-- This procedure sets the image rotation
--    A is the angle of rotation
--    ANGLE is the segment angle parameter table
begin
   ANGLE(0):=A;
   NEW_FRAM;
end ROTATE;


procedure ROTATEX3(S,C:float) is
-- This procedure calculates the rotation about the x axis
-- (y into z)
--    S and C are the sine and cosine of the rotation angle
--    TMAT is a 4 X 3 transformation matrix array
--    I is for stepping through the matrix elements
--    TMP is a temporary storage
   TMP:float;
begin
   for I in 1..4 loop
      TMP:=TMAT(I)(2)*C-TMAT(I)(3)*S;
      TMAT(I)(3):=TMAT(I)(2)*S+TMAT(I)(3)*C;
      TMAT(I)(2):=TMP;
   end loop;
end ROTATEX3;
```

```
procedure ROTATEY3(S,C:float) is
-- This procedure calculates the rotation about the y axis
-- (z into x)
--    S and C are the sine and cosine of the rotation angle
--    TMAT is a 4 X 3 transformation matrix array
--    I is for stepping through the matrix elements
--    TMP is a temporary storage
   TMP:float;
begin
   for I in 1..4 loop
      TMP:=TMAT(I)(1)*C+TMAT(I)(3)*S;
      TMAT(I)(3):=TMAT(I)(1)*S+TMAT(I)(3)*C;
      TMAT(I)(1):=TMP;
   end loop;
end ROTATEY3;


procedure ROTATEZ3(S,C:float) is
-- This procedure calculates the rotation about the z axis
-- (x into y)
--    S and C are the sine and cosine of the rotation angle
--    TMAT is a 4 X 3 transformation matrix array
--    I is for stepping through the matrix elements
--    TMP is a temporary storage
   TMP:float;
begin
   for I in 1..4 loop
      TMP:=TMAT(I)(1)*C-TMAT(I)(2)*S;
      TMAT(I)(2):=TMAT(I)(1)*S+TMAT(I)(2)*C;
      TMAT(I)(1):=TMP;
   end loop;
end ROTATEZ3;


procedure SCALES(SX,SY:float) is
-- This procedure sets the image scaling transformation
--    SX and SY are the scaling parameters
--    SCALE_X and SCALE_Y are the segment scaling parameter
--       tables
begin
   SCALE_X(0):=SX;
   SCALE_Y(0):=SY;
   NEW_FRAM;
end SCALES;


procedure SET_BACCLIP(ON_OFF:boolean) is
```

```
-- This is a user routine to set the back clipping flag
--    ON_OFF is the user's clipping flag setting
--    B_FLAG is the back clipping flag
begin
   B_FLAG:=ON_OFF;
end SET_BACCLIP;


procedure SET_FRONCLIP(ON_OFF:boolean) is
-- This is a user routine to set the front clipping flag
--    ON_OFF is the user's clipping flag setting
--    F_FLAG is the front clipping flag
begin
   F_FLAG:=ON_OFF;
end SET_FRONCLIP;


procedure SET_PARA(DX,DY,DZ:float) is
-- This procedure is for user input of the direction of
-- parallel projection
--    DX, DY and DZ are the new parallel projection vector
--       coordinates
--    PERS_FLAG is the perspective vs. parallel projection
--       flag
--    DXP, DYP and DZP are the permanent storage variables
--       for the direction of projection
--    RNDOFF is some small number greater than any
--       round-off error
begin
   if (abs(DX)+abs(DY)+abs(DZ)) < RNDOFF then
      put("NO DIRECTION OF PROJECTION");
      new_line;
   else
      PERS_FLAG:=false;
      DXP:=DX;
      DYP:=DY;
      DZP:=DZ;
   end if;
end SET_PARA;


procedure SET_PERS(X,Y,Z:float) is
-- This procedure indicates a perspective projection and
-- saves the center of projection
--    X, Y, and Z are the new center of projection
--       coordinates
--    XPCNTR, YPCTNTR, and ZPCNTR are the permanent storage
```

```
--       variables for the center of projection
--    PERS_FLAG is the perspective vs. parallel projection
--       flag
begin
   PERS_FLAG:=true;
   XPCNTR:=X;
   YPCNTR:=Y;
   ZPCNTR:=Z;
end SET_PERS;


procedure SET_PIX(X,Y,INTENSE:in integer) is
    X1,X2,Y1,Y2:integer;
    VIDEO_SEG,VIDEO_OFFSET:integer;
    M,DATA:byte;
begin -- SET_PIX
  X1:=((X / 8));
  Y1:=(((224-y) / 9));
  X2:=((X mod 8));
  Y2:=(((224-Y) mod 9));
  VIDEO_OFFSET:=-32766+(2048)*Y1+(128)*Y2+X1;
  M:=byte(2**(7-X2));
  case INTENSE is
      when   0 => VIDEO_SEG:=16C000;
      when   1 => VIDEO_SEG:=16D000;
      when   2 => VIDEO_SEG:=16E000;
  end case;
  DATA:=peek(VIDEO_SEG,VIDEO_OFFSET);
  DATA:=byte(LOR(INTEGER(DATA),INTEGER(M)));
  poke(VIDEO_SEG,VIDEO_OFFSET,DATA);
end SET_PIX;


procedure SET_PNT(INDEX,OP:integer;X,Y:float) is
-- This procedure replaces an instruction in the display
-- file
--    OP, X and Y are the replacement instructions
--    INDEX is the position of the instruction to be changed
--    DF_OP, DF_X and DF_Y are the arrays which together
--       make up the display file
--    DF_SIZE is the length of the display file
begin
   if INDEX > DFSIZE then
      put("DISPLAY FILE OVERFLOW");
      new_line;
   else
      DF_OP(INDEX):=OP;
```

```
      DF_X(INDEX):=X;
      DF_Y(INDEX):=Y;
   end if;
end SET_PNT;


procedure SET_VIEW(XL,XH,YL,YH:float) is
-- This procedure is used for specifying the viewport
-- boundary
--    XL and XH are the left and right viewport boundaries
--    YL and YH are the bottom and top viewport boundaries
--    VXL_HOLD, VXH_HOLD, VYL_HOLD and VYH_HOLD are the
--       storage variables for the viewport boundaries
begin
   if XL>XH or YL>YH then
      put("BAD VIEWPORT");
      new_line;
   else
      VXL_HOLD:=XL;
      VXH_HOLD:=XH;
      VYL_HOLD:=YL;
      VYH_HOLD:=YH;
   end if;
end SET_VIEW;


procedure SET_VIEWDEP(F_DIS,B_DIS:float) is
-- This procedure is a user routine to specify the position
-- of the front  and back clipping planes
--    F_DIS and B_DIS are the coordinates of the plane
--       distance from the view reference point along the
--       view plane normal
--    FRONT and BACK are the storage variables for the plane
--       position
begin
   if F_DIS>B_DIS then
      put("FRONT PLANE BEHIND THE BACK PLANE");
      new_line;
   else
      FRONT:=F_DIS;
      BACK:=B_DIS;
   end if;
end SET_VIEWDEP;


procedure SET_VIEWDIS(D:float) is
-- This procdure is used for changing the distance between
```

```
-- the view reference point and the view plane
--    V_DIS is the permanent storage variable for the view
--       distance
begin
   V_DIS:=D;
end SET_VIEWDIS;


procedure SET_VIEWPLANOR(DX,DY,DZ:float) is
-- This procedure is used for changing the view plane
-- normal
--    DX, DY and DZ are the new view plane normal vector
--       coordinates
--    D is the length of the user's specification vector
--    RNDOFF is some small number greater than any
--       round-off error
   D:float;
begin
   D:=sqrt(DX**2+DY**2+DZ**2);
   if D<RNDOFF then
      put("INVALID VIEW PLANE NORMAL");
      new_line;
   end if;
   DXN:=DX/D;
   DYN:=DY/D;
   DZN:=DZ/D;
end SET_VIEWPLANOR;


procedure SET_VIEWREFPT(X,Y,Z:float) is
-- This procedure is used for changing the view reference
-- point
--    X, Y and Z are the new view reference point
--       coordinates
--    XR, YR and ZR are permanent storage for the
--       reference point
begin
   XR:=X;
   YR:=Y;
   ZR:=Z;
end SET_VIEWREFPT;


procedure SET_VIEWUP(DX,DY,DZ:float) is
-- This procedure is used for changing the direction that
-- will be vertical on the image
--    DXUP, DYUP and DZUP are the permanent storage
```

```
--       variables for the view-up direction
--     RNDOFF is some small number greater than any
--       round-off error
begin
   if (abs(DX)+abs(DY)+abs(DZ)) < RNDOFF then
      put("NO SET-VIEW-UP DIRECTION");
      new_line;
   else
      DXUP:=DX;
      DYUP:=DY;
      DZUP:=DZ;
   end if;
end SET_VIEWUP;


procedure SET_VIS(SEG_NAME:integer;ON_OFF:boolean) is
-- This procedure is used to set the visibility attribute
--     SEG_NAME is the name of the segment
--     ON_OFF is the new visibility setting
--     VISIBLE is an array of visibility flags
--     NUM_SEGS is the size of the segment table
begin
  if (SEG_NAME < 1) or (SEG_NAME > NUM_SEGS) then
     put("INVALID SEGMENT NAME");
     new_line;
  else
     VISIBLE(SEG_NAME):=ON_OFF;
     if not ON_OFF then
        NEW_FRAM;
     end if;
   end if;
end SET_VIS;


procedure SET_WINDOW(XL,XH,YL,YH:float) is
-- This procedure is used for specifying the window
-- boundary
--     XL and XH are the left and right window boundaries
--     YL and YH are the bottom and top window boundaries
--     WXL_HOLD, WXH_HOLD, WYL_HOLD and WYH_HOLD are the
--       storage variables for the window boundaries
begin
   if XL>XH or YL>YH then
      put("BAD WINDOW");
      new_line;
   else
      WXL_HOLD:=XL;
```

```
      WXH_HOLD:=XH;
      WYL_HOLD:=YL;
      WYH_HOLD:=YH;
   end if;
end SET_WINDOW;


procedure S_CLIP_PT(OP:integer;X,Y,Z:float) is
-- This procedure saves a clipped polygon instruction
--    OP, X, Y and Z are a set of 3D drawing instructions
--    COUNT_OUT is a counter of the number of sides on
--       the clipped polygon
--    PFLAG indicates if a polygon is to be clipped
begin
   if PFLAG then
      COUNT_OUT:=COUNT_OUT+1;
      if COUNT_OUT<33 then
        PUT_IN_T(OP,X,Y,Z,COUNT_OUT);
      end if;
   else
      DOPROJ(OP,X,Y,Z);
   end if;
end S_CLIP_PT;


procedure S_FILL(ON_OFF: boolean) is
-- This procedure is used to set a flag indicating when a
-- polygon is to be filled
--    ON_OFF is the flag used for the fill setting
--    SOLID is the flag which indicates the filling of
--       polygons
begin
   SOLID:= ON_OFF;
end S_FILL;


procedure S_FILSTY(STYLE:integer) is
-- This procedure is used to set the polygon interior style
--    STYLE is the user's style request
begin
   DFE(-(30+STYLE));
end S_FILSTY;


procedure S_LINSTY(LSTYLE:integer) is
-- This procedure is used for changing line style
--    LSTYLE is the line-style specification
```

```
begin
   DFE(-LSTYLE);
end S_LINSTY;


procedure S_TRANSF(SEG_NAME:integer;SX,SY,A,TX,TY:float) is
-- This procedure is used to set the image transformation
-- parameters of a segment
--     SEG_NAME is the segment being transformed
--     SX, SY, A, TX and TY are the new image transformation
--        parameters
--     VISIBLE, SCALE_X, SCALE_Y, ANGLE, TRAN_X and TRAN_Y
--        are arrays for the attribute part of the segment
--        table
--     NUM_SEGS is the size of the segment table
begin
   if (SEG_NAME < 1) or (SEG_NAME > NUM_SEGS) then
      put("INVALID SEGMENT NAME");
      new_line;
   else
      SCALE_X(SEG_NAME):=SX;
      SCALE_Y(SEG_NAME):=SY;
      ANGLE(SEG_NAME):=A;
      TRANS_X(SEG_NAME):=TX;
      TRANS_Y(SEG_NAME):=TY;
      if VISIBLE(SEG_NAME) then
         NEW_FRAM;
      end if;
   end if;
end S_TRANSF;


procedure S_TRANSL(SEG_NAME:integer;TX,TY:float) is
-- This procedure is used to set the image translation for
-- an segment
--     SEG_NAME is the segment being transformed
--     TX and TY are the transformation parameters
--     TRANS_X and TRANS_Y are arrays containing the segment
--        translation parameter table
--     NUM_SEGS is the size of the segment table
begin
   if (SEG_NAME < 1) or (SEG_NAME > NUM_SEGS) then
      put("INVALID SEGMENT NAME");
      new_line;
   else
      TRANS_X(SEG_NAME):=TX;
      TRANS_Y(SEG_NAME):=TY;
```

```
      if VISIBLE(SEG_NAME) then
         NEW_FRAM;
      end if;
   end if;
end S_TRANSL;


procedure TRANSLAT(TX,TY:float) is
-- This procedure set the translation parameters for the
-- unnamed segment
--    TX and TY are the translation specfication
--    TRANS_X and TRANS_Y are part of the segment table for
--       translation
begin
   TRANS_X(0):=TX;
   TRANS_Y(0):=TY;
   NEW_FRAM;
end TRANSLAT;


procedure TRANSLAT3(TX,TY,TZ:float) is
-- This procedure multiplies the viewing transformation
-- matrix by a translation
--    TX, TY and TZ are the amount of the translation
--    TMAT is a 4 X 3 transformation matrix array
begin
   TMAT(4)(1):=TMAT(4)(1)+TX;
   TMAT(4)(2):=TMAT(4)(2)+TY;
   TMAT(4)(3):=TMAT(4)(3)+TZ;
end TRANSLAT3;


procedure UPDXVAL(E_EDGE:integer;S_EDGE:in out integer;
                  SCAN:float) is
-- This procedure updates points of intersection between
-- edges and the scanline
--    S_EDGE and E_EDGE are the limits of the current edge
--       list
--    SCAN is the current scanline
--    YMIN, XA and DX are arrays of edge information
--    B_EDGE and STP_EDGE are the limits on edges that are
--       considered for updating
--    K and L are for stepping through the edges
--    I is for stepping through the edges that are to
--       be shifted up
   B_EDGE,STP_EDGE:integer;
   L,K,I:integer;
```

```
begin
   STP_EDGE:=E_EDGE-1;
   B_EDGE:=S_EDGE;
   for K in B_EDGE..STP_EDGE loop
      if YMIN(K)<SCAN then
         XA(K):=XA(K)+DXARRAY(K);
         XSORT(B_EDGE,K);
      else
         S_EDGE:=S_EDGE+1;
         if S_EDGE<K then
            for I in K..S_EDGE loop
               YMIN(I):=YMIN(I-1);
               XA(I):=XA(I-1);
               DXARRAY(I):=DXARRAY(I-1);
            end loop;
         end if;
      end if;
   end loop;
end UPDXVAL;


procedure VIEW_TRANS(OP:integer;X,Y:float) is
-- This procedure calculates the viewing transformation of
-- a point
--    OP, X and Y are the intructions to be transformed
--    WXL, WYL, WSX, WSY, VXL and VYL are the window and
--       viewport parameters
--    X1 and Y1 are the transformed point
X1,Y1:float;
begin
   X1:=(X-WXL)*WSX+VXL;
   Y1:=(Y-WYL)*WSY+VYL;
   PUT_PNT(OP,X1,Y1);
end VIEW_TRANS;


procedure V_PLA_TRANS(X,Y,Z:in out float) is
-- This procedure transforms a point into the view plane
-- coordinate system
--    X, Y and Z are the coordinates of the point to be
--       transformed
--    TMAT is a 4 X 3 transformation matrix array
--    T is a three-element array to hold results until
--       calculations are finished
--    I is for stepping through the TMAT columns
   T:array (1..3) of float;
begin
```

```
   for I in 1..3 loop
      T(I):=X*TMAT(1)(I)+Y*TMAT(2)(I)+Z*TMAT(3)(I)
            +TMAT(4)(I);
   end loop;
   X:=T(1);
   Y:=T(2);
   Z:=T(3);
end V_PLA_TRANS;


procedure XCHANGE(A,B:in out float) is
-- This procedure is used to exchange two elements
--    A and B are the two elements to be exchanged
--    TEMP is a temporary storage variable
   TEMP:float;
begin
   TEMP:=B;
   B:=A;
   A:=TEMP;
end XCHANGE;


procedure XSORT(S_EDGE,K) is
-- This procedure was used to check the order of
-- the x value intersection
   L:=K;
begin
   while (L>S_EDGE) and then (XA(L)<XA(L-1)) loop
      XCHANGE(YMIN(L),YMIN(L-1));
      XCHANGE(XA(L),XA(L-1));
      XCHANGE(DXARRAY(L),DXARRAY(L-1));
      L:=L-1;
   end loop;
end XSORT;
```