NASA Technical Memorandum 100820
ICOMP-88-4

Two-Dimensional Graphics Tools for a Transputer Based Display Board

Graham K. Ellis
Lewis Research Center
Cleveland, Ohio

Prepared for the
Parallel Computing Workshop/OCCAM User Group Meeting
cosponsored by OSU/OCATE and INMOS
Portland, Oregon, April 11-13, 1988

NASA

INSTITUTE FOR COMPUTATIONAL MECHANICS IN PROPULSION
NASA LEWIS RESEARCH CENTER
ICOMP
CASE WESTERN RESERVE UNIVERSITY

TWO-DIMENSIONAL GRAPHICS TOOLS FOR A TRANSPUTER BASED DISPLAY BOARD

Graham K. Ellis*
Institute for Computational Mechanics in Propulsion
NASA Lewis Research Center
Cleveland, Ohio 44135

ABSTRACT

A package of two-dimensional graphics routines has been developed in an effort to standardize and simplify the user interface for a transputer based graphics display board. The routines available take advantage of the graphics board's capabilities while also presenting an intuitive approach for generating drawings. The routines allow a user to perform graphics rendering in a two-dimensional real-coordinate space without regard to the actual screen coordinates. Multiple windows, which can be placed arbitrarily on the screen, as well as the ability to use double-buffering techniques for smooth animations are also supported.

The routines are designed to be run on a transputer other than the graphics display board. The window and screen parameters are maintained locally. The conversion to device coordinates is also performed locally. The only data sent to the display board are control and device coordinate display commands.

The routines available include: rotation, translation, and scaling commands; absolute and relative point and line commands; circle, rectangle and polygon commands; and window and viewport definition commands.

---

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

INTRODUCTION

The ability to render graphics images is important for interpreting the results of many types of scientific and engineering simulations. Because the INMOS IMS B007 transputer-based graphics board[1] is supplied with simple routines that are based on integer device coordinates, a set of routines, the Transputer Graphics Package (TGP), has been developed that allows users to work in their own coordinate system. All TGP procedures are written in occam and British spellings are used throughout for compatibility reasons. TGP allows images to be described in two-dimensional real-coordinate space, the world coordinate system (WCS). The routines provided support all of the B007 graphics commands including multiple windows and screen double-buffering. Additionally, routines have been developed that allow translation, rotation and scaling of 2D objects.

The current implementation of TGP is designed to run on a processor other than the B007 graphics board. The connection architecture of the TGP-B007 system is discussed below.

The routines available in TGP free the application programmer from many of the details of the operation of the B007 display board. Most of the procedures provided perform the channel I/O automatically without user intervention. The low-level integer device coordinate B007 commands are also hidden from the user. However, in order to take full advantage of the TGP's capabilities, an understanding of the B007's operation is essential.

In addition to the TGP, two new B007 graphics commands have been developed. These new commands allow block transfers of pixel coordinate data to be sent to the B007 from the TGP. These new protocols reduce I/O overhead and allow increased graphics rendering performance using the B007.

This document discusses the configuration and operation of TGP. Some of the capabilities and limitations of both the B007 graphics board and the 2D display routines in TGP will also be discussed below.

TGP-B007 CONFIGURATON ARCHITECTURE

The current implementation of the TGP uses two processors in a pipeline: the TGP processor and the B007 graphics board. The TGP processor takes data from an application program in two-dimensional world coordinates and converts it to integer device coordinates (IDC). The IDC are sent to B007 graphics board and the B007 performs scan-conversion and display tasks. A block diagram of the processor configuration is shown in Figure 1.

The conversion of 2D world data into IDC on the TGP is performed

in three steps. The first step is to read or store the 2D model data in an array or user generated primitive data base. The user defines a world window in 2D world coordinates. The user then specifies the viewport size on the CRT screen that the world window will be mapped into. The placement of the world window into a viewport is specified in normalized device coordinates (NDC). The NDC are from (0,0) to (1,1) with the origin at the lower left corner of the screen. The advantage to placing viewports in this manner is that the actual screen resolution never appears in any of the user graphics calls. The last step is to convert from NDC to IDC. This transformation is totally transparent to the user. A tranformation from NDC to IDC is never invoked directly from a user procedure call. A block diagram of the 2D world coordinate to IDC conversion is shown in Figure 2.

The current implementation of the WCS to IDC conversion as shown above is not easily mapped onto multiple processors, for example, a pipeline of three processors. This is due to the sharing of global window and parametric data among the three stages shown in Figure 2. Intelligent buffer routines would be required to keep copies of relevant data on each processor in order to distribute this portion of the TGP routines onto several processors.

IMPLEMENTATION

The mathematics behind the two-dimensional graphics transformations used in TGP are well documented [2-5] and will not be discussed here. What will be covered is the method of handling the screen and window variables in the graphics tools package as well as how the global transformation matrix is used.

Because the graphics tool package runs on a processor other than the B007 display board, the window and screen variables such as world coordinates, viewport coordinates (position on the screen), and graphics cursor coordinates are maintained locally in the TGP and the only information sent to the B007 are the hardware control and device coordinate display commands.

In order to simplify the user interface, i.e. the procedure calls, many of the window and screen variables are hidden from the user in global variables. Because of the way Occam handles storage, a decision was made to simplify the parameter lists of the various procedures by modifying global variables from within a procedure without rescoping or putting additional variables in the parameter list. A side effect of this decision is that it is no longer possible to make a library out of TGP that a programmer can implement using the #USE statement in a program. The programmer must physically place a copy of the TGP source code in a program.

Another sacrifice made in order to keep the procedure parameter list small is that channels placed on the link to the B007 graphics board uses pre-defined channel names with global scope.

The only bookkeeping required for a programmer is to keep track of the windows defined starting with window number 0. An example of this is shown below. All other bookkeeping tasks are taken care of automatically by TGP. For example, TGP keeps track of active window, world and device coordinates in a globally scoped two-dimensional array that the various procedures can access as required. The most significant index is the window number and the least significant is a parameter such as the minimum world x-coordinate for that window. A corresponding integer array keeps a copy of the window numbers that the B007 graphics board has assigned to a given window.

TGP also contains procedures for translation, rotation, and scaling of arbitrary data. There is a 3x3 global transformation matrix, trans.2d, that is used for storing composite coordinate transforms. The trans.2d matrix can be modified by the following routines:

```
scale()
rotate()
translate()
```

The two-dimensional transformations are performed using homogeneous coordinates. A full discussion of the method can be found in References 2, 4, and 5. Composite transformations (essentially matrix multipliation) can be generated by consecutive calls to the routines described above. Note, however, that the generation of the composite matrix is order specific. A rotation then a translation is not the same as a translation followed by a rotation.

It is also the users responsibility to initialize the trans.2d matrix using the make.identity() procedure. An example of using the transformation routines is given below. The trans.2d matrix will be generated that will contain the information to translate and then rotate an array of data points. Note the user procedures use the WCS coordinates.

```
      .
      .
-- initialize trans.2d
make.identity(trans.2d)

-- translate x-coordinate 10.0 in WCS, y-coordinate same
translate(10.0 (REAL32), 0.0 (REAL32))

-- rotate 45 degrees about WCS origin (0,0)
rotate(45.0 (REAL32), 0.0 (REAL32), 0.0 (REAL32))

-- transform the data points
transform.points(num.points, x, y)
      .
      .
```

In addition to the 2D transformations mentioned above, TGP can perform both absolute and relative draw commands. The current graphics cursor for each window is maintainted in WCS by the TGP and any user specified absolute or relative move commands are translated into the appropriate B007 IDC commands and sent to the display board. The graphics cursor in the current implementation is currently uninitialized. It is the users responsibility to initialize the graphics cursor using the move.abs() procedure. Note that the B007 graphics display board does not support relative draw routines directly.

USER ROUTINES

A list of the user procedure routines along with the variables used in the parameter list is shown below.

Startup and Shutdown Procedures:

```
init.graphics()

init.db.graphics()

finit.graphics()
```

Geometric Transformation Procedures:

```
transform.points(VAL INT count, []REAL32 x, y)

transform.point(REAL32 x, y)

make.identity([3][3]REAL32 trans.matrix)

scale(VAL REAL32 scale.x, scale.y, x.fixed, y.fixed)

rotate(VAL REAL32 alpha, x.pivot, y.pivot)

translate(VAL REAL32 translate.x, translate.y)
```

Screen and Window Manipulation Procedures:

```
clip.line.2d(REAL32 x1, y1, x2, y2, BOOL display)

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

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

set.viewport.2d(VAL INT x.min, y.min, x.max, y.max,
                VAL INT viewport.number)

activate.viewport.2d(VAL INT viewport.number)
```

```
display.viewport.2d(VAL INT viewport.number)
select.screen(VAL INT screen.number)
```

Absolute and Relative Draw Procedures:

```
move.abs.2d(VAL REAL32 x, y)
move.rel.2d(VAL REAL32 dx, dy)
point.abs.2d(VAL REAL32 x, y)
point.rel.2d(VAL REAL32 dx, dy)
line.abs.2d(VAL REAL32x, y)
line.rel.2d(VAL REAL32 dx, dy)
draw.line.2d(VAL REAL32 x1, y1, x2, y2)
draw.rectangle.2d(VAL REAL32 x, y, x.length, y.length)
draw.polygon.2d(VAL INT num.sides, VAL []REAL32 buffer)
draw.circle.2d(VAL REAL32 x.center, y.center, radius)
draw.arc.2d(VAL REAL32 x1, y1, x2, y2, x3, y3)
```

Miscellaneous Display Procedures:

```
flip.screen()
activate.screen.2d(VAL INT screen.number)
display.screen.2d(VAL INT screen.number)
clear.screen(VAL INT colour)
clear.window(VAL INT colour)
select.colour.table(VAL INT number)
set.colour(VAL INT entry, red, green, blue)
fg.colour(VAL INT entry)
bg.colour(VAL INT entry)
fill.polygon(VAL INT x, y)
quick.fill.polygon(VAL INT x, y)
```

```
fill.polygon.2d(VAL REAL32 x, y)

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

int.line(VAL INT x1, y1, x2, y2)
```

New B007 Procedures:

```
pixel.line(VAL INT size, []INT buffer)

colour.line(VAL INT size, []INT buffer)
```

Internal Graphics System Procedures:

```
combine.transformations[3][3]REAL32 mat.a, mat.b)

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

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

g.send1(VAL INT32 command)

g.send2(VAL INT32 command1, VAL INT32 command2)

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

g.draw.line([2]INT p0, p1)
```

NEW B007 GRAPHICS COMMANDS

Two new graphics commands have been added to the B007 graphics board's driver software. The need for these new transfer protocols arose from having to send many individual c.draw.point commands to the B007. Because of the way link data transfers take place, multiple invocations of a

```
out ! c.draw.point; x; y
```

command can be quite slow. The slowdown arises because each invocation of a out ! variable command takes about 20 microseconds for the processor to set up. The two new B007 protocols take advantage of the variable length array transfer protocol[6]. The new protocols allow either a block of (x, y) coordinate pairs that assumes every pixel is the same color, or a block of (x, y, color) triplets to be sent using the

```
out ! size::buffer
```

occam command. The out ! size::buffer requires only one 20 microsecond set up by the main processor and the Direct Memory Access (DMA) link engines can perform the transfer without any further processor intervention[7].

Note that the new B007 commands use a common storage buffer for each command in the display board software. The format of the common storage buffer is appropriately retyped within the scope of the requested command. Because of the rescoping, the size in the size::buffer command is really

```
size := SIZE(x.variables.to.transfer).
```

An excerpt of the code added to the B007 driver software is shown below.

```
-- line.heap.size := (SQRT (x.screen.size^2 +
--                          y.screen.size^2) + const
-- where the constant lets line.heap.size be divisible by
-- both 2 and 3
-- line.heap.size gives enough storage for a diagonal line
-- on a 512x512 pixel screen to be passed in 1 block

VAL line.heap.size IS 2178 :
[line.heap.size]INT line.heap :
  .
  .
  .
IF
  graphics.command = c.pixel.line  -- block of same color
    .
    .
    [line.heap.size /2][2]INT pixel.buffer
    RETYPES line.heap :
      .
      .
      .
    SEQ
      .
      .
       in ? size::pixel.buffer
      .
      .
      .
  graphics.command = c.color.line -- block of diff. colors
    .
    .
    [line.heap.size / 3][3]INT pixel.buffer
    RETYPES line.heap :
      .
      .
      .
    SEQ
      .
      .
       in ? size::pixel.buffer
      .
      .
```

Depending upon the size of the data transfers and whether or not the drawing is being rendered in a window or just to the screen,

the two new protocols can give speed increases anywhere from 8 percent on a full screen draw to 3300 percent on a full-screen sized window draw when compared to the B007 c.plot.point command. Actual times for a typical example problem are shown in Table 1 below.

The example problem solved is as follows:

A complex function was evaluated over all pixel coordinates in a 2D region. Depending upon the quadrant of complex function value, one of four colors was assigned to the pixel coordinate. The function used in the example was:

$$g(z) = z^4 - 2z^3 + 1.25z^2 - 0.25z - 0.75$$

where $z = x + jy$. The quadrant plot was computed for the complex area $(-2, -j2)$ to $(2, j2)$.

The computation and TGP display processing were performed on a single T414 while the display was performed on a B007 board. The computed pixel colors were sent to the B007 for display using four different methods:

Drawing method	Time* Low-Pri ticks	Time Seconds
Draw into active screen:		
SEQ loop of c.plot.point	5,307,227	339.66
c.colour.line block transfer 512 pixel coordinates per block	4,909,576	314.21
Draw into active window (full screen size):		
SEQ loop of c.plot.point**	177,009,994	11,328.63
c.colour.line block transfer 512 pixel coordinates per block	5,254,740	336.30

* Rev. B T414 chip, 64 microseconds per low-pri tick
** Large time caused by the display board copying window memory to video memory for every pixel displayed.

Table 1. Typical speed increases using new drawing commands.

EXAMPLE USER CODE

A short example of the use of the 2D graphics tools provided in TGP is shown below. A world window is defined. The world window

is then mapped into a normalized device coordinate (NDC) viewport. The axes are from 0 to 1 in both the x and y coordinates. Also note that the origin in NDC is the lower left corner rather that the upper left corner which is normal for the integer device coordinates (IDC). A square is then drawn in the center of the window, and for each of 100 iterations, it is rotated 3.6 degrees and scaled down by a factor of 0.9 . Note that after the first transformation, the displayed figure is no longer a square. This is due to the way the composite transformation matrix is generated. A more complicated series of transformations would be required to keep the object a true square.

```
[4]INT x, y :                  -- store the coordinates here
VAL my.first.window IS 0 :     -- This is the only bookkeeping
                               -- I have to do.
SEQ
  init.graphics()

  -- set the window world coordinates
  set.window.2d(-80.0 (REAL32), -50.0 (REAL32),
                 80.0 (REAL32),  50.0 (REAL32), my.first.window)

  -- set the window size on the screen, starts at lower left,
  -- (0, 0) and goes to (0.75, 0.50) of whole screen
  set.viewport.2d(0.0  (REAL32),  0.0  (REAL32),
                  0.75 (REAL32),  0.50 (REAL32), my.first.window)

  -- turn this window on for drawing
  activate.viewport(my.first.window)

  -- select some colors
  fg.colour(31)
  bg.colour(4)
  clear.window(4)

  -- put it on the screen
  display.viewport.2d(my.first.window)

  -- init coordinates, due to the typing, must be able to
  -- modify the values in the parameter list
  x[0] := -10.0 (REAL32)
  y[0] := x[0]
  x[1] := x[0]
  y[1] := 10.0 (REAL32)
  x[2] := y[1]
  y[2] := y[1]
  x[3] := y[1]
  y[3] := x[0]

  make.identity(trans.2d)  -- trans.2d is the global
                           -- transformation matrix
  -- rotate 3.6 degrees, updates trans.2d matrix
  rotate(3.6 (REAL32), 0.0 (REAL32), 0.0 (REAL32))
```

```
-- scale by 0.9
scale(0.9 (REAL32), 0.9 (REAL32), 0.0 (REAL32), 0.0 (REAL32))

-- draw the square, rotate it, etc.
SEQ i = 0 FOR 100
  SEQ
    SEQ i = 0 FOR 3  -- draw the box
      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)  -- modifies x, y using trans.2d
finit.graphic()
```

Note in this example, the old box is not erased. Also, this is a single buffered routine. For double buffering the user must initialize the graphics using init.db.graphics() and put a flip.screen(), clear.window() after the transform points. This will give a smooth scrolling animation with no visible rendering occuring.

LIMITATIONS

The current implementation of TGP has a few limitations. These are primarily caused by the static memory allocation of Occam though some are limitations in TGP and the B007 driver software.

TGP currently does not directly allow moving windows once they are placed on the screen. Knowlegde of B007 operation can be used to circumvent this problem by direct use of low-level display routines.

Because of the method used for window storage allocation on the B007, it is not possible to resize windows once they are defined. Also, window memory can not be released after a window is allocated, i.e. can not "close" a window and free the window heap memory used by the window for a new one.

SUMMARY

A package of two-dimensional Occam graphics routines have been developed that allow model definition in 2D real-coordinate space. The package directly supports the IMS B007 graphics display board, although the current routines were designed to run on a transputer other than the one on the display board. Features supported include multiple windows, screen doubl-buffering, and 2D geometric transforms such as translation, rotation, and scaling.

In addition, two new data transfer protocols have been developed for the B007 graphics display board software. These new protocols allow blocks of pixel data to be sent to the B007 for display. The block data transfers speed communication by

decreasing the number of communication setups the main processor is required to perform.

REFERENCES

1. IMS B007 Evaluation Board Users Manual. INMOS, Bristol, England, June 1986.

2. Foley, J.D.; and VanDam, A.: Fundamentals of Interactive Computer Graphics. Addison-Wesley, 1982.

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

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

5. Rogers, D.F.; and Adams, J.A. Mathematical Elements for Computer Graphics. McGraw-Hill, 1976.

6. Pountain, D.: A Tutorial Introduction to OCCAM Programming. McGraw-Hill, 1987.

7. Atkin, P.: Performance Maximization. INMOS Technical Note 17, INMOS, Bristol, England, Mar. 1987.

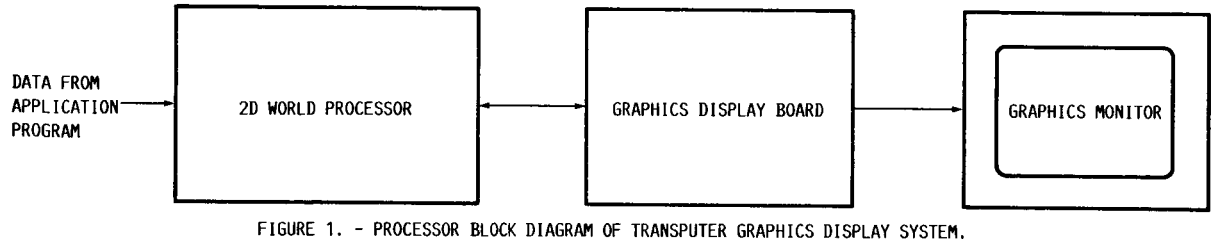

FIGURE 1. - PROCESSOR BLOCK DIAGRAM OF TRANSPUTER GRAPHICS DISPLAY SYSTEM.

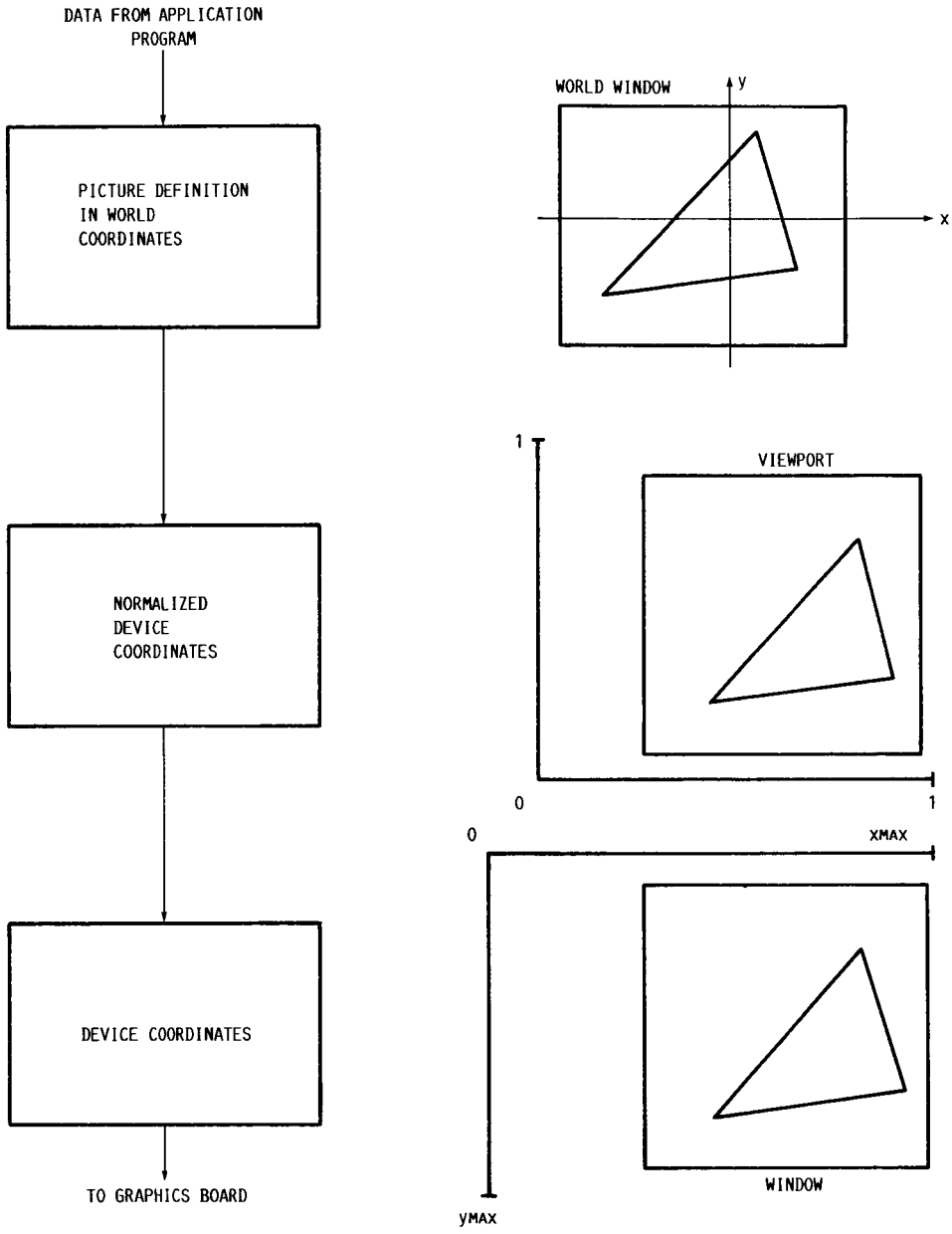

FIGURE 2. - IMPLEMENTATION OF 2D VIEWING ON 2D WORLD PROCESSOR.

NASA
National Aeronautics and Space Administration

Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA TM-100820 ICOMP-88-4		

4. Title and Subtitle	5. Report Date
Two-Dimensional Graphics Tools for a Transputer Based Display Board	6. Performing Organization Code

7. Author(s)	8. Performing Organization Report No.
Graham K. Ellis	E-4010
	10. Work Unit No. 505-33-1B

9. Performing Organization Name and Address	11. Contract or Grant No.
National Aeronautics and Space Administration Lewis Research Center Cleveland, Ohio 44135-3191	

12. Sponsoring Agency Name and Address	13. Type of Report and Period Covered
National Aeronautics and Space Administration Washington, D.C. 20546-0001	Technical Memorandum
	14. Sponsoring Agency Code

15. Supplementary Notes

Prepared for the Parallel Computing Workshop/OCCAM User Group Meeting cosponsored by OSU/OCATE and INMOS, Portland, Oregon, April 11-13, 1988. 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

A package of two-dimensional graphics routines has been developed in an effort to standardize and simplify the user interface for a transputer based graphics display board. The routines available take advantage of the graphics board's capabilities while also presenting an intuitive approach for generating drawings. The routines allow a user to perform graphics rendering in a two-dimensional real-coordinate space without regard to the actual screen coordinates. Multiple windows, which can be placed arbitrarily on the screen as well as the ability to use double-buffering techniques for smooth animations are also supported. The routines are designed to be run on a transputer other than the graphics display board. The window and screen parameters are maintained locally. The conversion to device coordinates is also performed locally. The only data sent to the display board are control and device coordinate display commands. The routines available include: rotation, translation, and scaling commands; absolute and relative point and line commands; circle, rectangle and polygon commands; and window and viewport definition commands.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
Parallel processing Transputer Graphics	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	16	A02

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