NASA Technical Memorandum 100974
ICOMP–88–13

User's Manual for the Two-Dimensional Transputer Graphics Toolkit

Graham K. Ellis
Institute for Computational Mechanics in Propulsion
Lewis Research Center
Cleveland, Ohio

August 1988

NASA

TABLE OF CONTENTS

USER'S MANUAL FOR THE TWO-DIMENSIONAL TRANSPUTER GRAPHICS TOOLKIT

Graham K. Ellis*
Institute for Computational Mechanics in Propulsion
National Aeronautics and Space Administration
Lewis Research Center
Cleveland, Ohio 44135

SUMMARY

The user manual for a two-dimensional graphics toolkit for a transputer based parallel processor is presented. The toolkit consists of a package of two-dimensional display routines that can be used for simulation visualizations. It supports multiple windows, double buffered screens for animations, and simple graphics transformations such as translation, rotation, and scaling. The display routines are written in occam to take advantage of the multiprocessing features available on transputers. The package is designed to run on a transputer separate from the graphics board.

INTRODUCTION

A description of the operation and features of the two-dimensional transputer graphics toolkit (TGT) is presented in this manual. An introduction to basic graphics terminology is given along with some examples using the TGT. A technical reference is provided that fully details the parameters required and procedure calls used by the TGT. Appendix A shows the computations used internally by TGT to generate graphics transformations. Appendix B contains the development of two-dimensional windowing and clipping algorithms used by the toolkit and, Appendix C contains a listing of the TGT routines in occam.

COMPUTER GRAPHICS BASICS

This section provides an introduction to some of the basic graphics and screen display terminology used in this manual.

Pixel

Pixel is an acronym for **picture element**. The pixels are the tiny dots on the display screen used in combinations to generate text and graphics. Each pixel can have 2 or more colors. Monochrome text or graphics has 2 colors, black and white (or black and green, etc.). Monochrome pixels require only 1 bit of storage for each pixel. Color pixels typically have anywhere from 4 bits to 32 bits of storage for each pixel. Four bits per pixel allows a pixel to be one of 16 unique colors. Thirty-two bits per pixel allows a pixel to be one of more than 4 billion colors. The number of bits per pixel is typically referred to as the number of bit-planes. An example of a four bit-plane display is shown in figure 1.

---

*Senior Research Associate (work funded under Space Act Agreement C99066G).

The transputer graphics board used in the NASA Lewis Research Center's Transputer Laboratory is the INMOS B007 graphics board (ref. 1). This board has 8 bits per pixel for a maximum of 256 different colors. A special technique is used on the B007 to increase the number of colors to choose from to a number greater than 256. This technique uses a color look-up table (LUT).

Color Look-up Table

A color look-up table is a method of increasing the number of possible shades of colors to choose from over the amount normally limited by the number of bits per pixel. The LUT uses one level of indirection to increase the number of shades available to the display monitor. Instead of each pixel representing a color, the pixel value is interpreted as containing an address of a color to send to the display screen. A diagram of the look-up table operation is shown in figure 2.

On the B007, the 8-bit pixel value points to one register out of 256 that contains the color to be displayed at that pixel. The color register contains a 18-bit value that specifies the color to be displayed. Each primary color for the monitor, red, green, and blue is allocated 6 bits. Thus, there are 262, 144 different shades available, but only 256 of those shades can be displayed at one time. The advantage in doing this is there is a decrease in the number bytes required to store a screen image. The B007 is a 512 by 512 pixel display medium performance graphics board. With a single byte per pixel, each screen requires 256K bytes for storage. If the full 18 bits were allocated for each pixel, the storage required for each screen would increase to 768K bytes.

Coordinate Systems

The TGT uses three different coordinate systems to generate graphics. The applications programmer uses only two of these systems. The three coordinate systems used are described below:

Integer Device Coordinate System (Absolute Screen Coordinate System). - This coordinate system is used internally by the TGT. Normally, the programmer does not call any routines using the Integer Device Coordinate System (IDC). IDC refers to the coordinate system used by the low-level B007 graphics board driver routines. The IDC origin is the upper left corner of the display monitor and has the (x, y) coordinate (0, 0). The x-coordinate increases to the right, and the y-coordinate increases downward. The B007 maximum x and y coordinate values are both 511. Using IDC would make any graphics rendering task difficult since most applications do not use integer coordinates between 0 and 511. All coordinates specified in IDC must be integers.

World Coordinate System. - The World Coordinate System (WCS) is a two-dimensional real coordinate space. The WCS is used by the programmer to describe the model of interest. The model is specified using whatever 2D coordinates are appropriate. WCS is a cartesian system and unlike IDC, the x-coordinate increases to the right and the y-coordinate increases upward.

The WCS origin is at (0.0, 0.0); however, the WCS can correspond with any position on the display screen. In order to relate the WCS to the IDC, a mapping from the world coordinates to integer coordinates must be specified using the normalized device coordinate system discussed below.

Normalized Device Coordinates. - The Normalized Device Coordinate (NDC) system is used internally by the TGT and by the programmer to specify the mapping of the WCS onto the display screen. The NDC origin is in the lower left corner and is at the (x, y) coordinates (0.0, 0.0). The maximum x and y screen value is 1.0 regardless of the actual number of x and y direction screen pixels. No assumption is made that the number of x and y pixels is equal. Note that like the WCS, NDC uses real coordinates.

World coordinate windows (see the description of windows and viewports below) are mapped onto the screen using NDC. The WCS window expressed in NDC is called a viewport. Internally in TGT, the viewport is mapped into an IDC window and is ready to be sent to the B007 for display. The mapping from the world coordinate system to normalized device coordinates to integer device coordinates is shown in figure 3.

The NDC coordinate system removes the actual display resolution from the user procedure calls. If, at some time in the future, a higher resolution display driver was added, TGT could be edited and users could recompile their programs and the screen output would be exactly the same except at a higher resolution. The window size is not a function of the number of pixels, it is a function of NDC which removes the screen resolution from any TGT user procedure calls.

Screens

The term screen in this document relates to the area of memory on the B007 that is displayed on the graphics monitor. The B007 has enough memory allocated for two screens. Only one screen can be displayed at a time. For a full explanation of the B007 features, see the B007 User Manual (ref. 1). Two screens are used for a technique called double-buffering. This technique is normally used for animations. One screen is displayed while the hidden screen is modified for the next display frame. The screens are "swapped": the nondisplayed screen is displayed and the currently displayed screen becomes hidden and can be updated for redisplay without the drawing modifications being seen. This is how smooth animations are performed.

Windows and Viewports

A window on the TGT is defined as a rectangular area in the WCS. A window must be mapped (copied) into a part of the screen memory to become visible. The mapping technique used by TGT is to specify the desired window placement on screen using NDC. The window mapped into NDC is called a viewport. The NDC viewport is mapped into IDC and it is called a window on the B007 board.

Fortunately, the IDC windows are never used directly by application programmers and the WCS window and the NDC viewport are the only coordinate systems needed by a programmer using TGT. An example of mapping a drawing in WCS to NDC to IDC is shown in figure 3.

It is possible to draw into a window and not display it. This is because the window memory is separate from the memory allocated for the two screens. The window memory is called the **window heap**. When the display window command is issued, the window heap is copied into the specified area of the screen memory. TGT supports multiple windows although once a window is defined, its size cannot change. Also, TGT does not support releasing window storage once a window is defined.

GETTING STARTED

The TGT source code should be included at or near the top of the application program that will use TGT. Because of the way the TGT routines are written, it is not possible to compile the TGT routines are written, it is not possible to compile the TGT procedures and use them as a library (ref. 2). The allocation of global storage requires that a user include the source code to TGT in a program. The source code is in a special file and can be copied into a users program. The source code for the TGT is on the NASA Lewis Transputer Laboratory host computer in the c:/d700c/animate subdirectory. The TGT file is called TGT.tsr.

The TGT source code can be compiled either using the D700C or the D700D compilers. The only routine that needs to be changed is the **rotate()** procedure. Both C and D versions of rotate are in the TGT source code. The procedure that is not required can be commented out and the compiler will ignore it.

The library for the B007 graphics board is in the c:/graphlib subdirectory and is called b007.tsr. Network programs using this library can be built using the D700C version of the compiler. To use the D700D version would require a slight modification of the B007 driver code.

If only the host B004 board and the B007 graphics board are required for a network, there are two loadable CODE PROGRAM folds in c:/d700c/animate that are compiled for either a T414 or T800 based B007 graphics board. These loadable folds can be used to load the B700 board. Link 2 on the B004 should be connected to link 1 on the B007.

With the current version of TGT, any unnecessary routines must either be deleted or commented out or the program will not compile. This is a compiler variable storage problem created by the size of TGT and the D700C version of the TDS. This is not required for the D700D version of the compiler.

Some of the windowing and geometric transformation procedures provided in TGT must be called in a specific order for proper application program operation. A world window must be defined using the **set.window.2d()** procedure before a viewport is defined using the **set.viewport.2d()** procedure. If this calling order is not followed, some of the internal variables used by **set.viewport.2d** will be uninitialized and the application program will not program correctly.

The geometric transformation procedures require the global transformation matrix, **trans.2d**, to be initialized to the identity matrix. This is performed with the **make.identity(trans.2d)** procedure call. Any calls to **scale()**, **rotate()**, or **translate()** will modify the global **trans.2d** matrix, and unless the matrix has been initialized, the application program will fail.

Several examples showing both windowing and geometric transformation procedures are listed below. The following examples are presented:

1. Window set-up
2. Multiple window set-up
3. Line drawing
4. Composite matrix transformation
5. Double buffered animation

Each example uses the following format for generating drawings:

```
--   include the TGT source code
--   initialize the graphics board
--   user specified TGT calls
--   shut down graphics board
```

EXAMPLES

Window Set-up

The code fragment shown below defines a window with the lower left corner at (-80.0, -50.0) and the upper right corner at (70.0, 25.0) in the world coordinate system. The window is placed on the display screen with the lower left corner at (0.0, 0.0) and the upper right corner at (0.75, 0.5) of the full screen size. The window is activated for drawing and is cleared to a light gray color which is color register 4 in the look-up table. The window is displayed and then the graphics board is shut down. Note that the window numbering is maintained by the application programmer and the window numbers start at number of 0.

The window-viewport-screen relationship for this example is shown in figure 4.

```
--include the TGT source code here
VAL my.window IS 0 :
SEQ
  init.graphics()
  set.window.2d(-80.0 (REAL32), -50.0 (REAL32),
                70.0 (REAL32), 25.0 (REAL32), my.window)
  set.viewport.2d(0.0 (REAL32), 0.0 (REAL32),
                0.75 (REAL32), 0.5 (REAL32), my.window)
  activate.viewport.2d(my.window)
  clear.window(4)
  display.viewport.2d(my.window)
  finit.graphics()
```

Multiple Window Set-up

The code fragment shown below defines and opens two windows on the display screen. One window is in the lower left corner of the screen and the other is in the uppper right corner. The lower left window is cleared to light gray and the upper right window is cleared to red. The last window drawn will overlap any currently displayed windows. This only affects screen memory and not the window heap storage. Note the window numbering is maintained by the application programmer and the window numbers start at number 0 with a maximum window number of 31.

The **set.window.2d()** procedure modifies the global internal storage used by the TGT. The **set.viewport.2d()** procedure sends windowing commands to the B007 graphics display board. The **set.window.2d()** procedure must be called before the **set.viewport.2d()** procedure. Once the viewports are defined, they can be activated, displayed and drawn into in any order.

```
-- include the TGT source code here
VAL my.win.one IS 0 :
VAL my.win.two IS 1 :
SEQ
  init.graphics()
  set.window.2d(-10.0 (REAL32), -100.0 (REAL32),
          25.0 (REAL32), 50.0 (REAL32), my.win.one)
  set.window.2d(0.0(REAL32), -10.0 (REAL32),
          43.0 (REAL32), 10.0 (REAL32), my.win.two)
  set.viewport.2d(0.0 (REAL32), 0.0 (REAL32),
          0.75 (REAL32), 0.75 (REAL32), my.win.one)
  activate.viewport.2d(my.win.one)
  clear.window(4)                          --light gray
  display.viewport.2d(my.win.one)
  set.viewport.2d(0.25 (REAL32), 0.25 (REAL32),
         1.0  (REAL32), 1.0 (REAL32), my.win.two)
  activate.viewport.2d(my.win.two)
  clear.window(31)                         --red
  display.viewport.2d(my.win.two)
  finit.graphics()
```

Line Drawing

In the code fragment shown below, a window is opened to the full screen size and cleared to 50 percent intensity white (gray). A diagonal line from the lower left corner to the lower upper right is drawn.

```
-- include the TGT source code here
VAL my.window IS 0 :
SEQ
  init.graphics()
  set.window.2d(-80.0 (REAL32), -50.0 (REAL32),
                70.0  (REAL32), 125.0 (REAL32), my.window)
  set.viewport.2d(0.0 (REAL32), 0.0 (REAL32),
                1.0 (REAL32), 1.0 (REAL32), my.window)
  activate.viewport.2d(my.window)
  clear.window(8)        -- 50 % intensity white (gray)
  bg.colour(8)          -- 50 % intensity white (gray)
  fg.colour(0)          -- black foreground pen
  draw.line.2d(-80.0 (REAL32), -50.0 (REAL32),
                70.0 (REAL32), 125.0 (REAL32), my.window)
  display.viewport.2d(my window)
  finit.graphics()
```

Composite Matrix Transformation

The code fragment shown below opens a window to the full screen size and clears it to 50 percent intensity white (gray). A box is drawn in black in the center of the window and is rotated 30 times through a displacement of 6 degrees. The x and y lengths of the line segments that make up the box are scaled by 0.9 for each rotation. The effect of this is the originally square box becomes distorted as it rotates and shrinks. The window is not cleared between consecutive drawings so all the boxes will be seen.

```
-- include the TGT source code here
VAL my.window IS 0 :
[4]REAL32 x, y :
SEQ
  init. graphics()
  set.window.2d(-100.0 (REAL32), -100.0 (REAL32),
                100.0 (REAL32), 100.0 (REAL32), my.window)
  set.viewport.2d(0.0 (REAL32), 0.0 (REAL32),
                  1.0 (REAL32), 1.0 (REAL32), my.window)
  activate.viewport.2d(my.window)
  clear.window(8)            -- 50 % intensity white (gray)
  bg.colour(8)              -- 50 % intensity white (gray)
  fg.colour(0)          -- black foreground pen

  -- initialize the box coordinates
  x[0] := -50.0 (REAL32)
  y[0] := x[0]
  x[1] := x[0]
  y[1] := 50.0 (REAL32)
  x[2] := y[1]
  y[2] := y[1]
  x[3] := y[1]
  y[3] := x[0]

  --set up composite tranformation matrix
  make.identity(trans.2d)
  rotate(6.0 (REAL32), 0.0 (REAL32), 0.0 (REAL32))
  scale(0.9 (REAL32), 0.9 (REAL32),
       (0.0 (REAL32), 0.0 (REAL 32))

  display.viewport.2d(my window)

  --draw the box and rotate and scale it 30 times
  SEQ i = 0 FOR 30
    SEQ
      SEQ i = 0 FOR 3
        draw.line.2d[x[i], y[i], x[i+1], y[i+1])
      draw.line.2d(x[0], y[0], x[3], y[3])
      transform.points(4, x, y)
  finit.graphics()
```

Double-Buffered Animation

The code fragment shown below opens a window to the full screen size. The window is cleared to 50 percent intensity white (gray) and the drawing pen is set to black. A box is drawn in the middle of the screen. While the box is displayed, the hidden screen has a scaled and rotated box drawn in it. The screens are swapped and the rotated, scaled box (distorted because the scaling scales the x and y projections of a line on the x an y axes) is displayed. The hidden screen is cleared and the next rotation, scaling and line drawing is performed. The screens are drawn in this manner for each time the box is rotated and scaled. Thus, the view sees a rotating box that smoothly decreases in size. None of the line drawing is seen since it all occurs on the hidden screen.

Note that the x and y coordinates are modified by each call to the **transform.points()** procedure. If the original data is required later, a copy of that data must be used with the TGT routines so the original data is not modified.

```
--include the TGT source code here
VAL my.window IS 0 :
[4]REAL32 x, y:
SEQ
  init.db.graphics()          -- initialize double buffered
                              -- graphics
  set.window.2d(-100.0 (REAL32), -100.0 (REAL32),
                100.0 (REAL32), 100.0 (REAL32), my.window)
  set.viewport.2d (0.0 (REAL32), 0.0 (REAL32),
                1.0 (REAL32), 1.0 (REAL32), my.window)
  activate.viewport.2d(my.window)
  clear.window(8)
  bg.colour(8)                -- 50 % intensity white (gray)
  fg.colour(0)                -- black foreground pen
  --initialize the box coordinates
  x[0] := -50.0 (REAL32)
  y[0] := x[0]
  x[1] := x[0]
  y[1] := 50.0 (REAL32)
  x[2] := y[1]
  y[2] := y[1]
  x[3] := y[1]
  y[3] := x[0]

  --set up composite transformation matrix
  make.identity(trans.2d)
  rotate(6.0 (REAL32), 0.0 (REAL32), 0.0 (REAL32))
  scale(0.9 (REAL32), 0.0 (REAL32), 0.0 (REAL32))
  --code continues on next page
```

```
--draw the box, rotate and scale it 30 times
SEQ i = 0 FOR 30
  SEQ
    SEQ i = 0 FOR 3
      draw.line.2d(x[i], y[i], x[i+1], y[i+1])
    draw.line.2d(x[0], y[0], x[3]), y[3])
    display.viewport.2d(my.window)
    flip.screen()
    clear.window(8)
    transform.points(4, x, y)
finit.graphics()
```

BYPASSING THE TRANSPUTER GRAPHICS TOOLKIT

Not all of the capabilities of the B007 graphics board are supported by the Transputer Graphics Toolkit. This section provides information on directly accessing the B007 graphics board when also using the TGT. Sending commands directly to the graphics board should only be performed by experienced users.

The input and output channels used by TGT to communicate with the B007 graphics board are called **to.graphic** and **from.graphic**. These channel names are globally scoped for internal TGT use. The channel names **to.graphic** and **from.graphic** can be used for direct communication with the B007. The channels are PLACEd onto the output and input of link 2 on the processor using the TGT. The programmer is responsible for sending all commands to and receiving any commands from the B007 graphics board. Use of the B007 is described in reference 1.

The B007 display driver software is provided as a loadable CODE PROGRAM fold and expects to input and output on link 1 (ref. 2). This file currently resides in c:/d700c/animate subdirectory on the NASA Lewis Transputer Laboratory host computer. Do not try to modify the B007 driver code. It was compiled under TDS 2.0 version D700C and will not compile under any of the later TDS releases without some slight changes.

COMMAND SUMMARY

This section lists the graphics procedures according to their function. This section is helpful if the exact procedure name is not known but the type of function required is known. Only the procedure names are listed in this section. The full description of each command can be found in the Command Reference section below.

The graphics commands have been split into the following groupings:

Startup and Shutdown Procedures
Geometric Transformation Procedures
Screen and Window Manipulation Procedures
Absolute and Relative Draw Procedures
Miscellaneous Display Procedures
New B007 Procedures
Internal Graphics System Procedures

Startup and Shutdown Procedures

finit.graphics
init.db.graphics
init.graphics

Geometric Transformation Procedures

make.identity
rotate
scale
transform.point
transform.points
translate

Screen and Window Manipulation Procedures

activate.viewport.2d
clip.line.2d
clip.point.2d
display.viewport.2d
move.viewport.position.2d
select.screen
set.viewport.2d
set.window.2d

Absolute and Relative Draw Procedures

draw.arc.2d
draw.circle.2d
draw.line.2d
draw.polygon.2d
draw.rectangle.2d
line.abs.2d
line.rel.2d
move.abs.2d
move.rel.2d
point.abs.2d
point.rel.2d

Miscellaneous Display Procedures

activate.screen
bg.colour
clear.screen
clear.window
display.screen
fg.colour
fill.polygon
fill.polygon.2d
flip.screen
int.line
quick.fill.polygon
quick.fill.polygon.2d
select.colour.table
set.colour

New B007 Procedures

pixel.line
colour.line

Internal Graphics System Procedures

c.draw.line
combine.transformations
g.draw.line
g.send
g.send1
g.send2
map.to.screen.coords

COMMAND REFERENCE

This section provides a detailed description of the TGT routines. The variables used in the parameter lists are explained and the global variables modified and routines called from a TGT procedure are listed. The descriptions are listed alphabetically.

The real numbers used in the example section of each command description leaves out the REAL32 type qualifier. This is done for readability and the type qualifier should be included in the application program. The examples shown previously use the correct method for declaring REAL32 variables.

activate.screen

Declaration: PROC activate.screen(VAL INT screen.number)

Usage: activate.screen(screen.number)

Parameters: screen.number: the number of the screen to activate for drawing. The value for **screen.number** must be 0 or 1.

Function: Activates the requested screen for drawing. It does not affect the displayed screem. This procedure will deactivate the active window and window mode. It does not affect the window heap allocation.

Procedures called:

g.send2()

Global variables used:

BOOL window.selected

Example: activate.screen(1)

Activates screen 1 for drawing. Does not affect the displayed screen. If screen 1 is displayed, any drawing commands executed will be seen. If screen 0 is displayed, the execution of drawing commands will be hidden.

activate.viewport.2d()

Declaration: PROC activate.viewport.2d(VAL INT viewport.number)

Usage: activate.viewport.2d(viewport.number)

Parameters: viewport.number: The number of the viewport to be activated for drawing.

Function: Selects the specified viewport for drawing. This is only for drawing. To display the viewport requires the **display.viewport.2d()** procedure.

Procedures called:

g.send2()

Global variables used:

active.window

Example: active.viewport.2d(4)

Activates viewport 4 for drawing. Any subsequent drawing commands issued will be performed in the WCS window number 4 and clipped to its boundary coordinates.

bg.colour()

Declaration: PROC bg.colour(VAL INT entry)

Usage: bg.colour(entry)

Parameters: entry: the color table entry to use for the background color.

Function: Changes the background drawing pen to the color in the color lookup table at position **entry**.

Procedures called:

g.send2()

Global variables used: None

Examples: bg.colour(124)

Selects the color in position 124 of the color look-up table to be the background pen. The current look-up table has 256 entries. The value for entry can be from 0 to 255.

c.draw.line()

Declaration: PROC c. draw.line(VAL INT x1, y1, x2, y2)

Usage: c.draw.line(x1, y1, x2, y2)

Parameters: x1, y1: IDC coordinate of the starting point

x2, y2: IDC coordinate of the end point

Function: Draws a line from (x1, y1) to (x2, y2) in IDC.

Procedures called: None

Global variables used:

```
CHAN OF ANY to.graphic
CHAN OF ANY from.graphic
```

Example: c.draw.line(25, 44, 288, 305)

Draws aline in IDC from (25, 44) to (288, 305).
This procedure is not normally used by application programs since it uses integer device coordinates.

clear.screen()

Declaration: PROC clear.screen(VAL INT colour)

Usage: clear.screen(colour)

Parameters: colour: the color table entry to use

Function: Clears the screen to the color in the look-up table specified by the value **colour**.

Procedures called:

g.send2()

Global variables used: None

Example: clear.screen(4)

Clears the screen to the color in the color lookup table at entry 4. This clears the active screen. It does not affect the window heap space but any windows displayed on the screen will be written over.

clear.window()

Declaration: PROC clear.window(VAL INT colour)

Usage: clear.window (colour)

Parameters: colour: the color table entry to use

Function: Clears the window to the color in the look-up table specified by the value **colour**.

Procedures called:

g.send2()

Global variables used: None

Example: clear.window(4)

Clears the window to the color in the color look-up table at entry 4. This clears the active window. It clears the window in the window heap area of memory. The window must be redisplayed to put the cleared window on the screen.

clip.line.2d()

Declaration: PROC clip.line.2d(REAL32 x1, y1, x2, y2,
BOOL display)

Usage: clip.line.2d(x1, y1, x2, y2, display)

Parameters: x1, y1: starting coordinate of the line to be clipped.

x2, y2: end coordinate of the line to be clipped

display : boolean value showing whether the line is on screen or not.

Function: Clips the line from (x1, y1) to (x2, y2) to the current WCS window coordinates. The boolean value **display** is set to TRUE if any of the line is on-screen and to FALSE if the line is totally off screen. The clipped lines coordinates are returned in the calling parameters.

Procedures called:

SC.code.2d()
reject.check()
exchange()

Global variables used:

[max.windows][index.size]REAL32 windows

Example:

clip.line.2d(x1, y1, x2, y2, displayed)

Clips the line from (x1, y1) to (x2, y2) and returns the clipped coordinates in (x1, y1) and (x2, y2). If the line is in the WCS window, display is TRUE else display is FALSE.

clip.point.2d()

Declaration: PROC clip. point.2d(VAL REAL32 x, y, BOOL display)

Usage: clip.point.2d(x, y, diaplay)

Parameters: x, y : The x and y coordinates of the point of be clipped.

display : boolean value showing whether the point (x, y) is in the WCS screen.

Function: Clips the point (x, y) to the current WCS window coordinates. The value display shows whether or not the point is in the WCS window.

Procedures called: None

Global variables used:

[max.windows][index.size]REAL32 windows

Example: clip.point.2d(x, y, display)

Clips the point x, y to the current WCS window coordinates. The value display is TRUE if the point is in the WCS window and it is FALSE if it is not.

colour.line()

Declaration: PROC colour.line(VAL INT size, []INT buffer)

Usage: colour.line(size, buffer)

Parameters: size: the number of (x, y, colour) coordinate triplets in buffer to draw.

buffer: buffer containing the data in (x, y, colour) format.

Function: Performs block transfer of pixel coordinate data. Each pixel in addition to the (x,y) data contains a color look-up table entry number to use for that point. Storage is in (x[0], y[0], colour[0], ..., x[n], y[n], colour[n]) order, where n = (size-1). The maximum number of triplets that can be transferred is 726. NOTE: The (x,y) coordinate data is stored in IDC **NOT** in the WCS.

Procedures called: None

Global variables used:

CHAN OF ANY to.graphic
CHAN OF ANY from.graphic

Example: colour.line(222, buffer)

Draws 222 pixels from the buffer array into the active screen or window. The color of each pixel is stored with the coordinate information in buffer. The format is consecutive triplets of (x, y, colour) data.

combine.transformations()

Declaration: PROC combine.transformations([3][3]REAL32 mat.a, mat.b)

Usage: combine.transformations(mat.a, mat.b)

Parameters: mat.a, mat.b: the matrices that are to be multiplied together. The multiplication order is [a][b]. The result is returned in mat.a.

Function: Multiplies the matrix mat.a by mat.b and returns the result in mat.a. This procedure is used to create composite matrix out of the scale, rotate, and translate procedures. This procedure is not normally used by application programmers.

Procedures called: None

Global variables used: None

Example: combine.transformations(trans.2d, temp.matrix)

Multiplies the trans.2d matrix by temp.matrix and returns the result in trans.2d.

display.screen()

Declaration: PROC display.screen(VAL INT screen.number)

Usage: display.screen(screen.number)

Parameters: screen.number: the number of the screen to display. The value for **screen.number** must be 0 or 1.

Function: Displays the specified screen. If the specified screen is already displayed, it does nothing.

Procedures called:

g.send2()

Global variables used: None

Example: display.screen(0)

Displays screen 0.

display.viewport.2d()

Declaration: PROC display.viewport.2d(VAL INT viewport.number)

Usage: display.viewport.2d(viewport.number)

Parameters: viewport.number : the number of the viewport to be copied into the active screen memory.

Function: Copies the specified viewport from **window heap** memory to the active **screen memory.** This procedure does not change the active viewport for drawing.

Procedures called:

map.to.screen.coords()
g.send()

Global variables used:

INT active.window
INT window.selected
[max.windows][index.size]REAL32 windows

Example: display.viewport.2d(3)

Copies the viewport 3 into the currently active screen's memory. The window may or may not be displayed on the monitor. This depends on whether the active screen is the one visible on the monitor.

draw.arc.2d

Declaration: PROC draw.arc.2d(VAL REAL32 x1, y1, x2, y2, x3, y3)

Usage: draw.arc.2d(x1, y1, x2, y2, x3,y3)

Parameters:
x1, y1,
x2, y2,
x3, y3 : The three points through which the arc will be drawn.

Function: Uses the three point arc routine to draw an arc in the WCS. Draws in the active viewpoint (WCS window).

Procedures called:

map.to.screen.coords()
g.send()

Global variables used: None

Example: draw.arc.2d(0.0, 0.0, 25.0, 25.0, -25.0, 25.0)

Draws an arc through the three world coordinate system points (0.0, 0.0), (25.0, 25.0), and (-25.0, 25.0).

draw.circle.2d()

Declaration: PROC draw.circle.2d(VAL REAL32 x.center, y.center, radius

Usage: draw.circle.2d(x.center, y.center, radius)

Parameters: x.center, y.center: WCS coordinates for the center of the circle.

radius: radius in WCS

Function: Render a circle in WCS with center (x.center, y.center) and radius radius. The circle is scaled to the viewport with the x-axis dimension. If the window to viewport mapping is not proportional, the circle will no longer be the correct aspect ratio, the y-axis length will be incorrect.

Procedures called:

map.to.screen.coordinates()
g.send()

Global variables used:

[max.windows][index.size]REAL32 windows

Example: draw.circle.2d (0.0, 0.0, 45.0)

Draws a circle in WCS with center (0.0, 0.0) and radius 45.0.

draw.line.2d()

Declaration: PROC draw.line.2d(VAL REAL32 x1, y1, x2, y2)

Usage: draw.line.2d(x1, y1, x2, y2)

Parameters: x1, y1: x and y coordinates of the first point
x2, y2: x and y coordinates of the second point

Function: Draws a line from (x1, y1) to (x2, y2) in WCS.

Procedures called:

map.to.screen.coords()
g.draw.line()

Global variables used: None

Example: draw.line.2d(-5.0, 0.0, 10.0, -7.0)

Draws a line from (-5.0, 0.0) to (10.0, -7.0) in the world coordinate system.

draw.polygon.2d()

Declaration: PROC draw.polygon.2d(VAL INT num.sides,
VAL []REAL32 buffer)

Usage: draw.polygon.2d(num.sides, buffer)

Parameters: num.sides: number of sides for the polygon

buffer: buffer containing the polygon sides in the order: x[0] = buffer[0], y[0] = buffer[1],

Function: Draws a polygon in the world coordinate system. Automatically closes the polygon connecting (x[0], y[0]) with (x[last], y[last]). The maximum number of sides currently supported is 100.

Procedures called:

map.to.screen.coords()

Global variables used:

CHAN OF ANY to.graphic
CHAN OF ANY from.graphic

Example: draw.polygon.2d(10, buffer)

Draws an 10 sided polygon using the points stored in buffer in increasing x, y order.

draw.rectangle.2d()

Declaration: PROC draw.rectangle.2d(VAL REAL32 x, y,
x.length, y.length)

Usage: draw.rectangle.2d(x, y, x.length, y.length)

Parameters: x, y: the top left coordinate of the rectangle.

x.length, y.length: length of the x and y sides of the rectangle.

Function: Draws a rectangle in WCS.

Procedures called:

map.to.screen.coords()
g.send()

Global variables used:

[max.windows][index.size]REAL32 windows

Example:

draw.rectangle.2d(1.0, 2.0, 5.0, 7.0)

Draws a rectangle with upper left corner at (1.0, 2.0) in the world coordinate system. The length of the x side is 5.0 units and the length of the y side is 7.0 units.

fg.colour()

Declaration: PROC fg.colour(VAL INT entry)

Usage: fg.colour(entry)

Parameters: entry: the color table entry to use for the foreground color.

Function: Changes the foreground drawing pen to the color in the color look-up table at position **entry.**

Procedures called:

g.send2()

Global variables used: None

Example: fg.colour(124)

Selects the color in position 124 of the color look-up table to be the foreground pen. The current look-up table has 256 entries. The value for entry can be from 0 to 255.

fill.polygon()

Declaration: PROC fill.polygon(VAL INT x, y).

Usage: fill.polygon(x, y)

Parameters: x, y: a point in IDC inside the polygon that is to be filled.

Function: An arbitrary polygon is filled with the currently selected foreground color. The polygon is selected by specifying a point inside it. This routine uses normalized device coordinates which is not normally used by applications programs. For a world coordinate system version see the procedure **fill.polygon.2d().**

Procedures called:
g.send()

Global variables used: None

Example: fill.polygon(125, 200)

Fills the polygon that has an interior point at (125, 200) in integer device coordinates. Will draw into the active screen or window.

fill.polygon.2d()

Declaration: PROC fill.polygon.2d(VAL REAL32 x, y)

Usage: fill.polygon.2d(x, y)

Parameters: x, y: a point in WCS inside the polygon that is to be filled.

Function: An arbitrary polygon is filled with the currently selected foreground color. The polygon is selected by specifying a point inside it. This routine uses the world coordinate system.

Procedures called:

map.to.screen.coords()
g.send()

Global variables used: None

Example: fill.polygon.2d(125.0, -200.0)

Fills the polygon that has an interior point at (125.0, -200.0) in the world coordinate system. It will draw into the active screen or window.

finit.graphics()

Declaration: PROC finit.graphics()

Usage: finit.graphics()

Parameters: None

Function: Sends **c.terminate** flag to B007 graphics board for termination. Ignores the reply from the B007.

Procedures called: None

Global variables used:

CHAN to.graphic
CHAN from.graphic

Example: finit.graphics()

Shuts down the B007 graphics board

flip.screen()

Declaration: PROC flip.screen()

Usage: flip.screen()

Parameters: None

Function: Swaps the currently display and nondisplayed screens. Used for double buffered animation. The nondisplayed screen is always active.

Procedures called:

g.send1()

Global variables used: None

Example: flip.screen()

Flips the currently displayed and nondisplayed screens.

g.draw.line()

Declaration: PROC g.draw.line([2]INT p0, p1)

Usage: g.draw.line(p0, p1)

Parameters: p0, p1: arrays of (x, y) coordinate data.

Function: draws a line in IDC from p0 to p1. The data is stored as p0[0] = x1; p0[1] = y1,; p1[0] = x2; p1[1] = y2.

Procedures called: None

Global variables used:

```
CHAN OF ANY to.graphic
CHAN OF ANY from.graphic
```

Example: g.draw.line(x0, x1)

Draws a line in IDC from the point x0 to the point x1. The arrays x0, and x1 each contain (x, y) coordinate data. This procedure is not normally used by applications programmers.

g.send()

Declaration: PROC g.send(VAL INT32 command, VAL []INT params)

Usage: g.send(command, params)

Parameters: command: the drawing command to send to the B007 graphics board.

parameters: array of data needed by the B007 to execute the specified command.

Function: Allows a graphics command and an arbitrary amount of data to be sent to the B007 graphics board. An example of a procedure that uses this command is **draw.rectangle()**.

Procedures called: None

Global variables used:

```
CHAN OF ANY to.graphic
CHAN OF ANY from.graphic
```

Example:

```
g.send(c.draw.rectangle, [x.screen, y.screen,
  x.length, y.length])
```

Sends the draw rectangle command to the B007 graphics board. The data sent is the upper left corner of the rectangle in IDC (x.screen, y.screen) and the length of each side: x.length and y.length. Note the data is surrounded by square brackets. The brackets effectively put that data in an array for transfer.

g.sendl()

Declaration: PROC g.sendl(VAL INT32 command)

Usage: g.sendl(command)

Parameters: command: the low-level command to send to the B007 graphics board.

Function: Used to send a single graphics command to the B007 graphics board. Some commands only require a single word to be sent to the B007.

Procedures called: None

Global variables used:

```
CHAN OF ANY to.graphic
CHAN OF ANY from.graphic
```

Example: g.sendl(c.flip.screen)

Tells the B007 to flip the active and inactive screens. Used for double-buffered animation.

g.send2()

Declaration: PROC g.send2(VAL INT32 command1,
VAL INT32 command2)

Usage: g.send2(command1, command2)

Parameters: command1, command2: the low-level commands to send to the B007 graphics board.

Function: Used to send two graphics commands to the B007 graphics board. Some commands only require two parameters to be sent to the B007.

Procedures called: None

Global variables used:

CHAN OF ANY to.graphic
CHAN OF ANY from.graphic

Example: g.send2(c.select.colour.table, 1)

Selects color look-up table 1 for good primary colors.

init.db.graphics()

Declaration: PROC init.db.graphics()

Usage: init.db.graphics()

Parameters: None

Function: Initializes the B007 for double-buffered screens selects the default color table 1, sets the background color pen to black, and the foreground color pen to white.

Procedures called:

g.send1()
g.send2()

Global variables used:

INT window.selected

Example: init.db.graphics()

Sets up the B007 for double buffered animations.

init.graphics()

Declaration: PROC init.graphics()

Usage init.graphics()

Parameters None

Function: Initializes the B007 graphics board for single buffered rendering. Screen 0 is selected and cleared to black. The foreground color is set to white and the background color is set to black.

Procedures called:

g.send2()

Global variables used:

INT window.selected

Example: init.graphics()

Initializes B007 graphics board for single buffered dislay.

int.line()

Declaration: PROC int.line(VAL INT x1, y1, x2, y2)

Usage: int.line(x1, y1, x2, y2)

Parameters: x1, y1: the starting point for the line in IDC

x2, y2: the end point for the line in IDC

Function: Draws a line from (x1, y1) to (x2, y2) in IDC.

Procedures called:

c.draw.line()

Global variables used:

CHAN OF ANY to.graphic
CHAN OF ANY from.graphic

Example: int.line(25, 25, 511, 229)

Draws a line from (25, 25) to (511, 229) in integer device coordinates. The line is drawn in the active window or screen.

line.abs.2d()

Declaration: PROC line.abs.2d(VAL REAL32 x, y)

Usage: line.abs.2d(x, y)

Parameters; x, y: The endpoint of the line in WCS coordinates.

Function: Draws a line from the current graphics cursor position (the graphics cursor position must be initialized by the user with the **move.abs.2d()** command) to the endpoint specified by (x, y) in WCS.

Procedures called:

move.abs.2d()
map.to.screen.coords()
g.draw.line()

Global variables used:

[max.windows][index.size]REAL32 windows

Example: line.abs.2d(100.0, 50.0)

Draws a line from the current graphics cursor position to a (100.0, 50.0) in the world coordinate system.

line.rel.2d()

Declaration: PROC line.rel.2d(VAL REAL32 dx,dy)

Usage: line.rel.2d(dx, dy)

Parameters: dx, dy: the distance in the x and y directions from the current graphics cursor position for the line endpoint. The values dx, and dy are in the WCS.

Function: Draws a line from the current graphics cursor position to a point dx in the x-direction and dy in the y-direction away. All coordinates are in the world coordinate system. The graphics cursor position must be initialized using the **move.abs.2d()** procedure.

Procedures called:

move.rel.2d()
map.to.screen.coords()
g.draw.line()

Global variables used:

[max.windows][index.size]REAL32 windows

Example: line.rel.2d(5.0, -10.0)

Draws a line from the current graphics cursor position in world coordinate system to a point 5.0 units in a x-direction and -10.0 units in the y-direction away from the current graphics position.

make identity()

Declaration: PROC make.identity ([3][3]REAL32 trans.matrix)

Usage: make.identity(trans.matrix)

Parameters: trans.matrix : A 3 by 3 array (matrix) to be set to the identity matrix.

Procedures called: None

Global variables used: None

Example: make.identity(my.matrix)

Sets the 3 by 3 array my.matrix to the identity matrix. All diagonal terms set to 1.0. All others set to 0.0.

map.to.screen.coords()

Declaration: PROC map.to.screen.coords(VAL REAL32 x, y,
INT x.screen, y.screen)

Usage: map.to.screen.coords(x, y, x.screen, y.screen)

Parameters: x, y: the coordinates in the WCS to be mapped into IDC.

x.screen, y.screen: the IDC coordinates for the (x, y) point in WCS.

Function: Maps a point in WCS for the active screen or window. The result is in IDC and is passed back in x.screen, y.screen. The routine makes use of the global data stored in the **windows** array.

NOTE: The window IDC system is relative to the window origin and not its position on screen. This is because window rendering is done in the B007 in the window heap space and not the screen memory.

Procedures called: None

Global variables used:

BOOL window.selected
[max.windows][index.size]REAL32 windows

Example: map.to.screen.coords(-5.0, -5.0, ix, iy)

Maps the point (-5.0, -5.0) in the active world coordinate system window to integer device coordinates. The result is passed back in ix and iy.

move.abs.2d()

Declaration: PROC move.abs.2d(VAL REAL32 x, y)

Usage: move.abs.2d(x, y)

Parameters: x, y: the point in the WCS to which the graphics cursor is to be moved.

Function: Causes the invisible graphics cursor maintained by internally by the TGT to be moved to the point (x, y) in the WCS)

Procedures called: None:

Global variables used:

[max.windows][index.size]REAL32 windows

Example: move.abs.2d(10.0, 11.0)

Moves the graphics cursor to the point (10.0, 11.0) in the current world coordinate system window.

move.rel.2d()

Declaration: PROC move.rel.2d(VAL REAL dx, dy)

Usage: move.rel.2d(dx, dy)

Parameters: dx, dy: the distance in the x and y directions to move the graphics cursor from the current position.

Function: Moves the current graphics cursor position a distance dx in the x-direction and dy in the y-direction in the world coordinate system.

Procedures called: None

Global variables used:

[max.windows][index.size]REAL32 windows

Example: move.rel.2d(1.0, -2.0)

Moves the graphics cursor to a point (1.0, -2.0) away. The cursor must be positioned initially with the **move.abs.d()** procedure.

move.viewport.position.2d()

Declaration: PROC move.viewport.position.2d(VAL INT viewport.number, VAL REAL32 x.min, y.min)

Usage: move.viewport.positon.2d(viewport.number, x.min, y.min)

Parameters: viewport.number: the viewport that is to be repositioned onscreen.

x.min, y.min: the new position of the lower left corner of the viewport. The position is expressed in NDC.

Function: Modifies the global parameters that contain the parameters for the window position on the display screen. This procedure does not display the repositioned viewport. The procedure **display.viewport.2d()** must be used for displaying the viewport.

Procedures called: None

Global variables used:

[max.windows][index.size]REAL32 windows

Example: move.viewport.position.2d(3, 0.0, 0.0)

Moves the lower left corner of viewport number 3 to the lower left corner of the screen. The viewport remains the same size as when it was defined using the **set.viewport.2d()** procedure.

pixel.line()

Declaration: PROC pixel.line(VAL INT size, []INT buffer)

Usage: pixel.line(size, buffer)

Parameters: size: the number of (x, y) pairs in buffer to draw.

buffer: buffer containing the data to plot. Data must be stored contiguously in (x, y) coordinate pairs.

Finction: Allows block transfers of pixel coordinate data. The maximum number of data points that can be transferred is 1089 (x, y) coordinate pairs. Storage is in (x[0], y[0], x[1], y1], ..., x[n], y[n]) order where n = (size - 1). The variable **size** is the number of (x, y) pairs to transfer. NOTE: The (x,y) coordinate data is in integer device coordinates and **NOT** in the world coordinate system.

Procedures called: None

Global variables used:

CHAN OF ANY to.graphic
CHAN OF ANY from.graphic

Example: pixel.line(100, buffer)

Draws the first 100 (x, y) coordinate pairs stored in the array buffer in the current foreground color in the current active window or screen.

point.abs.2d

Declaration: PROC point.abs.2d(VAL REAL x, y)

Usage: point.abs.2d(x, y)

Parameters: x, y: the coordinate in the WCS where a point is to be drawn.

Function: Draws a point in the current WCS window at the coordinate (x, y)

Procedures called:

map.to.screen.coords()
g.send()

Global variables used: None

Example: point.abs.2d(1.0, -2.0)

Draws a point at (1.0, -2.0) in the current WCS window. The graphics cursor is set to this coordinate.

point.rel.2d()

Declaration: PROC point.rel.2d(VAL REAL dx, dy)

Usage point.rel.2d(dx, dy)

Parameters: dx, dy: the distance in the x and y directions to move the ggraphics cursor from the current position and draw a point.

Function: Moves the current graphics cursor position a distance dx in the x-direction and dy in the y-direction and draws a point in the currently selected foreground color.

Procedures called:

map.to.screen.coords()
g.send()

Global varaibles used:

[max.windows][index.size]REAL32 windows

Example: point.rel.2d(1.0, -2.0)

Moves the graphics cursor to a point (1.0, -2.0) away and draws a point in the currently selected foreground color. The cursor must be positioned initially with the **move.abs.2d()** procedure.

quick.fill.polygon()

Declaration: PROC quick.fill.polygon(VAL INT x, y)

Usage: quick.fill.polygon(x, y)

Parameters: x, y: point inside the polygon to be filled. The coordinate (x, y) is specified in IDC.

Function: An simple convex polygon is filled with the currently selected foreground color. The polygon is selected by specifying a point inside it. This routine uses IDC. For a WCS version of this command see **quick.fill.polygon.2d()**.

Procedures called:

g.send()

Global variables used: None

Example: quick.fill.polygon(100, 333)

Fills only the simple convex polygon that contains the point (100, 333). The coordinate is specified in integer device coordinates.

quick.fill.polygon.2d()

Declaration: PROC quick.fill.polygon.2d(VAL REAL32 x, y)

Usage: quick.fill.polygon.2d(x, y)

Parameters: x, y: point inside the polygon to be filled. The coordinate (x, y) is specified in WCS.

Function: A simple convex polygon is filled with the currently selected foreground color. The polygon is selected by specifying a point inside it. The algorithm assumes no interior angles of the polygon exceed 180 degrees. This routine uses the WCS.

Procedures called:

g.send()

Global variables used: None

Example: quick.fill.polygon.2d(100.0, -25.5)

Fills only the simple convex polygon that contains the point (100.0, -25.5). The coordinate is specified in the world coordinate system.

rotate() D700C Version

Declaration: PROC rotate(VAL REAL32 alpha, x.pivot, y.pivot)

Usage rotate(alpha, x.pivot, y.pivot)

Parameters: alpha: the desired rotation angle in degrees.

x.pivot, y.pivot: the point in the world coordinate system about which rotation occurs.

Function: The rotate routine modifies the trans.2d matrix for a rotation of alpha degrees about the stationary point (x.pivot, y.pivot).

Procedures called:

make.identity()
radian.equiv()
COSP()
SINP()
combine.transformations()

Global variables used:

[3][3]REAL32 trans.2d

Example: rotate(45.0, 0.0, 0.0)

Set up the trans.2d matrix for a positive 45.0 degree rotation about the origin (0.0, 0.0) in world coordinate system.

rotate() D700D Version

Declaration: PROC rotate(VAL REAL32 alpha, x.pivot, y.pivot)

Usage: rotate(alpha, x.pivot, y.pivot)

Parameters: alpha: the desired rotation angle in degrees.

x.pivot, y.pivot: the point in the world coordinate system about which rotation occurs.

Function: The rotate routine modifies the trans.2d matrix for a rotation of alpha degrees about the stationary point (x.pivot, y.pivot).

Procedures called:

make.identity()
COS()
SIN()
FUNCTION radian.equiv()
combine.tranformations()

Global varaibles used:

[3][3] REAL32 trans.2d

Example: rotate(45.0, 0.0, 0.0)

Set up the trans.2d matrix for a positive 45.0 degree rotation about the origin (0.0, 0.0) in the world coordinate system.

scale()

Declaration: PROC scale(VAL REAL32 scale.x, scale.y,
x.fixed, y.fixed)

Usage: scale(scale.x, scale.y, x.fixed, y.fixed)

Parameters: scale.x, scale.y: scale factor for the x and y axis dimensions.

x.fixed, y.fixed: the stationary (x, y) coordinate of the object being scaled.

Function: The scale routine modifies the global transformation matrix trans.2d to scale a data set by scale.x, scale.y about the stationary point (x.fixed, y.fixed)

Procedures called:

combine.transformations()
make.identity()

Global variables used:

[3][3]REAL32 trans.2d

Example: scale(0.5, 1.0, 0.0, 0.0)

The trans.2d matrix has been configured to scale the x-dimension of some data to 0.5 while not changing the y-axis dimension. The scaling is from the stationary point (0.0, 0.0) in the world coordinate system.

select.colour.table()

Declaration: PROC select.colour.table(VAL INT number)

Usage: select.colour.table(number)

Parameters: number: the number of the B007 color table to use. The value for number should be 0 or 1.

Function: Selects which color table to use on the B007 graphics board. The table selection specifies how the bits in the byte address pointer are to be interpreted. Color table 1 is normally used because it has good primary color scales. See the INMOS B007 user manual for more details (ref. 1).

Procedures called:

g.send2()

Global variables used: None

Example: select.colour.table(1)

Selects color table 1 which has good primary color scales.

select.screen()

Declaration: PROC select.screen(VAL INT screen.number)

Usage: select.screen(screen.number)

Parameters: screen.number : screen to activate

Function: Activates the requested screen number (must be 0 or 1) and turns off the window mapping feature of TGT. Using this command allows drawing directly into the screen map. Can also use this command to disable double-buffering once it is enabled. The **display.screen()** command also needs to be used when disabling double-buffering.

Procedures called:

g.send2()

Global variables used:

window.selected

Example: select.window(0)

Screen number 0 is selected for drawing. The internal TGT variable **window.selected** is set to FALSE.

set.colour()

Declaration: PROC set.colour(VAL INT entry, red, green, blue)

Usage: set.colour(entry, red, green, blue)

Parameters: entry: the color look-up table entry to modify

red, green, blue: the value of the red, green and blue intensities to be stored in the look-up table. The value for each intensity can be from 0 to 63 (6-bits of intensity information). For more details see reference 1.

Function: Modifies the color look-up table to the specified values for red, green and blue.

Procedures called:

g.send()

Global variables used: None

Example: set.color(2, 12, 23, 61)

Sets the color table entry 2 to: red intensity, 12; blue intensity, 23; green intensity, 61.

set.viewport.2d()

Declaration: PROC set.viewport.2d(VAL REAL32 x.min, y.min,
x.max, y.max,
VAL INT viewport.number)

Usage: set.viewport.2d(x.min, y.min, x.max, y.max, viewport.number)

Parameters: x.min, y.min: the lower left corner of the viewport in NDC.

x.max, y.max: the upper right corner of the viewport in NDC.

viewport.number: the number of the WCS window that this viewport is associated with.

Function: Maps the WCS window given by **viewport.number** into NDC. The procedure will not allow viewport parameters outside the range [0.0, 1.0] to be used. The appropriate scale factors to map from WCS to NDC are computed in this procedure. The viewport (window) selected is activated for drawing.

NOTE: The window size and position on the view screen is generated using this command. The position of the window can be modified using the **move.viewport.position.2d()** procedure.

Procedures called:

map.to.screen.coords()

Global variables used:

INT active.window
BOOL window.selected
[max.windows][index.size]REAL32 windows
CHAN to.graphic
CHAN from.graphic

Example: set.viewport.2d(0.1, 0.1, 0.5, 0.5, 4)

The WCS window number 4 is mapped into viewport number 4 in NDC. The onscreen placement of the viewport is with the lower left corner at (0.1, 0.1) and upper right corner at (0.5, 0.5)) in NDC.

set.window.2d()

Declaration: PROC set.window.2d(VAL REAL32 x.min, y.min,
x.max, y.max,
VAL INT window.number)

Usage: set.window.2d(xmin, y.min, x.max, y.max,
window.number)

Parameters: x.min, y.min: the lower left corner of the window in the WCS.

x.max, y.max: the upper right corner of the window in the WCS.

window.number: the number assigned to this window. Window numbering should start a window number 0.

Function: Allows the user to define a WCS window for graphics rendering. Will not do anything if maximum number of windows is exceeded. The value is currently set to 32. This procedure must be called before using the set.viewport.2d() procedures.

Procedures called: None

Global variables used:

[max.windows][index.size]REAL32 windows

Example: set.window.2d(-25.0, -50.0, 100.0 0.0, 1)

User WCS window number 1 is defined to be from the lower left corner at (-25.0, 50.0) to the upper right corner at (100.0, 0.0) in the world coordinate system. Note at this point, the placement of the world window on screen is not specified. This must be done using **set.viewport.2d()**.

transform.point()

Declaration: PROC transform.point(REAL32 x, y)

Usage: transform.point(x, y)

Parameters: x, y: The WCS x and y coordinate of the point to be transformed by the trans.2d matrix.

Function: Performs a matrix multiplication of the point (x,y) by the trans.2d matrix. Note that x and y must be able to be modified (call by reference) since the new x, y is passed back in the parameter list.

Procedures called: None

Global variables used:

[3] [3] REAL32 trans.2d

Example: transform.point(x, y)

Transform the point (x, y) in the world coordinate system by the trans.2d global transformation matrix.

transform.points()

Declaration: PROC tranform.points(VAL INT count,
[]REAL32 x, y)

Usage: transform.points(count, x, y)

Parameters: count : the number of points to transform
x, y : the array of x and y coordinates to to be transformed.

Funtion: Performs a matrix multiplication on the coordinate arrays x and y by the trans.2d matrix. The number of points transformed is count.

Procedures called: None

Global Variables used:

[3][3] REAL32 trans.2d

Example: transform.points(100, x, y)

Transforms the first 100 data points in the x, and y coordinate arrays by the trans.2d global transformation matrix.

translate()

Declaration: PROC translate(VAL REAL32 translate.x,
translate.y)

Usage: translate(translate.x, translate.y)

Parameters: translate.x, translate.y: distance for translating x, and y coordinates.

Function: Modifies the trans.2d array for x, and y translation.

Procedures called:

make.identity()
combiine.transformations()

Global variables used:

[3][3]REAL32 trans.2d

Example: translate(2.0, 3.0)

Transforms the trans.2d matrix so it will translate data 2.0 units in the x-direction and 3.0 units in the y-direction in the world coordinate system.

APPENDIX A

GRAPHICS TRANSFORMATIONS

This section gives a brief overview of the two-dimensional (2D) graphics transformations used in the Transputer Graphics Toolkit (TGT). A more complete presentation of this information can be found in references 3 to 5.

The development of translation, scaling, and rotation operations in 2D are presented along with the extension to homogeneous coordinates for the matrix representation of the transformations. The global variables used by TGT are also presented so an understanding of the transformations performed can be obtained.

Translation

A translation is a straight-line movement from one position in space to another. In 2D space, the translation from one point P(x , y) to another point P(x', y') can be performed by:

$$x' = x + T_x \tag{1}$$

$$y' = y + T_y \tag{2}$$

where

Tx ≡ translation distance in the x-direction

T_y ≡ translator distance in the y-direction

The translation distance from the original point is (Tx, Ty).

Scaling

Scaling is used to alter the size of an object. Points can be scaled by multiplying the coordinate P(x, y) by scale factors Sx, and Sy to generate the new coordinates (x', y'):

$$x' = r \cdot \cos(\phi) \cdot \cos(\Theta) - r \cdot \sin(\phi) \cdot \sin(\Theta) \tag{11}$$

$$y' = r \cdot \sin(\phi) \cdot \cos(\Theta) + r \cdot \cos(\phi) \cdot \sin(\Theta) \tag{12}$$

where r is the distance from the point from the origin.

Since

$$x = r \cdot \cos(\phi) \tag{13}$$

$$y = r \cdot \sin(\phi) \tag{14}$$

equations 11 and 12 can be rewritten as

$$x' = x \cdot \cos(\phi) - y \cdot \sin(\theta) \quad (15)$$

$$y' = y \cdot \cos(\phi) - x \cdot \sin(\theta) \quad (16)$$

As with the scaling operation, the rotation can be performed about a fixed point, P(Xr, Yr), called the rotation point or pivot point. Rotation with respect to this arbitrary fixed point is shown in figure 6. The transformation equations for this case are:

$$x' = X_r + (x - Xr) \cdot \cos(\theta) - (y - Y_r) \cdot \sin(\theta) \quad (17)$$

$$y' = Y_r + (y - Yr) \cdot \cos(\theta) + (x - X_r) \cdot \sin(\theta) \quad (18)$$

The placement of the pivot point is totally arbitrary and if the rotation point is set to (Xr, Yr) = (0, 0), the original rotation equations are obtained.

If only small angles are to be used, the trigonometric functions can be replaced with the following approximations:

$$\cos(\theta) \approx 1 \quad (19)$$

$$\sin(\theta) \approx \theta \quad (20)$$

where θ is the angle in radians. These small angle approximations are not used in the transputer graphics toolkit.

Matrix Represention of Graphics Transformations

Typical transformations performed by many applications are not merely a single scaling, rotation, or translation, but a combination of the three. An efficient approach combining the various transformations would be to generate a matrix from which the final coordinate could be computed from the iinital coordinate. The general form of the matrix representations of the three transformations are:

$x' = x + T_x$ translation
$y' = y + T_y$

$x' = x * S_x$ scaling
$y' = y * S_y$

$x' = x * R_x$ rotation
$y' = y * R_y$

Notice the translation is a matrix addition while the scaling and rotation operations are matrix multiplications. Because the translation is expressed as a matrix addition, there is no 2 by 2 matrix than can be used for all three transformations.

If the points to be transformed are expressed in homogeneous coordinates, all three transformations can be expressed as matrix multiplications. In homogeneous coordinates, point P(x, y) is represented as P(W*x, W*y, W) for $W \neq 0$.

When given a homogeneous coordinate representation for a point P(X, Y, W), a 2D cartesian coordinate representation for the point P(x, y) can be found by x = X/W and y = Y/W. For the 2D transformations used in the TGT routines, W will always be 1 and the division need not be performed. In some three-dimensional viewing operations, W will be different than 1.

With the coordinates for 2D space now represented as a 3-element vector, 3 by 3 transformation matrices can be written for the translation, rotation, and scaling operations. The 2D plane in the 3D space (x, y, w) can be thought of as the plane in 3D space with the coordinates (x, y, 1).

The transformation matrix for translation is:

$$[x', y', 1] = [x, y, 1] \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ T_x & T_y & 1 \end{bmatrix} \qquad (21)$$

where

$T_x \equiv$ translation distance in the x-direction.

$T_y \equiv$ translation distance in the y-direction.

The transformation matrix for scaling is

$$[x', y', 1] = [x, y, 1] \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad (22)$$

where

$S_x \equiv$ scale factor for x-axis

$S_y \equiv$ scale factor for y-axis

and the rotation transformation matrix is:

$$[x', y', 1] = [x, y, 1] \begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad (23)$$

where

$\theta \equiv$ rotation angle

Composite 2D Transforms

Now that the transformations can be represented as matrices, a composite transformation matrix can be generated that represents any arbitrary sequence of translations, rotations, and scalings. The composite transformation matrix can be built by using matrix multiplications. The matrices can be multiplied in the order desired to generate the composite transformation matrix. Rememeber that matrix multiplication is order dependant:

$$[A][B] \neq [B][A]$$

Thus, a translation followed by a rotation is not the same as a rotation followed by a translation.

Transputer Graphics Toolkit Transformations

The TGT has a global, 3 by 3 transformation matrix that is used for building a composite transformation matrix. This global transformation matrix is a [3][3]REAL32 array called **trans.2d.** To properly create a composite transformation matrix, first the trans.2d matrix must be set to the identity matrix with the **make.identity()** procedure. After it is initialized to the identity matrix the TGT transformation routines, **translate()**, **rotate()**, and **scale()**, can be used. These routines directly manipulate the trans.2d matrix to generate the desired composite transformation matrix. After the composite trans.2d matrix is built, data can be modified (multiplied) by the matrix by using either the **transform.point()**, or **transform.points()** procedures. An example of this procedure is shown below:

The code fragment shown below opens a window to the full screen size and clears it to 50 percent intensity white (gray). A box is drawn in black in the center of the window and is rotated 30 times through a displacement of 6 degrees. The x and y lengths of the line segments that make up the box are scaled by 0.9 for each rotation. The effect of this is the originally square box becomes distorted as it rotates and shrinks. The window is not cleared between consecutive drawings so all the boxes will be seen.

```
--include the TGT source code here
VAL my.window IS 0:
[4]REAL32 x, y:
SEQ
  init.graphics()
```

```
set.window.2d(-100.0 (REAL32), -100.0 (REAL32),
              100.0 (REAL32), 100.0 (REAL32), my window)
set.viewport.2d(0.0 (REAL32), 0.0 (REAL32),
              1.0 (REAL32), 1.0 (REAL32), my window)
activate.viewport(my.window)
clear.window(8)
bg.colour(8)           -- 50% intensity white (gray)
fg.colour(0)           -- black foreground pen


-- initialize the box coordinates
x[0] := -50.0 (REAL32)
y[0] := x[0]
x[1] := x[0]
y[1] := 50.0 (REAL32)
x[2] := y[1]
y[2] := y[1]
x[3] := y[1]
y[3] := x[0]

-- set up composite transformation matrix
make.identity(trans.2d)
rotate(6.0 (REAL32), 0.0 (REAL32), 0.0 REAL32))
scale(0.9 (REAL32), 0.0 (REAL32), 0.0 (REAL32))

display.viewport(my window)

--draw the box and rotate and scale it 30 times
SEQ i = 0 FOR 30
  SEQ
     SEQ i = 0 FOR 3
       draw.line.2d[x[i], y[i], x[i+1), y(i+1])
     draw.line.2d(x[0], y[0], x[3], y[3])
     transform.points(4, x, y)

finit.graphics()
```

APPENDIX B

WINDOWING AND CLIPPING

Windowing

Application programmers require the ability to define objects in a world coordinate system (WCS). A world coordinate system is any cartesian coordinate system a programmer finds convenient. Objects defined in the WCS are mapped by the TGT procedures first into Normalized Device Coordinates (NDC) and then into Integer Device Coordinates (IDC). Normalized device coordinates are used by the applications programmer to specify window placement and sizing on the display screen. The IDC are used by the transputer graphics display board to perform all of its graphics rendering computations. These coordinate systems are shown in figure 3.

Normalized device coordinates are defined with the origin in the lower left corner of the screen and the maximum x and y coordinate is 1.0. The normalized device coordinate system is used for WCS window placement and sizing on the display screen. The equations required to map from the WCS to NDC are shown below:

$$S_{xv} = \frac{x_{vmax} - x_{vmin}}{x_{wmax} - x_{wmin}} \tag{24}$$

$$S_{yv} = \frac{x_{vmax} - x_{vmin}}{x_{wmax} - x_{wmin}} \tag{25}$$

where

S_{xv}, S_{yv} — x and y viewport scale factors

X_{vmax}, Y_{vmax} — x and y coordinates of the upper right corner of the viewport expressed in NDC

X_{vmin}, Y_{vmin} — x and y coordinates of the lower left corner of the viewport expressed in NDC

X_{wmax}, Y_{wmax} — the x and y coordinates of the upper right corner of the WCS window

X_{wmin}, Y_{wmin} — the x and y coordinates of the lower left corner of the WCS window

The mapping from NDC to IDC is performed internally by the TGT. The IDC are used by the INMOS B007 transputer graphics board for its graphics primitive computations. The B007 has a maximum resolution of 512 pixels in both the horizontal and vertical directions. The integer screen coordinates vary from 0 to 511. Once an object is defined in NDC, the mapping to IDC is quite easy. The equations used to map from NDC to IDC are shown below:

$$X_{ndc} = (x - x_{wmin}) \cdot S_{xv} + X_{vmin} \quad (26)$$

$$Y_{ndc} = 1.0 - ((y - y_{wmin}) \cdot S_{yv} + Y_{vmin}) \quad (27)$$

where

x, y ≡ x and y coordinate of the WCS point to be converted to NDC

X_{ndc}, Y_{ndc} ≡ the NDC coordinates of the points converted from WCS

$$X_{idc} = X_{ndc} \cdot X_{screen} \quad (28)$$

$$Y_{idc} = Y_{ndc} \cdot Yscreen \quad (29)$$

where

X_{screen}, Y_{screen} ≡ the horizontal and vertical resolution of the video display screen in pixels.

X_{idc}, Y_{idc} ≡ the coordinate in integer device coordinates of the original point P(x, y) in WCS.

However, the B007 graphics board uses a **window heap** for window display storage display storage that is totally separate from the display screen storage, so the coordinate system used to draw in a window is always relative to the window origin regardless of where that window is displayed on the screen. The routines in TGT automatically take this into account and the consequences of such a mapping are totally transparent to the application programmer.

Line Clipping

The Cohen-Southerland line-clipping algorithm (CS) is used in the **clip.line.2d()** procedure in the TGT. This algorithm is discussed in detail in references 3 to 5. Although the line drawing routines used in TGT could make use of this clipping algorithm, the current release does not because the B007 graphics board performs raster clipping in its own display routines with considerable speed. The clipping routine is provided in case an application requires such operations.

Basically, the CS line-clipping algorithm uses a four digit binary code called a **region code** for each line endpoint. The region code identifies the endpoint as lying in one of nine regions relative to the desired display window. The regions and their respective codes are shown in figure 7. The region numbering is used to respective codes are shown in figure 7. The region numbering is used to represent one of four possible line endpoint positions relative to the display window. The following position bits are turned on if the line endpoint lies in a specific region:

bit 1 - endpoint to the left of the window
bit 2 - endpoint to the right of the window
bit 3 - endpoint below the window
bit 4 - endpoint above the window

The bit values can quickly be determined by comparing the line endpoint (x, y) to the window boundaries. Once the region codes have been determined for the line endpoints, a quick check can be performed to establish which lines are completely inside or outside the window boundaries. Lines that are not either completely inside or outside the window boundary are checked for intersection with the window boundaries.

New endpoints are computed for any lines intersecting a window boundary using the following equations (see fig. 8):

For an intersection with a vertical window boundary:

$$y = y1 + m(x - x1)$$

For an intersection with a horizontal boundary:

$$x = x1 + (y - y1) / m$$

where

$$m = (y2 - y1) / (x2 - x1)$$

Notice that a division is required in the CS clipping algorithm. Since division is fast on the floating-point transputer, this is not a problem; however, if it is a problem, there is another line-clipping routine called **midpoint subdivision** which does not require any division to clip a line to a window boundary (refs. 3 and 4).

APPENDIX C

TRANSPUTER GRAPHICS TOOLKIT SOURCE CODE

This sections contains the source code for all of the transputer graphics toolkit procedures.

The program was listed using the file lister program provided as Example 17 in the D700C release of the Transputer Development System (TDS). The lines beginning with {{{ or }}} represent the folds in the TDS text editor (ref. 6). They can be interpreted as comments in the listing below.

```
+-------------------------------------------------------------------------+
|   FILE: TGT.lis                                        SIZE:   32134 bytes |
|  SAVED: Tue Jul 05 15:10:54 1988                       PAGE:       1       |
+-------------------------------------------------------------------------+

**List of Fold**       TGT graphics routines
**List of File**       TGT.tsr
**List all lines
**Excluding : NO LIST folds
{{{  TGT graphics routines
{{{  graphics channel definitions
CHAN OF ANY to.graphic, from.graphic:
VAL link0.in  IS 4:
VAL link1.in  IS 5:
VAL link2.in  IS 6:
VAL link3.in  IS 7:

VAL link0.out IS 0:
VAL link1.out IS 1:
VAL link2.out IS 2:
VAL link3.out IS 3:

PLACE to.graphic   AT link2.out:
PLACE from.graphic AT link2.in:
}}}
{{{  constants
[3][3]REAL32 trans.2d:       -- "global" 2-D transformation matrix

{{{  screen size definitions
VAL screen.width    IS        512 :
VAL screen.height   IS        512 :
VAL screen.width.r  IS        511.0 (REAL32):
VAL screen.height.r IS        511.0 (REAL32):
VAL max.windows     IS          32 :
VAL max.viewports   IS max.windows :
}}}
{{{  primitive size definitions
VAL max.sides       IS         100 :
VAL max.points      IS          50 :
}}}

BOOL display :
BOOL window.selected :

VAL r0.0 IS 0.0 (REAL32) :
VAL r1.0 IS 1.0 (REAL32) :
VAL r2.0 IS 2.0 (REAL32) :

INT active.window :

{{{  windows array indicies
VAL index.size    IS 12 :

VAL x.world.min   IS  0 :
VAL x.world.max   IS  1 :
VAL y.world.min   IS  2 :
VAL y.world.max   IS  3 :
VAL x.view.min    IS  4 :
```

```
+-------------------------------------------------------------------------+
|   FILE: TGT.lis                                        SIZE:   32134 bytes |
|  SAVED: Tue Jul 05 15:10:54 1988                       PAGE:       2       |
+-------------------------------------------------------------------------+

VAL y.view.min    IS  5 :
VAL x.view.max    IS  6 :
VAL y.view.max    IS  7 :
VAL pen.x         IS  8 :
VAL pen.y         IS  9 :
VAL vp.scale.x    IS 10 :
VAL vp.scale.y    IS 11 :
)))
[max.windows][index.size]REAL32 windows :
[max.windows]INT b007.win.num :  -- carry along B007 "internal" window vals
)))
(((  graphic commands to to B007
(((  commands
VAL c.plot.point          IS   1 (INT32) :
VAL c.draw.line           IS   2 (INT32) :
VAL c.draw.circle         IS   3 (INT32) :
VAL c.draw.arc            IS   4 (INT32) :
VAL c.draw.rectangle      IS   5 (INT32) :
VAL c.draw.polygon        IS   6 (INT32) :
VAL c.fill.polygon        IS   7 (INT32) :
VAL c.move                IS   8 (INT32) :
VAL c.move.rel            IS   9 (INT32) :

VAL c.clear.screen        IS  10 (INT32) :
VAL c.select.screen       IS  11 (INT32) :
VAL c.display.screen      IS  12 (INT32) :
VAL c.flip.screen         IS  13 (INT32) :
VAL c.copy.screen         IS  14 (INT32) :
VAL c.clear.window        IS  15 (INT32) :
VAL c.select.window       IS  16 (INT32) :
VAL c.display.window      IS  17 (INT32) :
VAL c.set.window          IS  18 (INT32) :
VAL c.set.draw.mode       IS  19 (INT32) :

VAL c.draw.char           IS  20 (INT32) :
VAL c.define.char         IS  21 (INT32) :
VAL c.write.string        IS  22 (INT32) :
VAL c.untyped.string      IS  23 (INT32) :
VAL c.write.number        IS  24 (INT32) :
VAL c.scroll              IS  25 (INT32) :
VAL c.jump.scroll         IS  26 (INT32) :
VAL c.rotate              IS  27 (INT32) :
VAL c.reflect.x           IS  28 (INT32) :
VAL c.reflect.y           IS  29 (INT32) :

VAL c.line.feed           IS  30 (INT32) :
VAL c.carriage.return     IS  31 (INT32) :
VAL c.set.colour          IS  32 (INT32) :
VAL c.select.fg.colour    IS  33 (INT32) :
VAL c.select.bg.colour    IS  34 (INT32) :
VAL c.select.colour.table IS  35 (INT32) :
VAL c.set.mask.reg        IS  36 (INT32) :
VAL c.set.xwidth          IS  37 (INT32) :
```

```
+-------------------------------------------------------------------------------+
|   FILE: TGT.lis                                          SIZE:   32134 bytes  |
| SAVED: Tue Jul 05 15:10:54 1988                          PAGE:       3        |
+-------------------------------------------------------------------------------+

VAL c.set.yheight          IS  38 (INT32) :

VAL c.line.frequency       IS  40 (INT32) :
VAL c.frame.rate           IS  41 (INT32) :
VAL c.interlace            IS  42 (INT32) :
VAL c.pixel.clock          IS  43 (INT32) :
VAL c.init.crt             IS  44 (INT32) :
VAL c.quick.fill           IS  45 (INT32) :

VAL c.terminate            IS 255 (INT32) :

{{{  new block transfer protocols
-- added to do block transfers of pixels:
-- c.pixel.line expects size::data with data in an (x, y) format
-- c.color.line expects size::data with data in and (x, y, color) format
-- NOTE size is only the MOST significant value in a 2-D array

VAL c.pixel.line           IS  99 (INT32) :
VAL c.color.line           IS 100 (INT32) :
VAL c.color.line.16        IS 101 (INT32) :
VAL c.RL.line              IS 102 (INT32) :
}}}
}}}
{{{  return codes
VAL e.ok                        IS    0 (INT32) :
VAL e.out.of.drawing.range      IS   -1 (INT32) :
VAL e.invalid.screen            IS   -2 (INT32) :
VAL e.invalid.window            IS   -3 (INT32) :
VAL e.no.window.store           IS   -4 (INT32) :
VAL e.too.many.windows          IS   -5 (INT32) :
VAL e.unknown.drawing.mode      IS   -6 (INT32) :
VAL e.invalid.rotation          IS   -7 (INT32) :
VAL e.invalid.colour            IS   -8 (INT32) :
VAL e.char.out.of.range         IS   -9 (INT32) :
VAL e.string.length.exceeded    IS  -10 (INT32) :
VAL e.invalid.colour.table      IS  -11 (INT32) :
}}}
{{{  PROC g.draw.line
PROC g.draw.line ([2]INT p0, p1)
  INT32 reply :
  SEQ
    to.graphic ! c.draw.line; INT32 (p0[0]); INT32 (p0[1]); INT32(p1[0]);
                        INT32(p1[1])
    from.graphic ? reply
:
}}}
{{{  PROC c.draw.line
PROC c.draw.line (VAL INT p0, p1, p2, p3)
  INT32 reply :
  SEQ
    to.graphic ! c.draw.line; INT32 (p0); INT32 (p1); INT32(p2);
                        INT32(p3)
    from.graphic ? reply
```

```
+-------------------------------------------------------------------------+
|   FILE: TGT.lis                                    SIZE:   32134 bytes  |
|  SAVED: Tue Jul 05 15:10:54 1988                   PAGE:       4        |
+-------------------------------------------------------------------------+

:
}}}
{{{  PROC g.send
PROC g.send (VAL INT32 command, VAL []INT params)
  --send a command and recieve reply
  INT32 reply :
  SEQ
    to.graphic ! command
    SEQ i = 0 FOR SIZE params
      to.graphic ! INT32 params[i]
    from.graphic ? reply
:
}}}
{{{  PROC g.send1
PROC g.send1 (VAL INT32 command)
  --send a command an recieve reply
  INT32 reply :
  SEQ
    to.graphic ! command
    from.graphic ? reply
:
}}}
{{{  PROC g.send2
PROC g.send2 (VAL INT32 com1, VAL INT com2)
  --send a two param command an receive reply
  INT32 reply :
  SEQ
    to.graphic ! com1; INT32 com2
    from.graphic ? reply
:
}}}
{{{  PROC init.graphics
PROC init.graphics()
  INT reply :
  SEQ
    {{{  B007 initialization
    g.send2(c.select.screen, 0)
    g.send2(c.clear.screen, 0)
    g.send2(c.display.screen, 0)
    g.send2 (c.select.colour.table, 1) --good primary scales
    g.send2 (c.select.bg.colour, 0)    --black
    g.send2 (c.select.fg.colour, 15)   --white
    window.selected := FALSE
    }}}
:
}}}
{{{  PROC init.db.graphics
PROC init.db.graphics()
  INT reply :
  SEQ
    {{{  B007 initialization
    g.send2 (c.select.colour.table, 1) --good primary scales
    g.send1 (c.flip.screen)            --switch to alternating screens
```

```
+-------------------------------------------------------------------------+
|   FILE: TGT.lis                                    SIZE:   32134 bytes  |
|  SAVED: Tue Jul 05 15:10:54 1988                   PAGE:       5        |
+-------------------------------------------------------------------------+

    g.send2 (c.select.bg.colour, 0)     --black
    g.send2 (c.select.fg.colour, 15)    --white
    g.send1 (c.flip.screen)
    window.selected := FALSE
    }}}
:
}}}
{{{  PROC finit.graphics
PROC finit.graphics ()
  INT32 reply :
  SEQ
    to.graphic ! c.terminate
    from.graphic ? reply
:
}}}
}}}

{{{  PROC select.screen
PROC select.screen(VAL INT screen.number)
  IF
    (screen.number = 0) OR (screen.number = 1)
      SEQ
        g.send2(c.select.screen, screen.number)
        window.selected := FALSE
    TRUE
      SKIP
:
}}}
{{{  PROC map.to.screen.coords
PROC map.to.screen.coords(VAL REAL32 x, y, INT x.screen, y.screen)
  --NOTE: x, y are in window coordinates, not viewport.
  REAL32 temp.x, temp.y:
  REAL32 dx, dy :
  SEQ
    {{{  COMMENT write statistics
    :::A COMMENT FOLD
    {{{  write statistics
    write.full.string(screen, "map.to.screen.coords: x = ")
    write.real32(screen, x, 5, 2)
    write.full.string(screen, "  ")
    write.full.string(screen, "map.to.screen.coords: y = ")
    write.real32(screen, y, 5, 2)
    newline(screen)
    }}}
    }}}
    IF
      window.selected
        REAL32 scale.x, scale.y :
        SEQ
          {{{  compute relative to window coords

          temp.x := ((x - windows[active.window][x.world.min]) *
                     windows[active.window][vp.scale.x])
```

```
+-------------------------------------------------------------------------+
|   FILE: TGT.lis                                        SIZE:   32134 bytes |
|  SAVED: Tue Jul 05 15:10:54 1988                       PAGE:       6       |
+-------------------------------------------------------------------------+

          temp.y := (windows[active.window][y.view.max] -
                     windows[active.window][y.view.min]) -
                     ((y - windows[active.window][y.world.min]) *
                     windows[active.window][vp.scale.y])

          x.screen := (INT TRUNC (temp.x * screen.width.r))
          y.screen := (INT TRUNC (temp.y * screen.height.r))
          }}}
        TRUE
          SEQ
            {{{  compute relative to screen coords
            {{{  window-to-viewport mapping
            temp.x := ((x -
                        windows[active.window][x.world.min]) *
                        windows[active.window][vp.scale.x]) +
                        windows[active.window][x.view.min]

            temp.y := rl.0 -
                     (((y -
                        windows[active.window][y.world.min]) *
                        windows[active.window][vp.scale.y]) +
                        windows[active.window][y.view.min])
            }}}
            {{{  viewport-to-screen mapping
            x.screen := (INT TRUNC (temp.x * screen.width.r))
            y.screen := (INT TRUNC (temp.y * screen.height.r))
            }}}
            }}}
      {{{  COMMENT write xformed coords
      :::A COMMENT FOLD
      {{{  write xformed coords
      write.full.string(screen, " x.screen = ")
      write.int(screen, x.screen, 0)
      write.full.string(screen, "  ")
      write.full.string(screen, " y.screen = ")
      write.int(screen, y.screen, 0)
      newline(screen)
      }}}
      }}}
:
}}}
{{{  PROC transform.points
PROC transform.points(VAL INT count, []REAL32 x, y)
  REAL32 temp.x:

  SEQ i = 0 FOR count
    SEQ
      temp.x := ((x[i]*trans.2d[0][0]) + (y[i]*trans.2d[1][0]))+ trans.2d[2][0
].
      y[i] := ((x[i]*trans.2d[0][1]) + (y[i]*trans.2d[1][1])) + trans.2d[2][1]

      x[i] := temp.x
```

```
+-------------------------------------------------------------------------+
|   FILE: TGT.lis                                    SIZE:   32134 bytes  |
|  SAVED: Tue Jul 05 15:10:54 1988                   PAGE:       7        |
+-------------------------------------------------------------------------+

:
}}}
{{{  PROC transform.point
PROC transform.point(REAL32 x, y)
  REAL32 temp.x:
  SEQ
    temp.x := ((x * trans.2d[0][0]) + (y * trans.2d[1][0]))+ trans.2d[2][0]
    y := ((x * trans.2d[0][1]) + (y * trans.2d[1][1])) + trans.2d[2][1]
    x := temp.x
:
}}}
{{{  transformation procedures, modify matrix trans.2d
{{{  PROC make.identity
PROC make.identity([3][3]REAL32 trans.matrix)
  SEQ i = 0 FOR 3
    SEQ j = 0 FOR 3
      IF
        i = j
          trans.matrix[i][j] := r1.0
        TRUE
          trans.matrix[i][j] := r0.0
:
}}}
{{{  PROC combine.transformations
PROC combine.transformations([3][3]REAL32 mat.a, mat.b)
  [3][3]REAL32 temp.matrix:
  SEQ
    SEQ i = 0 FOR 3
      SEQ j = 0 FOR 3
        temp.matrix[i][j] := (((mat.a[i][0] * mat.b[0][j]) +
                              (mat.a[i][1] * mat.b[1][j])) +
                              (mat.a[i][2] * mat.b[2][j]))
    SEQ i = 0 FOR 3
      SEQ j = 0 FOR 3
        mat.a[i][j] := temp.matrix[i][j]
:
}}}
{{{  PROC scale
PROC scale(VAL REAL32 scale.x, scale.y, x.fixed, y.fixed)
  [3][3]REAL32 temp.matrix :
  REAL32 s.x, s.y, x.f, y.f :

  SEQ
    s.x := scale.x
    s.y := scale.y
    x.f := x.fixed
    y.f := y.fixed
    make.identity(temp.matrix)
    temp.matrix[0][0] := s.x
    temp.matrix[1][1] := s.y
    temp.matrix[2][0] := (r1.0 - s.x) * x.f
    temp.matrix[2][1] := (r1.0 - s.y) * y.f
    combine.transformations(trans.2d, temp.matrix)
```

```
+-------------------------------------------------------------------------+
|   FILE: TGT.lis                                    SIZE:   32134 bytes  |
|  SAVED: Tue Jul 05 15:10:54 1988                   PAGE:       8        |
+-------------------------------------------------------------------------+

:
}}}
{{{  COMMENT PROC rotate (D700C version)
:::A COMMENT FOLD
{{{  PROC rotate (D700C version)
PROC rotate(VAL REAL32 alpha, x.pivot, y.pivot)
  #USE "\tds2trig\sincos.tsr"
  [3][3]REAL32 temp.matrix:
  REAL32 alph, cos.alpha, sin.alpha:

  {{{  PROC radian.equiv
  PROC radian.equiv(REAL32 angle)
    angle := angle * ((3.14159265 (REAL32)) / (180.0 (REAL32)))
  :
  }}}
  SEQ
    alph := alpha
    make.identity(temp.matrix)
    radian.equiv(alph)
    COSP(cos.alpha, alph)
    SINP(sin.alpha, alph)
    temp.matrix[0][0] := cos.alpha
    temp.matrix[0][1] := sin.alpha
    temp.matrix[1][0] := (-sin.alpha)
    temp.matrix[1][1] := cos.alpha
    temp.matrix[2][0] := (x.pivot * (r1.0 - cos.alpha)) + (y.pivot * sin.alpha
)
    temp.matrix[2][1] := (y.pivot * (r1.0 - cos.alpha)) - (x.pivot * sin.alpha
)
    combine.transformations(trans.2d, temp.matrix)
:
}}}
}}}
{{{  PROC rotate (D700D version)
{{{  FUNCTION radian.equiv
REAL32 FUNCTION radian.equiv(VAL REAL32 angle)
  VAL degrees.to.radians IS (3.14159265 (REAL32)) / (180.0 (REAL32)) :
  REAL32 result :
  VALOF
    result := angle * degrees.to.radians
    RESULT result
:
}}}
PROC rotate(VAL REAL32 alpha, x.pivot, y.pivot)
  #USE snglmath
  [3][3]REAL32 temp.matrix:
  REAL32 angle, cos.alpha, sin.alpha:

  SEQ
    make.identity(temp.matrix)
    angle := radian.equiv(alpha)
    cos.alpha := COS(angle)
    sin.alpha := SIN(angle)
```

```
+-------------------------------------------------------------------------+
|   FILE: TGT.lis                                        SIZE:   32134 bytes |
|  SAVED: Tue Jul 05 15:10:54 1988                       PAGE:       9       |
+-------------------------------------------------------------------------+

    temp.matrix[0][0] := cos.alpha
    temp.matrix[0][1] := sin.alpha
    temp.matrix[1][0] := (-sin.alpha)
    temp.matrix[1][1] := cos.alpha
    temp.matrix[2][0] := (x.pivot * (r1.0 - cos.alpha)) + (y.pivot * sin.alpha
)
    temp.matrix[2][1] := (y.pivot * (r1.0 - cos.alpha)) - (x.pivot * sin.alpha
)
    combine.transformations(trans.2d, temp.matrix)
:
}}}
{{{  PROC translate
PROC translate(VAL REAL32 translate.x, translate.y)
  [3][3]REAL32 temp.matrix:
  REAL32 t.x, t.y:

  SEQ
    t.x := translate.x     -- assignable statements
    t.y := translate.y
    make.identity(temp.matrix)
    temp.matrix[2][0] := t.x
    temp.matrix[2][1] := t.y
    combine.transformations(trans.2d, temp.matrix)
:
}}}
}}}
{{{  clipping procedures
{{{  PROC clip.line.2d
PROC clip.line.2d(REAL32 x1, y1, x2, y2,
                  BOOL display)

  {{{  COMMENT Cohen-Southerland clipping algorithm
  :::A COMMENT FOLD
  {{{  Cohen-Southerland clipping algorithm
  Cohen-Southerland clipping algorithm for line P1 = (x1, y1) to
  P2 = (x2, y2).
  }}}
  }}}

  {{{  PROC SC.code.2d
  PROC SC.code.2d(REAL32 x, y, INT outcode)
    SEQ
      outcode := 0
      IF
        {{{  point left of window
        x < windows[active.window][x.world.min]
          outcode := outcode \/ 1
        }}}
        {{{  SKIP
        TRUE
          SKIP
        }}}
      IF
```

```
+-------------------------------------------------------------------------------+
|   FILE: TGT.lis                                          SIZE:   32134 bytes  |
|  SAVED: Tue Jul 05 15:10:54 1988                         PAGE:      10        |
+-------------------------------------------------------------------------------+

        {{{  point right of window
        x > windows[active.window][x.world.max]
          outcode := outcode \/ 2
        }}}
        {{{  SKIP
        TRUE
          SKIP
        }}}
      IF
        {{{  point below window
        y < windows[active.window][y.world.min]
          outcode := outcode \/ 4
        }}}
        {{{  SKIP
        TRUE
          SKIP
        }}}
      IF
        {{{  point above window
        y > windows[active.window][y.world.max]
          outcode := outcode \/ 8
        }}}
        {{{  SKIP
        TRUE
          SKIP
        }}}
  :
  }}}
  {{{  PROC reject.check
  PROC reject.check(VAL INT outcode1, outcode2, BOOL reject)
    SEQ
      IF
        (outcode1 /\ outcode2) <> 0
          reject := TRUE
        TRUE
          reject := FALSE
  :
  }}}
  {{{  PROC accept.check
  PROC accept.check(VAL INT outcode1, outcode2, BOOL accept)
    SEQ
      IF
        (outcode1 \/ outcode2) = 0
          accept := TRUE
        TRUE
          accept := FALSE
  :
  }}}
  {{{  PROC exchange
  PROC exchange(REAL32 x1, y1, x2, y2, INT outcode1, outcode2)
    [2]REAL32 temp:
    INT temp.mask:
```

```
+-------------------------------------------------------------------------+
|   FILE: TGT.lis                                    SIZE:   32134 bytes  |
|  SAVED: Tue Jul 05 15:10:54 1988                   PAGE:      11        |
+-------------------------------------------------------------------------+

    SEQ
      temp[0] := x1
      temp[1] := y1
      x1 := x2
      y1 := y2
      x2 := temp[0]
      y2 := temp[1]
      temp.mask := outcode1
      outcode1 := outcode2
      outcode1 := temp.mask
  :
  }}}
  {{{  clipping routine
  BOOL done, accept, reject:
  INT outcode1, outcode2:
  REAL32 slope:

  SEQ
    done := FALSE
    WHILE NOT done
      SEQ
        {{{  get position codes for both points
        SC.code.2d(x1, y1, outcode1)
        SC.code.2d(x2, y2, outcode2)
        accept.check(outcode1, outcode2, accept)
        }}}
        IF
          {{{  trivial accept check
          accept
            SEQ
              done := TRUE
              display := TRUE
          }}}
          TRUE
            SEQ
              reject.check(outcode1, outcode2, reject)
              IF
                {{{  trivial reject check
                reject
                  SEQ
                    done := TRUE
                    display := FALSE
                }}}
                TRUE
                  SEQ
                    {{{  make sure (x1, y1) is outside window
                    IF
                      {{{  switch points so (x1, y1) is outside window
                      outcode1 = 0
                        exchange(x1, y1, x2, y2, outcode1, outcode2)
                      }}}
                      TRUE
                        SKIP
```

```
+-------------------------------------------------------------------------+
|  FILE: TGT.lis                                        SIZE:   32134 bytes |
| SAVED: Tue Jul 05 15:10:54 1988                       PAGE:       12      |
+-------------------------------------------------------------------------+

                    }}}
                    IF
                      {{{  vertical line
                      x1 = x2
                        SEQ
                          IF
                            {{{  line completely through window
                            (outcode1 \/ outcode2) = 6
                              IF
                                {{{  y2 > y1
                                y2 > y1
                                  SEQ
                                    y2 := windows[active.window][y.world.max]
                                    y1 := windows[active.window][y.world.min]
                                }}}
                                {{{  y1 < y2
                                TRUE
                                  SEQ
                                    y1 := windows[active.window][y.world.max]
                                    y2 := windows[active.window][y.world.min]
                                }}}
                            }}}
                            {{{  line extends above window
                            outcode1 = 8
                              y1 := windows[active.window][y.world.max]
                            }}}
                            {{{  line extends below window
                            outcode1 = 6
                              y1 := windows[active.window][y.world.min]
                            }}}
                            {{{  SKIP
                            TRUE
                              SKIP
                            }}}
                      }}}
                      {{{  any other line
                      TRUE
                        SEQ
                          slope := (y2 - y1) / (x2 - x1)
                          IF
                            {{{  left of window
                            (outcode1 /\ 1) <> 0
                              SEQ
                                y1 := y1 + ((windows[active.window][x.world.mi
n] - x1) * slope)
                                x1 := windows[active.window][x.world.min]
                            }}}
                            {{{  right of window
                            (outcode1 /\ 2) <> 0
                              SEQ
                                y1 := y1 + ((windows[active.window][x.world.ma
x] - x1) * slope)
                                x1 := windows[active.window][x.world.max]
```

```
+-------------------------------------------------------------------------+
|   FILE: TGT.lis                                          SIZE:   32134 bytes |
|  SAVED: Tue Jul 05 15:10:54 1988                         PAGE:      13      |
+-------------------------------------------------------------------------+

                              }}}
                              {{{  below window
                              (outcode1 /\ 4) <> 0
                                SEQ
                                  x1 := x1 + ((windows[active.window][y.world.mi
n] - y1) / slope)
                                  y1 := windows[active.window][y.world.min]
                              }}}
                              {{{  above window
                              (outcode1 /\ 8) <> 0
                                SEQ
                                  x1 := x1 + ((windows[active.window][y.world.ma
x] - y1) / slope)
                                  y1 := windows[active.window][y.world.max]
                              }}}
                              TRUE
                                SKIP
                        }}}
  }}}
:
}}}
{{{  PROC clip.point.2d
PROC clip.point.2d(VAL REAL32 x, y, BOOL display)
  SEQ
    IF
      ((x < windows[active.window][x.world.min]) OR
       (x > windows[active.window][x.world.max]) OR
       (y < windows[active.window][y.world.min]) OR
       (y > windows[active.window][y.world.max]))
        display := FALSE
      TRUE
        SKIP
:
}}}
}}}
{{{  PROC set.window.2d
PROC set.window.2d(VAL REAL32 x.min, y.min, x.max, y.max,
                   VAL INT window.num)
  IF
    window.num < max.windows
      SEQ
        windows[window.num][x.world.min] := x.min
        windows[window.num][x.world.max] := x.max
        windows[window.num][y.world.min] := y.min
        windows[window.num][y.world.max] := y.max
    TRUE
      SKIP
:
}}}
{{{  PROC set.viewport.2d
PROC set.viewport.2d(VAL REAL32 x.min, y.min, x.max, y.max,
                     VAL INT viewport.num)
  SEQ
```

```
+-------------------------------------------------------------------------------+
|   FILE: TGT.lis                                          SIZE:   32134 bytes  |
|  SAVED: Tue Jul 05 15:10:54 1988                         PAGE:      14        |
+-------------------------------------------------------------------------------+

    IF
      (x.min < r0.0) OR (y.min < r0.0) OR (x.max > r1.0) OR (y.max > r1.0)
        SKIP
      TRUE
        INT x.size, y.size :
        INT x.win.min, y.win.min :
        INT x.win.max, y.win.max :
        SEQ
          active.window := viewport.num  -- for map.to.screen.coords
          window.selected := FALSE -- don't need to save current val.
          windows[viewport.num][vp.scale.x] := (x.max - x.min) /
                        (windows[viewport.num][x.world.max] -
                         windows[viewport.num][x.world.min])
          windows[viewport.num][vp.scale.y] := (y.max - y.min) /
                        (windows[viewport.num][y.world.max] -
                         windows[viewport.num][y.world.min])
          windows[viewport.num][x.view.min] := x.min
          windows[viewport.num][x.view.max] := x.max
          windows[viewport.num][y.view.min] := y.min
          windows[viewport.num][y.view.max] := y.max
          map.to.screen.coords(windows[viewport.num][x.world.min],
                               windows[viewport.num][y.world.min],
                               x.win.min, y.win.min)
          map.to.screen.coords(windows[viewport.num][x.world.max],
                               windows[viewport.num][y.world.max],
                               x.win.max, y.win.max)
          x.size := x.win.max - x.win.min
          y.size := y.win.min - y.win.max

          to.graphic ! c.set.window; x.size; y.size
          from.graphic ? b007.win.num[viewport.num]
          window.selected := TRUE
:
}}}
{{{  PROC activate.viewport.2d
PROC activate.viewport.2d(VAL INT viewport.number)
  SEQ
    active.window := viewport.number
    g.send2(c.select.window, b007.win.num[viewport.number])
:
}}}
{{{  PROC display.viewport.2d
PROC display.viewport.2d(VAL INT viewport)
  INT x.min, y.max :
  INT old.active.window :
  BOOL old.window.selected :
  SEQ
    old.active.window := active.window
    active.window := viewport
    old.window.selected := window.selected
    window.selected := FALSE  -- display window in screen coords
    map.to.screen.coords(windows[viewport][x.world.min],
                         windows[viewport][y.world.max],
```

```
                          x.min, y.max)
    g.send(c.display.window, [b007.win.num[viewport], x.min, y.max])
    active.window := old.active.window
    window.selected := old.window.selected
:
}}}
{{{  PROC move.viewport.position.2d
PROC move.viewport.position.2d(VAL INT viewport.number,
                               VAL REAL32 x.min, y.min)
  REAL32 dx, dy :
  SEQ
    dx := windows[viewport.number][x.view.max] -
          windows[viewport.number][x.view.min]
    dy := windows[viewport.number][y.view.max] -
          windows[viewport.number][y.view.min]
    windows[viewport.number][x.view.min] := x.min
    windows[viewport.number][y.view.min] := y.min
    windows[viewport.number][x.view.max] := x.min + dx
    windows[viewport.number][y.view.max] := y.min + dy
    --display.viewport(viewport.number)
:
}}}
{{{  absolute, relative, and cursor commands
{{{  PROC move.abs.2d
PROC move.abs.2d(VAL REAL32 x, y)
  SEQ
    windows[active.window][pen.x] := x
    windows[active.window][pen.y] := y
:
}}}
{{{  PROC move.rel.2d
PROC move.rel.2d(VAL REAL32 dx, dy)
  SEQ
    windows[active.window][pen.x] := windows[active.window][pen.x] + dx
    windows[active.window][pen.y] := windows[active.window][pen.y] + dy
:
}}}
{{{  PROC point.abs.2d
PROC point.abs.2d(VAL REAL32 x, y)
  BOOL display:
  INT x.screen, y.screen:
  SEQ
    --clip.point.2d(x, y, display)
    display := TRUE
    IF
      display
        SEQ
          map.to.screen.coords(x, y, x.screen, y.screen)
          g.send(c.plot.point, [x.screen, y.screen])
      TRUE
        SKIP
:
}}}
```

```
+-------------------------------------------------------------------------+
|   FILE: TGT.lis                                    SIZE:   32134 bytes  |
|  SAVED: Tue Jul 05 15:10:54 1988                   PAGE:      16        |
+-------------------------------------------------------------------------+

{{{  PROC point.rel.2d
PROC point.rel.2d(VAL REAL32 dx, dy)
  BOOL display:
  REAL32 x, y:
  INT x.screen, y.screen:
  SEQ
    move.rel.2d(dx, dy)
    x := windows[active.window][pen.x]
    y := windows[active.window][pen.y]
    -- clip.point.2d(x, y, display)
    display := TRUE
    IF
      display
        SEQ
          map.to.screen.coords(x, y, x.screen, y.screen)
          g.send(c.plot.point, [x.screen, y.screen])
      TRUE
        SKIP
:
}}}
{{{  PROC line.abs.2d
PROC line.abs.2d(VAL REAL32 x, y)
  BOOL display:
  REAL32 x1, y1,
         x2, y2:
  [2]INT point1, point2:
  SEQ
    x2 := x
    y2 := y
    x1 := windows[active.window][pen.x]
    y1 := windows[active.window][pen.y]
    move.abs.2d(x, y)                          -- update cursor position
    -- clip.line.2d(x1, y1, x2, y2, display)
    display := TRUE
    IF
      display
        SEQ
          map.to.screen.coords(x1, y1, point1[0], point1[1])
          map.to.screen.coords(x2,  y2, point2[0], point2[1])
          g.draw.line(point1, point2)
      TRUE
        SKIP
:
}}}
{{{  PROC line.rel.2d
PROC line.rel.2d(VAL REAL32 dx, dy)
  BOOL display:
  REAL32 x1, y1,
         x2, y2:
  [2]INT point1, point2:
  SEQ
    x1 := windows[active.window][pen.x]
    y1 := windows[active.window][pen.y]
```

```
    move.rel.2d(dx, dy)
    x2 := windows[active.window][pen.x]
    y2 := windows[active.window][pen.y]
    -- clip.line.2d(x1, y1, x2, y2, display)
    display := TRUE
    IF
      display
        SEQ
          map.to.screen.coords(x1, y1, point1[0], point1[1])
          map.to.screen.coords(x2, y2, point2[0], point2[1])
          g.draw.line(point1, point2)
      TRUE
        SKIP
:
}}}
}}}
{{{  graphics primitives
{{{  PROC draw.line.2d
PROC draw.line.2d(VAL REAL32 x1, y1, x2, y2)
  [2]INT point1, point2 :
  REAL32 tx1, ty1, tx2, ty2 :  -- assignable parameters
  SEQ
    tx1 := x1
    ty1 := y1
    tx2 := x2
    ty2 := y2
    --clip.line.2d(tx1, ty1, tx2, ty2, display)
    display := TRUE
    IF
      display
        SEQ
          map.to.screen.coords(x1, y1, point1[0], point1[1])
          map.to.screen.coords(x2, y2, point2[0], point2[1])
          g.draw.line(point1, point2)
      TRUE
        SKIP
:
}}}
{{{  PROC draw.rectangle.2d
PROC draw.rectangle.2d(VAL REAL32 x, y, x.length, y.length)
  INT x.screen, y.screen :
  INT x.len.scn, y.len.scn :
  REAL32 scale.x, scale.y :
  SEQ
    map.to.screen.coords(x, y, x.screen, y.screen)
    map.to.screen.coords(x.length, y.length, x.len.scn, y.len.scn)
    {{{  scale sides
    scale.x := x.length / (windows[active.window][x.world.max] -
                           windows[active.window][x.world.min])
    x.len.scn := (INT TRUNC (scale.x *
                  ((windows[active.window][x.view.max]-
                    windows[active.window][x.view.min]) *
                    screen.width.r)))
```

```
+-------------------------------------------------------------------------+
|  FILE: TGT.lis                                        SIZE:   32134 bytes |
| SAVED: Tue Jul 05 15:10:54 1988                       PAGE:       18      |
+-------------------------------------------------------------------------+

    scale.y := y.length / (windows[active.window][x.world.max] -
                           windows[active.window][x.world.min])
    y.len.scn := (INT TRUNC (scale.y *
                  ((windows[active.window][y.view.max]-
                    windows[active.window][y.view.min]) *
                    screen.width.r)))
    }}}
    g.send(c.draw.rectangle, [x.screen, y.screen, x.len.scn, y.len.scn])
:
}}}
{{{  PROC draw.polygon.2d
PROC draw.polygon.2d(VAL INT num.sides,
                     VAL []REAL32 buffer)
  [max.points]INT x.tmp, y.tmp :
  INT count :
  INT reply :
  SEQ
    count := 0
    SEQ i = 0 FOR num.sides
      SEQ
        map.to.screen.coords(buffer[count], buffer[count+1], x.tmp[i], y.tmp[i
])
        count := count + 2
    to.graphic ! c.draw.polygon; num.sides
    SEQ i = 0 FOR num.sides
      to.graphic ! x.tmp[i]; y.tmp[i]
    from.graphic ? reply
:
}}}
{{{  PROC draw.circle.2d
PROC draw.circle.2d(VAL REAL32 x.center, y.center, radius)
  INT x.cen, y.cen, rad :
  REAL32 scale.x :
  SEQ
    map.to.screen.coords(x.center, y.center, x.cen, y.cen)
    {{{  scale radius
    scale.x := radius / (windows[active.window][x.world.max] -
                         windows[active.window][x.world.min])
    rad := (INT TRUNC (scale.x *
                  ((windows[active.window][x.view.max]-
                    windows[active.window][x.view.min]) *
                    screen.width.r)))
    }}}
    g.send(c.draw.circle, [x.cen, y.cen, rad])
:
}}}
{{{  PROC draw.arc.2d
PROC draw.arc.2d(VAL REAL32 x1, y1, x2, y2, x3, y3)
  INT ix1, iy1, ix2, iy2, ix3, iy3 :
  SEQ
    map.to.screen.coords(x1, y1, ix1, iy1)
    map.to.screen.coords(x2, y2, ix2, iy2)
    map.to.screen.coords(x3, y3, ix3, iy3)
```

```
    {{{  COMMENT write statements
    :::A COMMENT FOLD
    {{{  write statements
    write.int(screen, ix1, 0)
    write.int(screen, iy1, 0)
    write.int(screen, ix2, 0)
    write.int(screen, iy2, 0)
    write.int(screen, ix3, 0)
    write.int(screen, iy3, 0)
    }}}
    }}}
    g.send(c.draw.arc, [ix1, iy1, ix2, iy2, ix3, iy3])
:
}}}
}}}
{{{  misc. screen and window routines
{{{  PROC flip.screen
PROC flip.screen()
  g.send1(c.flip.screen)
:
}}}
{{{  PROC activate.screen
PROC activate.screen(VAL INT screen.number)
  SEQ
    g.send2(c.select.screen, screen.number)
    window.selected := FALSE
:
}}}
{{{  PROC display.screen
PROC display.screen(VAL INT screen.number)
  g.send2(c.select.screen, screen.number)
:
}}}
{{{  PROC clear.screen
PROC clear.screen(VAL INT colour)
  g.send2(c.clear.screen, colour)
:
}}}
{{{  PROC select.colour.table
PROC select.colour.table(VAL INT number)
  g.send2(c.select.colour.table, number)
:
}}}
{{{  PROC set.colour
PROC set.colour(VAL INT entry, red, green, blue)
  g.send(c.set.colour, [entry, red, green, blue])
:
}}}
{{{  PROC fg.colour
PROC fg.colour(VAL INT entry)
  g.send2(c.select.fg.colour, entry)
:
}}}
```

```
+-------------------------------------------------------------------------------+
|   FILE: TGT.lis                                          SIZE:   32134 bytes  |
|  SAVED: Tue Jul 05 15:10:54 1988                         PAGE:      20        |
+-------------------------------------------------------------------------------+

{{{  PROC bg.colour
PROC bg.colour(VAL INT entry)
  g.send2(c.select.bg.colour, entry)
:
}}}
{{{  PROC int.line
PROC int.line(VAL INT x1, y1, x2, y2)
  c.draw.line(x1, y1, x2, y2)
:
}}}
{{{  PROC clear.window
PROC clear.window(VAL INT colour)
  g.send2(c.clear.window, colour)
:
}}}
{{{  PROC pixel.line
PROC pixel.line(VAL INT size, []INT buffer)
  -- assumes buffer is packed (x, y) from a 2d array
  INT reply :
  SEQ
    to.graphic ! c.pixel.line; size::buffer
    from.graphic ? reply
:
}}}
{{{  PROC colour.line
PROC colour.line(VAL INT size, []INT buffer)
  -- assumes buffer is packed (x, y, color) from a 2d array
  INT reply :
  SEQ
    to.graphic ! c.color.line; size::buffer
    from.graphic ? reply
:
}}}
{{{  PROC fill.polygon
PROC fill.polygon(VAL INT x, y)
  g.send(c.fill.polygon, [x, y])
:
}}}
{{{  PROC quick.fill.polygon
PROC quick.fill.polygon(VAL INT x, y)
  g.send(c.quick.fill, [x, y])
:
}}}
{{{  PROC fill.polygon.2d
PROC fill.polygon.2d(VAL REAL32 x, y)
  INT i.x, i.y :
  SEQ
    map.to.screen.coords(x, y, i.x, i.y)
    g.send(c.fill.polygon, [i.x, i.y])
:
}}}
{{{  PROC quick.fill.polygon.2d
PROC quick.fill.polygon(VAL REAL32 x, y)
```

```
+-------------------------------------------------------------------------------+
|   FILE: TGT.lis                                          SIZE:   32134 bytes  |
|  SAVED: Tue Jul 05 15:10:54 1988                         PAGE:      21        |
+-------------------------------------------------------------------------------+

  INT i.x, i.y :
  SEQ
    map.to.screen.coords(x, y, i.x, i.y)
    g.send(c.quick.fill, [i.x, i.y])
:
}}}
}}}
}}}
```

REFERENCES

1. IMS B007 Evaluation Board User Manual. INMOS Corp., Colorado Springs, CO, 1986.

2. Ellis, G.K.: Two-Dimensional Graphics Tools for a Transputer Based Display Board. NASA TM-100820, 1988.

3. Foley, J.D.; and Van Dam, A.: Fundamentals of Interactive Computer Graphics. Addison-Wesley, Reading, MA, 1982.

4. Hearn, D.; and Baker, M.P.: Computer Graphics. Prentice-Hall, 1986.

5. Newman, W.M.; and Sproull, R.F.: Principles of Interactive Computer Graphics. 2nd ed., McGraw-Hill, 1979.

6. Transputer Development System 2.0 User Manual. 72 TDS 111000, INMOS Corp., Colorado Springs, CO, 1987.

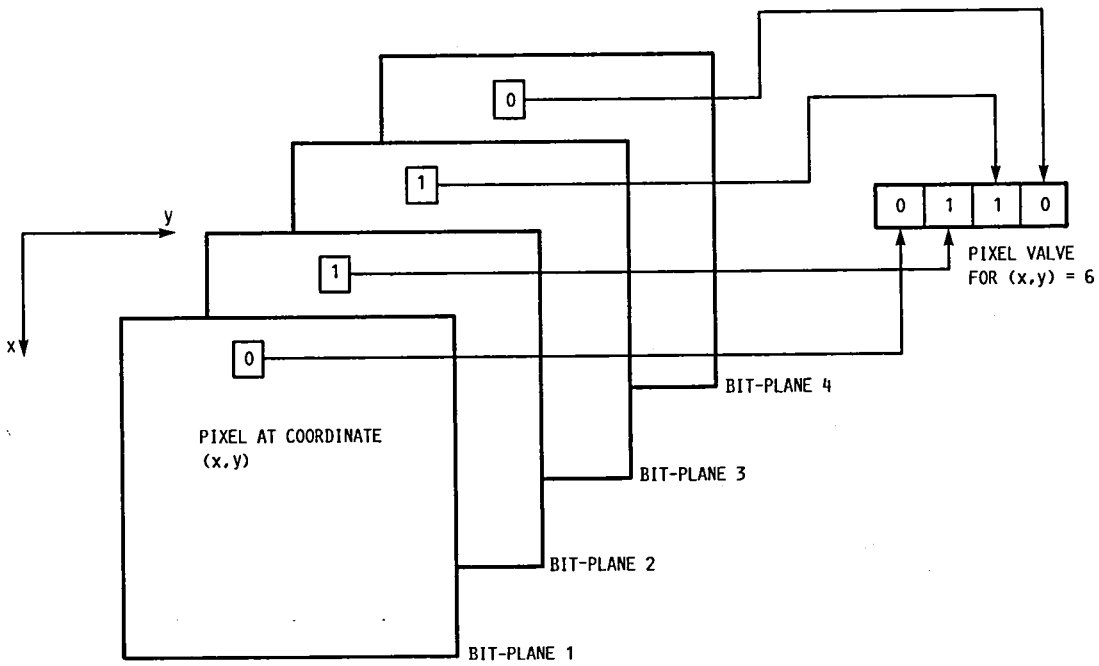

FIGURE 1. - EXAMPLE OF A FOUR BIT-PLANE SCREEN MEMORY.

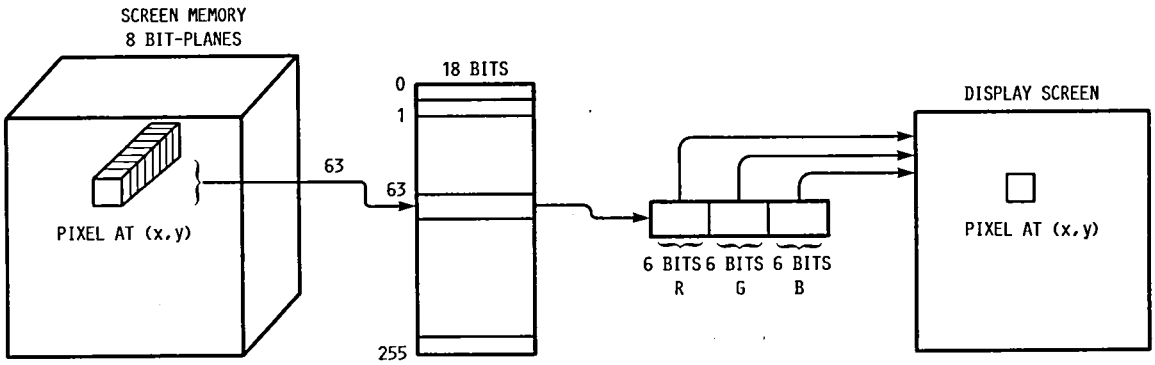

FIGURE 2. - COLOR LOOK-UP TABLE OPERATION.

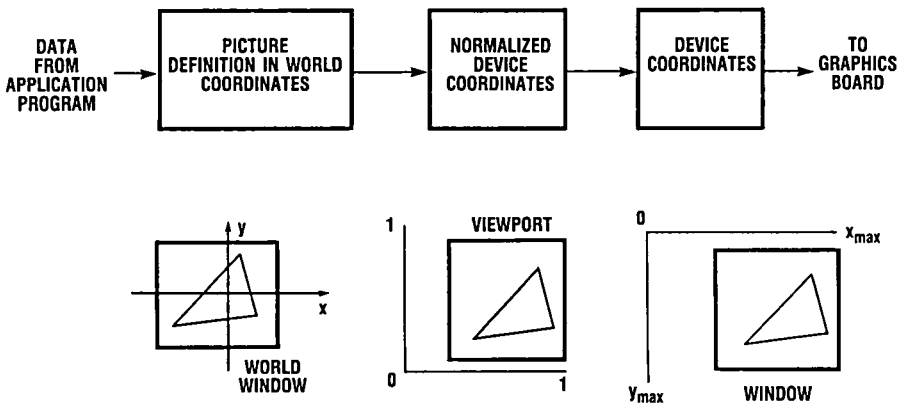

FIGURE 3. - MAPPING FROM WORLD COORDINATE SYSTEM TO NORMALIZED DEVICE COORDINATES TO INTEGER DEVICE COORDINATES.

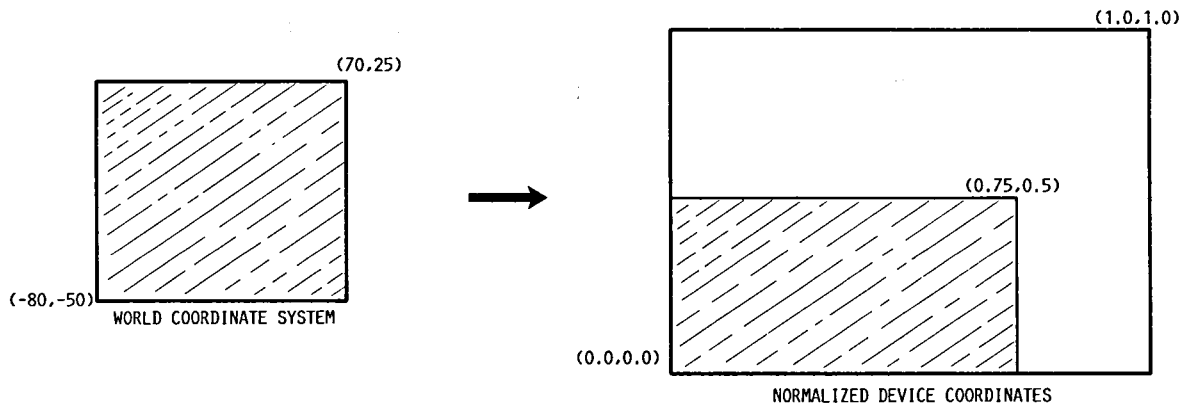

FIGURE 4. - SCREEN, WINDOW AND VIEWPORT RELATIONS FOR THE WINDOW SETUP EXAMPLE.

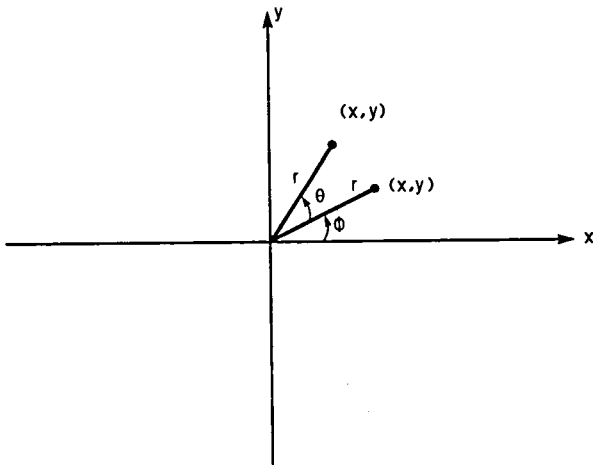

FIGURE 5. - ROTATION OF A POINT ABOUT THE ORIGIN.

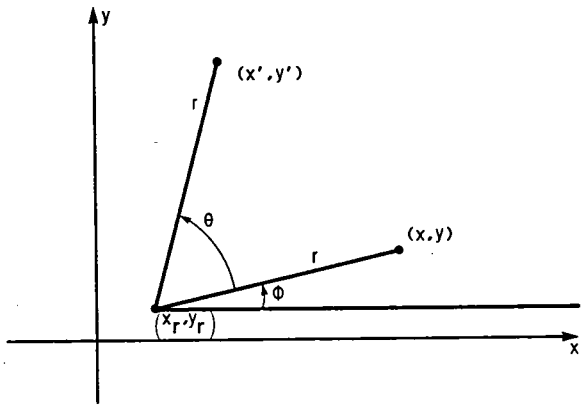

FIGURE 6. - ROTATION OF A POINT ABOUT AN ARBITRARY POINT.

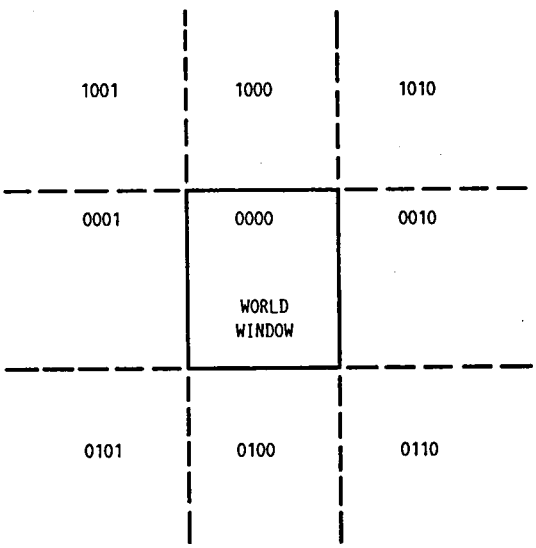

FIGURE 7. - THE NINE CLIPPING REGIONS AND THEIR RESPECTIVE CLIPPING CODES.

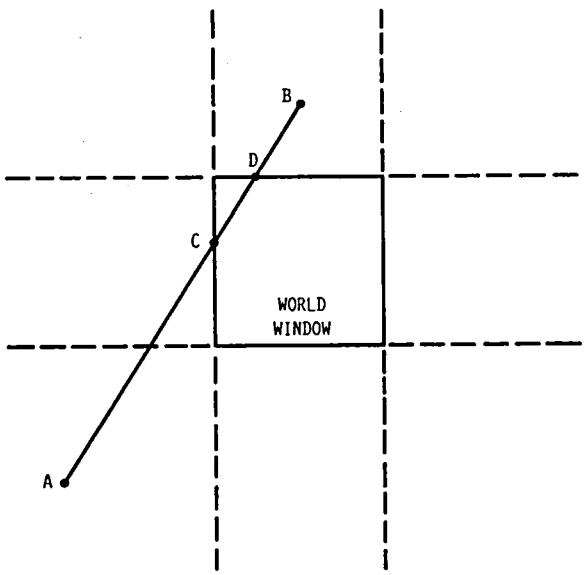

FIGURE 8. - LINE CLIPPING FROM LINE AB TO CD.

NASA
National Aeronautics and Space Administration

Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA TM-100974 ICOMP-88-13		

4. Title and Subtitle
User's Manual for the Two-Dimensional Transputer Graphics Toolkit

5. Report Date
August 1988

6. Performing Organization Code

7. Author(s)
Graham K. Ellis

8. Performing Organization Report No.
E-4266

10. Work Unit No.
505-63-1B

9. Performing Organization Name and Address
National Aeronautics and Space Administration
Lewis Research Center
Cleveland, Ohio 44135-3191

11. Contract or Grant No.

13. Type of Report and Period Covered
Technical Memorandum

12. Sponsoring Agency Name and Address
National Aeronautics and Space Administration
Washington, D.C. 20546-0001

14. Sponsoring Agency Code

15. Supplementary Notes
Graham K. Ellis, Senior Research Associate at the Institute for Computational Mechanics in Propulsion, NASA Lewis Research Center (work funded under Space Act Agreement C99066G).

16. Abstract
The user manual for the two-dimensional graphics toolkit for a transputer based parallel processor is presented. The toolkit consists of a package of two-dimensional display routines that can be used for simulation visualizations. It supports multiple windows, double buffered screens for animations, and simple graphics transformations such as translation, rotation, and scaling. The display routines are written in occam to take advantage of the multiprocessing features available on transputers. The package is designed to run on a transputer separate from the graphics board.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
Graphics Transputer Parallel processing	Unclassified - Unlimited Subject Category 61

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No of pages	22. Price*
Unclassified	Unclassified	104	A06

*For sale by the National Technical Information Service, Springfield, Virginia 22161