THE UNIFIED GRAPHICS SYSTEM
FOR FORTRAN 77
INTERNAL OPERATIONS AND MAINTENANCE MANUAL

ROBERT C. BEACH
COMPUTATION RESEARCH GROUP
STANFORD LINEAR ACCELERATOR CENTER
STANFORD, CALIFORNIA 94309
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>SECTION</th>
<th>DESCRIPTION</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>AN INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1</td>
<td>A BRIEF DESCRIPTION OF THE CONTROL BLOCKS</td>
<td>1</td>
</tr>
<tr>
<td>1.2</td>
<td>A DESCRIPTION OF A PROGRAM IN EXECUTION</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>THE CONTROL BLOCKS</td>
<td>4</td>
</tr>
<tr>
<td>2.1</td>
<td>THE MAIN COMMUNICATION AREA (MCA)</td>
<td>4</td>
</tr>
<tr>
<td>2.2</td>
<td>THE DEVICE-DEPENDENT AREA (DDA)</td>
<td>5</td>
</tr>
<tr>
<td>2.3</td>
<td>THE DDA EXTENSION (DDX)</td>
<td>10</td>
</tr>
<tr>
<td>2.4</td>
<td>THE PICTURE OPTIONS TABLE (POT)</td>
<td>12</td>
</tr>
<tr>
<td>2.5</td>
<td>THE SYMBOL DEFINITIONS</td>
<td>12</td>
</tr>
<tr>
<td>2.6</td>
<td>THE ERROR MESSAGE MODULE (EMM)</td>
<td>13</td>
</tr>
<tr>
<td>3</td>
<td>INTERNAL SUBROUTINES</td>
<td>14</td>
</tr>
<tr>
<td>3.1</td>
<td>MISCELLANEOUS MODULES</td>
<td>14</td>
</tr>
<tr>
<td>3.1.1</td>
<td>SUBROUTINE UGB001</td>
<td>14</td>
</tr>
<tr>
<td>3.1.2</td>
<td>SUBROUTINE UGB002</td>
<td>14</td>
</tr>
<tr>
<td>3.1.3</td>
<td>SUBROUTINE UGB003</td>
<td>15</td>
</tr>
<tr>
<td>3.1.4</td>
<td>SUBROUTINE UGB004</td>
<td>15</td>
</tr>
<tr>
<td>3.1.5</td>
<td>SUBROUTINE UGB005</td>
<td>16</td>
</tr>
<tr>
<td>3.1.6</td>
<td>SUBROUTINE UGB006</td>
<td>16</td>
</tr>
<tr>
<td>3.1.7</td>
<td>SUBROUTINE UGB007</td>
<td>16</td>
</tr>
<tr>
<td>3.1.8</td>
<td>SUBROUTINE UGB008</td>
<td>17</td>
</tr>
<tr>
<td>3.1.9</td>
<td>SUBROUTINE UGB009</td>
<td>17</td>
</tr>
<tr>
<td>3.1.10</td>
<td>SUBROUTINE UGB010</td>
<td>18</td>
</tr>
<tr>
<td>3.1.11</td>
<td>SUBROUTINE UGB011</td>
<td>18</td>
</tr>
<tr>
<td>3.1.12</td>
<td>SUBROUTINE UGB012</td>
<td>18</td>
</tr>
<tr>
<td>3.1.13</td>
<td>SUBROUTINE UGB013</td>
<td>19</td>
</tr>
<tr>
<td>3.1.14</td>
<td>SUBROUTINE UGB014</td>
<td>19</td>
</tr>
<tr>
<td>3.1.15</td>
<td>SUBROUTINE UGB015</td>
<td>20</td>
</tr>
<tr>
<td>3.2</td>
<td>PICTURE GENERATION MODULES</td>
<td>20</td>
</tr>
<tr>
<td>3.2.1</td>
<td>SUBROUTINES UGC001, UGC002, UGC003, AND UGC004</td>
<td>20</td>
</tr>
<tr>
<td>3.2.2</td>
<td>SUBROUTINES UGD001, UGD002, AND UGD003</td>
<td>21</td>
</tr>
<tr>
<td>3.2.3</td>
<td>SUBROUTINES UGE001 AND UGE002</td>
<td>22</td>
</tr>
<tr>
<td>3.2.4</td>
<td>SUBROUTINES UGF001, UGF002, AND UGF003</td>
<td>23</td>
</tr>
<tr>
<td>3.2.5</td>
<td>SUBROUTINES UGG001, UGG002, UGG003, AND UGG004</td>
<td>24</td>
</tr>
<tr>
<td>3.3</td>
<td>OPERATING SYSTEM DEPENDENT MODULES</td>
<td>25</td>
</tr>
<tr>
<td>3.3.1</td>
<td>SUBROUTINE UGZ001</td>
<td>25</td>
</tr>
<tr>
<td>3.3.2</td>
<td>SUBROUTINE UGZ002</td>
<td>25</td>
</tr>
<tr>
<td>3.3.3</td>
<td>SUBROUTINE UGZ003</td>
<td>26</td>
</tr>
<tr>
<td>3.3.4</td>
<td>SUBROUTINE UGZ004</td>
<td>26</td>
</tr>
<tr>
<td>3.3.5</td>
<td>SUBROUTINE UGZ005</td>
<td>26</td>
</tr>
<tr>
<td>3.3.6</td>
<td>SUBROUTINES UGZ006 AND UGZ007</td>
<td>27</td>
</tr>
<tr>
<td>4</td>
<td>THE INTERNAL FORMAT OF A GRAPHIC SEGMENT</td>
<td>28</td>
</tr>
<tr>
<td>4.1</td>
<td>Marker data</td>
<td>29</td>
</tr>
<tr>
<td>4.2</td>
<td>Line data</td>
<td>29</td>
</tr>
<tr>
<td>4.3</td>
<td>Text data</td>
<td>30</td>
</tr>
<tr>
<td>4.4</td>
<td>Extended text data</td>
<td>31</td>
</tr>
<tr>
<td>4.5</td>
<td>Polygon fill data</td>
<td>31</td>
</tr>
<tr>
<td>4.6</td>
<td>Device-dependent data</td>
<td>32</td>
</tr>
<tr>
<td>4.7</td>
<td>Three-dimensional marker data</td>
<td>32</td>
</tr>
<tr>
<td>4.8</td>
<td>Three-dimensional line data</td>
<td>33</td>
</tr>
<tr>
<td>4.9</td>
<td>Three-dimensional text data</td>
<td>33</td>
</tr>
</tbody>
</table>
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>THE DEVICE-DEPENDENT MODULE</td>
<td>35</td>
</tr>
<tr>
<td>5.1</td>
<td>OPEN THE GRAPHIC DEVICE</td>
<td>35</td>
</tr>
<tr>
<td>5.2</td>
<td>CLOSE THE GRAPHIC DEVICE</td>
<td>36</td>
</tr>
<tr>
<td>5.3</td>
<td>CLEAR SCREEN OR WINDOW</td>
<td>36</td>
</tr>
<tr>
<td>5.4</td>
<td>MANIPULATE THE SCREEN</td>
<td>36</td>
</tr>
<tr>
<td>5.5</td>
<td>BEGIN A GRAPHIC SEGMENT</td>
<td>37</td>
</tr>
<tr>
<td>5.6</td>
<td>TERMINATE A GRAPHIC SEGMENT</td>
<td>37</td>
</tr>
<tr>
<td>5.7</td>
<td>MANIPULATE A GRAPHIC SEGMENT</td>
<td>38</td>
</tr>
<tr>
<td>5.8</td>
<td>GRAPHIC INQUIRY</td>
<td>38</td>
</tr>
<tr>
<td>5.9</td>
<td>GRAPHIC DISPLAY</td>
<td>40</td>
</tr>
<tr>
<td>5.10</td>
<td>MISCELLANEOUS CONTROL</td>
<td>41</td>
</tr>
<tr>
<td>5.11</td>
<td>MODIFY STATUS OF A CONTROL UNIT</td>
<td>41</td>
</tr>
<tr>
<td>5.12</td>
<td>ENABLE OR DISABLE A CONTROL UNIT</td>
<td>42</td>
</tr>
<tr>
<td>5.13</td>
<td>OBTAIN AN EVENT</td>
<td>42</td>
</tr>
<tr>
<td>5.14</td>
<td>SAMPLE A CONTROL UNIT</td>
<td>43</td>
</tr>
<tr>
<td>5.15</td>
<td>THREE-DIMENSIONAL VIEWING TRANSFORMATIONS</td>
<td>43</td>
</tr>
<tr>
<td>5.16</td>
<td>DIFFERENCES BETWEEN CURRENT AND EARLIER VERSIONS</td>
<td>44</td>
</tr>
<tr>
<td>6</td>
<td>EXTENDING THE UNIFIED GRAPHICS SYSTEM</td>
<td>45</td>
</tr>
<tr>
<td>6.1</td>
<td>ADDING A NEW GRAPHIC DEVICE</td>
<td>45</td>
</tr>
<tr>
<td>6.2</td>
<td>TRANSFERRING THE SYSTEM TO ANOTHER COMPUTER</td>
<td>46</td>
</tr>
<tr>
<td>6.2.1</td>
<td>OPERATING SYSTEM DEPENDENT MODULES</td>
<td>46</td>
</tr>
<tr>
<td>6.2.2</td>
<td>THE OPTIONS SCANNER</td>
<td>47</td>
</tr>
<tr>
<td>6.2.3</td>
<td>PROCESS THE ERROR MESSAGES</td>
<td>47</td>
</tr>
<tr>
<td>6.2.4</td>
<td>PROCESS THE CHARACTER SETS</td>
<td>48</td>
</tr>
<tr>
<td>6.2.5</td>
<td>THE ERROR PROCESSOR</td>
<td>49</td>
</tr>
<tr>
<td>6.2.6</td>
<td>THE LINE SCISSORING MODULE</td>
<td>49</td>
</tr>
<tr>
<td>6.2.7</td>
<td>THE LINE STRUCTURE MODULE</td>
<td>49</td>
</tr>
<tr>
<td>6.2.8</td>
<td>PROCESS THE CHARACTER SETS, PART 2</td>
<td>50</td>
</tr>
<tr>
<td>6.2.9</td>
<td>THE CHARACTER GENERATOR MODULE</td>
<td>50</td>
</tr>
<tr>
<td>6.2.10</td>
<td>THE POLYGON SCISSORING MODULE</td>
<td>50</td>
</tr>
<tr>
<td>6.2.11</td>
<td>THE THREE-DIMENSIONAL LINE SCISSORING MODULE</td>
<td>51</td>
</tr>
<tr>
<td>6.2.12</td>
<td>GRAPHIC SEGMENT GENERATION MODULES</td>
<td>51</td>
</tr>
<tr>
<td>6.2.13</td>
<td>GRAPHIC ALGORITHMS MODULE</td>
<td>51</td>
</tr>
<tr>
<td>6.2.14</td>
<td>MISCELLANEOUS CHARACTER CONTROL MODULES</td>
<td>53</td>
</tr>
<tr>
<td>6.2.15</td>
<td>A NON-INTERACTIVE DUMMY-DEVICE</td>
<td>53</td>
</tr>
<tr>
<td>6.2.16</td>
<td>AN INTERACTIVE DUMMY-DEVICE</td>
<td>54</td>
</tr>
<tr>
<td>6.2.17</td>
<td>THE ACTUAL DEVICE-DEPENDENT MODULES</td>
<td>54</td>
</tr>
<tr>
<td>6.3</td>
<td>ADDING A NEW GRAPHIC PRIMITIVE</td>
<td>55</td>
</tr>
<tr>
<td>7</td>
<td>THE ALGORITHMS USED IN CERTAIN MODULES</td>
<td>57</td>
</tr>
<tr>
<td>7.1</td>
<td>THE SCISSORING ALGORITHMS</td>
<td>57</td>
</tr>
<tr>
<td>7.2</td>
<td>THE LINE STRUCTURE ALGORITHM</td>
<td>60</td>
</tr>
<tr>
<td>7.3</td>
<td>THE CHARACTER GENERATION ALGORITHM</td>
<td>61</td>
</tr>
<tr>
<td>7.4</td>
<td>THE PROJECTION ALGORITHMS</td>
<td>62</td>
</tr>
<tr>
<td>7.5</td>
<td>THE SMOOTH CURVE INTERPOLATION ALGORITHM</td>
<td>67</td>
</tr>
<tr>
<td>7.6</td>
<td>THE CROSS-HATCHING ALGORITHM</td>
<td>70</td>
</tr>
<tr>
<td>7.7</td>
<td>THE AXIS LABELING ALGORITHMS</td>
<td>72</td>
</tr>
<tr>
<td>7.8</td>
<td>THE CONTOUR PLOT ALGORITHMS</td>
<td>74</td>
</tr>
<tr>
<td>7.9</td>
<td>THE MESH SURFACE ALGORITHM</td>
<td>77</td>
</tr>
<tr>
<td>7.10</td>
<td>THE TWO-DIMENSIONAL HISTOGRAM ALGORITHMS</td>
<td>80</td>
</tr>
</tbody>
</table>
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>SECTION</th>
<th>DESCRIPTION</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>THE ORGANIZATION OF THE DATA SETS</td>
<td>81</td>
</tr>
<tr>
<td>8.1</td>
<td>THE DATA SETS ON THE VAX COMPUTERS</td>
<td>81</td>
</tr>
<tr>
<td>8.2</td>
<td>THE DATA SETS ON THE IBM COMPUTERS</td>
<td>83</td>
</tr>
<tr>
<td>8.3</td>
<td>NONSTANDARD FORTRAN-77 CONSTRUCTIONS</td>
<td>85</td>
</tr>
<tr>
<td>8.4</td>
<td>SOURCE CODE DIFFERENCES</td>
<td>86</td>
</tr>
<tr>
<td>9</td>
<td>THE SUPPORTED GRAPHIC DEVICES</td>
<td>88</td>
</tr>
<tr>
<td>9.1</td>
<td>THE CONSOLE ON WHEELS (COW) OF THE SLC PROJECT</td>
<td>88</td>
</tr>
<tr>
<td>9.2</td>
<td>THE DEC GIGI COLOR GRAPHICS TERMINAL</td>
<td>88</td>
</tr>
<tr>
<td>9.3</td>
<td>THE DEC VAXSTATIONS</td>
<td>88</td>
</tr>
<tr>
<td>9.4</td>
<td>THE GRINNELL GMR-27 DISPLAY SYSTEM</td>
<td>89</td>
</tr>
<tr>
<td>9.5</td>
<td>THE IBM 3179 G COLOR GRAPHICS DISPLAY STATION</td>
<td>90</td>
</tr>
<tr>
<td>9.6</td>
<td>THE IBM 5080 GRAPHICS SYSTEM</td>
<td>91</td>
</tr>
<tr>
<td>9.7</td>
<td>THE IMAGEN LASER PLOTTERS</td>
<td>91</td>
</tr>
<tr>
<td>9.8</td>
<td>THE METHEUS OMEGA 300 DISPLAY CONTROLLER</td>
<td>92</td>
</tr>
<tr>
<td>9.9</td>
<td>THE POSTSCRIPT LANGUAGE</td>
<td>93</td>
</tr>
<tr>
<td>9.10</td>
<td>THE PRINTRONIX (MODEL MVP) PLOTTER</td>
<td>93</td>
</tr>
<tr>
<td>9.11</td>
<td>THE SEIKO GR-1105 COLOR GRAPHICS TERMINAL</td>
<td>94</td>
</tr>
<tr>
<td>9.12</td>
<td>THE SLAC EXPERIMENTAL SLAVE SCOPE</td>
<td>95</td>
</tr>
<tr>
<td>9.13</td>
<td>THE TALARIS LASER PLOTTERS</td>
<td>95</td>
</tr>
<tr>
<td>9.14</td>
<td>THE TEKTRONIX 4010 SERIES TERMINALS</td>
<td>96</td>
</tr>
<tr>
<td>9.15</td>
<td>TEKTRONIX 4010/4014 EMULATORS</td>
<td>97</td>
</tr>
<tr>
<td>9.16</td>
<td>THE TEKTRONIX 4027 COLOR GRAPHICS TERMINAL</td>
<td>97</td>
</tr>
<tr>
<td>9.17</td>
<td>THE TEKTRONIX 4105 COMPUTER DISPLAY TERMINAL</td>
<td>98</td>
</tr>
<tr>
<td>9.18</td>
<td>THE TEKTRONIX 4207 COMPUTER DISPLAY TERMINAL</td>
<td>98</td>
</tr>
<tr>
<td>9.19</td>
<td>THE TEKTRONIX 4510 COLOR GRAPHICS RASTERIZER</td>
<td>99</td>
</tr>
<tr>
<td>9.20</td>
<td>THE VERSATEC ELECTROSTATIC PLOTTER</td>
<td>100</td>
</tr>
<tr>
<td>9.21</td>
<td>THE X-WINDOWS PROTOCOL</td>
<td>102</td>
</tr>
<tr>
<td>9.22</td>
<td>GRAPHIC PSEUDO-DEVICES</td>
<td>102</td>
</tr>
<tr>
<td>9.23</td>
<td>A GENERIC WORKSTATION</td>
<td>103</td>
</tr>
<tr>
<td>10</td>
<td>COMPARISONS WITH OTHER SYSTEMS</td>
<td>122</td>
</tr>
<tr>
<td>10.1</td>
<td>A HISTORY OF THE UNIFIED GRAPHICS SYSTEM</td>
<td>122</td>
</tr>
<tr>
<td>10.2</td>
<td>THE GRAPHICS COMPATIBILITY INTERFACE</td>
<td>128</td>
</tr>
<tr>
<td>10.3</td>
<td>DISSPLA</td>
<td>128</td>
</tr>
<tr>
<td>10.4</td>
<td>GINO-F</td>
<td>129</td>
</tr>
<tr>
<td>10.5</td>
<td>THE GRAPHICS COMPATIBILITY SYSTEM</td>
<td>129</td>
</tr>
<tr>
<td>10.6</td>
<td>THE GRAPHICS DISPLAY SYSTEM</td>
<td>130</td>
</tr>
<tr>
<td>10.7</td>
<td>THE INTEGRATED GRAPHICS SYSTEM</td>
<td>130</td>
</tr>
<tr>
<td>10.8</td>
<td>THE GENERAL PURPOSE GRAPHICS SYSTEM</td>
<td>131</td>
</tr>
<tr>
<td>10.9</td>
<td>THE GRAPHICAL DATA DISPLAY MANAGER</td>
<td>132</td>
</tr>
<tr>
<td>10.10</td>
<td>THE ACM-SIGGRAPH STANDARD PROPOSAL</td>
<td>132</td>
</tr>
<tr>
<td>10.11</td>
<td>THE GKS STANDARD</td>
<td>133</td>
</tr>
</tbody>
</table>

REFERENCES | 135
SECTION 1: AN INTRODUCTION

This document describes the internal operation of the Unified Graphics System. In addition, recommendations are made for extending the system and getting it running on other computers. This document will assume that the reader is familiar with the Unified Graphics System Programming Manual [Bea81a] and the Unified Graphics System Algorithms Manual [Bea81b].

The version of the Unified Graphics System that is described here is coded almost entirely in FORTRAN-77 [ANS78]. A strong effort has been made to limit nonstandard constructions so that the system should be relatively easy to move to other computers. It would be an extensive project, however, to get the Unified Graphics System to compile on earlier versions of FORTRAN because the system makes significant use of character string variables, PARAMETER statements, the IF-THEN-ELSE construction, and the nonstandard INCLUDE statement. There are also a few Assembler Language modules in the system, but these are small and should be relatively easy to implement on other computers.

SECTION 1.1: A BRIEF DESCRIPTION OF THE CONTROL BLOCKS

There are a number of common blocks within the Unified Graphics System which are used to contain control information. The first of these is the Main Communication Area (MCA). It contains information about the state of the system, in particular, it keeps track of how many graphic devices are open.

Another pair of common blocks is the Device-Dependent Area (DDA) and the Device-Dependent Area Extension (DDX). These contain information about the active graphic device. The DDA contains information that must be kept about every type of device, while the DDX contains information that only applies to a specific type of device. The DDA has the same format for each graphic device, but the DDX is distinct for each different type of device.

Another common block, the Picture Options Table (POT), contains the options scanning table for the options used by the graphic segment generators. Putting this in a static common block means that it does not have to be duplicated in a large number of subroutines.

The Error Message Module (EMM) is another common block. In addition to the error messages themselves, the block contains such items as the unit number for the printed error messages.

Finally, there are the symbol definitions. These consist of four static common blocks. The common blocks define the marker symbols, the basic character set, the extended/simplex character set, and the extended/duplex character set.
SECTON 1.2: A DESCRIPTION OF A PROGRAM IN EXECUTION

This section will give a brief description of some of the operations that are performed at execution time. For definiteness, we will assume that an application program will do the following:

1. A TEKTRONIX terminal is opened for interactive use, and a graphic segment is sent to it.
2. The VERSATEC plotter is opened, and the same graphic segment is sent to it.
3. The TEKTRONIX is made the active device, and the program waits for a response from the terminal.
4. Both of the graphic devices are closed.

The rest of this section will examine what happens internally when this program executes.

To open the TEKTRONIX, the program calls UGOPEN. UGOPEN saves the identification supplied to it in the MCA and begins the initialization of the DDA. Then UGOPEN activates the device-dependent code for the TEKTRONIX, by calling subroutine UGZ002. In the current implementations, activating the device-dependent code simply means that its address in memory is obtained. Next UGOPEN calls the device-dependent code. The device-dependent code adds more data to the DDA and initializes the DDX. When control returns to UGOPEN, it finishes the initialization of the DDA. Any of the other subroutines in the Unified Graphics System can now communicate with the TEKTRONIX.

The application program prepares a graphic segment by calling UGINIT, UGLINE, UGTEXT, etc. These subroutines use the POT but do not enter the device-dependent code. When UGWRIT is called, a complex handshaking sequence is performed between UGWRIT and the device-dependent code. When, for example, UGWRIT encounters text in the graphic segment, it calls the device-dependent code to ask if it can draw characters of the specified size with a hardware character generator. If the device-dependent code answers no, then UGWRIT uses the basic character set to draw the characters with line segments; if the answer is yes, the device-dependent code also returns enough information so that UGWRIT can clip the characters at the window boundaries before passing the character string to the device-dependent code. In either case, the picture elements are written to the screen.

When UGOPEN is called to initialize the VERSATEC, some additional work must be done. UGOPEN detects that a graphic device is already open, so it deactivates the TEKTRONIX by allocating memory to contain copies of the DDA and DDX and moves these common blocks into the allocated memory. Opening of the VERSATEC then proceeds in the same manner that the TEKTRONIX did.

Now, the application program can call UGWRIT again to transmit the graphic segment to the VERSATEC. Since the TEKTRONIX has a hardware character generator and the VERSATEC does not, the handshaking sequence may proceed very differently for this second graphic device.
When the application program calls UGSLCT to make the TEKTRONIX active again, it detects that another device is already active. UGSLCT therefore deactivates the VERSATEC in the manner described above, and activates the TEKTRONIX by moving the data from the previously allocated memory into the DDA and DDX. The allocated memory is retained because it is presumed that the TEKTRONIX might be deactivated again.

The application program then enables the keyboard with UGENAB and waits for an event with UGEVT. Both of these operations cause the device-dependent code to be entered. In fact, the wait operation is performed within the device-dependent code.

After the event is received, the program calls UGCLOS with the ALL option to close both graphic devices. UGCLOS calls the device-dependent code to allow it to perform any necessary clean-up operations on the TEKTRONIX, frees the allocated blocks for the DDA and DDX, deactivates the device-dependent code by calling subroutine UGZ002, and removes the identification of the TEKTRONIX from the MCA. UGCLOS then activates the VERSATEC and repeats the close sequence for it. The termination operations for the VERSATEC are more important than for the TEKTRONIX because there will be partial output buffers for that device which must be written out.
SECTION 2: THE CONTROL BLOCKS

This section contains detailed information about the various control blocks used by the Unified Graphics System. Most of the control blocks are contained within the NUCLEUS. It is useful to notice that the names of the control blocks within the NUCLEUS all consist of the letters "UGA" followed by three numbers.

Strictly speaking, these common blocks violate the FORTRAN-77 standard because they include character string variables with real and integer items. The standard apparently objects to this because a common block could have two different declarations in two different subroutines and thus get the effect of overlaying characters and arithmetic variables. Since there is no standard for the number of characters in an arithmetic variable, this results in computer dependencies. The Unified Graphics System does not violate the intent of the standard because the common block declaration is the same in all subroutines. In addition, the FORTRAN-77 standard itself is inconsistent because there is no standard that assures that real and integer variables are always the same length, thus putting real and integer variables in a common block can also result in computer dependencies.

SECTION 2.1: THE MAIN COMMUNICATION AREA (MCA)

The Main Communication Area (MCA) is a common block contained in the NUCLEUS. It contains information about the overall state of the Unified Graphics System. The declaration for this control block is incorporated into a module by including a module named UGMCACBK. The contents of that module are:

CONTROL BLOCK FOR THE MAIN COMMUNICATION AREA.
SAVE /UGA001/
C NUMBER OF DEVICES THAT MAY BE OPEN AT ONE TIME.
INTEGER MCAZ1
PARAMETER (MCAZ1=32)
C THE DECLARATION OF THE COMMON BLOCK.
COMMON /UGA001/
X MCAID,
X MCACN,MCACP,
X MCAOI,MCAOP,
X MCAIC
C MAIN COMMUNICATION AREA IDENTIFICATION.
CHARACTER*8 MCAID
C NAME OF ACTIVE EXTENDED CHARACTER SET.
CHARACTER*8 MCACN
C POINTER TO ACTIVE EXTENDED CHARACTER SET.
INTEGER MCACP
C IDENTIFICATION OF ALL OPEN DEVICES.
INTEGER MCAOI(MCAZ1)
C POINTERS TO THE ALLOCATED DDA'S OF ALL OPEN DEVICE.
INTEGER MCAOP(MCAZ1)
C NUMBER OF FULLY INTERACTIVE DEVICES CURRENTLY OPEN.
INTEGER MCAIC

Notice that the dimension of the arrays MCAOI and MCAOP are defined by the parameter MCAZ1. When these arrays are searched by the modules that reference them, the bounds for the search is again MCAZ1. This means that if the number of devices that are permitted to be open at one time is to be changed, the only statement that must be modified is the PARAMETER statement that assigns a value to MCAZ1. Then, all modules using the MCA must be recompiled. Actually, in this case, one of the error messages in the Error Message Module should also be changed.

The MCA is used by the following modules:
NUCLEUS, UGOPEN, UGCLOS, UGSLCT, UGINFO, UGWRIT, UGFONT, UGCTOL,
and in the device-dependent module for PDEVUGS.

SECTION 2.2: THE DEVICE-DEPENDENT AREA (DDA)

The Device Dependent Area (DDA) is a common block contained in the NUCLEUS. It contains information about the current status of the active graphic device. The declaration for this control block is incorporated into a module by including a module named UGDDACBK. The contents of that module are:

C CONTROL BLOCK FOR THE DEVICE-DEPENDENT AREA FOUNDATION.
SAVE /UGA003/
C MAXIMUM NUMBER OF INTERACTIVE CONTROLS.
INTEGER DDAZ1
PARAMETER (DDAZ1=8)
C MAXIMUM KEYBOARD STRING LENGTH.
INTEGER DDAZ2
PARAMETER (DDAZ2=128)
C NUMBER OF ITEMS IN THE BUTTON ARRAY.
INTEGER DDAZ3
PARAMETER (DDAZ3=2)
C MAXIMUM NUMBER OF STROKE SEGMENTS.
INTEGER DDAZ4
PARAMETER (DDAZ4=128)
C MAXIMUM NUMBER OF SHIELDS.
INTEGER DDAZ5
PARAMETER (DDAZ5=4)
C NUMBER OF WORDS IN THE DEVICE-DEPENDENT AREA FOUNDATION.
INTEGER DDAZZ
PARAMETER (DDAZZ=224)
C THE DECLARATION OF THE COMMON BLOCK.
COMMON /UGA003/
    X DDAID,DDALG,DDAPA,DDALX,DDAPX,DDACX,
    X DDAAI,DDAAT,DDAAC,DDAAN,
X DDAIL, DDADM, DDADF, DDAIC, DDA BC,
X DDAKX, DDAKY, DDAKS, DDAK N, DDAKF,
X DDABF,
X DDASL, DDAST, DDASN,
X DDABD, DDABE, DDABX, DDABY,
X DDADS, DDADD, DDADX, DDADY, DDADA,
X DDAWA, DDAWS, DDAWD, DDAW X, DDAW Y, DDA TR,
X DDASA, DDASF, DDASH,
X DDAS3, DDASV,
X DDASW, DDASO, DDAS E, DDASU, DDAS N,
X DDAS3M, DDAS3P, DDAS T,
X DDAXV, DDAXO, DDAXE, DDAXU, DDAXN,
X DDAY W

C DEVICE-DEPENDENT AREA FOUNDATION IDENTIFICATION.
CHARACTER*8 DDAID
C NUMBER OF WORDS IN THE DDA FOUNDATION.
INTEGER DDALG
C POINTER TO THE ALLOCATED BLOCK FOR THE DDA FOUNDATION.
INTEGER DDAPA
C NUMBER OF WORDS IN THE DEVICE-DEPENDENT EXTENSION OF THE DDA.
INTEGER DDALX
C POINTER TO THE ALLOCATED BLOCK FOR THE DEVICE-DEPENDENT
C EXTENSION OF THE DDA.
INTEGER DDAPX
C POINTER TO THE ACTUAL COMMON BLOCK FOR THE DEVICE-DEPENDENT
C EXTENSION.
INTEGER DDACX
C INDEX OF THE DEVICE IN THE MCA'S LIST OF OPEN DEVICES.
INTEGER DDACI
C DEVICE TYPE AS SPECIFIED IN SUBROUTINE UGOPEN.
CHARACTER*8 DDAAT
C NAME OF DEVICE-DEPENDENT SUBROUTINE.
INTEGER DDAM
C INTERACTION LEVEL OF DEVICE:
C 1 MEANS NON-INTERACTIVE,
C 2 MEANS SLAVE-DISPLAY, AND
C 3 MEANS FULLY INTERACTIVE.
INTEGER DDAIL
C DRAWING MEDIUM PROPERTIES:
C 1 MEANS NON-ERASABLE MEDIUM,
C 2 MEANS RASTER-SCAN DEVICE, AND
C 3 MEANS REFRESH DISPLAY DEVICE.
INTEGER DDADM
C DISPLAY DIMENSION:
C 2 FOR A TWO-DIMENSIONAL DEVICE, AND
C 3 FOR A THREE-DIMENSIONAL DEVICE.
INTEGER DDADF
C INTERACTIVE CONTROLS FLAGS; THE ENTRIES IN THE ARRAY REFER TO
C THE FOLLOWING CONTROL UNITS:
C 1 KEYBOARD,
C 2 PICK,
C 3 BUTTON,
C 4 STROKE,
5 locator, and
6 valuator.
Any additional entries are not used. The values of the
entries are:
0 means not available on the device, and
1 means available on the device except for button and
valuator where the value gives the number of buttons or
valuators respectively.

INTEGER DDAIC(DDAZ1)

Enabled interactive controls; the entries in the array are
the same as the preceding array. The values are:
0 means disabled, and
+ means enabled.

INTEGER DDABC(DDAZ1)

Keyboard x and y coordinates.

INTEGER DDAKX,DDAKY

Keyboard character string.
Character*(DDAZ2) DDASK

Keyboard character string length.

INTEGER DDAKN

Keyboard upper case translation flag:
0 means translate lower case to upper case, and
1 means do not translate lower case to upper case.

INTEGER DDAKF

Button flags:
0 bit means light is off, and
1 bit means light is on.

INTEGER*4 DDABF(DDAZ3)

Maximum stroke length as a ratio of full screen (precision is
(32,30)).

INTEGER DDASL

Maximum stroke time (in hundredths of a second).

INTEGER DDAST

Stroke table length.

INTEGER DDASHN

Basic two-dimensional drawing area.

INTEGER DDABD(2,2)

Extension possibilities:
0 means no extension is possible, and
1 means extension is possible.

INTEGER DDABE(2,2)

Centimeters per raster unit in the x and y direction.

REAL DDABX,DDABY

Drawing space limits.

REAL DDADS(2,2)

Drawing space device limits.

INTEGER DDAADD(2,2)

Centimeters per unit in the x and y direction in the drawing
space.

REAL DDAWX,DDADY

Affinity value.

REAL DDADA

Two-dimensional view port limits.

REAL DDAWA(2,2)

Two-dimensional window limits.
REAL DDAWS(2,2)
C TWO-DIMENSIONAL WINDOW DEVICE LIMITS.
INTEGER DDAWD(2,2)
C CENTIMETERS PER UNIT IN THE X AND Y DIRECTION IN THE VIEW
C PORT.
REAL DDAWY
C AUXILIARY TWO-DIMENSIONAL TRANSFORMATION DATA.
REAL DDAATR(8)
C NUMBER OF CURRENTLY ACTIVE SHIELDS.
INTEGER DDASA
C FLAGS FOR ACTIVE SHIELDS:
C 0 MEANS SHIELD IS NOT AVAILABLE, AND
C 1 MEANS SHIELD IS AVAILABLE.
INTEGER DDASF(DDAZS)
C SHIELD DEFINITIONS.
REAL DDASH(2,2,DDAZS)
C BASIC THREE-DIMENSIONAL DRAWING VOLUME.
INTEGER DDA3D(3,2)
C THREE-DIMENSIONAL VIEW PORT.
REAL DDA3V(2,2)
C THREE-DIMENSIONAL WORLD VOLUME.
REAL DDA3W(3,2)
C THREE-DIMENSIONAL OBJECT VOLUME.
REAL DDA3O(3,2)
C THREE-DIMENSIONAL EYE POINT.
REAL DDA3E(3)
C THREE-DIMENSIONAL UP DIRECTION.
REAL DDA3U(3)
C THREE-DIMENSIONAL PROJECTION FLAG:
C 0.0 MEANS PARALLEL PROJECTION, AND
C + MEANS NEAR SCISSORING VALUE FOR A POINT PROJECTION.
REAL DDA3N
C THE PROJECTION MATRIX.
REAL DDA3M(3,4)
C EQUATION OF THE NEAR SCISSORING PLANE.
REAL DDA3P(4)
C AUXILIARY THREE-DIMENSIONAL TRANSFORMATION DATA.
REAL DDA3T(24)
C THREE-DIMENSIONAL VIEW PORT IN DEVICE UNITS.
INTEGER DDA XV(2,2)
C THREE-DIMENSIONAL OBJECT VOLUME IN DEVICE UNITS.
INTEGER DDA XO(3,2)
C THREE-DIMENSIONAL EYE POINT IN DEVICE UNITS.
INTEGER DDA XE(3)
C THREE-DIMENSIONAL UP DIRECTION IN DEVICE UNITS (PRECISION IS
C (32,30)).
INTEGER DDA XU(3)
C THREE-DIMENSIONAL PROJECTION FLAG IN DEVICE UNITS:
C 0 MEANS PARALLEL PROJECTION, AND
C + MEANS NEAR SCISSORING VALUE FOR A POINT PROJECTION
C (PRECISION IS (32,30)).
INTEGER DDA XN
C INITIAL WRITE GIVEN FLAG:
C 0 MEANS FIRST GRAPHIC SEGMENT HAS NOT BEEN WRITTEN, AND
C 1 MEANS FIRST GRAPHIC SEGMENT HAS BEEN WRITTEN.
INTEGER DDAFW
C THE FOLLOWING ARRAY OVERLAYS THE ENTIRE DDA FOUNDATION.
INTEGER DDARY(DDAZZ)
EQUIVALENCE (DDARY(1), DDAID)

All of the arbitrary decisions about maximum counts relative to
the DDA are assigned by the parameters DDAZ1 through DDAZ5.
These counts can be changed by simply changing the associated
PARAMETER statement and recompiling the appropriate modules. It
is important to notice that any modification that changes the
length of the DDA requires that the value of DDAZZ be changed
also. If DDAZZ is not correct, the Unified Graphics System will
not correctly allocate storage for the DDA's of inactive devices.

The first few items in the DDA are used in maintaining the
active-device/inactive-device relations. When a graphic device
is first opened, an unused entry in MCAOI is set to contain the
identification supplied in UGOPEN and the corresponding entry in
MCAOP is set to zero. The entries DDAPA and DDAPX are also set
to zero. These zeros indicate that a block of allocated storage
is not yet been allocated for the DDA (or for the DDX in the case
of DDAPX). The device-dependent code is activated in UGOPEN by
calling subroutine UGZ002 and saving the returned address in
DDAAC. Any of the subroutines may execute the device-dependent
code by using DDAAC in conjunction with subroutine UGZ006. The
open section of the device-dependent code must set DDACX to the
address of the common block containing the DDX, by using
subroutine UGZ005. It must also set DDALX to the length of the
DDX.

A graphic device is deactivated explicitly when UGSLCT is called
or implicitly when a new graphic device is opened by UGOPEN. To
deactivate a graphic device, DDAPX is first checked to see if a
block of allocated storage is available to hold the DDX. If
DDAPX is zero, a block is allocated, using subroutine UGZ003, and
its address is saved in DDAPX. Then, using DDAPX, DDACX, and
DDALX, the DDX is moved into this allocated block. Next, if
DDAPA is zero, a block of storage is allocated to hold the DDA
and its address is saved in DDAPA and in the appropriate entry in
MCAOP. Finally, the DDA is moved into the allocated block. In
this state, both the DDA and the DDX are ready to be reused by
another graphic device.

Once the DDA and DDX are ready for reuse, it is a simple matter
to activate a graphic device. First, MCAOI is searched to find
the identification of the device to be activated. When the
identification is found, the address of the allocated block for
the DDA is found in the corresponding entry of MCAOP. Using this
address and DDALG, the DDA is moved into the common block. Then
using DDAPX, DDACX, and DDALX, the common block containing the
DDX is restored.

The advantage of this scheme is that very few subroutines (only
UGOPEN, UGCLOS, and UGSLCT) must know anything about any inactive
devices. All other subroutines simply use the data in the common
blocks and act as if there was only a single graphic device available. The device-dependent code, except for the setting of DDACX and DDALX, acts as if it is the only graphic device around. It should also be noted that secondary storage blocks are not allocated for the DDA and DDX unless a graphic device is actually deactivated; programs that only use a single graphic device do not waste memory space.

Starting with DDAIL are a large number of items that describe the graphic device. Most of these items are initialized by UGOPEN before the device-dependent module is called, or after the device-dependent module has returned. The device-dependent module, however, is responsible for setting a few of these items. DDAIL, DDADM, and DDADF must be set by the device-dependent module if they are to have values other than their default values of 1, 1, and 2, respectively. If the device has any interactive controls, they must be indicated in DDAIC. The extent of the basic drawing space must be set in DDABD and the spacing of the raster units must be saved in DDABX and DDABY. The extension possibilities must be set in DDABE for devices like drum plotters. For a graphic device with three-dimensional capability, the extent of the three-dimensional coordinate system must be given in DDA3D.

The DDA is used by the following modules:

NUCLEUS, UGOPEN, UGCLOS, UGSLCT, UGINFO, UGMCTL,
UGRIT, UGPICT, UGENAB, UGDSAB, UGEVNT, UGECTL,
UGDSPC, UGWDOV, UGSHLD, UG3WRD, UG3TRN, UGB003,
UGB004, UGB005, UGB006, UGB007, UGB008, UGB009,
UGB010, UGB011, UGB012, UGB013,
and all of the device-dependent modules.

SECTION 2.3: THE DDA EXTENSION (DDX)

The DDA Extension (DDX) also contains information about the active graphic device. The information in the DDA applies to all types of graphic devices; the information in the DDX applies only to a specific graphic device. There is a different DDX for each type of graphic device. The DDX is a common block contained within the device-dependent module. The DDX for the TEKTRONIX 4010 Series Terminals when sequential data sets are being prepared is:

C CONTROL BLOCK FOR THE DEVICE-DEPENDENT AREA EXTENSION FOR
C GENERATING DISPLAY FILES FOR THE TEKTRONIX 4010 SERIES
C DEVICES IN A SEQUENTIAL DATA SET.

SAVE /UGTS00/

C NUMBER OF CHARACTERS IN OUTPUT RECORD.

INTEGER DDXZ1
PARAMETER (DDXZ1=72)

C NUMBER OF CHARACTERS IN OUTPUT CARD.
INTEGER DDXZ2
PARAMETER (DDXZ2=80)
C CONTROL BLOCK LENGTH.
INTEGER DDXZZ
PARAMETER (DDXZZ=36)
C THE DECLARATION OF THE COMMON BLOCK.
COMMON /UGTS00/
   X DDXID,
   X DDXBF, DDXBN,
   X DDXTT,
   X DDXNC,
   X DDXPI, DDXPN, DDXPL, DDXDL,
   X DDXIL, DDXPV, DDXDC, DDXOP,
   X DDXIO
C DEVICE-DEPENDENT AREA EXTENSION IDENTIFICATION.
   CHARACTER*8 DDXID
C STORAGE BUFFER FOR THE OUTPUT RECORD.
   CHARACTER*(DDXZ2) DDXBF
C NUMBER OF CHARACTERS CURRENTLY IN THE OUTPUT RECORD.
   INTEGER DDXBN
C TERMINAL TYPE FLAG:
C 0 MEANS STANDARD TEKTRONIX 4010,
C 1 MEANS RETRO-GRAPHICS ADM-3A, AND
C 2 MEANS RETRO-GRAPHICS VT100.
   INTEGER DDXTT
C NUMBER OF NULL RECORDS AFTER CLEAR.
   INTEGER DDXNC
C PICTURE IDENTIFICATION (ALPHABETIC PART).
   CHARACTER*4 DDXPI
C PICTURE IDENTIFICATION (NUMERIC PART).
   INTEGER DDXPN
C PICTURE ALIAS.
   CHARACTER*8 DDXPL
C PICTURE ALIAS (DEFERRED VALUE).
   CHARACTER*8 DDXDL
C INTENSITY LEVEL FOR POINTS.
   INTEGER DDXIL
C PICTURE AVAILABLE FLAG:
C 0 MEANS NO PREVIOUS PICTURE IS AVAILABLE, AND
C 1 MEANS A PICTURE HAS PREVIOUSLY BEEN PRODUCED.
   INTEGER DDXPV
C DEFERRED CLEAR FLAG:
C 0 MEANS THE START OF A NEW PICTURE HAS NOT BEEN DEFERRED,
C 1 MEANS THE START OF A NEW PICTURE HAS BEEN DEFERRED.
   INTEGER DDXDC
C LAST POSITIONING ORDER.
   CHARACTER*4 DDXOP
C INPUT/OUTPUT IDENTIFICATION VALUE.
   INTEGER DDXIO
C THE FOLLOWING ARRAY OVERLAYS THE ENTIRE DDA EXTENSION.
   INTEGER DDXRY(DDXZZ)
   EQUIVALENCE (DDXRY(1), DDXID)

The DDX also contains a parameter, DDXZZ, which defines the length of the control block. This value is stored in the DDA (in
MAINTENANCE MANUAL

DDALX) at open time and is used to allocate storage for the DDX when it is not active. At first, it may seem that obtaining duplicate storage for the DDX is unnecessary, but it must be remembered that the Unified Graphics System allows a graphic device to be opened more than once. For example, a program can create more than one sequential data set for a TEKTRONIX 4010 at one time.

SECTION 2.4: THE PICTURE OPTIONS TABLE (POT)

The Picture Options Table (POT), is a common block within the NUCLEUS that contains the current default values of the picture options and the options scanning table that defines the input structure for UGOPTN. Its name is UGA002 and its declaration is incorporated into a module by including a module named UGPOTCBK. A second group of declarations is contained in the module UGPOTDCL. This item contains the declarations for the output structure for UGOPTN corresponding to the input structure in UGPOTCBK.

The POT is used by the following modules:
- NUCLEUS, UGMARK, UGLINE, UGPMRK, UGPLIN, UGTEXT,
- UGXTXT, UGPFIL, UG3MRK, UG3LIN, UG3PMK, UG3PLN,
- UG3TXT, UGDDAT, UGDEFL.

SECTION 2.5: THE SYMBOL DEFINITIONS

There are four symbol definition modules and all of them are common blocks. The symbol definition modules are:
1. The Marker Symbols. The name of this common block is UGA011.
2. The Basic Character Set. The name of this common block is UGA012.
3. The Extended/Simplex Character set. The name of this common block is UGA013.
4. The Extended/Duplex Character set. The name of this common block is UGA014.

The NUCLEUS contains the marker symbols and the basic character set. The NUCLEUS also contains dummy versions of the extended character sets; the actual data for the extended character sets is in SIMPLEX and DUPLEX. If any of these symbol definitions is modified, it should not be necessary to recompile the modules that reference them; the symbol definitions are self-defining tables.

A quick glance at the source code for any of these character set modules will show that they could not be generated directly. A
support program, named CHARPROC, is available which reads data consisting of a human readable description of the character sets and punches FORTRAN declaration and data statements that are part of these modules.

SECTION 2.6: THE ERROR MESSAGE MODULE (EMM)

The Error Message Module (EMM) is a common block within the NUCLEUS that contains items like the output unit number for the error messages, the print counts for the error messages, and the error messages themselves. Its name is UGA021 and its declaration is incorporated into a module by including a module named UGEMSCBK.

Without some help, the generation and maintenance of the error messages would be quite tedious. For this reason, a support program named ERRMPROC is available. This program reads data consisting of a human readable description of the error messages and punches FORTRAN declaration statements for UGEMSCBK and declaration and data statements for NUCLEUS.

The EMM is used by the following modules:
NUCLEUS, UGMCTL, UGRERR.
SECTION 3: INTERNAL SUBROUTINES

This section describes a number of internal subroutines. These subroutines are not called directly by the user of the Unified Graphics System but are instead called by other subroutines in the system.

In the following descriptions of the subroutines, floating point or character string arguments are always described as such; if nothing is said about the data type of a parameter, it is fixed point. All arguments described as character strings must be character string literals or of type CHARACTER. Almost all arguments represent input to these subroutines; when an argument is an output variable, it will be explicitly described as such and will be underlined in the list of parameters in the calling sequence. In some cases, the underlined parameter may be used for both input and output.

SECTION 3.1: MISCELLANEOUS MODULES

The subroutines described in this section are used to perform a miscellaneous set of functions.

SECTION 3.1.1: SUBROUTINE UGB001

This subroutine will move a subset of one array into another array. This subroutine is not usually called directly; instead, it is called through subroutine UGZ006 or UGZ007 when only the addresses of the arrays are known. The principal use of this subroutine is moving the copies of the DDA and DDX between their permanent and temporary storage locations when a graphic device is activated or deactivated.

The calling sequence is:

```
CALL UGB001(ARRAY1, INDEX1, ARRAY2, INDEX2, LENGTH)
```

The parameters in the calling sequence are:

- `ARRAY1` The target array.
- `INDEX1` The starting index in the target array.
- `ARRAY2` The source array.
- `INDEX2` The starting index in the source array.
- `LENGTH` The number of full words to be moved.

SECTION 3.1.2: SUBROUTINE UGB002

This subroutine is used by the graphic segment generators to find space in a graphic segment for new data. It receives a mode
specification and a data word count. It then checks to see if this data will fit into a graphic segment. If it will fit, a new mode specification is inserted, if necessary, and all pointers in the segment are updated. If it will not fit, an error indication is returned.

The calling sequence is:
CALL UGB002(MODE,NMODE,NDATA,SEGMENT,INDEX)

The parameters in the calling sequence are:
- **MODE** The mode specification block.
- **NMODE** The number of words in the mode specification block. If this number is negative, it indicates that a new mode specification must be inserted and the absolute value is used for the count.
- **NDATA** The number of data words to be added.
- **SEGMENT** The graphic segment which is to have the data added to it.
- **INDEX** The index of the start of the data within the graphic segment or a zero if the data will not fit.

**SECTION 3.1.3: SUBROUTINE UGB003**

This subroutine may be used to transform a point in the world coordinate system to the same point in the device coordinate system. The transformation data is obtained from the DDA.

The calling sequence is:
CALL UGB003(XWCORD,YWCORD,XDCORD,YDCORD)

The parameters in the calling sequence are:
- **XWCORD** The floating point X coordinate in the world coordinate system.
- **YWCORD** The floating point Y coordinate in the world coordinate system.
- **XDCORD** The X coordinate in the device coordinate system.
- **YDCORD** The Y coordinate in the device coordinate system.

**SECTION 3.1.4: SUBROUTINE UGB004**

This subroutine may be used to transform a point in the device coordinate system to the same point in the world coordinate system. The transformation data is obtained from the DDA.

The calling sequence is:
CALL UGB004(XDCORD,YDCORD,XWCORD,YWCORD)

The parameters in the calling sequence are:
- **XDCORD** The X coordinate in the device coordinate system.
- **YDCORD** The Y coordinate in the device coordinate system.
- **XWCORD** The floating point X coordinate in the world
coordinate system.
YWCORD  The floating point Y coordinate in the world coordinate system.

SECTION 3.1.5: SUBROUTINE UGB005

This subroutine is used to process the two-dimensional transformation data. It is called whenever a new two-dimensional window and view port is available. The transformation data (DDATR) and the shield data (DDASA and DDASF) are initialized.

The calling sequence is:
CALL UGB005

The calling sequence has no parameters.

SECTION 3.1.6: SUBROUTINE UGB006

This subroutine may be used to transform a point in the three-dimensional world coordinate system to the same point in the normalized three-dimensional coordinate system. The transformation data is obtained from the DDA.

The calling sequence is:
CALL UGB006(XWCORD,YWCORD,ZWCORD,XNCORD,YNCORD,ZNCORD)

The parameters in the calling sequence are:
XWCORD  The floating point X coordinate in the three-dimensional world coordinate system.
YWCORD  The floating point Y coordinate in the three-dimensional world coordinate system.
ZWCORD  The floating point Z coordinate in the three-dimensional world coordinate system.
XNCORD  The floating point X coordinate in the normalized three-dimensional coordinate system.
YNCORD  The floating point Y coordinate in the normalized three-dimensional coordinate system.
ZNCORD  The floating point Z coordinate in the normalized three-dimensional coordinate system.

SECTION 3.1.7: SUBROUTINE UGB007

This subroutine may be used to transform a point in the normalized three-dimensional world coordinate system to the same point in the three-dimensional device coordinate system. The transformation data is obtained from the DDA.

The calling sequence is:
CALL UGB007(XNCORD,YNCORD,ZNCORD,XDCORD,YDCORD,ZDCORD)
The parameters in the calling sequence are:

**XNCORD** The floating point X coordinate in the normalized three-dimensional world coordinate system.

**YNCORD** The floating point Y coordinate in the normalized three-dimensional world coordinate system.

**ZNCORD** The floating point Z coordinate in the normalized three-dimensional world coordinate system.

**XDCORD** The X coordinate in the three-dimensional device coordinate system.

**YDCORD** The Y coordinate in the three-dimensional device coordinate system.

**ZDCORD** The Z coordinate in the three-dimensional device coordinate system.

**SECTION 3.1.8: SUBROUTINE UGB008**

This subroutine may be used to transform a point in the three-dimensional world coordinate system to the same point in the three-dimensional device coordinate system. The transformation data is obtained from the DDA.

The calling sequence is:

```call ubg008(xncord, yncord, zncord, xdcord, ydcoord, zdcoord)```

The parameters in the calling sequence are:

**XWCORD** The floating point X coordinate in the three-dimensional world coordinate system.

**YWCORD** The floating point Y coordinate in the three-dimensional world coordinate system.

**ZWCORD** The floating point Z coordinate in the three-dimensional world coordinate system.

**XDCORD** The X coordinate in the three-dimensional device coordinate system.

**YDCORD** The Y coordinate in the three-dimensional device coordinate system.

**ZDCORD** The Z coordinate in the three-dimensional device coordinate system.

**SECTION 3.1.9: SUBROUTINE UGB009**

This subroutine may be used to transform a point in the three-dimensional device coordinate system to the same point in the three-dimensional world coordinate system. The transformation data is obtained from the DDA.

The calling sequence is:

```call ubg009(xdcord, ydcoord, zdcoord, xcord, ycord, zcord)```

The parameters in the calling sequence are:

**XDCORD** The X coordinate in the three-dimensional device coordinate system.
YDCORD  The Y coordinate in the three-dimensional device coordinate system.
ZDCORD  The Z coordinate in the three-dimensional device coordinate system.
XWCORD  The floating point X coordinate in the three-dimensional world coordinate system.
YWCORD  The floating point Y coordinate in the three-dimensional world coordinate system.
ZWCORD  The floating point Z coordinate in the three-dimensional world coordinate system.

SECTION 3.1.10: SUBROUTINE UGB010

This subroutine is used to do the initial processing of three-dimensional transformation data. It is called whenever a new three-dimensional view port and three-dimensional world volume is available. The three-dimensional view parameters (DDA30, DDA3E, DDA3U, DDA3N, and DDA3V) are initialized from the available data.

The calling sequence is:
CALL UGB010

The calling sequence has no parameters.

SECTION 3.1.11: SUBROUTINE UGB011

This subroutine is used to do the final processing of three-dimensional transformation data. It is called whenever a new three-dimensional object volume, eye point, upward direction, and projection flag are available. The basic items computed are the three-dimensional transformation data (DDA3T) and the three-dimensions to two-dimensions projection matrix (DDA3M). In addition, the near scissoring plane (DDA3P) and the processed three-dimensional view parameters (DDAXO, DDAXE, DDAXU, and DDAXN) are initialized.

The calling sequence is:
CALL UGB011(FLAG)

The parameter in the calling sequence is:
FLAG  An error flag: 0 means processing was complete, and 1 means the upward direction was invalid.

SECTION 3.1.12: SUBROUTINE UGB012

This subroutine may be used to transform a point in the three-dimensional world coordinate system to the same point in the user three-dimensional view port coordinate system. The transformation data is obtained from the DDA.
The calling sequence is:
CALL UGB012(XWCORD, YWCORD, ZWCORD, XVCORD, YVCORD)

The parameters in the calling sequence are:

- **XWCORD**: The floating point X coordinate in the three-dimensional world coordinate system.
- **YWCORD**: The floating point Y coordinate in the three-dimensional world coordinate system.
- **ZWCORD**: The floating point Z coordinate in the three-dimensional world coordinate system.
- **XVCORD**: The floating point X coordinate in the user three-dimensional view port coordinate system.
- **YVCORD**: The floating point Y coordinate in the user three-dimensional view port coordinate system.

SECTION 3.1.13: SUBROUTINE UGB013

This subroutine may be used to transform a point in the user three-dimensional view port coordinate system to the same point in the device three-dimensional view port coordinate system. The transformation data is obtained from the DDA.

The calling sequence is:
CALL UGB013(XVCORD, YVCORD, XDCORD, YDCORD)

The parameters in the calling sequence are:

- **XVCORD**: The floating point X coordinate in the user three-dimensional view port coordinate system.
- **YVCORD**: The floating point Y coordinate in the user three-dimensional view port coordinate system.
- **XDCORD**: The X coordinate in the device three-dimensional view port coordinate system.
- **YDCORD**: The Y coordinate in the device three-dimensional view port coordinate system.

SECTION 3.1.14: SUBROUTINE UGB014

This subroutine may be used to normalize a three-dimensional vector. The vector is also tested to make sure it is not too near zero in length to be useful.

The calling sequence is:
CALL UGB014(VECT, FLAG)

The parameters in the calling sequence are:

- **VECT**: The floating point vector to be normalized. This parameter should be an array of dimension 3.
- **FLAG**: An error flag: 0 means the vector could not be normalized, and 1 means it was normalized.
SECTION 3.1.15: SUBROUTINE UGB015

This subroutine may be used to compute the cross product of two three-dimensional vectors.

The calling sequence is:
CALL UGB015(VCT1,VCT2,VCT3)

The parameters in the calling sequence are:
VCT1 The floating point first input vector. This parameter should be an array of dimension 3.
VCT2 The floating point second input vector. This parameter should be an array of dimension 3.
VCT3 The floating point cross product that is to be computed. This parameter should be an array of dimension 3.

SECTION 3.2: PICTURE GENERATION MODULES

This section describes a group of subroutines that are used in generating pictures. They perform such functions as scissoring and shielding, line structure generation, and character stroke generation.

SECTION 3.2.1: SUBROUTINES UGC001, UGC002, UGC003, AND UGC004

These subroutines may be used to scissor and shield line segments. Subroutine UGC001 is used to supply the scissoring limits and initialize the process. Subroutine UGC002 may supply a number of optional shield specifications. Subroutine UGC003 is used to supply a line end point to the scissoring module, and subroutine UGC004 is used to retrieve a line end point from the scissoring module. The normal sequence of calls is the following:
1. UGC001 is called to initialize processing,
2. UGC002 is called a number of times to supply the optional shield specifications,
3. UGC003 is called to supply an end point to the module,
4. UGC004 is called repeatedly to retrieve end points of lines until it signals that no more data is available, and
5. Step 3 is repeated until no more input is available.

The calling sequences are:
CALL UGC001(SLIMS)
CALL UGC002(SLIMS)
CALL UGC003(BBIT,XCOORD,YCOORD)
CALL UGC004(BBIT,XCOORD,YCOORD)
The parameters in the calling sequences are:

**SLIMS**  A floating point array of dimension 2 by 2 which contains the scissoring or shielding limits. SLIMS(1,1) is the low X value, SLIMS(2,1) is the low Y value, SLIMS(1,2) is the high X value, and SLIMS(2,2) is the high Y value.

**BBIT**  The blanking bit or termination flag: 0 means move without drawing and 1 means draw. For UGC004, -1 means no more data is available.

**XCOORD**  The floating point X coordinate of a point or line end point.

**YCOORD**  The floating point Y coordinate of a point or line end point.

The actual line scissoring is done in subroutine UGC005, and the actual line shielding is done in subroutine UGC006. Subroutine UGC007 is used to determine the relation between a point and the scissoring or shielding limits. These subroutines will not be described here. The algorithm that is used in these subroutines will be described in Section 7.

**SECTION 3.2.2: SUBROUTINES UGD001, UGD002, AND UGD003**

These subroutines may be used to generate line structure. Solid lines may be broken down into "DASHED" lines, "DOTTED" lines, or "DOTDASH" lines. Subroutine UGD001 is used to initialize the process for a new series of line segments, subroutine UGD002 is used to supply a line end point to the line structure generating module, and subroutine UGD003 is used to retrieve a point or line end point from the line structure generating module. The normal sequence of calls is the following:

1. UGD001 is called to initialize processing,
2. UGD002 is called to supply an end point to the module,
3. UGD003 is called repeatedly to retrieve points or end points of lines until it signals that no more data is available, and
4. Step 2 is repeated until no more input is available.

The calling sequences are:

CALL UGD001(FLAG,XCMU,YCMU)
CALL UGD002(BBIT,XCOORD,YCOORD)
CALL UGD0003(BBIT,XCOORD,YCOORD)

The parameters in the calling sequences are:

**FLAG**  Line structure flag: 1 means "DASHED", 2 means "DOTTED", and 3 means "DOTDASH".

**XCMU**  A floating point value giving the centimeters per unit distance in the X direction.

**YCMU**  A floating point value giving the centimeters per unit distance in the Y direction.

**BBIT**  The blanking bit or termination flag: 0 means move without drawing and 1 means draw. For UGD003, 2 means a point is available and -1 means no more data is available.

**XCOORD**  The floating point X coordinate of a point or line end point.
point.

YCOORD The floating point Y coordinate of a point or line end point.

The algorithm that is used in these subroutines will be described in Section 7.

SECTION 3.2.3: SUBROUTINES UGE001 AND UGE002

These subroutines may be used to break a primary/secondary pair of character strings down into individual strokes. Two distinct operations may be performed:

1. Generate the actual stroke end points and return them, or:
2. Scan the characters and return the coordinates of the end of the string. The end of the string may be:
   A. The center of the last character, or:
   B. The center of the next character.

This subroutine is used to produce the marker symbols, the basic character set, and both extended character sets; the only requirement is that the correct symbol definitions be supplied to this subroutine. Subroutine UGE001 is used to initialize the process for a new pair of character strings. Subroutine UGE002 is used to obtain the next stroke end point. If the strokes are only being scanned, UGE002 need only be called once following the call the UGE001.

The calling sequences are:

CALL UGE001(FGDR,FGEQ,FGSC,FGMS,XCNTR,YCNTR,CSIZE,CANGL,
             YFACT,DATA,FGER)

CALL UGE002(CHRP,CHRS,CHRN,DATA,BBIT,XCOORD,YCOORD,SIZECH)

The parameters in the calling sequence are:

FGDR Drawing flag: 0 means generate the strokes, 1 means scan for last, and 2 means scan for next.
FGEQ Spacing flag: 0 means equal spacing is required, 1 means proportional spacing should be used if possible.
FGSC Special character flag: 0 means do not process superscript, subscript, etc. control characters, 1 means process these characters.
FGMS Missing secondary characters flag: 0 means CHRS will be a dummy argument, 1 means it will be given.
XCNTR The floating point X coordinate of the center of the first character.
YCNTR The floating point Y coordinate of the center of the first character.
CSIZE The floating point size of the characters.
CANGL The floating point angle the characters make with the horizontal.
YFACT The floating point Y factor multiplier for the output.
DATA The data structure defining the symbols.
FGER A flag which will be nonzero if DATA is invalid.
CHRP The primary character string.
The secondary character string.

The number of characters in CHRP and CHRS.

The blanking bit or termination flag: -1 means no more data is available, 0 means move without drawing, 1 means draw.

The floating point X coordinate of a stroke end point or the end of string X coordinate for a scan.

The floating point Y coordinate of a stroke end point or the end of string Y coordinate for a scan.

The floating point size change factor. This is only available when BBIT is -1.

A subroutine called UGE003 is used to do the transformation of the coordinates to their final coordinate system. This subroutine will not be described here. The algorithm that is used in these subroutines will be described in Section 7.

SECTION 3.2.4: SUBROUTINES UGF001, UGF002, AND UGF003

These subroutines may be used to scissor polygons. Subroutine UGF001 is used to supply the scissoring limits and initialize the process. Subroutine UGF002 is used to supply a polygon to the scissoring module, and subroutine UGF003 is used to retrieve a polygon from the scissoring module. The normal sequence of calls is the following:

1. UGF001 is called to initialize processing,
2. UGF002 is called to supply a polygon to the module,
3. UGF003 is called repeatedly to retrieve polygons until it signals that no more data is available, and
4. Step 2 is repeated until no more input is available.

The calling sequences are:

CALL UGF001(SLIMS)
CALL UGF002(XARRAY,YARRAY,NPTS,ERFG)
CALL UGF003(XARRAY,YARRAY,NPTS)

The parameters in the calling sequences are:

SLIMS A floating point array of dimension 2 by 2 which contains the scissoring limits. SLIMS(1,1) is the low X value, SLIMS(2,1) is the low Y value, SLIMS(1,2) is the high X value, and SLIMS(2,2) is the high Y value.

XARRAY The floating point X coordinates of the vertices of the polygon.

YARRAY The floating point Y coordinates of the vertices of the polygon.

NPTS The number of vertices in the polygon. The first and last vertices of the polygon must be identical. The input polygon must not have more than 32 vertices, the output polygons will have no more than 64 vertices. For UGF003, -1 means no more data is available.

ERFG An error flag: 0 means no error, and 1 means an internal table has overflown.
The actual line scissoring is done in subroutine UGF004. This subroutine will not be described here. The algorithm that is used in these subroutines will be described in Section 7.

SECTION 3.2.5: SUBROUTINES UGG001, UGG002, UGG003, AND UGG004

These subroutines may be used to scissor three-dimensional line segments. The line segments may be scissored against a plane or a rectangular parallelepiped. Subroutine UGG001 is used to initialize for scissoring against a plane, while subroutine UGG002 is used to initialize for scissoring against a rectangular parallelepiped. Subroutine UGG003 is used to supply a line end point to the scissoring module, and subroutine UGG004 is used to retrieve a line end point from the scissoring module. The normal sequence of calls is the following:
1. UGG001 or UGG002 is called to initialize processing,
2. UGG003 is called to supply an end point to the module,
3. UGG004 is called repeatedly to retrieve end points of lines until it signals that no more data is available, and
4. Step 2 is repeated until no more input is available.

The calling sequences are:
CALL UGG001(PLANE)
CALL UGG002(SLIMS)
CALL UGG003(BBIT,XCOORD,YCOORD,ZCOORD)
CALL UGG004(BBIT,XCOORD,YCOORD,ZCOORD)

The parameters in the calling sequences are:
PLANE A floating point array of dimension 4 which contains the scissoring plane. The points on the positive side of the plane are the ones which will be saved.
SLIMS A floating point array of dimension 3 by 2 which contains the rectangular parallelepiped scissoring limits. SLIMS(1,1) is the low X value, SLIMS(2,1) is the low Y value, SLIMS(3,1) is the low Z value, SLIMS(1,2) is the high X value, SLIMS(2,2) is the high Y value, and SLIMS(3,2) is the high Z value.
BBIT The blanking bit or termination flag: 0 means move without drawing and 1 means draw. For UGG004, -1 means no more data is available.
XCOORD The floating point X coordinate of a line end point.
YCOORD The floating point Y coordinate of a line end point.
ZCOORD The floating point Z coordinate of a line end point.

Subroutine UGG005 is used to determine the relation between a point and the rectangular parallelepiped scissoring limits. This subroutine will not be described here. The algorithm that is used in these subroutines will be described in Section 7.
SECTION 3.3: OPERATING SYSTEM DEPENDENT MODULES

The subroutines described below are very dependent on the operating system and must usually be written in Assembler Language. Although the operations are performed in an operating system dependent manner, they represent functions which can be performed on almost all operating systems.

SECTION 3.3.1: SUBROUTINE UGZ001

This subroutine is used to terminate execution. It is used when a terminal error has been encountered and no further processing is possible. The subroutine should produce a memory dump, generate a subroutine traceback, enter the debugger, or do anything else that is appropriate on the specific computer. This subroutine is called by UGRERR when a level 4 error has been encountered and by many other subroutines when an "impossible" situation has arisen.

The calling sequence is:
CALL UGZ001

The calling sequence has no parameters.

SECTION 3.3.2: SUBROUTINE UGZ002

This subroutine may be used to "activate" or "deactivate" a named module. When a module is activated, the address of the module is returned. If the module cannot be found, a zero is returned. When a module is deactivated, both its name and address should be given. In the current implementations of the Unified Graphics System, a subroutine is activated by simply looking its address up in a table and deactivation does not require any action. However, the Unified Graphics System has been written with the possibility in mind that, in future implementations, activation may mean that a module is dynamically loaded into memory and deactivation may mean that it is deleted from memory. The modules that this subroutine must operate on are the extended character sets and the device-dependent modules.

The calling sequence is:
CALL UGZ002(FLAG, NAME, POINTER)

The parameters in the calling sequence are:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLAG</td>
<td>The operation indicator: 0 means activate and 1 means deactivate.</td>
</tr>
<tr>
<td>NAME</td>
<td>The name of the module to be activated or deactivated. This argument must be a character string of 8 characters with the name padded on the right with blanks.</td>
</tr>
<tr>
<td>POINTER</td>
<td>The address of the module.</td>
</tr>
</tbody>
</table>
SECTION 3.3.3: SUBROUTINE UGZ003

This subroutine may be used to allocate or deallocate a block of memory. When a block of memory is allocated, the address of the block is returned. When a block of memory is deallocated, its address and length must both be given. Current implementations of the Unified Graphics System are under operating systems which support this function, but this subroutine could operate correctly under operating systems which do not support dynamic allocation of memory. In that case, UGZ003 could simply supply blocks of memory from a larger block that is internal to UGZ003.

The calling sequence is:

CALL UGZ003(FLAG,SIZE,POINTER)

The parameters in the calling sequence are:
- **FLAG** The operation indicator: 0 means allocate and 1 means deallocate.
- **SIZE** The size of the block of memory. This value must be given in full words.
- **POINTER** The address of the block of memory.

SECTION 3.3.4: SUBROUTINE UGZ004

This subroutine may be used to determine the address of a subroutine within the current load module.

The calling sequence is:

CALL UGZ004(SUBR,POINTER)

The parameters in the calling sequence are:
- **SUBR** The subroutine whose address is needed. In FORTRAN calling programs, this argument should be declared as EXTERNAL.
- **POINTER** The address of the subroutine.

SECTION 3.3.5: SUBROUTINE UGZ005

This subroutine may be used to determine the address of a data element (not a character string) within the current load module.

The calling sequence is:

CALL UGZ005(DATA,POINTER)

The parameters in the calling sequence are:
- **DATA** The data item whose address is needed.
- **POINTER** The address of the data item.
SECTION 3.3.6: SUBROUTINES UGZ006 AND UGZ007

These two subroutines may be used to execute a subroutine when the address of the subroutine and/or the addresses of some of the subroutine’s arguments are known. The address of the subroutine must have been generated by subroutines UGZ002 or UGZ004. The addresses of the arguments must have been produced by subroutines UGZ003 or UGZ005.

These two subroutines are identical in all respects except their name. UGZ006 is used by the device-independent modules. In particular, the device-independent modules use UGZ006 to execute the device-dependent code. Any use of UGZ006 in the device-dependent code would therefore imply recursive use of UGZ006. Since these subroutines are not recursive, UGZ007 is supplied for use within the device-dependent code.

The calling sequences are:
CALL UGZ006(SADDR,NARG,IARG,ARG1,ARG2,...)
CALL UGZ007(SADDR,NARG,IARG,ARG1,ARG2,...)

The parameters in the calling sequence are:
SADDR The address of the subroutine to be called.
NARG The number of arguments that are given by addresses.
IARG An array containing the indices of the arguments given by addresses.
ARG1 The first actual argument.
ARG2 The second actual argument.
... ...
SECTION 4: THE INTERNAL FORMAT OF A GRAPHIC SEGMENT

This section describes, in detail, the content of a graphic segment. A programmer normally constructs graphic segments by calling the procedures UGLINE, UGTEXT, etc. It is possible for a programmer to construct graphic segments directly from the information given here. However, it should be remembered that most of the checking for invalid data is done when the graphic segment is generated; little checking for invalid graphic segments is done when the segment is converted to device-dependent orders. It is also possible that changes or additions may have to be made to some of these descriptions in the future.

A graphic segment is an array of full words reserved to contain picture description information. Basically, the content of a graphic segment is a few items of information at the beginning, followed by "blocks" of picture data. Each block begins with a "mode specification" and contains either markers, lines, text, or other such information. In detail, this is:

- SEGMENT(1): Current length of the graphic segment (N).
- SEGMENT(2): The index of the start of the first block (J).
  
  At present, this value is always 4.
- SEGMENT(3): The index of the start of the last block (K).
- SEGMENT(J): The start of the first block.
- SEGMENT(K): The start of the last block.
- SEGMENT(N+1): The maximum length of the graphic segment, that is, the dimension of the array minus one.

Notice that if a graphic segment is to be saved in a data set, SEGMENT(1) gives the number of words which must be written. The value in SEGMENT(N+1) need not be saved because its value will depend on the size of the array that contains the graphic segment when it is retrieved.

When procedure UGINIT is called with the CLEAR option, it resets the graphic segment to the following:

- SEGMENT(1): 3
- SEGMENT(2): 4
- SEGMENT(3): 0
- SEGMENT(4): NSEG-1

where NSEG is the value of the third argument of UGINIT. When the RESET option is used, UGINIT sets:

- SEGMENT(SEGMENT(1)+1): NSEG-1

The CONTINUE option instructs UGINIT to do the same thing as for CLEAR unless the last mode specification was for two-dimensional line or three-dimensional line data. In that case it sets:

- SEGMENT(1): N, where N is 12 for two-dimensional line data and 13 for three-dimensional line data.
- SEGMENT(2): 4
- SEGMENT(3): 4
- SEGMENT(4)...SEGMENT(N): A copy of the block specification for two-dimensional lines or three-dimensional lines and the last end point from the line data with
the blanking bit set to zero.

**SEGMENT(N+1): NSEG-1**

The graphic segment generators work by moving the word with the maximum length down and inserting a block of data into the segment. **SEGMENT(1)**, and if necessary **SEGMENT(3)**, are then reset. The graphic segment generators use the pointer to the last block to avoid adding a new block specification when it is not needed.

**SECTION 4.1: MARKER DATA**

A marker data block consists of the following:

- **SEGMENT(I): 1**
- **SEGMENT(I+1):** The number of fullwords in the block.
- **SEGMENT(I+2):** The intensity level: 1 means VDIM, 2 means DIM, 3 means MEDIUM, 4 means BRIGHT, and 5 means VBRIGHT.
- **SEGMENT(I+3):** The color: 1 means WHITE, 2 means RED, 3 means GREEN, 4 means BLUE, 5 means YELLOW, 6 means MAGENTA, 7 means CYAN, and 8 means BLACK.
- **SEGMENT(I+4):** The blinking mode: 1 means STEADY and 2 means BLINK.
- **SEGMENT(I+5):** The pick identification.
- **SEGMENT(I+6):** The floating point SIZE or DSIZE value. If the value is positive, it represents SIZE, if it is negative, it represents DSIZE.
- **SEGMENT(I+7):** The marker value: -1 for a point or 0 through 9 for a marker symbol.

Following this are a group of X and Y coordinates in floating point form.

**SECTION 4.2: LINE DATA**

A line data block consists of the following:

- **SEGMENT(I): 2**
- **SEGMENT(I+1):** The number of fullwords in the block.
- **SEGMENT(I+2):** The intensity level: 1 means VDIM, 2 means DIM, 3 means MEDIUM, 4 means BRIGHT, and 5 means VBRIGHT.
- **SEGMENT(I+3):** The color: 1 means WHITE, 2 means RED, 3 means GREEN, 4 means BLUE, 5 means YELLOW, 6 means MAGENTA, 7 means CYAN, and 8 means BLACK.
- **SEGMENT(I+4):** The blinking mode: 1 means STEADY and 2 means BLINK.
- **SEGMENT(I+5):** The pick identification.
- **SEGMENT(I+6):** The line structure: 1 means SOLID, 2 means
DASHED, 3 means DOTTED, and 4 means DOTDASH.
Following this are a group of X and Y coordinates, in floating point form, of the end points of the line segments. The blanking bit is in the least significant bit of the Y coordinate. This is one of the places where the source code for the IBM computers differs from that of the VAX computers; the least significant bit of a floating point number is in a very unusual position on the VAX computers.

SECTION 4.3: TEXT DATA

A text data block consists of the following:

SEGMENT(I): 3
SEGMENT(I+1): The number of fullwords in the block.
SEGMENT(I+2): The intensity level: 1 means VDIM, 2 means DIM, 3 means MEDIUM, 4 means BRIGHT, and 5 means VBRIGHT.
SEGMENT(I+3): The color: 1 means WHITE, 2 means RED, 3 means GREEN, 4 means BLUE, 5 means YELLOW, 6 means MAGENTA, 7 means CYAN, and 8 means BLACK.
SEGMENT(I+4): The blinking mode: 1 means STEADY and 2 means BLINK.
SEGMENT(I+5): The pick identification.
SEGMENT(I+6): The floating point SIZE or DSIZE value. If the value is positive, it represents SIZE; if it is negative, it represents DSIZE.
SEGMENT(I+7): The floating point ANGLE value.
SEGMENT(I+8): The justification value: 1 means LEFT, 2 means RIGHT, and 3 means CENTER.
SEGMENT(I+9): The generation flag: 1 means NORMGN, 2 means HARDGN, and 3 means SOFTGN.
Following this may be an arbitrary number of sub-blocks consisting of the following:
SEGMENT(I): The number of fullwords in the sub-block.
SEGMENT(I+1): The floating point X coordinate of the character string.
SEGMENT(I+2): The floating point Y coordinate of the character string.
SEGMENT(I+3): The number of characters in the character string.
SEGMENT(I+4): The start of the character string. If the characters in the character string do not fill an integral number of words, the end of the string is padded with blanks.
SECTION 4.4: EXTENDED TEXT DATA

An extended text data block consists of the following:

SEGMENT(I): 4
SEGMENT(I+1): The number of fullwords in the block.
SEGMENT(I+2): The intensity level: 1 means VDIM, 2 means DIM,
3 means MEDIUM, 4 means BRIGHT, and 5 means VBRIGHT.
SEGMENT(I+3): The color: 1 means WHITE, 2 means RED, 3 means
GREEN, 4 means BLUE, 5 means YELLOW, 6 means
MAGENTA, 7 means CYAN, and 8 means BLACK.
SEGMENT(I+4): The blinking mode: 1 means STEADY and 2 means
BLINK.
SEGMENT(I+5): The pick identification.
SEGMENT(I+6): The floating point SIZE or DSIZE value. If the
value is positive, it represents SIZE; if it is
negative, it represents DSIZE.
SEGMENT(I+7): The floating point ANGLE value.
SEGMENT(I+8): The justification value: 1 means LEFT, 2 means
RIGHT, and 3 means CENTER.
SEGMENT(I+9): The fixed size flag: 1 means NOFIXSIZE and 2 means
FIXSIZE.

Following this may be an arbitrary number of sub-blocks
consisting of the following:
SEGMENT(I): The number of fullwords in the sub-block.
SEGMENT(I+1): The floating point X coordinate of the character
string.
SEGMENT(I+2): The floating point Y coordinate of the character
string.
SEGMENT(I+3): The number of characters in the character
strings.
SEGMENT(I+4): The start of the primary character string. If
the characters in the character string do not
fill an integral number of words, the end of the
string is padded with blanks.

... SEGMENT(I+L): The start of the secondary character string
where L=(SEGMENT(I+3)+3)/4. If the characters
in the character string do not fill an integral
number of words, the end of the string is padded
with blanks.

... 

SECTION 4.5: POLYGON FILL DATA

A polygon fill data block consists of the following:
SEGMENT(I): 5
SEGMENT(I+1): The number of fullwords in the block.
SEGMENT(I+2): The intensity level: 1 means VDIM, 2 means DIM,
3 means MEDIUM, 4 means BRIGHT, and 5 means
VBRIGHT.
SEGMENT(I+3): The color: 1 means WHITE, 2 means RED, 3 means GREEN, 4 means BLUE, 5 means YELLOW, 6 means MAGENTA, 7 means CYAN, and 8 means BLACK.
SEGMENT(I+4): The blinking mode: 1 means STEADY and 2 means BLINK.
SEGMENT(I+5): The pick identification.
SEGMENT(I+6): The number of vertices in the polygon.
Following this are a group of X and Y coordinates, in floating point form, of the vertices of the polygon. The last vertex is always identical to the first vertex.

SECTION 4.6: DEVICE-DEPENDENT DATA

A device-dependent data block consists of the following:
SEGMENT(I): 6
SEGMENT(I+1): The number of fullwords in the block.
SEGMENT(I+2): The intensity level: 1 means VDIM, 2 means DIM, 3 means MEDIUM, 4 means BRIGHT, and 5 means VBRIGHT.
SEGMENT(I+3): The color: 1 means WHITE, 2 means RED, 3 means GREEN, 4 means BLUE, 5 means YELLOW, 6 means MAGENTA, 7 means CYAN, and 8 means BLACK.
SEGMENT(I+4): The blinking mode: 1 means STEADY and 2 means BLINK.
SEGMENT(I+5): The pick identification.
Following this may be an arbitrary number of sub-blocks consisting of the following:
SEGMENT(I): The number of fullwords in the sub-block.
SEGMENT(I+1): The floating point X coordinate of the device-dependent data.
SEGMENT(I+2): The floating point Y coordinate of the device-dependent data.
SEGMENT(I+3): The number of characters in the device-dependent data.
SEGMENT(I+4): The start of the device-dependent data. If the characters in the device-dependent data do not fill an integral number of words, the end of the string is padded with NULL characters.

SECTION 4.7: THREE-DIMENSIONAL MARKER DATA

A three-dimensional marker data block consists of the following:
SEGMENT(I): 7
SEGMENT(I+1): The number of fullwords in the block.
SEGMENT(I+2): The intensity level: 1 means VDIM, 2 means DIM, 3 means MEDIUM, 4 means BRIGHT, and 5 means
SEGMENT(I+3): The color: 1 means WHITE, 2 means RED, 3 means GREEN, 4 means BLUE, 5 means YELLOW, 6 means MAGENTA, 7 means CYAN, and 8 means BLACK.

SEGMENT(I+4): The blinking mode: 1 means STEADY and 2 means BLINK.

SEGMENT(I+5): The pick identification.

SEGMENT(I+6): For compatibility with two-dimensional markers, this value is always -0.015.

SEGMENT(I+7): For compatibility with two-dimensional markers, this value is always -1.

Following this are a group of X, Y, and Z coordinates in floating point form.

SECTION 4.8: THREE-DIMENSIONAL LINE DATA

A three-dimensional line data block consists of the following:

SEGMENT(I): 8

SEGMENT(I+1): The number of fullwords in the block.

SEGMENT(I+2): The intensity level: 1 means VDIM, 2 means DIM, 3 means MEDIUM, 4 means BRIGHT, and 5 means VBRIGHT.

SEGMENT(I+3): The color: 1 means WHITE, 2 means RED, 3 means GREEN, 4 means BLUE, 5 means YELLOW, 6 means MAGENTA, 7 means CYAN, and 8 means BLACK.

SEGMENT(I+4): The blinking mode: 1 means STEADY and 2 means BLINK.

SEGMENT(I+5): The pick identification.

SEGMENT(I+6): For compatibility with two-dimensional lines, this value is always 1.

Following this are a group of X, Y, and Z coordinates, in floating point form, of the end points of the line segments. The blanking bit is in the least significant bit of the Y coordinate.

SECTION 4.9: THREE-DIMENSIONAL TEXT DATA

A three-dimensional text data block consists of the following:

SEGMENT(I): 9

SEGMENT(I+1): The number of fullwords in the block.

SEGMENT(I+2): The intensity level: 1 means VDIM, 2 means DIM, 3 means MEDIUM, 4 means BRIGHT, and 5 means VBRIGHT.

SEGMENT(I+3): The color: 1 means WHITE, 2 means RED, 3 means GREEN, 4 means BLUE, 5 means YELLOW, 6 means MAGENTA, 7 means CYAN, and 8 means BLACK.

SEGMENT(I+4): The blinking mode: 1 means STEADY and 2 means BLINK.
SEGMENT(I+5): The pick identification.
SEGMENT(I+6): The floating point DSIZE value. For compatibility with two-dimensional text, this value is always negative.
SEGMENT(I+7): The floating point ANGLE value.
SEGMENT(I+8): The justification value: 1 means LEFT, 2 means RIGHT, and 3 means CENTER.
SEGMENT(I+9): For compatibility with two-dimensional text, this value is always 2.

Following this may be an arbitrary number of sub-blocks consisting of the following:
SEGMENT(I): The number of fullwords in the sub-block.
SEGMENT(I+1): The floating point X coordinate of the character string.
SEGMENT(I+2): The floating point Y coordinate of the character string.
SEGMENT(I+3): The floating point Z coordinate of the character string.
SEGMENT(I+4): The number of characters in the character string.
SEGMENT(I+5): The start of the character string. If the characters in the character string do not fill an integral number of words, the end of the string is padded with blanks.

...
SECTION 5: THE DEVICE-DEPENDENT MODULE

A device-dependent module consists of a common block, the DDX for the device, and a group of subroutines that perform the device-dependent functions. The names of the common block and subroutines within a device-dependent module always consist of the letters "UG", followed by two identifying letters for the device, followed by a two digit number. The digits associated with the common block are always "00" and the digits associated with the principal entry point are always "01". The device-independent modules always enter the device-dependent modules through this principal entry point, and its calling sequence is the same for all graphic devices. For example, the name of the DDX for the VERSATEC plotter using fan-fold paper is UGVFO00, and the principal entry point is UGVFO01. The device-dependent module also includes a large number of subroutines with names UGVFO02, UGVFO03, etc.

The calling sequence for the principal entry point in the device-dependent module is:

CALL UGXX01(DDIN, DDST, DDEX)

The parameters in the calling sequence are:

DDIN An integer array containing input information. In particular, DDIN(1) always contains a value that specifies the operation to be performed.

DDST A character string that is used for both input and output.

DDEX An integer array containing output information. In particular, DDEX(1) always contains a value that specifies if the operation was successful or not.

The following sections describe each of the operations that the device-dependent modules must perform. There are ten operations that may be requested of non-interactive or slave-display devices, and four additional operations that may be requested of interactive graphic devices. If the device can process three-dimensional data, then a fifteenth operation may be requested.

SECTION 5.1: OPEN THE GRAPHIC DEVICE

This operation is requested when UGOPEN is called. The meaning of the arguments for this operation are:

1. DDIN(1): 1, the operation identification.
2. DDST: The options list supplied to UGOPEN.
3. DDEX(1): 0 if the operation was successful, 1 if the operation was unsuccessful, and -1 if a dummy device-dependent module was called.

There are a group of items in the DDA that must be set correctly if the device-independent modules are to work right. DDALX and
DDACX must be initialized if UGSLCT is to work. DDAAT should be set to the string supplied to UGOPEN so UGINFO can return this value when requested. The items DDAIL, DDAHM, DDADF, and DDAIC are important because the device-independent modules rely on these values to decide what the graphic device can do. DDABD, DDABX, and DDABY must be set so that the device-independent modules can access the screen. For a graphic device with three-dimensional capability, the extent of the three-dimensional coordinate system must be given in DDA3D.

SECTION 5.2: CLOSE THE GRAPHIC DEVICE

This operation is requested when UGCLOS is called. The meaning of the arguments for this operation are:
1. DDIN(1): 2, the operation identification.
   DDIN(2): 0 means normal close function, and
            1 means NOCLEAR is given.
2. DDST: The options list supplied to UGCLOS.
3. DDEX(1): 0 if the operation was successful, and
            1 if the operation was unsuccessful.

SECTION 5.3: CLEAR SCREEN OR WINDOW

This operation is requested when it is explicitly asked for by UGPICT and at other times when it is necessary. The meaning of the arguments for this operation are:
1. DDIN(1): 3, the operation identification.
   DDIN(2): 0 means clear full screen, and
            1 means clear the current window specified by DDAND.
2. DDST: The alias name, when it is appropriate, or 8 blank characters.
3. DDEX(1): 0 if the operation was successful, and
            1 if the operation was unsuccessful.

The clear-window operation will only be requested of raster-scan graphic devices.

SECTION 5.4: MANIPULATE THE SCREEN

This operation is requested when it is explicitly asked for by UGPICT and at other times when it is necessary. The meaning of the arguments for this operation are:
1. DDIN(1): 4, the operation identification.


**MAINTENANCE MANUAL**

DDIN(2): 1 means turn the display unit on, and 2 means turn the unit off.

2. DDST: This parameter is not used.

3. DDEX(1): 0 if the operation was successful, and 1 if the operation was unsuccessful.

This operation can only be performed on a few slave-display or interactive devices. If the device-dependent module cannot perform this operation, it may simply ignore it.

**SECTION 5.5: BEGIN A GRAPHIC SEGMENT**

This operation is requested when UGWRIT begins operating. The meaning of the arguments for this operation are:

1. DDIN(1): 5, the operation identification.
   DDIN(2): The graphic segment identification.
   DDIN(3): The address of the graphic segment.
   DDIN(4): 0 means draw the segment, and 1 means erase the segment.
   DDIN(5): 0 means NOPICK, and 1 means PICK.
   DDIN(6): 0 means INCLUDE, and 1 means OMIT.

2. DDST: The options list supplied to UGWRIT.

3. DDEX(1): 0 if the operation was successful, 1 if the operation was unsuccessful, and -1 if the entire graphic segment has been successfully processed. In this case, the graphic segment will not be broken down into individual graphic items by UGWRIT.

   DDEX(2): For three-dimensional graphic devices:
   0 means scissor to world volume, and 1 means scissor to object volume. For two-dimensional devices, this item is not usually used, but it should be set to 0 for consistency.

   Except for interactive refresh graphic displays, there is usually not too much to do. The erase operation will only be requested of raster-scan graphic devices, and the PICK/NOPICK and INCLUDE/OMIT operations will only be requested for refresh display devices.

   The only time the address of the graphic segment in DDIN(3) is used, is in the device-dependent module for PDEVUGS.

**SECTION 5.6: TERMINATE A GRAPHIC SEGMENT**

This operation is requested when UGWRIT is about to finish executing. The meaning of the arguments for this operation are:
1. DDIN(1): 6, the operation identification.
2. DDST: This parameter is not used.
3. DDEX(1): 0 if the operation was successful, and 1 if the operation was unsuccessful. This error normally means that the graphic segment does not fit into the display file.

SECTION 5.7: MANIPULATE A GRAPHIC SEGMENT

This operation is requested by UGPICT when a segment manipulation operation is called for. The meaning of the arguments for this operation are:

1. DDIN(1): 7, the operation identification.
   DDIN(2): The graphic segment identification.
   DDIN(3): 0 means do not delete the segment, and 1 means delete the segment.
   DDIN(4): 0 means do not change PICK status, 1 means set status to NOPICK, and 2 means set status to PICK.
   DDIN(5): 0 means do not change INCLUDE/OMIT status, 1 means set status to INCLUDE, and 2 means set status to OMIT.
2. DDST: The options list supplied to UGPICT.
3. DDEX(1): 0 if the operation was successful, and 1 if the operation was unsuccessful.

This function is only requested for refresh graphic devices.

SECTION 5.8: GRAPHIC INQUIRY

This operation is requested when subroutine UGWRIIT must know what the graphic device can display. The meaning of the arguments for this operation are:

1. DDIN(1): 8, the operation identification.
   DDIN(2): 1 means points, 2 means lines, 3 means text, 4 means polygon fill, 5 means device-dependent data, 6 means three-dimensional points, 7 means three-dimensional lines, and 8 means three-dimensional text.
   DDIN(3): The intensity level: 1 means VDIM, 2 means DIM, 3 means MEDIUM, 4 means BRIGHT, and 5 means VBRIGHT.
   DDIN(4): The color: 1 means WHITE, 2 means RED, 3 means GREEN, 4 means BLUE, 5 means YELLOW, 6 means MAGENTA, 7 means CYAN, and 8 means BLACK.
DDIN(5): The blinking mode: 1 means STEADY, and 2 means BLINK.

DDIN(6): The pick identification.
For lines only:
DDIN(7): The line structure: 1 means SOLID, 2 means DASHED, 3 means DOTTED, and 4 means DOTDASH.

For text only:
DDIN(7): The required size in raster units.
DDIN(8): The angle in degrees. This value is always between 0 and 359.
DDIN(9): The generation flag: 1 means NORMGN, and 2 means HARDGN.

For polygon fill only:
DDIN(7): The number of vertices in the polygon. This value may be as large as 64, and the first and last points are always identical.
DDIN(8): The X coordinate of the first vertex.
DDIN(9): The Y coordinate of the first vertex.

For three-dimensional lines only:
DDIN(7): Always 1 for compatibility with the two-dimensional lines.

For three-dimensional text only:
DDIN(7): The required size in raster units.
DDIN(8): The angle in degrees. This value is always between 0 and 359.
DDIN(9): Always 2 for compatibility with two-dimensional text.

1. DDST: This parameter is not used.
2. DDEX(1): 0 if the data can be processed, and 1 if the data cannot be processed.

For two-dimensional or three-dimensional text only:
DDEX(2): Delta X offset from center.
DDEX(3): Delta Y offset from center.
DDEX(4): Delta X between characters.
DDEX(5): Delta Y between characters.

The device-dependent module must always indicate that two-dimensional points and two-dimensional solid lines can be processed. A device will only be asked to process three-dimensional data if it has identified itself as a three-dimensional device, however, in that case, it must indicate that all three-dimensional primitives can be processed. Any of the other items can be simulated by the device-independent modules. The intensity level, color, and blink mode data should be saved for succeeding Graphic-Display functions. The reason that the vertices are supplied in the polygon fill inquiry is that some devices may only be able to generate certain kinds of polygons; this gives the device-dependent code the opportunity to examine the polygon in detail. The polygon will again be supplied in the graphic display operation.
SECTION 5.9: GRAPHIC DISPLAY

This operation is requested when subroutine UGWRIT has a display item ready for the device-dependent module. If the graphic inquiry response is correct, the device-dependent module will never be asked to perform an impossible action. The meaning of the arguments for this operation are:

1. **DDIN(1):** 9, the operation identification.
   - **DDIN(2):** 1 means points,
     - 2 means lines,
     - 3 means text,
     - 4 means polygon fill,
     - 5 means device-dependent data,
     - 6 means three-dimensional points,
     - 7 means three-dimensional lines, and
     - 8 means three-dimensional text.
   - **DDIN(3):** For points only:
   - **DDIN(4):** The X coordinate of the point.
   - For lines only:
   - **DDIN(3):** The X coordinate of the line end point.
   - **DDIN(4):** The Y coordinate of the line end point.
   - **DDIN(5):** The blanking bit: 0 means blank and 1 means draw.
   - For text only:
   - **DDIN(3):** The X coordinate of the first character.
   - **DDIN(4):** The Y coordinate of the first character.
   - For polygon fill only:
   - **DDIN(3):** The number of vertices in the polygon. This value may be as large as 64 and the first and last points are always identical.
   - **DDIN(4):** The X coordinate of the first vertex.
   - **DDIN(5):** The Y coordinate of the first vertex.
   - ... 
   - For device-dependent data only:
   - **DDIN(3):** The X coordinate associated with the data.
   - **DDIN(4):** The Y coordinate associated with the data.
   - For three-dimensional points only:
   - **DDIN(3):** The X coordinate of the point.
   - **DDIN(4):** The Y coordinate of the point.
   - **DDIN(5):** The Z coordinate of the point.
   - For three-dimensional lines only:
   - **DDIN(3):** The X coordinate of the line end point.
   - **DDIN(4):** The Y coordinate of the line end point.
   - **DDIN(5):** The Z coordinate of the line end point.
   - **DDIN(6):** The blanking bit: 0 means blank and 1 means draw.
   - For three-dimensional text only:
   - **DDIN(3):** The X coordinate of the first character.
   - **DDIN(4):** The Y coordinate of the first character.
   - **DDIN(5):** The Z coordinate of the first character.
   - **DDIN(6):** The delta-X offset of the text.
   - **DDIN(7):** The delta-Y offset of the text.
   - **DDST:** For two-dimensional or three-dimensional text, this contains the string to be displayed. For
device-dependent data, this parameter contains the data.

3. DDEX(1): 0 if the data can be processed, and 1 if the operation was unsuccessful. This error normally means that the graphic data does not fit into the current graphic segment.

If a graphic inquiry responds with a "data cannot be processed" signal because of line structure, UGWRIT responds by re-inquiring with a specification of solid lines. UGWRIT then breaks the line down into its constituent pieces and sends graphic display requests. Notice that this means, in the case of DOTTED or DOTDASH lines, that point data will be supplied after a graphic inquiry for solid lines.

SECTION 5.10: MISCELLANEOUS CONTROL

This operation is requested by UGMCTL when necessary. The meaning of the arguments for this operation are:

1. DDIN(1): 10, the operation identification.
   DDIN(2): 1 means BEEP.
2. DDST: This parameter is not used.
3. DDEX(1): 0 if the operation was successful, and 1 if the operation was unsuccessful.

If the device-dependent module cannot perform this operation, it may simply ignore it.

SECTION 5.11: MODIFY STATUS OF A CONTROL UNIT

This operation is requested by UGECTL when necessary. The meaning of the arguments for this operation are:

1. DDIN(1): 11, the operation identification.
   DDIN(2): 1 means keyboard data is ready, that is, the DDA variables DDAKX, DDAKY, DDAKN, and DDAKS have been changed,
   3 means button data is ready, that is, the DDA variable DDABF has been changed,
   4 means stroke data is ready, that is, the DDA variables DDASL, DDAST, and DDASN have been changed.
2. DDST: This parameter is not used.
3. DDEX(1): 0 if the operation was successful, and 1 if the operation was unsuccessful.

There is often very little that has to be done when this operation is requested.
SECTION 5.12: ENABLE OR DISABLE A CONTROL UNIT

This operation is requested by subroutine UGENAB and UGDSAB. The meaning of the arguments for this operation are:

1. DDIN(1): 12, the operation identification.
   DDIN(2): 0 means disable, and
   1 means enable the control unit.
   DDIN(3): 1 means KEYBOARD,
   2 means PICK,
   3 means BUTTON,
   4 means STROKE,
   5 means LOCATOR, and
   6 means VALUATOR.

2. DDST: This parameter is not used.

3. DDEX(1): 0 if the operation was successful, and
   1 if the operation was unsuccessful.

If the open function is correct, the device-dependent module will never be asked to perform an impossible action.

SECTION 5.13: OBTAIN AN EVENT

This operation is requested by subroutine UGEVNT. The meaning of the arguments for this operation are:

1. DDIN(1): 13, the operation identification.
   DDIN(2): The wait time in hundredths of a second, or a minus one.

2. DDST: For a KEYBOARD event, this is where the character string is saved.

3. DDEX(1): 0 means no event is available, that is, a time-out has occurred,
   1 means a KEYBOARD event is available,
   2 means a PICK event is available,
   3 means a BUTTON event is available, and
   4 means a STROKE event is available.

For KEYBOARD events only:
   DDEX(2): The number of characters in DDST.

For PICK events only:
   DDEX(2): The identification of the graphic segment.
   DDEX(3): The pick identification of the graphic item.

For BUTTON events only:
   DDEX(2): The button index.

For STROKE events only:
   DDEX(2): The number of points in the stroke.
   DDEX(3): The first X coordinate.
   DDEX(4): The first Y coordinate.
   DDEX(5): The second X coordinate.
   DDEX(6): The second Y coordinate.

... ... ...

If the open function is correct, the device-dependent module will never be asked to perform an impossible action.
SECTION 5.14: SAMPLE A CONTROL UNIT

This operation is requested by subroutine UGECTL. The meaning of the arguments for this operation are:

1. DDIN(1): 14, the operation identification.
   
   For a VALUATOR only:
   
   DDIN(2): 6 a VALUATOR is to be sampled.
   
2. DDST: This parameter is not used.

For LOCATOR data only:

3. DDEX(1): 5 LOCATOR data is available.
   
   DDEX(2): The X coordinate.
   
   DDEX(3): The Y coordinate.

For VALUATOR data only:

4. DDEX(2): The valuator value as an integer with a precision of (32,30).

If the open function is correct, the device-dependent module will never be asked to perform an impossible action.

SECTION 5.15: THREE-DIMENSIONAL VIEWING TRANSFORMATIONS

This operation will only be requested of a device that has identified itself as a three-dimensional graphic device. The operation is requested when a new three-dimensional viewing transformation is available or when the transformation must be read from the graphic device. The meaning of the arguments for this operation are:

1. DDIN(1): 15, the operation identification.
   
   DDIN(2): 1 means PUT, and
   
   2 means GET.

2. DDST: This parameter is not used.

3. DDEX(1): 0 if the operation can be performed, and
   
   1 if the operation cannot be performed.

The PUT operation must indicate that the operation has been successfully performed. The GET operation means that the projection parameters are to be retrieved from the three-dimensional graphic device itself. If this is not possible, the device-dependent code should indicate that the operation was unsuccessful.

For the PUT operation, the data is available in:

- DDAXV (4 words) The view port.
- DDAXO (6 words) The object volume.
- DDAXE (3 words) The eye point.
- DDAXU (3 words) The upward direction.
- DDAXN (1 word) The projection flag.

For the GET operation, the above items must be set by the device-dependent code if a successful operation is being signaled.
SECTION 5.16: DIFFERENCES BETWEEN CURRENT AND EARLIER VERSIONS

In the latter part of 1984 and the beginning of 1985, a three-dimensional capability was added to the Unified Graphics System. At that time, the polygon fill and device-dependent data primitives were also added. This activity required that many changes be made to the system. In particular, the old device-dependent modules had to be modified although most of these changes were trivial unless they involved using one of the new features. This section will enumerate these changes:

1. The initialization of DDAIL and DDADM in the open process is unnecessary if they are being set to unity. Setting them to one, however, will not cause any problems.

2. In the Being-a-Graphic-Segment process, the value of DDEX(2) should now be set. It is vital that this value be set correctly for three-dimensional graphic devices. It is also important that it be set to zero for some other devices.

3. The Graphic-Inquiry process had a number of changes made.
   A. First of all, the pick identification was added near the beginning of DDIN so the other items were pushed back by one slot. This means that the value that used to be in DDIN(6) is now in DDIN(7), etc. Failure to accommodate this change will cause disastrous and unpredictable results.
   B. The color BLACK was added, so the device-dependent code for color devices have to be modified to recognize this. Failure to accommodate this change may cause unpredictable results.
   C. The earlier device-dependent code required that the angle of a line of text match the device capability exactly while the new code relaxes this requirement. For example, on the TEKTRONIX 4010, it formerly was necessary for the angle to be exactly zero before the character generator would be used; now a leeway of plus or minus four degrees is allowed. This is more consistent with the way character size is processed. Failure to make this change will simply mean that the device-dependent code works the way it used to.

4. If the graphic device supports the polygon fill primitive, it should be incorporated into the device-dependent code. However, failure to do this will simply mean that polygon fill will be simulated like it is on a device that does not support polygon fill.

5. Finally, the device-dependent code must be recompiled so that the declaration of the expanded DDA is included. Because the basic control blocks, MCA, DDA, etc. were changed, it is impossible to combine modules from the old version with the new version.
The subroutines compiled in this step should be inserted into the library that is used at LINK-EDIT time.

The programmer must make an early decision as to what graphic devices will be supported because subroutine UGZ002 contains references to all of the device-dependent modules. If a dummy device-dependent module is omitted at this stage, it can easily be added later; the only device-dependent module that is immediately important is UGZZ01. UGZZ01 is the name of a device-dependent module that will be used in some of the following steps.

SECTION 6.2.2: THE OPTIONS SCANNER

The options scanning subroutine is also an Assembler Language subroutine. It is coded in Assembler Language for two reasons. First, its arguments are structures of mixed type and it is difficult to handle such arguments in FORTRAN, and second, the subroutine must execute as fast as possible because it is called a very large number of times in the course of running an application program. Even when UGOPTN runs as fast as possible, much of the time spent within the Unified Graphics System is actually spent within UGOPTN.

The test program is:

TSTOPTNS

The module to be checked out is:

UGOPTN

The subroutine compiled in this step should be inserted into the library that is used at LINK-EDIT time.

SECTION 6.2.3: PROCESS THE ERROR MESSAGES

This step does not actually check out any new Unified Graphics System modules; instead, it generates source statements which will become part of other modules.

The processing program is:

ERRMPROC

The production run is:

ERRMESG1

ERRMPROC reads data cards that describe all of the error messages in the Unified Graphics System. The program then punches this information as FORTRAN declaration and data statements. The punched data is divided into two sections by separator records. The first part should be inserted into UGEMSCBK, and the second part goes into NUCLEUS.
The input data in ERRMESG1 is much easier to read and maintain than the FORTRAN source records are. If error messages must ever be added to the system, the way to do it is to modify the data in ERRMESG1, rerun ERRMPROC to produce the new declaration and data records, and insert these records into the appropriate modules.

Actually, it is seldom necessary to do this when moving to a new computer because the modules on the source computer should have these declaration in them.

SECTION 6.2.4: PROCESS THE CHARACTER SETS

This step, like the last one, generates source statements which will become part of other modules.

The processing program is:
CHARPROC
The production runs are:
CHARSET1, CHARSET2

CHARPROC reads data cards that describe a character set and punches this information as FORTRAN declaration and data statements. The punched data is divided into two sections by separator records. The first part contains the declaration statements and the second part contains the data statements.

The output of CHARSET1 is the marker symbols and should be inserted at the appropriate place in NUCLEUS. The output of CHARSET2 is the basic character set and also goes in NUCLEUS.

There is a problem with running CHARPROC at this stage. CHARPROC includes a section of code that uses the Unified Graphics System subroutines to plot the character set on a non-interactive graphic device. Since these subroutines are not available at this stage, this will not work yet. The easiest way to get around this is to supply dummy entry points to the subroutines that are called. At this stage, the simplest thing to do is prepare a temporary module which contains dummy entry points to: UGINIT, UGPLIN, UGTEXT, UGOPEN, UGCLOS, UGWRIT, UGPICT, UGDSPC

These dummy entry points should simply return when they are called. Later, when an actual graphic device is available, the program may be rerun to get all of the output.

It is important to observe the differences in this step between the VAX computers and the IBM computers. The program CHARPROC is essentially identical on all computers. The input data in CHARSET1 and CHARSET2 are identical except for the translation between EBCDIC and ASCII. However, the output from CHARPROC is quite distinct on the different machines. The reason for this is that the characters are sorted, within CHARPROC, on their primary-secondary character pair so that subroutines UGE001 and UGE002 can do a binary search of the character pair table. The
differences between ASCII and EBCDIC assure that the output is different.

Like the error messages, this step may not be necessary when moving the system to a new computer because the modules already contain these declarations. However, this will only be the case if the source and target computers have the same collating sequence.

SECTION 6.2.5: THE ERROR PROCESSOR

The error processor will be checked out in this step.

The test programs are:
  TSTERRPR, TSTERRMG

The modules to be checked out are:
  UGRERR, UGXERR, NUCLEUS, UGMCA CKB K, UGPOTCBK, UGDDACBK, UGEMSCBK, UGERRCBK

The subroutines UGRERR and UGXERR should be inserted into the library that is used at LINK-EDIT time. The module NUCLEUS is not put into the library but remains as a separate module that is included at LINK-EDIT time.

TSTERRPR checks to see if most of the error processor options work and TSTERRMG prints all of the error messages. The modules UGMCA CKB K, ..., UGERRCBK are all included text modules, most of which must be available so NUCLEUS will compile.

SECTION 6.2.6: THE LINE SCISSORING MODULE

This step requires that the line scissoring module be processed and checked out.

The test program is:
  TSTLSCIS

The modules to be checked out are:
  UGC000, UGC001

Under VM, UGC001 is broken down into the seven modules:
  UGC001, UGC002, UGC003, UGC004, UGC005, UGC006, UGC007

The subroutines compiled in this step should be inserted into the library that is used at LINK-EDIT time.

SECTION 6.2.7: THE LINE STRUCTURE MODULE

This step requires that the line structure module be processed and checked out.
The test program is:
TSTLSTRU
The modules to be checked out are:
UGD000, UGD001
Under VM, UGD001 is broken down into the three modules:
UGD001, UGD002, UGD003

The subroutines compiled in this step should be inserted into the library that is used at LINK-EDIT time.

SECTION 6.2.8: PROCESS THE CHARACTER SETS, PART 2

This is another step in which source statements are generated for other modules.

The processing program is:
CHARPROC
The production runs are:
CHARSET3, CHARSET4

CHARPROC is the same program that was used earlier. The output of CHARSET3 is inserted into SIMPLEX and the output of CHARSET4 goes into DUPLEX.

SECTION 6.2.9: THE CHARACTER GENERATOR MODULE

This step requires that the character generator module be processed and checked out.

The test program is:
TSTCHGEN
The modules to be checked out are:
UGE000, UGE001, SIMPLEX, DUPLEX
Under VM, UGE001 is broken down into the three modules:
UGE001, UGE002, UGE003

The subroutines UGE001 (and UGE002 and UGE003) should be inserted into the library that is used at LINK-EDIT time. The modules SIMPLEX and DUPLEX are not put into the library but remains as separate modules that are included at LINK-EDIT time.

SECTION 6.2.10: THE POLYGON SCISSORING MODULE

This step requires that the polygon scissoring module be processed and checked out.

The test program is:
TSTPSCIS
The modules to be checked out are:
UGF000, UGF001
Under VM, UGF001 is broken down into the four modules:
UGF001, UGF002, UGF003, UGF004

The subroutines compiled in this step should be inserted into the library that is used at LINK-EDIT time.

SECTION 6.2.11: THE THREE-DIMENSIONAL LINE SCISSORING MODULE

This step requires that the three-dimensional line scissoring module be processed and checked out.

The test program is:
TST3DSCS
The modules to be checked out are:
UGG000, UGG001
Under VM, UGG001 is broken down into the five modules:
UGG001, UGG002, UGG003, UGG004, UGG005

The subroutines compiled in this step should be inserted into the library that is used at LINK-EDIT time.

When this step is complete, most of the internal subroutines in the Unified Graphics System are ready. The remaining steps will deal mostly with user called subroutines.

SECTION 6.2.12: GRAPHIC SEGMENT GENERATION MODULES

This step requires that the graphic segment generation modules be processed and checked out. Most of these subroutines contain minor amounts of nonstandard FORTRAN-77 code. The problem is that a combination of fixed point, floating point, and character data must be packed into a graphic segment and this operation cannot be done in standard FORTRAN-77.

The test program is:
TSTSEGGN
The modules to be checked out are:
UGB002, UGINIT, UGMARK, UGLINE, UGPMRK, UGPLIN,
UGTEXT, UGXTXT, UGPFIL, UG3MRK, UG3LIN, UG3PMK,
UG3PLN, UG3TXT, UGDDAT, UGDEFL, UGPOTDCL

The subroutines compiled in this step should be inserted into the library that is used at LINK-EDIT time.

SECTION 6.2.13: GRAPHIC ALGORITHMS MODULES

This step will describe a number of test programs and their associated modules. With the exception of UGCNVF, work on these
subroutines can be deferred until later if desired; UGCNVF, however, is used by some of the device-dependent modules and graphic device test programs.

The test program for the numeric conversion module is:
   TSTAGCNV
The module to be checked out is:
   UGCNVF

The test program for the geometric projection modules is:
   TSTAGPRJ
The modules to be checked out are:
   UGTRAN, UGPROJ
Under VM, UGTRAN is broken down into the three modules:
   UGTRAN, UGTRN1, UGTRN2

The test program for the smooth curve interpolator is:
   TSTAGSCI
The module to be checked out is:
   UGSCIN

The test program for the cross-hatching module is:
   TSTAGXCH
The module to be checked out is:
   UGXCH

The test program for the axis generation modules is:
   TSTAGAXS
The modules to be checked out are:
   UGLNAX, UGLGAX, UGLNDX, UGLGDX

The test program for the concatenating contour plot module is:
   TSTAGCTR
The module to be checked out is:
   UGCNTR
Under VM, UGCNTR is broken down into the five modules:
   UGCNTR, UGCNT1, UGCNT2, UGCNT3, UGCNT4

The test program for the simple contour plot module is:
   TSTAGQCT
The module to be checked out is:
   UGQCTR
Under VM, UGQCTR is broken down into the two modules:
   UGQCTR, UGQCT1

The test program for the mesh surface module is:
   TSTAGMSH
The module to be checked out is:
   UGMESH
Under VM, UGMESH is broken down into the five modules:
   UGMESH, UGME1, UGME2, UGME3, UGME4

The test program for the two-dimensional line-drawn histogram module is:
   TSTAG2DH
The module to be checked out is:
UG2DHG
Under VM, UG2DHG is broken down into the seven modules:
UG2DHG, UG2DH1, UG2DH2, UG2DH3, UG2DH4, UG2DH5, UG2DH6

The test program for the two-dimensional polygon-fill histogram module is:
TSTAG2DP
The module to be checked out is:
UG2DHP
Under VM, UG2DHP is broken down into the two modules:
UG2DHP, UG2DH7

The subroutines compiled in these steps should be inserted into the library that is used at LINK-EDIT time.

SECTION 6.2.14: MISCELLANEOUS CHARACTER CONTROL MODULES

This step requires that some miscellaneous character control modules be processed and checked out. Work on these subroutines can also be deferred until later if desired.

The test program is:
TSTCH2LN
The modules to be checked out are:
UGFONT, UGCTOL

The subroutines compiled in this step should be inserted into the library that is used at LINK-EDIT time.

SECTION 6.2.15: A NON-INTERACTIVE DUMMY-DEVICE

This step includes a dummy-device that is used to check out a large group of subroutines. Subroutine UGWRIT contains some nonstandard FORTRAN-77 code because it must unpack the information in a graphic segment.

The test program is:
TSTPDEV1
The modules to be checked out are:
UGB003, UGB004, UGB005, UGB006, UGB007, UGB008,
UGB009, UGB010, UGB011, UGB012, UGB013, UGB014,
UGB015, UGOPEN, UGCLOS, UGSLCT, UGINFO, UGMCTL,
UGWRIT, UGPICT, UGDSPC, UGWDOW, UGSHLD, UG3WRD,
UG3TRN

The subroutines compiled in this step should be inserted into the library that is used at LINK-EDIT time.
This is a relatively complicated test sequence. In addition to performing an extensive test of the functions of a non-interactive graphic device, the test uses multiple graphic devices. Multiple graphic devices are tested by opening the same dummy-device twice and generating two distinct output files.

SECTION 6.2.16: AN INTERACTIVE DUMMY-DEVICE

This step includes an interactive dummy-device to check out the interactive control modules.

The test program is:

TSTPDEV2

The modules to be checked out are:

UGENAB, UGDSAB, UGEVNT, UGECTL

The subroutines compiled in this step should be inserted into the library that is used at LINK-EDIT time.

With the completion of this step, all of the device-independent modules are complete. The test programs have checked a wide variety of functions; however, additional checking and verification will be required when some device-dependent modules are available.

SECTION 6.2.17: THE ACTUAL DEVICE-DEPENDENT MODULES

The final step in getting the Unified Graphics System to run on another computer is to develop the device-dependent modules. This will normally be the most time consuming and difficult step. Because of their inherent simplicity, the pseudo-devices are good ones to start with although they do not test all possibilities and the ability to see an actual picture is much more encouraging.

The device-dependent modules are not put into the library used at LINK-EDIT time, but remain as separate modules that are included at LINK-EDIT time.

One of the problems with the device-dependent modules is that these modules are necessarily also dependent on the operating system. For example, in the device-dependent modules for the TEKTRONIX 4010 series terminals, there are a few subroutines at the end of the module that communicate with the actual output data set or graphic device. To get the TEKTRONIX 4010 to run on another computer, these subroutines, and only these subroutines, should have to be modified.

There are a number of test programs that can be used to give the device-dependent modules an extensive check. These test programs are:
1. TESTNIDD: This test program will subject any non-interactive display device to an extensive series of tests. The test includes checks of all of the picture description parameters, checks of the many character generation options, and a number of scissoring and shielding tests. The only item not tested at all is device-dependent data.

2. TESTSLDD: This test program will subject any slave-display device to an extensive series of tests. All of the checks of the previous program are performed in this program also. In addition, screen manipulation tests for raster-scan or refresh display may be performed.

3. TESTINDD: This test program will subject any interactive display device to an extensive series of tests. In addition to the tests of the previous programs, this program includes tests for all of the interactive control units.

4. TESTMPCS: This test program is different from the preceding ones. TESTMPCS can run in the non-interactive, slave-display, or interactive mode. The program produces a number of interesting pictures on the graphic device. The interaction is quite primitive, but the pictures are very complicated.

5. TEST3DPC: This test program generates a number of three-dimensional pictures. The pictures should help demonstrate just how useful a three-dimensional device is. The program requires an interactive device.

6. TESTMDHG: This test program is written in PL/1 and is only available on the IBM computer. It can run in the non-interactive, slave-display, or interactive mode. The program produces a number of interesting designs on the graphic device. The interaction is also quite primitive for this test module.

There are many control modules that have been prepared to execute the programs listed above on the various supported graphic devices. Those that invoke TESTNIDD, TESTSLDD, or TESTINDD all have "XX" as their first two letters in their names while those that invoke TESTMPGS all have "XP" as their first two characters and those that invoke TEST3DPC have "X3" as their first two characters.

SECTION 6.3: ADDING A NEW GRAPHIC PRIMITIVE

The Unified Graphics System supports graphic primitives consisting of two-dimensional markers, line segments, and character strings; filled polygons; three-dimensional markers, lines, and character strings; and device-dependent data. The system is designed so that additional primitives can be added. For purposes of illustration, we will describe the work necessary
to add a circle as a graphic primitive. Hardware circle and arc generators are available on some graphic devices so it would be a reasonable addition to the Unified Graphics System.

The steps required to add a new graphic primitive to the system are:

1. The most formidable problem in adding a graphic primitive is to decide exactly what is to happen on a graphic device that cannot produce the primitive within its hardware. For a circle, this is not a particularly difficult decision, but other possible primitives can present some difficult problems. For example, when the polygon fill primitive was added, there was considerable uncertainty about what the default action should be.

2. The format of the data within a graphic segment must be specified. This should follow the pattern set by the existing primitives.

3. A subroutine similar to UGMARK, UGLINE, etc. should be written to pack the circle data into a graphic segment. A test program, TSTSEGGN, is available to test existing graphic primitive generators and this program should be expanded to test the new subroutine. This step may require that new options be added to the POT and will certainly require that new error messages be added to the EMM. If the POT is changed, UDDEFIL will have to be expanded. Changes to the POT or EMM require that all modules that reference them be recompiled. When new error messages are added, the test program TSTERRMG should be expanded so that it generates these new messages.

4. It may be necessary to write a group of subroutines like UGE001 and UGE002 (which operate on character strings) to generate line segment data for the graphic devices that do not have a circle generator. A separate test program, similar to TSTCHGEN, should be written to test this module.

5. UGWRIT must be modified so that it recognizes circle data in the graphic segment and passes it on to the device-dependent module. The Graphic-Inquiry and Graphic-Display functions will have to be expanded to include requests to draw circles.

At this stage, a circle primitive would be available on all graphic devices. The circle would, however, always be drawn by line segments. The following step remedies this:

6. If a graphic device has a hardware circle generator, its device-dependent module should be modified to use the new graphic inquiry and graphic display functions. It is difficult to make this series of modifications and additions in a manner that assures that a user always gets a valid load module. In particular, if a user LINK-EDITs a program while the POT or EMM is being changed, the load module is invalid.
SECTION 7: THE ALGORITHMS USED IN CERTAIN MODULES

A few of the algorithms used in the Unified Graphics System can use some additional explanation. In many cases, the explanations which follow will refer to the actual source code.

SECTION 7.1: THE SCISSORING ALGORITHMS

There are three distinct scissoring algorithms used within the Unified Graphics System. These algorithms are used to:

1. Scissor a two-dimensional line against the current two-dimensional window and shields,
2. Scissor a polygon against the current two-dimensional window, and
3. Scissor a three-dimensional line against a plane (the near scissoring plane) or against a rectangular parallelepiped (the world or object volumes).

These algorithms will all be described in this section.

The two-dimensional line scissoring modules are used to eliminate that part of a line segment extending outside the current window or within the current shields. The basic algorithm is well known and is described on pages 123 and 124 of [New73] and in Section 3.1 of [Rog85]. The algorithm will be redescribed here as it applies to the Unified Graphics System.

Consider the problem of determining the intersection points between a line segment and the boundary of a rectangle. The first thing that is done is to determine a set of four flags for each end point that encodes its relation to the boundary limits. Each flag specifies the relation between the point and a specific boundary line. Each of the four flags is encoded as a single bit: a zero means the point is on the acceptable side of that boundary line and a one means the point is on the outer side of the boundary line. The order of the bits is shown in Figure 7.1.1. The extended sides of the rectangle divide the plane into nine areas. The values of the flags in each of these areas are also shown in Figure 7.1.1. This encoding is performed in subroutine UGC007.

Now suppose these flags have been determined for the end points of a line segment and suppose these flags are in FLG1 and FLG2. The following statements may then be verified:

1. If ((FLG1.EQ.0).AND.(FLG2.EQ.0)), the line segment lies entirely within the rectangle.
2. If ((FLG1.AND.FLG2).NE.0), the line segment lies completely outside the rectangle.
3. In the other cases, (FLG1.OR.FLG2) contains a bit for each boundary line that the line segment crosses.
In the third case, the number of bits turned on may range from one to four. If only one bit is on, we know immediately which boundary line to intersect with the line segment. If more than one bit is on, the line segment can be intersected with any of the indicated boundaries. The computed point replaces the appropriate end point of the line segment and the entire process is repeated with the shortened line segment. In the end, this scheme always computes that part of the line segment within the rectangle.

![Figure 7.1.1: The End Point Flags for Line Scissoring.](image)

Subroutine UGC005 does the actual line scissoring with respect to the window. This subroutine is relatively simple because its output is, at most, single line segment. Subroutine UGC006 does the line scissoring with respect to a single shield. UGC006 is a bit more complicated because its output can consist of zero, one, or two line segments. In the most complicated sequence, a single line segment passed into the line scissoring module can result in five output line segments with six distinct parts of the original line segment being eliminated.

The polygon scissoring algorithm is used to determine which parts of a polygon-fill primitive lie within the current window. Basically, the algorithm works as follows:

1. The given polygon is inserted into a buffer. In the beginning this buffer will only contain a single polygon but as the algorithm progresses, it may contain more than one polygon.

2. One of the four boundary lines on the window is used to scissor off the part of the polygon on the wrong side of the boundary. The result may be an empty buffer, or, one or more polygons in the buffer. For example, Figure 7.1.2 shows a polygon and the low Y boundary of
The problem of scissoring a polygon against a single straight line may be restated as follows:

1. The input is a buffer containing one or more polygons and a horizontal or vertical straight line.
2. The output is zero or more polygons in the buffer such that all of these polygons are on the "keep side" of the straight line and not on the "reject side".

The algorithm works as follows:

1. The "current polygon pointer" is set to the first polygon in the buffer.
2. A pass is made through the current polygon and each vertex is compared to the scissoring line.
   A. If all of the points in the polygon are on the keep side of the scissoring line, the current polygon pointer is incremented to the next polygon in the buffer.
   B. If all of the points in the polygon are on the reject side of the scissoring line, the polygons in the buffer following the current polygon are moved up to overlay the current polygon.
   C. If points occur on both sides of the scissoring line, then the two intersection points with the smallest X or Y values are retained. In Figure 7.1.2, these are the points labeled P and Q. The polygon is then split into two parts along the line from P to Q.
3. Control is then transferred back to step 2.
The algorithm continues until all polygons have been accepted at step 2A, or until no polygons are left in the buffer. It is important to notice that the intermediate steps in this algorithm can require considerably more space to hold all the temporary polygons than is required to hold either the input or the output polygons.

There is not much information available on polygon scissoring. One useful article is [Lia83]. Unfortunately, the algorithm described there does not produce multiple disjoint polygons in a case like that shown in Figure 7.1.2. Instead, the output is always a single polygon with joining paths around the edges of the window. However, an algorithm equivalent to the one used here is described in Section 3.16 of [Rog85].

The three-dimensional line scissoring module works very similarly to the two-dimensional module when a rectangular parallelepiped is involved. The principal difference is that an additional pair of flags, ZHI and ZLO in Figure 7.1.1, must be added. The logical comparisons then work exactly as in the two-dimensional case.

When a three-dimensional line segment must be scissored against an arbitrary plane, the end points of the line are substituted into the equation of the plane. The plane has been set up so that a positive value results if the end point is on the keep side of the plane. If both end points of the line give a positive value, the line is kept; if both end points give a negative value, the line segment is discarded. If the signs are different, the segment is intersected with the plane and one part of the segment is discarded.

SECTION 7.2: THE LINE STRUCTURE ALGORITHM

The line structure generating module, UGD001, is actually a relatively simple piece of code. It is based on a number of flags which are retained and manipulated as line structure is generated along a series of line segments. These flags are:

- **SZDS**: The size of a dash (in centimeters).
- **SZBK**: The size of a blank space (in centimeters).
- **FGAV**: Data available flag (0 means all output has been processed for the input data, and 1 means output is still available).
- **FGLO**: Last complete operation flag (0 means the last thing that happened was that a line end point with a BBIT of 0 was supplied, 1 means a blank segment was processed, 2 means a dash was drawn, and 3 means a point was drawn).
As a line sequence is generated, starting with a blank vector, the line structure generation module calculates accumulated chord length along the line. This chord length is maintained in centimeters. It is then a relatively simple thing to evaluate points on the line segments to get the proper spacing of the dots and dashes. To maintain this information, the following variables are used:

- **PNT1** The first point on the line segment under consideration.
- **TVL1** The accumulated chord length at PNT1.
- **PNT2** The terminal point on the line segment under consideration.
- **TVL2** The accumulated chord length at PNT2.
- **TVAL** The accumulated chord length of the last output point generated.

The equation of the line between PNT1 and PNT2 is maintained in the form:

\[
X = AX \cdot t + BX,
\]
\[
Y = AY \cdot t + BY,
\]

where \( t \) is accumulated chord length in centimeters.

Using the above information, the individual section of the line structure generation module can easily be understood. Suppose for example, a dashed line is being generated and the last complete operation was to draw a blank (FGLO=1). This means that the module must work on generating a dash of length SZDS. When the module is called with a request for data it first compares

\[
(TVAL+SZDS) > TVL2
\]

1. If \((TVAL+SZDS) > TVL2\) then the given segment must be drawn in its entirety. In this case, FGAV is set to zero and PNT2 is returned.

2. If \((TVAL+SZDS) \leq TVL2\) then the dash will terminate within the current line segment. In this case, TVAL is incremented by SZDS, FGLO is set to 2 to indicate that a dash was drawn, and finally, the line segment is evaluated at TVAL and this point is returned.

The other sub-sections of code in this module have a similar interpretation.

**SECTION 7.3: THE CHARACTER GENERATION ALGORITHM**

The character generation module, UGE001, must be supplied with a data structure that defines the character set. This data structure is a self-defining structure containing two-byte integers and character data. It contains the following information:

- **CHRID** A character string of 8 characters containing some identifying information. This item is not actually used.
- **CHRTP** An integer containing the type of the data (1 means
marker symbols. 2 means basic character set, 3 means extended/simplex, and 4 means extended/duplex).

CHRNC  The number of characters in the character set. This value is -1 in the dummy versions of the extended character sets contained in the NUCLEUS.

CHRCS  The normal spacing of the characters in terms of the unit movements given in CHRLT. This value is always 6, except for the extended/duplex character set where it is 24.

CHRCT  The character pair table. Each character in the character set is defined by its primary and secondary characters. This table must be sorted with the entries considered as two-byte integers. The purpose of ordering CHRCT is so that subroutine UGE001 can do a binary search to find a given character pair. The exception to this ordering is that each character set must contain a character with a character pair of "$$" which is forced to sort last. This symbol is used to define the symbol used for invalid character pairs.

CHROT  The character offset table. For each entry in CHRCT, this table contains an entry which gives the index of the start of the data in CHRLT.

CHRLT  The line segment table. The first entry in the table for each character contains the number of line end points in the character (NSEG) and the difference in the spacing between fixed spaced characters and proportionally spaced characters (XDSP) encoded as:

\[ 128 \times \text{NSEG} + (\text{XDSP} + 64) \]

The line end points are encoded as:

\[ 16384 \times \text{BBIT} + 128 \times (\text{DELX} + 64) + (\text{DELY} + 64) \]

where BBIT is the blanking bit and DELX and DELY are the delta motion values.

The majority of the duplex characters were designed by A. V. Hershey and are described in [Her67]. A few of the duplex characters, and all of the other character sets were designed by the author.

SECTION 7.4: THE PROJECTION ALGORITHMS

There are a number of distinct two-dimensions to three-dimensions projection algorithms used in the Unified Graphics System. The algorithms described here are:

1. The point and parallel projections used in subroutine UGTRAN, and
2. The point and parallel projections used for the three-dimensional primitives defined by UG3MRK, UG3LIN, UG3TXT, ext.

The algorithms used in UGTRAN are described first because they are more general; the second algorithms are really special cases of the first.
The basic product of subroutine UGTRAN is a 3x4 matrix which defines a projective transformation from three-dimensions into two-dimensions. The orthogonal transformation is just a special case of a projective transformation where the projection point has been moved to infinity. The 3x4 matrix, M, transforms a point P in three-space with the coordinates (X,Y,Z) into a point Q in two-space with the coordinates (t,u). The coordinate transformation is carried out by:

\[
\begin{bmatrix}
    X \\
    Y \\
    Z
\end{bmatrix} =
\begin{bmatrix}
    m_{11} & m_{12} & m_{13} & m_{14} \\
    m_{21} & m_{22} & m_{23} & m_{24} \\
    m_{31} & m_{32} & m_{33} & m_{34}
\end{bmatrix}
\begin{bmatrix}
    X \\
    Y \\
    Z \\
    1
\end{bmatrix}
\]

where L is a scalar. In subroutine UGPROJ, Equation (1) is expanded into three scalar equations for L*t, L*u, and L. The third value is divided into the first two to obtain t and u.

\[
\begin{bmatrix}
    t \\
    u \\
    L
\end{bmatrix} =
\begin{bmatrix}
    m_{11} & m_{12} & m_{13} & m_{14} \\
    m_{21} & m_{22} & m_{23} & m_{24} \\
    m_{31} & m_{32} & m_{33} & m_{34}
\end{bmatrix}
\begin{bmatrix}
    X \\
    Y \\
    Z \\
    1
\end{bmatrix}
\]

Figure 7.4.1: The Definition of a Projective Transformation.

First, we shall derive the form of the matrix, M, for a projective transformation. Figure 7.4.1 shows the data used in this derivation. In the following, we shall assume that VDIR, HDIR, and UDIR are all unit vectors. These, and all other vectorial quantities, will be represented by column vectors.
From Figure 7.4.1, it is evident that:

\[ E = \text{REFP} + \text{EYED} \times \text{HDIR}, \]
\[ Q = \text{REFP} + \text{SCRD} \times \text{VDIR} + t' \times \text{HDIR} + u' \times \text{UDIR}, \]
where \( t' \) and \( u' \) are related to \( t \) and \( u \) by:

\[ t' = \text{TVR1} \times (t - \text{XBAR}), \]
\[ u' = \text{TVR2} \times (u - \text{YBAR}), \]
and \( \text{TVR1}, \text{TVR2}, \text{XBAR}, \) and \( \text{YBAR} \) are given by:

\[ \text{TVR1} = \frac{\text{SCRZ} \times (\text{XHI} - \text{XLO})}{\text{XHI} - \text{XLO}}, \]
\[ \text{TVR2} = \frac{\text{SCRZ} \times (\text{YHI} - \text{YLO})}{\text{YHI} - \text{YLO}}, \]
\[ \text{XBAR} = \frac{\text{XHI} + \text{XLO}}{2}, \]
\[ \text{YBAR} = \frac{\text{YHI} + \text{YLO}}{2}. \]

Since \( E, P, \) and \( Q \) lie on a straight line, the vectors \( (P-E) \) and \( (Q-E) \) are parallel and, therefore, are proportional. Let the constant of proportionality be \( L \), so that we have:

\[ L \times (Q-E) = (P-E). \]

Substituting Equation (2) into Equation (5) gives:

\[ L \times [t' \times \text{EYED} + u' \times \text{UDIR} + \text{SCRD} \times \text{VDIR}] = (P-E). \]

However, this equation may be written in matrix form as:

\[ L \times \begin{bmatrix} t' \times \text{EYED} \\ L \times \text{TVR1} \times (t - \text{XBAR}) - \text{EYED} \\ \text{SCRD} \end{bmatrix} = (P-E), \]

where the columns of \( M1 \) are \( \text{HDIR}, \text{UDIR}, \) and \( \text{VDIR} \). Now let \( M2 \) be the inverse of \( M1 \). The inverse of \( M1 \) will exist unless \( \text{HDIR}, \text{UDIR}, \) and \( \text{VDIR} \) are coplanar and, in that case, a valid projection does not exist. Multiplying Equation (6) by \( M2 \) and using Equation (3) gives:

\[ \begin{bmatrix} \text{TVR1} \times (t - \text{XBAR}) - \text{EYED} \\ \text{TVR2} \times (u - \text{YBAR}) \\ \text{SCRD} \end{bmatrix} = M2 \times (P-E). \]

The column vector on the left-hand side of Equation (7) may be factored into:

\[ \begin{bmatrix} \text{TVR1} & 0 & -(\text{TVR1} \times \text{XBAR} + \text{EYED}) \\ 0 & \text{TVR2} & -\text{TVR2} \times \text{YBAR} \\ 0 & 0 & \text{SCRD} \end{bmatrix} \times \begin{bmatrix} t \\ u \\ 1 \end{bmatrix}. \]

Next, the inverse of the 3x3 matrix in (8) may be written as:

\[ \begin{bmatrix} 1/\text{TVR1} & 0 & \text{TVR3} \\ 0 & 1/\text{TVR2} & \text{TVR4} \\ 0 & 0 & 1/\text{SCRD} \end{bmatrix} \]

where \( \text{TVR3} \) and \( \text{TVR4} \) are defined by:

\[ \text{TVR3} = (\text{TVR1} \times \text{XBAR} + \text{EYED}) / (\text{TVR1} \times \text{SCRD}), \]
\[ \text{TVR4} = \text{YBAR} / \text{SCRD}. \]

Now Equation (7) may be multiplied through by Matrix (9) to obtain:
\[
L^*|u| = M3^*(P - E) \\
1 \\
L \\
\]

where \( M3 \) is Matrix (9) times \( M2 \). Finally, Equation (10) may be put in the form of Equation (1) by writing:

\[
M = (M3!(-M3^*E))
\]

This last equation states that the first three columns of \( M \) are the same as those of \( M3 \), while the last column of \( M \) is the column vector \(-M3^*E\). An alternate way of writing this last equation is:

\[
\begin{bmatrix}
1 & 0 & 0 & -XE \\
M3^* & 0 & 1 & -YE \\
0 & 0 & 1 & -ZE \\
\end{bmatrix}
\]

where the eye point is \((XE, YE, ZE)\).

The derivation of an orthogonal transformation for \( UGTRAN \) is somewhat similar. In an orthogonal projection, the projection is through \( P \), parallel to \( VDIR \). Thus, the vector \((P-Q)\) is parallel to \( VDIR \) and, therefore, proportional to it. Let the constant of proportionality be \( K \) so that we have:

\[
P-Q = K*VDIR,
\]

and Equations (3) and (4) remain valid. From Equations (11) we get:

\[
t'^*HDIR + u'^*UDIR + SCRD*VDIR = P + K*VDIR - REFP.
\]

This can be put into matrix form as:

\[
\begin{bmatrix}
t' \\
M1^* & u' \\
[SCRD] & L \\
\end{bmatrix} = P + K*VDIR - REFP
\]

where \( M1 \) is the same as before. Equation (13) is similar to Equation (6) except that \( EYED \) is zero. Proceeding as before, Equation (13) can be transformed into the equivalent of Equation (10):

\[
\begin{bmatrix}
t' \\
M3^*(P + K*VDIR - REFP) \\
1 \\
L \\
\end{bmatrix}
\]

where \( M3 \) is the same as before except that \( EYED \) is zero. Equation (14) can be written as three scalar equations. These three equations define \( t \), \( u \), and \( 1 \) as linear functions of \( X \), \( Y \), \( Z \), and \( K \). The third equation can be solved for \( K \) and this value can be replaced in the first two equations. The result is \( t \) and \( u \) defined as linear functions of \( X \), \( Y \), and \( Z \). The coefficients of these equations become the first two rows of \( M \) and the third row is set to \((0, 0, 0, 1)\). The third row forces \( L=1 \) which does not affect the transformation in the first two rows. Actually, in subroutine \( UGPROJ \), the fourth entry in the third row is not unity because it is more convenient to avoid dividing all the other entries in the matrix by a common factor and instead set
that entry to the common divisor.

The array TRANS, which is generated by UGTRAN, contains other information in addition to the matrix M. The complete definition of the content of TRANS is as follows:

- TRANS(1) ... TRANS(12): The transformation matrix stored by rows.
- TRANS(13) ... TRANS(15): The coordinates of REFP.
- TRANS(16) ... TRANS(18): A unit vector in the direction of HDR.
- TRANS(19) ... TRANS(21): A unit vector in the direction of UDR.
- TRANS(22) ... TRANS(24): A unit vector in the direction of VDIR.
- TRANS(25): The value of EYED.
- TRANS(26): The value of SCRD.
- TRANS(27): The value of SCRZ.
- TRANS(28): The value of XLO.
- TRANS(29): The value of XHI.
- TRANS(30): The value of YLO.
- TRANS(31): The value of YHI.

Subroutine UGTRAN only makes use of the transformation matrix, but subroutine UGMESH makes use of REFP and VDIR to decide how the surface is to be generated.

The three-dimensional to two-dimensional projection matrix for use with the three-dimensional primitives defined by UG3MRK, UG3LIN, UG3TXT, etc. will now be discussed. The derivation of the projection matrix is very similar to the earlier derivations; the only difference is that things are now much simpler. Instead of the full derivation, we shall only display the results. To do this, we shall introduce the additional variables:

- V1: A unit vector giving the horizontal direction of the three-dimensional window.
- V2: A unit vector giving the vertical direction of the three-dimensional window.
- V3: A unit vector pointing from E toward the center of the object volume.
- D1: The distance from E to the center of the object volume.
- D2: The half size of the three-dimensional window.

Using these variables, the matrix for the point projection is:

\[
\begin{bmatrix}
[XHI-XLO & 0 & XHI+XLO] \\
0 & YHI-YLO & YHI+YLO \\
0 & 0 & 2
\end{bmatrix}
\begin{bmatrix}
D1 & 0 & 0 \\
0 & D1 & 0 \\
0 & 0 & D2
\end{bmatrix}
\begin{bmatrix}
1 & 0 & 0 & -XE \\
V1 & V2 & V3 & 0 \\
0 & 0 & 1 & -YE \\
0 & 0 & 0 & -ZE
\end{bmatrix}
\]

The matrix for the parallel projection is:
where the first two rows of $M_4$ are given by:

$$
\begin{bmatrix}
T & 1 & 0 & 0 & -X_e \\
V_1 & 0 & 1 & 0 & -Y_e \\
V_2 & 0 & 0 & 1 & -Z_e
\end{bmatrix}
$$

and the third row is $(0, 0, 0, 1)$.

Additional useful information about projective and orthogonal transformations can be found in [New73 and Car78]. A complete derivation in the style displayed here can be found in [Bea91].

SECTION 7.5: THE SMOOTH CURVE INTERPOLATION ALGORITHM

The basic operation performed by the interpolation algorithm used in subroutine UGSCIN is to produce a curve between the middle two of four given points in such a manner that two adjacent curve segments join smoothly. If $K_0$, $K_1$, $K_2$, and $K_3$ are the given points, and they are represented as column vectors, then the form of the interpolation curve is:

$$K(t) = [K_0 \ K_1 \ K_2 \ K_3] * M * [t^3 \ t^2 \ t \ 1]$$

where $M$ is a $4 \times 4$ matrix, and the parameter $t$ runs from zero at $K_1$ to one at $K_2$.

Let $D_1$ be the distance between $K_0$ and $K_1$, $D_2$ be the distance between $K_1$ and $K_2$, and $D_3$ be the distance between $K_2$ and $K_3$. In the uniform case, the values of $D_1$, $D_2$, and $D_3$ are set to unity. Also let $A_1$ be the tension value at $K_1$ and $A_2$ be the tension at $K_2$. This data is shown in Figure 7.5.1 along with the tangent vectors to the curve at $K_1$ and $K_2$.

The matrix $M$ may be derived in the following manner. First, determine a second degree parametric curve through $K_0$, $K_1$, and $K_2$. This curve has the form:

$$L(u) = A_1 * u^2 + B_1 * u + C$$

where $L(0) = K_0$, $L(D_1) = K_1$, and $L(D_1+D_2) = K_2$. The derivative of this curve is evaluated at $u = D_1$ to determine a tangent vector $T_1$ at $K_1$. A similar scheme is used to determine a tangent vector $T_2$ at $K_2$. Finally, a third degree parametric curve is passed through $K_1$ with derivative $A_1 * T_1$ and through $K_2$ with derivative $A_2 * T_2$. When this procedure is carried out, the third degree parametric
curve may be put into the form of Equation (1) with:

\[
\begin{bmatrix}
-D1M2-D1P2 & -2*D1M2+2*D1P2 & D1M2-D1P2 & 0
-D1M2-D2P3+2 & 2*D1M2+D2P3-3 & -D1M2 & 1
-D2M3+D1P2-2 & D2M3-2*D1P2+3 & D1P2 & 0
D2M3+D2P3 & -D2M3-D2P3 & 0 & 0
\end{bmatrix}
\]

where the values of the symbols in Equation (2) are:

\[
D1M2 = A1*(D1-D2)/D1,
D1P2 = A1*D1/(D1+D2),
D2M3 = A2*(D2-D3)/D3,
D2P3 = A2*D3/(D2+D3).
\]

Notice that the method of determining the tangent vector at \(K2\) is independent of whether it is being found for the curve segment between \(K1\) and \(K2\) or the curve segment between \(K2\) and \(K3\). Thus, the two curve segments join smoothly.

Equation (2) determines M if there is a point on either side of \(K1\) and \(K2\). The interpolation scheme allows other conditions at the ends of the curve; the ends of the curve may alternatively be specified by a given tangent vector or a zero second derivative. Now suppose a tangent vector is given instead of \(K0\). In this case, Equation (1) has \(K0\) replaced by \(T1\) and the matrix \(M\) becomes:

\[
\begin{bmatrix}
-1D2P3+2 & D2P3-3 & 0 & 1
-D2M3-2 & D2M3+3 & 0 & 0
[D2M3+D2P3 & -D2M3-D2P3 & 0 & 0
\end{bmatrix}
\]

If a tangent vector is given instead of \(K3\), then \(K3\) is replaced.
by $T_2$ in Equation (1) and $M$ becomes:

$$
M = \begin{bmatrix}
D_1M_2-D_1P_2 & -2*D_1M_2+2*D_1P_2 & D_1M_2-D_1P_2 & 0 \\
D_1M_2+2 & 2*D_1M_2-3 & -D_1M_2 & 1 \\
D_1P_2-2 & -2*D_1P_2+3 & D_1P_2 & 0 \\
A_2*D_2 & -A_2*D_2 & 0 & 0 \\
\end{bmatrix}
$$

If the interpolation curve is to have a zero second derivative at $K_1$, then the matrix becomes:

$$
M = \begin{bmatrix}
0 & 0 & 0 & 0 \\
-D_2P_3/2+1/2 & 0 & D_2P_3/2-3/2 & 1 \\
-D_2M_3/2-3/2 & 0 & D_2M_3/2+3/2 & 0 \\
D_2M_3/2+D_2P_3/2 & 0 & -D_2M_3/2-D_2P_3/2 & 0 \\
\end{bmatrix}
$$

Notice that the value of $K_0$ is immaterial in this case because the first row of $M$ is zero. If the interpolation curve is to have a zero second derivative at $K_2$, then $M$ becomes:

$$
M = \begin{bmatrix}
D_1M_2/2-D_1P_2/2 & -3*D_1M_2/2+3*D_1P_2/2 & D_1M_2-D_1P_2 & 0 \\
-D_1M_2/2+1/2 & 3*D_1M_2/2-3/2 & -D_1M_2 & 1 \\
D_1P_2/2-1/2 & -3*D_1P_2/2+3/2 & D_1P_2 & 0 \\
0 & 0 & 0 & 0 \\
\end{bmatrix}
$$

All of the normal interpolation possibilities are covered by the preceding five values for $M$. However, there is the possibility that someone might ask for a complete interpolation from only two points. In this case, there are two possibilities for each end of the curve so we get four additional matrices. If tangent vectors are given at both ends, then, in Equation (1), $K_0$ is replaced by $T_1$, $K_3$ is replaced by $T_2$, and $M$ becomes:

$$
M = \begin{bmatrix}
A_1*D_2 & -2*A_1*D_2 & A_1*D_2 & 0 \\
2 & -3 & 0 & 1 \\
-2 & 3 & 0 & 0 \\
A_2*D_2 & -A_2*D_2 & 0 & 0 \\
\end{bmatrix}
$$

If $T_1$ is given at $K_1$ and the second derivative is to be zero at $K_2$, then $M$ is:

$$
M = \begin{bmatrix}
A_1*D_2/2 & -3*A_1*D_2/2 & A_1*D_2 & 0 \\
1/2 & -3/2 & 0 & 1 \\
-1/2 & 3/2 & 0 & 0 \\
0 & 0 & 0 & 0 \\
\end{bmatrix}
$$

If the second derivative is to be zero at $K_1$ and $T_2$ is given at $K_2$, then:

$$
M = \begin{bmatrix}
0 & 0 & 0 & 0 \\
1/2 & 0 & -3/2 & 1 \\
-1/2 & 0 & 3/2 & 0 \\
A_2*D_2/2 & -A_2*D_2/2 & 0 & 0 \\
\end{bmatrix}
$$

Finally if the second derivative is to be zero at $K_1$ and $K_2$, then
The fact that this interpolation scheme is independent of its coordinate system is a result of the property that when Equation (1) is expanded into the form:

\[ K(t) = e(t)K_0 + f(t)K_1 + g(t)K_2 + h(t)K_3 \]

it is true that:

\[ e(t) + f(t) + g(t) + h(t) = 1. \]  \( (4) \)

In the cases where one or more of \( K_0 \) or \( K_3 \) is replaced by a tangent vector or eliminated by a zero second derivative, then Equation (4) should only sum the coefficients of the \( K \)'s. Thus, in Equation (2), the columns sum to \((0, 0, 0, 1)\). In Equation (3), the first row must be eliminated and the remaining columns summed to obtain \((0, 0, 0, 1)\).

A complete discussion of this method of interpolation may be found in [Bea91], especially in Chapter 4.

SECTION 7.6: THE CROSS-HATCHING ALGORITHM

The algorithm used in subroutine UGXHCH is basically the following:

1. A family of parallel lines are set up which define the cross-hatching.
2. The lines in this family which actually intersect the region to be cross-hatched are determined.
3. For each of these lines, the following operations are performed:
   A. The line of cross-hatching is intersected with the region boundary to obtain all of the intersection points.
   B. The intersection points are sorted so that they are in order along the line. This order alternates, in direction along the line, in an attempt to minimize movement on pen plotters.
   C. At this point, there will be an even number of points available and they are output by blanking to the first point, drawing to the second point, blanking to the third point, etc.

The data which describes the lines of cross-hatching are:

\[ XCRD \]  The X coordinate of a point through which a line of cross-hatching will pass.
\[ YCRD \]  The Y coordinate of a point through which a line of cross-hatching will pass.
ANGL  The angle at which the lines of cross-hatching will be drawn.

SPAC  The spacing between the lines of cross-hatching.

This data is shown in Figure 7.6.1.

First, the equations of the lines of cross-hatching will be developed. A unit vector along a line of cross-hatching is:

\[ \begin{bmatrix} \cos(\text{ANGL}) \\ \sin(\text{ANGL}) \end{bmatrix} \]

Therefore, a unit vector perpendicular to one of these lines is:

\[ \begin{bmatrix} -\sin(\text{ANGL}) \\ \cos(\text{ANGL}) \end{bmatrix} \]

A reference point on each line of cross-hatching may be obtained by moving along Vector (1) in integral multiples of \( \text{SPAC} \) from the point \((X\text{CRD},Y\text{CRD})\). Therefore, a point on each line of cross-hatching is given by:

\[ [X\text{CRD}-\text{INT1}\times \text{SPAC}\times \sin(\text{ANGL}), Y\text{CRD}+\text{INT1}\times \text{SPAC}\times \cos(\text{ANGL})] \]

Where \( \text{INT1} \) takes on all integral values. Thus, the parametric equations of the family of lines forming the cross-hatching are:

\[
X = \cos(\text{ANGL})t + [X\text{CRD}-\text{INT1}\times \text{SPAC}\times \sin(\text{ANGL})], \\
Y = \sin(\text{ANGL})t + [Y\text{CRD}+\text{INT1}\times \text{SPAC}\times \cos(\text{ANGL})],
\]

where \( \text{INT1}=\ldots,-1,0,1,2,\ldots \). These equations may also be displayed as:

\[
X = AXt + (BX+\text{INT1}\times CX) = AXt + DX, \\
Y = AYt + (BY+\text{INT1}\times CY) = AYt + DY,
\]

where \( \text{INT1}=\ldots,-1,0,1,2,\ldots \) and the A's, B's, and C's are constants and the D's depend only on the line of cross-hatching.

Figure 7.6.1: The Lines of Cross-Hatching.
under consideration.

The next problem is to determine which of the lines in Equations (2) actually intersect the region. To do this, we compute the (non-integral) value of INT1 in Equations (2) which determines a line passing through each point in the definition of the region boundary. By computing the minimum and maximum values of this set of numbers and rounding up or down to an integer, we obtain the values of INT1, which bound the region to be cross-hatched. The computation for INT1 is done by eliminating $t$ in Equations (2), solving for INT1, and simplifying. The result is:

$$INT1 = \frac{AX*(Y-BY)-AY*(X-BX)}{SPAC}.$$

The last problem to be considered is that of computing the intersections of the line of cross-hatching with the region boundary. To do this, consider the function:

$$D(X,Y) = AX*X - AX*Y + (AX*DY-AY*DX).$$

The value of this function is zero if the point $(X,Y)$ is on the line, is positive on one side of the line, and is negative on the other side. Further, $\text{ABS}(D(X,Y))$ is the distance from the point $(X,Y)$ to the line of cross-hatching. By testing each adjacent pair of points on the boundary, in turn, to find a place where the function $D$ gives a change in sign, we find the position of all possible intersections between the region boundary and the line of cross-hatching. Suppose a pair of points are $(X1,Y1)$ and $(X2,Y2)$ and the respective values of $D(X,Y)$ are $D1$ and $D2$. Then the point of intersection is given by:

$$X1 = \frac{\text{ABS}(D1)*X2+\text{ABS}(D2)*X1}{\text{ABS}(D1)+\text{ABS}(D2)},$$
$$Y1 = \frac{\text{ABS}(D1)*Y2+\text{ABS}(D2)*Y1}{\text{ABS}(D1)+\text{ABS}(D2)}.$$

The point determined by these equations is, except for round-off error, on the line of cross-hatching. The next step is to determine the value of $t$ in Equations (2) corresponding to $(X1,Y1)$. The value of $t$ which gives the closest point on the line of cross-hatching to the point $(X1,Y1)$ is given by:

$$t = AX*(X1-DX) + AY*(Y1-DY).$$

After all such $t$'s are computed, they are sorted into ascending or descending order, and the points are recomputed from the value of $t$ using Equations (2), and passed on to the given line end point subroutine.

SECTION 7.7: THE AXIS LABELING ALGORITHMS

The problem addressed by subroutine UGLNDX is to take a given range on an axis and expand that range a minimal amount and divide it into a number of equal length segments, such that each segment begins and ends on a "round number". For the purposes of this subroutine, a "round number" is defined as a number of the form:

$$N*(P*10**K)$$

where $N$, $P$, and $K$ are integers and $P$ is one of the set {1, 2, 5}. 
The values of \( P \) and \( K \) are fixed for all labels, and the factor in parenthesis is the segment length. The actual labels correspond to a series of consecutive values for \( N \).

A simple algorithm for determining the labels according to these constraints when the number of segments is given, is described in [Dix65]. If \( S \) is the number of segments required, then a first approximation for \( K \) is:

\[
K = \text{GINT}[\log_{10}(\frac{\text{HIDATA}-\text{LODATA}}{S})]
\]

where \( \text{GINT}[X] \) is the greatest integer in \( X \). If \( (\text{HIDATA}-\text{LODATA})/10^{2K} \) is greater than the largest of the \( P \)'s, then \( K \) is incremented by 1. Then the smallest \( P \) is found which is greater than or equal to \( (\text{HIDATA}-\text{LODATA})/10^{2K} \). This procedure gives a starting value for \( P \) and \( K \). Now let:

\[
s = P*10^{2K},
\]

\[
\text{LOLAB} = \text{GINT}[\frac{(\text{HIDATA}+\text{LODATA})/2-s*s/2}{s}]*s,
\]

\[
\text{HILAB} = \text{LOLAB}+s*S.
\]

Thus \( \text{LOLAB} \) and \( \text{HILAB} \) are round numbers, the distance between them is divided into \( S \) segments of length \( s \), and \( s \) is also a round number. However, there is no guarantee that \( \text{LOLAB} \) and \( \text{HILAB} \) bracket the data. The final step is, therefore, to check that the data is bracketed. If it is not, two things may be tried: first \( \text{LOLAB} \) and \( \text{HILAB} \) may be incremented by \( s \), or if this does not work, \( P \) and \( K \) must be increased to the next permissible values.

The basic addition to the algorithm described in [Dix65] is that the algorithm is applied to a range of segment counts. The algorithm is applied for all segment counts between \((\text{MINLAB}-1)\) and \((\text{MAXLAB}-1)\). Subroutine UGLNDX then selects the segment count which minimizes the expansion of the range. Other algorithms of this nature have been described in [Gia64 and Lew73].

The algorithm used in UGLGDX is much more primitive than the one in UGLNDX. In the first place, the difference between \( \text{LOLAB} \) and \( \text{HILAB} \) always represents an integral number of full cycles. Also, if the number of full cycles does not lie between \( \text{LOLAB} \) and \( \text{HILAB} \), the labels between the full cycles are not very suitable.

The algorithm works be determining \( \text{DHIX} \) and \( \text{DLOX} \) such that the data is bracketed by \( 10^{\text{DLOX}} \) and \( 10^{\text{DHIX}} \). Then, tentative output values are computed by:

\[
\text{LOLAB} = 10^{\text{DLOX}},
\]

\[
\text{HILAB} = 10^{\text{DHIX}},
\]

\[
\text{NLAB} = \text{INT}(\text{DHIX}-\text{DLOX}+1.0).
\]

If \( \text{NLAB} \) is not acceptable, an attempt is made to increase (decrease) it by dividing (multiplying) the cycle into 2, 5, or 10 parts.
SECTION 7.8: THE CONTOUR PLOT ALGORITHMS

There are two contour plot algorithms in the Unified Graphics System. The first, incorporated into subroutine UGCNTR, produces the contour lines as concatenated line segments. The second algorithm, in subroutine UGQCTR, produces the line segments in a disorganized manner. Subroutine UGQCTR, however, is much simpler and does not need any auxiliary work space. A short description of both of these algorithms will be given.

Within subroutine UGCNTR, there are a few basic variables which must be understood. The first of these are IROW and ICOL. These variables specify the "row" and "column" of the surface patch under consideration. IROW takes on values from 3 to MDIM, and ICOL takes on values from 3 to NDIM. Thus, the indices of the Z values of the corner points of the surface patch are as shown in Figure 7.8.1. This figure also gives the values of ISID, the surface patch side index.

The array WKAREA is used to contain a flag for each surface patch boundary in the surface. These flags can take on the value "processed" or "unprocessed". The details of how these flags are stored will be given later. For now, the reader should keep in mind that the ISID=2 side of the (IROW,ICOL)-th surface patch is
the same line segment as the ISID=0 side of the (IROW,ICOL+1)-th surface patch. Similarly the ISID=3 side of the (IROW,ICOL)-th surface patch is the same as the ISID=1 side of (IROW+1,ICOL)-th patch.

The basic algorithm can then be stated as follows:

1. A contour value is selected and the flags in WKAREA are set to unprocessed.
2. The line segments along the boundary of the whole surface are checked to see if they are marked unprocessed and if the contour line intersects the line segment. If the answer is no, the line segment is marked processed. If the answer is yes, the curve is followed until it terminates at another boundary line segment and all line segments that are encountered are marked processed.
3. The ISID=0 line segment of each of the surface patches is checked to see if it is marked unprocessed and if the contour line intersects the line segment. If the answer is no, the line segment is marked processed. If the answer is yes, the curve is followed until it terminates at the initial line segment and all encountered line segments are marked processed.

Step 2 generates all contour lines which begin and end on the surface boundary while Step 3 generates all closed curves. By checking the boundary first, we can be certain that these curves will be drawn as a single concatenated curve. Notice, in Step 3, that it is not necessary to check all four boundaries of each surface; the ISID=0 side is enough since any closed curve must cross an ISID=0 side of some surface patch.

---

**Figure 7.8.2: Ambiguities with Four Intersections.**

The problem of following a contour line is reasonably straightforward. The only problem comes when the number of sides intersected by the contour line is four. It is easy to see that the number of intersections is always zero, two, or four. The
results for zero or two intersections are obvious, but four intersections present three possibilities, as shown in
Figure 7.8.2. When a surface is defined by a mesh of points, as in this subroutine, there is not enough information available to
decide between these possibilities. The actual solution is almost certainly one of the asymmetric solutions. However, in
this algorithm, we choose the third configuration because it is the most symmetric. This choice is very simply accomplished in
the algorithm; if a surface patch is entered by the ISID-th side, the side opposite this side is checked for the exit side first
before the adjacent sides are checked. The side opposite the ISID-th side is the MOD(ISID+2,4)-th side. The adjacent sides
are the MOD(ISID+1,4)-th and MOD(ISID+3,4)-th. Thus, following
the curve is a simple process.

The only thing still to be discussed is the manner that the "processed" and "unprocessed" flags are stored in WKAREA. The
flags are stored as a single bit, 0 meaning unprocessed and 1 meaning processed, in WKAREA with 30 bits per word. The bits are
packed and unpacked by multiplying and dividing by powers of two. To explain how an individual bit is found, suppose we wish to
find the bit for the (IROW,ICOL,ISID)-th line segment. If ISID is two, we reset ISID to zero and increase ICOL by one. If ISID
is three, we reset ISID to one and increase IROW by one. The bit number, starting with zero, of the bit that must be accessed is
then:

$$NBIT = 2\times[(IROW-3)\times(NDIM-1)+(ICOL-3)] + ISID.$$  \hspace{1cm} (1)

Then let:

$$MWRD = 1 + NBIT/30,$$
$$MBIT = 1 + MOD(NBIT,30).$$

Then the required bit is the MBIT-th bit of WKAREA(MWRD). Notice that Equation (1) shows that the actual number of words required
in WKAREA is:

$$\left[(MDIM-2)\times(NDIM-1)+(NDIM-2)+15\right]/15.$$

The number MDIM*NDIM/15 that is given in the Unified Graphics Algorithms Manual [Bea81b] is only an approximation that is
correct for all cases except when MDIM or NDIM is quite small. By using this method of counting the bits, there are a few bits
that do not actually get used.

Subroutine UGQCTR is much simpler. For each contour value, each rectangular surface patch is examined. For each surface patch,
the four boundaries are examined. If one corner point of a boundary line is above the contour value and the other is below
the contour value, then the intersection of the contour line and the boundary is determined by linear interpolation. As long as
exactly two intersection points per surface patch are found, they are simply joined together. When four intersections are found,
they are joined together in the same manner that UGCNTR uses. The collection of all of these lines will form a contour plot of
the surface. Unfortunately, there are two things wrong with this simple algorithm. First, most graphic devices work more
efficiently when the contour lines are drawn as a sequence of
concatenated line segments instead of drawing them in some arbitrary order (think of all the extra "pen-up", "pen-down", and movement with the pen up on a mechanical plotter). The second problem is associated with the Unified Graphics System itself; line structure cannot successfully be applied to a curve that is not drawn as concatenated line segments. The reason is that line structure must start over after each blank vector, and the dashes (for example) will not be evenly spaced along the entire length of the line.

An algorithm very similar to the one used in subroutine UGCNTR is described in [Cot69]. Additional information about contour plotting will be found in [IBM--; Mor68, and War78]. I was also able to examine a listing of CERN's contour plotting subroutine in GD3 [Mil76], and the idea of searching the boundary first and then the interior came from there.

SECTION 7.9: THE MESH SURFACE ALGORITHM

The algorithm used in subroutine UGMESH is basically very simple. Its implementation, however, does get somewhat complicated. To begin, consider the case where the lines on the surface are to be drawn in only one direction, and assume that this direction is roughly perpendicular to the viewing direction. Also, suppose that a picture of the upper side of the surface is to be developed. The first step is to project the surface line which is closest to the observer into the drawing plane. This line may be drawn in full because there is nothing closer to the observer which could hide it. This line now becomes a "height function", as shown in the picture on the left in Figure 7.9.1. This height function has a value of minus infinity outside the domain of the projected line.

![Height Function](image)

Figure 7.9.1: Development of the Height Function.
The next step in developing the picture is to project the next surface line into the picture plane. Only those parts of the second line that are above the height function are actually drawn. The height function is then modified so that it represents the highest point of either line. In the picture on the right in Figure 7.9.1, the actual height function is shown by the solid lines. Notice that this height function can be a very jagged function. Successive lines in the surface may be drawn in the same manner.

In determining what part of a projected surface line is above the height function, two simplifying assumptions are made:

1. If both end points of one of the short straight line segments are below the height function, then the entire segment is assumed to be below the height function and the line segment is not drawn.
2. If both end points are above the height function, the entire segment is assumed to be visible and it is drawn.

When exactly one end point is above the height function, then the actual intersection with the height function is determined and a line is drawn from the visible point to the point of intersection.

It is a common misconception that a complete picture of a mesh surface is produced by first processing the lines in one direction, and then, independently, processing the lines in the other direction. Figure 7.9.2 demonstrates that this assumption is false.

In view of the above discussion, the complete algorithm may be stated as follows:

1. The mesh surface is examined to determine which edge is closer to the viewer. Suppose these lines are the $V_2=\text{constant}$ lines where $V_2$ is either $X$ or $Y$. The lines in the other direction are the $V_1=\text{constant}$ lines.
2. The height function is initialized to a constant with a value of minus infinity.
3. For each $V_2=\text{constant}$ line, starting at the line closest to the viewer, the following is done:
   A. Each straight line segment on the $V_2=\text{constant}$ line is examined. The part of it that is above the height function is drawn, and the height function is modified to include this new information.
   B. If this is not the last $V_2=\text{constant}$ line, then the segments of the $V_1=\text{constant}$ lines between the current $V_2=\text{constant}$ line and the next $V_2=\text{constant}$ line are examined. The part of these line segments that is above the height function is drawn and the height function is modified.

When the bottom side of the mesh surface is being developed, the height function is initialized to plus infinity, and lines are drawn only when they are below the current height function.
The height function itself is contained in the array WKAREA in the calling sequence to subroutine UGMESH. The function is saved as a list of \((X, Y)\) coordinates of the break points in the height function. As Figure 7.9.1 shows, some of the segments in the height function are vertical. To contain the height function, the array WKAREA is divided up into three word segments. Each segment contains two halfword fixed point values and two floating point values, as follows:

- **HFFP** - A halfword fixed point value which gives the index in WKAREA of the start of the three word segment containing the \((X, Y)\) coordinate of the next point of the height function in a leftward direction.
- **HFBP** - A halfword which gives the index of the start of the three word segment containing the \((X, Y)\) coordinate of the next point in a rightward direction.
- **HFXC** - The \(X\) coordinate of the point on the height function.
- **HFYC** - The \(Y\) coordinate of the point.

The rightmost point of the height function is always in WKAREA(1)…WKAREA(3) and the leftmost point is always in WKAREA(4)…WKAREA(6). The height function is thus maintained as a concatenation of points defining the break points. By maintaining the function as a doubly linked list, the many modifications that must be made to this function can be accomplished in a reasonably fast time.
The algorithm used in subroutine UGMESH is based on the information in [Wri72 and Barl72]. In fact, Figure 7.9.2 is taken directly from [Wri72]. Other algorithms of this nature have been described in [Kub68, Wil72, and Wat74]. Algorithms which have fewer restrictions have also been described, for example [And82], but they usually require more computer time. Additional examples of this type of computer generated picture will be found in [Pru73 and Pru75].

SECTION 7.10: THE TWO-DIMENSIONAL HISTOGRAM ALGORITHMS

There are two algorithms described in this section. The first produces pictures for line-drawn two-dimensional histograms and is incorporated into subroutine UG2DHG. The second produces pictures for graphic devices which have a polygon-fill capability and is incorporated into subroutine UG2DHP.

The algorithm used in subroutine UG2DHG is similar to the one used in UGMESH; a height function is maintained and each new line segment is checked against it and the portion below the height function is eliminated.

However, the subroutines in UGMESH had to be rewritten because certain assumptions made for the mesh surface are not valid for a two-dimensional histogram. For example, if both ends of a line segment are below the height function, then it is not valid to assume that the entire line segment is invisible.

The restriction that the transformation must be a parallel transformation is dictated by the use of a height function. If a projective transformation is used, the top of a column can appear larger than the bottom, and the height function is no longer single valued. One simplifying assumption that is possible, however, is that the height function never splits a line segment into two visible pieces by hiding the middle.

The algorithm used in subroutine UG2DHP is actually quite simple. First of all, only the sides of the columns closest to the eye point need be considered; the back faces can be eliminated immediately. The algorithm then draws the columns in such a manner that the columns farthest from the eye point are drawn first. Thus, if a column partially hides another column, the closer column will be drawn last and drawing it will wipe out the hidden part of the earlier column. No height functions are required, so no large temporary storage arrays are needed. Also, because nothing like a height function is needed, either a point or parallel projection is permitted.
SECTION 8: THE ORGANIZATION OF THE DATA SETS

In addition to the executable modules and their source, the Unified Graphics System also consists of a large number of test programs and support procedures. Most of the test programs have been described in earlier sections. These test programs include information which describe how they are to be executed.

There are a few important support modules that are available on all versions of the Unified Graphics System. These modules are:

1. INVNTRY: This module is used to produce a printed listing of the current state of the Unified Graphics System. On the IBM computer, the Unified Graphics System mini-disk should be the H disk when this EXEC is run.

2. CONTROL: This module is used to compile and/or assemble the modules in the Unified Graphics System. It also performs a number of operations that are necessary in maintaining the data sets. The module itself contains a full description of what it can do. The IBM version of this module includes a scheme that allows modules to be changed while other users have access to the mini-disk. It works by renaming an existing module so that it has a type of JUNK, and then putting the new module on the mini-disk with the original name. Later, all of the modules with a type of JUNK can be erased. On the VAX computer, the Unified Graphics System directory should be the current directory when this command file is executed; on the IBM computer, the Unified Graphics System mini-disk should be the H disk.


SECTION 8.1: THE DATA SETS ON THE VAX COMPUTERS

Of the many VAX computers at SLAC, it is normal for only one of these computers to have a complete and up-to-date version of the Unified Graphics System; other VAX computers will probably have abbreviated versions of the system. The computer with the latest version of the Unified Graphics System is unpredictable and will vary with time.

Many of the device-dependent modules consist of both a FORTRAN and an Assembler Language part. Thus, there is, for example, a module named VEP12FF.FOR and another module named VEP12FF.MAR. The CONTROL support procedure contains a function which does both a FORTRAN compile and a MACRO assembly. The resulting output files are concatenated together. It is vital that the person recompiling these modules know if the module is written in FORTRAN, Assembler Language, or a combination of the two. In one
case, the generic workstation, the device-dependent module consists of a FORTRAN and a C Language part.

A backup/distribution tape for the VAX computers consists of a copy of the Unified Graphics System directory prepared with the DCL statements:

```
$ DEFINE UGSYSTEM <UGS-directory>
$ COPY UGSYSTEM:*./*;* MTAK:
```

A command file named TAPCTRL.COM is available to create and verify a backup/distribution tape. A command file named TAPGSEQ.CMD is also available to write all of the source modules onto an unlabeled blocked sequential ASCII tape. The source modules are written to tape in four files, the *.FOR modules, the *.MAR modules, the *.C modules, and the *.COM modules. Individual modules are separated by IEBUPDTE control cards. IEBUPDTE is an IBM utility program. The files on this tape have a logical record length of 80 and a block size of 4000. A second command file, named TAPVSEQ.CMD is available to verify this tape.

The statements shown above include a DEFINE statement to equate the symbol UGSYSTEM to the identification of the directory containing the Unified Graphics System. It would be most convenient for the majority of users if this symbol were put into the System Logical Name Table, however, this document will not assume that this has been done.

This symbol, UGSYSTEM, is used throughout the source code. All of the INCLUDE statements and all of the command files address the Unified Graphics System directory symbolically by means of this symbol.

Most of the command files begin by invoking another command file named UGSETUP.COM. The purpose of this file is to enable the command files to be run in an interactive or batch mode. If the directory containing the Unified Graphics System is not the default directory for a batch job, then the file UGSETUP.COM should be copied into the default directory. The basic problem is that batch jobs are handled very differently on the different VAX computers at SLAC. In particular, sometimes the LOGIN.COM file is invoked and sometimes it is not. If the LOGIN.COM file is not executed, it is difficult to assure that the UGSYSTEM symbol is properly defined. The use of the UGSETUP.COM file can overcome this difficulty.

There is no need at SLAC to have a complete copy of the Unified Graphics System directory on each machine. If the full source code is not transferred, a considerable amount of disk space will be saved and the transfer will take much less time. To make this easy, a command file named MINUGSX.COM is supplied. To obtain a minimal version of the Unified Graphics System with only the executable modules and a few auxiliary modules, the user should do the following. First, MINUGSX.COM should be brought over DECnet to an empty directory. Second, the command file MINUGSX.COM itself should be invoked. MINUGSX.COM will bring the remaining modules over DECnet. The user will then probably have
to modify the copy of UGSETUP.COM that was brought over.

A similar command file, MINUGSV.COM, is available for moving the system to a VAXSTATION. When this command file is used, only those modules pertinent to the VAXSTATION are transferred.

SECTION 8.2: THE DATA SETS ON THE IBM COMPUTERS

On the IBM computers, the data sets are currently on the 198 mini-disk of the RCB account. Under the RCB account this mini-disk is the "H" disk. The files that must be accessed by a user of this system have a mode number of one or four while the other files have a mode number of zero. In this way a user is not swamped with hundreds of irrelevant names when this mini-disk is obtained.

Most of the device-dependent modules on this machine also consist of a FORTRAN and an Assembler Language part. There is, for example, a module named VEP12FF FORTRAN H0 and another module named VEP12FF ASSEMBLE H0. The CONTROL EXEC program will compile and assemble these two pieces and put the output together to form a module named VEP12FF TEXT H1. It is vital that the person recompiling these modules know if the module is written in FORTRAN, Assembler Language, or a combination of the two.

On the IBM computers, the files that are brought in by an INCLUDE statement are contained in a MACLIB named UGFTNMAC. The contents of this MACLIB are declarations of COMMON blocks. The macros used by the Assembler Language subroutines are contained in a MACLIB named UGASMMAC. These macros consist of entry and exit protocols and their support functions. They also define symbolic names for the registers.

A backup/distribution tape for the IBM computer can consist of a VM format tape produced with the TAPE DUMP command. It is also possible to produce a tape of the source modules in a form that is independent of the VM operating system. This latter tape is usually a labeled tape and contains sequential files with a logical record length of 80 and a block size of 4000. The files on this tape and their DSNAME's are:

4. UGFTNSRC: The FORTRAN source modules. Individual members are separated by IEBUPDTE control cards.
5. UGASMSRC: The Assembler Language source modules. Individual members are separated by IEBUPDTE control cards.
6. UGFTNMAC: The FORTRAN macros for INCLUDE statements. Individual members are separated by IEBUPDTE control cards.

7. UGASMMAC: The Assembler Language macros. Individual members are separated by IEBUPDTE control cards.

8. UGPLIDCL: The PL/1 declarations for the FORTRAN subroutines. Individual members are separated by IEBUPDTE control cards.

9. UGTSTEXE: Support and test EXEC files. Individual members are separated by IEBUPDTE control cards.

10. UGTSTFTN: Support and test FORTRAN programs. Individual members are separated by IEBUPDTE control cards.

11. UGTSTPLI: Support and test PL/1 programs. Individual members are separated by IEBUPDTE control cards.

12. UGTSTDATA: Support and test data files. Individual members are separated by IEBUPDTE control cards.

This tape is prepared and verified using the programs described in [Bea82].

The documentation for the Unified Graphics System is only available on the IBM computers. The source for the documents has been prepared as input to the FORMAT Text Processing Program. FORMAT is coded in FORTRAN-66 and is available from:

SHARE Program Library Agency
Triangle Universities Computation Center
Post Office Box 12076
Research Triangle Park, North Carolina 27709
Telephone: (919) 549-0671 (Ext. 283)

as Distribution No. 360D-06.0.007. A short description of FORMAT is available in [Ber69] while a complete description is given in [Ehr71].

The programs that generate the figures for the Programming Manual, Algorithms Manual, and Internal Operation and Maintenance Manual, respectively, are contained in modules named UGPGMDOC, UGALGDOC, and UGINTDOC with a type of FORTRAN and EXEC. A common constituent of these programs is the parameter named FACT. When FACT is given a value of 1.0, the pictures are produced at exactly the size needed in the documents. A value greater than 1.0 produces the pictures at a larger than needed size so that they can be photographically reduced if necessary.

The VAX version of the Unified Graphics System has proven to be quite portable to other installations; the IBM version has not. The basic problem seems to be that VM is the most abysmally deficient operating system imaginable. Every installation is forced to create a vast plethora of MODULE's and EXEC files to try to overcome this problem. The Unified Graphics System, in most instances, tries to avoid these items. It avoids these items to make the Unified Graphics System more portable and also because they seem to be even less stable than the basic VM. However, when the Unified Graphics System is moved to another installation, it appears that the extensions and modifications at that installation can interfere with things that the Unified
Graphics System is trying to do. There is some specific information on the system-dependencies in the later sections on the supported graphic devices.

SECTION 8.3: NONSTANDARD FORTRAN-77 CONSTRUCTIONS

Although a strong effort was made to use only standard FORTRAN-77 [ANS78], it was necessary to use a few nonstandard constructions. This section will try to describe these constructions and give the reasons for using them.

The INCLUDE statement is extensively used to insert COMMON block declarations into the subroutines. Something of this nature was absolutely necessary to assure that the declarations were identical in all subroutines. If this facility is not available on a computer where the Unified Graphics System is needed, a simple pre-processor could be written to overcome this problem.

COMMON blocks contain both numeric and character string variables. There really seems to be little reason for this restriction in the FORTRAN-77 standard. The rationalization seems to be that this will prevent a user from declaring a COMMON block differently in two subroutines to obtain the effect of equivalencing of numeric and character data. In the first place, the Unified Graphics System does not try to use COMMON blocks in this questionable manner. In the second place, the logical extension of this argument means that INTEGER and REAL variables should not be allowed in the same COMMON block because equivalencing these can cause strange results on a computer where these variables are not of the same length.

Numeric and character string variables are used together in an EQUIVALENCE statement. This is of course very machine dependent, but it is also absolutely necessary in a few cases. One case is in the generation and use of the graphic segments. FORTRAN-77 provides very little support for collecting numeric and character data together into a single unit. The Unified Graphics System uses this language extension to perform that function. The second place where this equivalencing is done is within the device-dependent code. After an X and Y coordinate has been developed, it is often necessary to pack them into a character string. One of the most efficient way to do this is to use equivalenced variables.

The nonstandard built-in functions IAND, IOR, and NOT are used to perform logical arithmetic on integer variables. This facility is used in constructing the graphic segments and in the device-dependent code. It is needed to pack the blanking bit into a graphic segment without reserving another full word, and is absolutely necessary to prepare the output for some graphic devices. The only subroutines affected in the device-independent
part of the system are:

UGINIT, UGLINE, UGPLIN, UG3LIN, UG3PLN, UGWRIT, UGCTOL, UGC005, UGC006

The manipulation of COMMON blocks to obtain the active graphic device effect is totally outside the normal facilities of FORTRAN-77. However, this was accomplished with a very small number of computer dependent subroutines, and the function of these subroutines should be transportable to most other computers. Their net effect was a substantial simplification of the device-dependent modules and a fairly efficient way to transfer to the device-dependent code.

SECTION 8.4: SOURCE CODE DIFFERENCES

This section describes the differences between the source code for the VAX version and the IBM version of the Unified Graphics System.

In the device-independent part of the Unified Graphics System, the biggest difference is in the operating system dependent subroutines:
UGZ001, UGZ002, UGZ003, UGZ004, UGZ005, UGZ006, UGZ007

which are written in Assembler Language or very system-dependent FORTRAN. There is also the options scanning subroutine:
UGOPTN

which is written in Assembler Language for efficiency reasons. The syntax of the INCLUDE statement is different on the VAX and IBM computers also. This affects almost all subroutines in a minor way. Finally, the masks for the low order bit in a floating point word are different. These masks occur in:
UGINIT, UGLINE, UGPLIN, UG3LIN, UG3PLN, UGWRIT

Other than these few differences, the device-independent code is identical on both machines.

The device-dependent modules contain more differences than the device-independent modules. Many device-dependent modules involve some Assembler Language or very system-dependent FORTRAN subroutines. In general, the system-dependent subroutines are the Assembler Language modules and the subroutines at the end of the FORTRAN device-dependent module. If a graphic device is available on both computers, the subroutines at the beginning of the FORTRAN device-dependent module are often very similar or identical on both machines.

The modules like:
DPIC4010
are completely different. On the VAX computers, these are written in very system-dependent FORTRAN. On the IBM computers, they are written in Assembler Language. A FORTRAN version on the
IBM computers would consist of nothing but calls to Assembler Language subroutines so it is easier to put the whole thing in Assembler Language. The module:

PDEVUGSI
has a few differences also. First, the interaction for a slave-display graphic device is different. Second, the reading of a file whose name is entered at the keyboard is a big problem on the IBM computers. This required an Assembler Language subroutine to issue FILEDEF commands.

There are also a few differences in the test and support programs. The open statements for unit 7 is different in:

ERRMPROC, CHARPROC
and the call to UGOPEN is different in:

CHARPROC
The interaction for slave-display devices is different in:

TESTSLDD, TESTMPCS

In summary, except for the Assembler Language modules and part of the device-dependent code, the differences between the VAX and IBM versions are very minor.
SECTION 9: THE SUPPORTED GRAPHIC DEVICES

This section gives additional information on how the individual graphic devices were interfaced with the Unified Graphics System. To understand some of the more caustic comments given below, it is important to remember that graphics at SLAC has always been an extremely low priority item. Also, with a few exceptions, graphic devices at SLAC are selected by members of private empires who have little knowledge about graphics and no idea of what they want to use the devices for.

SECTION 9.1: THE CONSOLE ON WHEELS (COW) OF THE SLC PROJECT

The device-dependent module for the VAX computers is simple and straightforward. It is the code in the INTEL 8086 that does all of the work of maintaining the display file and effectively upgrades the device from a raster-scan unit to a refresh device. Of all of the things that microprocessors are used for in graphic devices, its use in this device seems the most rational to me. Unfortunately, a number of extensions which limit portability have been added to this device-dependent code.

The device-dependent code for the VAX computers and the code in the INTEL 8086 were written by Eric Linstadt of the Instrumentation and Control Group at SLAC.

SECTION 9.2: THE DEC GIGI COLOR GRAPHICS TERMINAL

This unit is a relatively straight-forward device. It is fully described in [DEC81a and DEC81b]. The only problem is that the user must assure that the many options on the terminal must be in the state that the Unified Graphics System expects. In particular, the Unified Graphics System does not use the graphic prefix character.

SECTION 9.3: THE DEC VAXSTATIONS

The programming interface for these devices is described in [DEC86]. The description, in particular the part used by the Unified Graphics System, is reasonably straightforward. This device is different from most of the other devices in that the device-dependent code does not communicate with the hardware directly, but instead calls the UIS subroutines described in
Since these subroutines already isolate the user from the vagaries of the actual hardware, the device-dependent code is relatively short and simple.

The one thing in the device-dependent code that is not particularly obvious, is the handling of the INCLUDE/OMIT functions. When the Unified Graphics System allocates the virtual-display, it makes it much larger than necessary. Only the part of the virtual-display between zero and one in X and Y is displayed on the screen. When a graphic segment is put into the OMIT state, it is translated to an off-screen position. When it is put into the INCLUDE state, it is simply translated back to its original position.

The choices that were made for the BUTTON keys is not very satisfying at all. The most natural choice for the keys would have been the F1 through F20 keys. Unfortunately, DEC has preempted the use of a number of them and there is no way for the Unified Graphics System to override DEC's use of them. By using the scheme adopted by the ANSI terminal emulator described in [IBM84d], this device-dependent code is at least consistent with something else.

SECTION 9.4: THE GRINNELL GMR-27 DISPLAY SYSTEM

If an installation had only one GRINNELL Display System things would be quite simple; there is nothing very complicated in driving this device. However, two things have made things very confusing at SLAC.

1. The first group of units were ordered with no common features. They differed in the number of pixels in the X and Y directions, the number of memory planes, the character generator, and many other ways. This greatly complicated the device-dependent code and also severely limited the extent to which one unit could backup another. Many of the purchased options on this group of units are useless. For example, some monochrome units have a large number of memory planes and other units had 1024 by 1024 resolution at a time when CRT's using this resolution were very expensive and no better visually than a 512 by 512 resolution unit.

2. When SLAC finally woke up, it was found that it is almost impossible to get two identical units from GRINNELL. GRINNELL has no internal standards on the significance of "channels", "sub-channels", or the output bits from the look-up table. Indeed, it sometimes seems as if GRINNELL was trying to see how many units they could produce without ever making two exactly alike. The only way to determine the meaning of the channel/sub-channel data is to study the documentation [GR177] and circuit diagrams that came
with the individual unit. These differences are the reason for the many BWTYPE and COLTYPE options in UGOPEN.

A "device-driver" for the GRINNELL on the VAX operating system had to be written at SLAC. This work was done by Arthur J. Leino of the Data Analysis Group.

SECTION 9.5: THE IBM 3179 G COLOR GRAPHICS DISPLAY STATION

The only information that has been found about these devices is contained in [IBM85b] and that document assumes most of the information in [IBM85a]. Unfortunately, both of these documents together are totally inadequate and IBM considers any additional information to be "proprietary". In addition, the device was apparently designed to be as devious as possible, probably so other manufacturers could not produce copies of it and undercut IBM's inflated price. As a result, the device-dependent code for this device will probably be very unreliable.

An example of the obvious omissions of information in the original version of the manual is the fact that the manuals did not specify the coordinates of the lower left pixel (it is not 0,0). And the manuals never alluded in any way to the fact that the locator device works on a different coordinate system than the graphic primitives. Both of these omissions have been partially corrected in later revisions to the manual, but there is still much missing information. For example, there is no adequate explanation of the Assembler Language macros that should be used to control this device.

An example of the unnecessary complexity of the device is the data that must be sent to write graphic primitives to the screen. Suppose, for example, that a series of concatenated line segments are to be drawn. Basically, as each end point becomes available, it is added to the end of a buffer containing the previous end points. Then, three distinct counters in the previous part of the record must be incremented; two of these counters are two-byte integers and one of them is a one-byte integer. In addition, the two two-byte counters must necessarily have an odd number of bytes between them so one of them is guaranteed to not lie on an even byte boundary. As a result, the code that generates the device-dependent orders is extremely inefficient when compared to other devices. The device-dependent data is also much longer than that required for most other devices. For example, it takes a two-byte record to put the cross-hairs on the screen of a TEKTRONIX 4010 but 128 bytes to put the graphic cursor on the screen of this device.

The Unified Graphics System only supports this device as an interactive terminal. A cursory examination of the device might
indicate that it could also be used in the non-interactive mode or the slave-display mode. This is, however, not the case. It cannot be used in the slave-display mode because any user attempt to put a prompt on the screen and read the input will cause the screen to clear, thus wiping out any previously drawn picture. It would be difficult to use in a non-interactive mode also because of the very long output records that are required. For consistency with devices like the TEKTRONIX 4010, the output records should be 80 characters long, but some of the record headers are nearly that long. Therefore, a fairly complicated program would be required to concatenate the 80 character records together into longer records and pass them on to the device.

The device-dependent code for this device is not very portable to other VM installations. The reasons for this are not clear.

SECTION 9.6: THE IBM 5080 GRAPHICS SYSTEM

The device-dependent code for this device was written by Lennox E. Sweeney of the SLAC Computing Services. The graphic device itself, and the system support for it are described in [IBM84a, IBM84b, and IBM84c].

There are a large number of device-dependent options available on this device. These items seriously hinder the transportability of programs from this device to another device and are not described in [Bea81a].

The device-dependent code for this device is definitely not portable to other VM installations. It contains many SLAC dependencies that usually have to be removed before use can be made of it elsewhere.

SECTION 9.7: THE IMAGEN LASER PLOTTERS

The Model 8/300 Laser Printer/Plotter is a relatively simple and straightforward device as far as the Unified Graphics System is concerned. These units are fully described in [IMA83]. On the VAX, the support software produced by Kellerman & Smith is usually used to send a graphics file prepared by the Unified Graphics System to the IMAGEN. That software is described in [K&S88].

The Unified Graphics System uses the IMPRESS language to draw the pictures. However, the Unified Graphics System only uses a very small subset of the functions that the laser plotters are capable of performing. The only commands within the IMPRESS language
that are used are "SET-PEN", "SET-TEXTURE", "SET-PATH", "DRAW-
PATH", "FILL-PATH", "END-PAGE", and "END-OF-FILE". A limited
number of document header items are also added to the beginning
of a file by the Unified Graphics System.

On the IBM computer, each record produced by the Unified Graphics
System has a blank character as its first character and an 'X' as
its last character. Both of these characters are stripped off by
various levels of the support programs before being transmitted
to the laser plotters. The presence of these characters means
that the device-dependent code is probably not usable at other
IBM installations; it is also the reason the VAX and IBM files
are different.

The Unified Graphics System is sometimes criticized for not using
the built-in fonts of the IMAGEN. There are a number of reasons
for not doing this. First, the available documentation does not
specify what fonts are available and does not give any
information about them. The Unified Graphics System would at
least have to know how big they are. Second, the built-in fonts
seem to be different from one unit to another. That means that
any size determined from one unit would not necessarily apply to
another unit. By ignoring the built-in fonts, the Unified
Graphics System avoids a large number of device-dependencies.

The device-dependent code for IMGNIBM on the VAX was added
against my wishes. From my point of view, all it does is
propagate some of the absurdities from the IBM computers over to
the relatively clean VAX. For some unknown reason, I have had
considerable trouble persuading most people that the use of the
programs PDEVUGSE and PDEVUGSD is better. At least the addition
of IMGNIBM may keep some people from doing even more stupid and
wasteful things. If I had not acted, SLAC would have had
multiple, incompatible versions of the Unified Graphics System on
the VAX computers.

IMAGEN produces a large number of different models. Some of
these will work correctly with the device-dependent code in the
Unified Graphics System and some will not. It is very difficult
to tell which ones will work correctly from the documentation.

Differences include the width of lines and the number of pixels
in the X and Y directions.

SECTION 9.8: THE MEHTHEUS OMEGA 300 DISPLAY CONTROLLER

This is basically a very straightforward device. The orders that
can be sent to the device are described in [MET85]. The orders
are sent to the device with almost-normal SYS$QIO's. The exact
form of these statements was obtained from the module named
XA.FOR in some demonstration programs that were supplied with the
device.
The device itself is controlled through a standard device-driver supplied by DEC so its installation was unusually smooth and trouble free.

SECTION 9.9: THE POSTSCRIPT LANGUAGE

The PostScript Language is described in [AD085a and AD085b]. The Unified Graphics System, in fact, only uses an extremely small part of the PostScript Language. The PostScript statements produced by the Unified Graphics System are all very straightforward.

The file produced by this device-dependent code defines a number of PostScript macros. The purpose of these macros is to try to reduce the size of the files produced. The macros do accomplish this but they have the bad effect of reducing the legibility of the files.

There is one anomaly in the device-dependent code. The size of the characters, as defined by the manuals, seems to be incorrect. An extra factor of 5/3 had to be introduced to get characters of the proper size. The problem now is that this factor may keep the device-dependent code from working correctly on some PostScript devices.

The monochrome data was tested on a LaserWriter and the color data was tested on a QMS ColorScript 100, Model 10p. The normal picture boundaries of one-half inch were designed for the LaserWriter. The QMS printer cannot plot that close to the end of the paper. When using that device, it is necessary to supply the UGOPEN options items of XMIN=392 and XMAX=3088; if you do not supply these items, you will lose part of the picture and no error indications will be given. These limitations are described in the QMS documentation [QMS--] in the section on Page Types in Chapter 5.

When a file with color orders is sent to the LaserWriter, problems appear. The main problem is that the LaserWriter interprets the color orders as orders to draw lines and other items with various halftone-like patterns. When a thin line is drawn this way, it can miss the halftone dots and disappear completely.

SECTION 9.10: THE PRINTRONIX (MODEL MVP) PLOTTER

The PRINTRONIX (Model MVP) Printer/Plotter is a relatively simple and straightforward device as far as the Unified Graphics System
is concerned. The unit is fully described in [PRI82].

The Unified Graphics System produces pictures for the Medium Resolution Mode (Mode 002) with 14 by 11 inch paper. The High Resolution Mode is not used because it slows printing and plotting down by a factor of two and gives very asymmetric plotting densities (120 dots per inch horizontally by 72 dots per inch vertically). The High Speed Plot Mode is not used because its resolution is inadequate.

The standard DEC line printer driver may be used when using this device as a printer or as a plotter, however, it must have its characteristics set with a command similar to:

```
$ SET PRINTER/WIDTH=133/PRINTALL/LOWERCASE <printer-name>
```

The WIDTH item is necessary because a scan line consists of 132 bytes of plot data preceded by a control byte to get the unit into full-dot plot mode. PRINTALL allows the control byte to be transmitted to the device; if PRINTALL is not given, the PRINTRONIX will not get into plot mode and the plot data will appear as alphabetic garbage. LOWERCASE prevents lower case alphabetic characters from being translated into upper case; if this is not given, the result is that every sixth dot in a scan line is not plotted.

SECTION 9.11: THE SEIKO GR-1105 COLOR GRAPHICS TERMINAL

This device is a mixture of the good and the bad. On the one hand, the resolution and the quality of the display is very good. On the other hand, it tries to be all things to all people, and the result is a very complicated and ill-defined device.

The documentation [SEI85] was clearly written by a technical writer and not by someone really knowledgeable about the device. The writer understood the simple aspects of the device and provided hundreds of trivial examples and pages of descriptive material. When things get more complicated, and one needs to know how the various commands interact with each other, no guidance is provided. For example, the device is allegedly able to do pan and zoom in the refresh mode. The document is totally inadequate in the area. Another area that is even more poorly described is the possible implementation of the PICK control unit with their "HIT TEST" commands.

In the raster-scan mode, the graphic segments are written directly to the graphic memory planes, and the segment memory is not used. In the refresh mode, all segments are stored in segment memory. The oldest segment will always have a name of "S001". The next oldest will have a name of "S002", etc. If a group of segments are in memory, and an early one is deleted, then the later segments are all renamed.
Few problems were encountered getting this device to work in the slave-display modes. The interactive modes, however, resulted in some surprises. The principal problem is the way that the VAX operating system responds to escape sequences sent to it by the terminal. For some reason, the escape character itself, and the next character, which is always an "r" in this case, disappear. The reasons for this are unclear and little time was spent trying to trace the problem down.

The device-dependent code was prepared so that it would support either the basic unit or the VT100 emulator. The two units are very similar, but certain orders are different for the two versions. Since none of the basic units are now available at SLAC, this part of the code has never been tested.

SECTION 9.12: THE SLAC EXPERIMENTAL SLAVE SCOPE

This graphic device was designed and constructed at SLAC. The documentation is a little sparse, but the orders for the device are described on SLAC blueprint Number GP 445-275-00 RO. This is basically a simple and straightforward device and the device-dependent code reflects this. The length of the device-dependent code is entirely due to the fact that the device is a refresh display device. The graphic device is connected to the VAX computers through a CAMAC crate controller. All input/output operations performed by the Unified Graphics System for this device are done by calling subroutines CAMDIO and CAMIOP.

The CAMAC "device-driver" and the CAMDIO and CAMIOP subroutines were written at SLAC.

SECTION 9.13: THE TALARIS LASER PLOTTERS

The Talaris printer/plotters are relatively simple and straightforward devices as far as the Unified Graphics System is concerned. The EXCL language that the Unified Graphics System uses to drive it is described in [TAL87]. According to the manual, the EXCL language is very similar to that needed by DEC's LN03 Plus printer/plotter. The manual also suggests a certain structure for an output file and the Unified Graphics System follows those suggestions very closely. The only real problem with the device is that the output files tend to be quite large; they seem to be about 4 or 5 times larger than an equivalent TEKTRONIX 4010 file. It would be possible for the Unified Graphics System to generate slightly more compact files, but only at the expense of spending more time to do it.
One of the reasons the Unified Graphics System does not use the built-in fonts of the TALARIS is that the documentation does not give much information about them. The Unified Graphics System at least needs to know how big they are and what the offsets from the center are. This information could probably be obtained by running a series of tests but this was not done. This way, the output will also be more consistent with the earlier VERSATEC and IMAGEN devices.

The device-dependent code was checked out on a TALARIS 1590 Printstation using only letter size paper.

SECTION 9.14: THE TEKTRONIX 4010 SERIES TERMINALS

The TEKTRONIX 4010 series terminals are described in [TEK73]. The device-dependent code on the VAX computers is extremely straightforward; the VAX operating system in use at SLAC (VMS) fully supports ASCII terminals.

Things are more complicated on the IBM computers. The device-dependent code uses the RDTERM and WRTERM macros to read from and write to the terminal. Because the device-dependent code uses these macros, it is limited to line-mode use. Since the IBM computers operate in EBCDIC and the TEKTRONIX operates in ASCII, it is necessary that the characters be translated when they are sent between these devices. The device-dependent code assumes that the IBM computers will translate between EBCDIC and ASCII on input and output according to the following table:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7F</td>
<td></td>
<td></td>
<td>0C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>18</td>
<td></td>
<td></td>
<td></td>
<td>1D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1F</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>07</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1A</td>
</tr>
<tr>
<td>4</td>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5E</td>
<td>2E</td>
<td>3C</td>
<td>2B</td>
<td>28</td>
<td>27</td>
</tr>
<tr>
<td>5</td>
<td>26</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>21</td>
<td>24</td>
<td>2A</td>
<td>29</td>
<td>3B</td>
<td>7C</td>
</tr>
<tr>
<td>6</td>
<td>2D</td>
<td>2F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2C</td>
<td>25</td>
<td>5F</td>
<td>3E</td>
<td>3F</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>60</td>
<td>3A</td>
<td>23</td>
<td>40</td>
<td>27</td>
<td>3D</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>61</td>
<td>62</td>
<td>63</td>
<td>64</td>
<td>65</td>
<td>66</td>
<td>67</td>
<td>68</td>
<td>69</td>
<td></td>
<td>7B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>6A</td>
<td>6B</td>
<td>6C</td>
<td>6D</td>
<td>6E</td>
<td>6F</td>
<td>70</td>
<td>71</td>
<td>72</td>
<td></td>
<td></td>
<td>7D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td>73</td>
<td>74</td>
<td>75</td>
<td>76</td>
<td>77</td>
<td>78</td>
<td>79</td>
<td>7A</td>
<td></td>
<td>5B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

It is not critical how the entries identified with "---" are translated; they will never actually occur. In the noninteractive and slave-display modes, some of the characters in
the table will not be used. This translate table is different from those at most other IBM installations and that can cause trouble when people try to use the Unified Graphics System at other installations. The use of the RDTERM and WRTERM macros may also cause difficulty at other installations.

There is a set of device-dependent code for the TEKTRONIX 4010 series devices that is not documented in [Bea81a]. It is obtained by using TEK401Z instead of TEK4010 in the link-edit step and TESTDEV instead of TEK4010 in UGOPEN. This device-dependent code simulates a refresh display device with a full set of interactive controls. The pick, button, stroke, locator, and valuator control units are all simulated with the keyboard. The simulation is not very good; the primary reason for preparing this code was to test certain modules including the test program TESTINDD.

SECTION 9.15: TEKTRONIX 4010/4014 EMULATORS

A real TEKTRONIX 4010 is described in [TEK73] and the TEKTRONIX 4014 is described in [TEK74]. To utilize this device-dependent code, the user will also have to be very familiar with the target device. For example, the information necessary to produce the UGOPEN options items examples for the Programming Manual were obtained from [MOD88] for the MODGRAPH GX-2000 and from [IMA83] for the IMAGEN.

Even though TEKTRONIX 4010/4014 emulators have been around for a long time, this device-dependent code is a late addition to the Unified Graphics System. The code on the IBM computers has all of the problems that are described in the later sections on the TEKTRONIX 4105 and TEKTRONIX 4207.

SECTION 9.16: THE TEKTRONIX 4027 COLOR GRAPHICS TERMINAL

This unit is described in [TEK78a and TEK78b]. In contrast to the TEKTRONIX 4010, this device is not a straightforward device and the documentation is of little help with all of the problems and pitfalls that await the user.

While this device can be made to work on the VAX computers, it would be extremely difficult to use it from the IBM computers. One of the problems is that the terminal accepts data at a much faster rate than it can process that data. If this goes unchecked, buffer overflow occurs and no indication, other than a scrambled picture, is given. On the VAX computers, each record is terminated with a report command (!REP 00) and the next record
is not sent until the answer is received. Unfortunately, the report must be read with no echo, and that does not seem to be possible on the IBM computers at SLAC. Using the locator control unit also requires that echo be suppressed. Limited use probably could be made of this device on the IBM computers but only by running at a very low BAUD rate.

SECTION 9.17: THE TEKTRONIX 4105 COMPUTER DISPLAY TERMINAL

This device is described in [TEK87a]. The description is reasonably good and the device-dependent code on the VAX is fairly straightforward.

The situation on the IBM computer is not straightforward at all. When the code for this device was generated, it was wanted for use in the full-screen mode. At SLAC, ASCII terminals are supported in the full-screen mode through the use of the Yale Emulator described in [IBM84d]. This means that the device-dependent code does not manage a real TEKTRONIX 4105 but instead tries to work with a simulated IBM 3270. The Yale emulator has a "transparent mode" that allows ASCII data to be sent to the simulated terminal. Unfortunately, the documentation is grossly inadequate. This also means that the device-dependent code on the VAX computers has little in common with the code on the IBM computer.

The device-dependent code on the IBM computer uses the relatively new CONSOLE macro to read and write to the terminal. The use of that macro limits the use of this device to full-screen mode.

Another problem with this device-dependent code is that it was never intended to drive a real TEKTRONIX 4105 at SLAC. Instead, it was intended to be used with personal computers emulating a TEKTRONIX 4105. At the time this decision was made at SLAC, the 4105 was a discontinued model and was no longer being produced or sold. This has meant that the device-dependent code has never been tested on a real TEKTRONIX 4105; instead it has been tested on a TEKTRONIX 4207 simulating a 4105.

SECTION 9.18: THE TEKTRONIX 4207 COMPUTER DISPLAY TERMINAL

This device is described in [TEK87b]. The device-dependent code for this device is very similar, although much more complicated, than that of the TEKTRONIX 4105 and most of the statements about that device apply here.
On this device, the Unified Graphics System always uses three of the four memory planes for color. The BLINK options item for subroutine UGOPEN is a way for the user to decide if the fourth memory plane is to be used for a second intensity level or for blinking. The color indices that are assigned for 8 through 15 are not as convoluted as they might appear at first. One of the last problems with this device that was resolved, was the color of the framing box in the pan and zoom functions. The color is always that of index 15 and some contortions were required so that that always turned out to be a reasonable value. In particular, it is important that it does not blink.

The device-dependent code on the IBM computer uses the relatively new CONSOLE macro to read and write to the terminal. The use of that macro limits the use of this device to full-screen mode.

This terminal is actually a very versatile device. It can do almost everything that an IBM 5080 can do except three-dimensional rotation. In the graphics area, it does everything and more that a VAXSTATION can do and it does some things, like the PICK function, much better. The only instance where the TEKTRONIX 4207 is inferior is in screen resolution, but even that deficiency is partially overcome by the easily used pan and zoom functions.

The terminal also demonstrates how abysmally bad both VM and the ASCII Terminal Emulator [IBM84d] are for trying to do real interactive work. The device-dependent code on the IBM computer contains an immense amount of redundancies, contortions, and black magic to try to get around the many problems that were encountered. Since the Terminal Emulator is so poorly documented and understood at SLAC, this device-dependent code may be very prone to failure as new releases of the operating system are installed or other changes are made.

SECTION 9.19: THE TEKTRONIX 4510 COLOR GRAPHICS RASTERIZER

This device uses a slight modification and extension (for line width) of the orders for the TEKTRONIX 4105. The rasterizer is described in [TEK86].

The device-dependent code was actually checked out on a TEKTRONIX 4510A connected to a TEKTRONIX 4693D plotter.
SECTION 9.20: THE VERSATEC ELECTROSTATIC PLOTTER

The processing that must be done on the IBM computers is very different from the processing on the VAX computers for this graphic device.

On the VAX computers, the device-dependent code must do the rasterization. Since a quick calculation indicates that a single rasterized picture would contain nearly 500,000 bytes of data, it is clear that some data compression is required. The device-dependent code therefore proceeds as follows:
1. The entire picture is broken down into line segments, and these line segments are saved in blocks of length MAXNSA words.
2. A block of storage is available to hold one "band" of the picture. A band consists of 32 consecutive scan lines in a picture. For each band in the picture, a pass is made through all of the saved line segments to fill in the bits in the band.
3. The data in the band is compressed and written to the output data set. The output data set looks like a standard print data set to the operating system.

The format of the compressed data was designed at SLAC but was very strongly influenced by the information in [Fro75]. Graphic data for the VERSATEC printer/plotter is identified by a 'CC'X (hexadecimal) value in the carriage-control character. Such a "pseudo-print-line" of data may contain up to 133 characters. A single print line may contain the data for many scan lines, or a single scan line may require many print lines. The scan lines are specified by giving the differences between the current scan line and the previous scan line. This scheme serves to compress the picture in a vertical direction. Compression in the horizontal direction is done by what amounts to run-length coding as described below.

The compressed picture data consists of "command bytes" and "data bytes" concatenated together in a print line. The command bytes are divided into two fields, a "prefix" of two bits, and a "count" of six bits. The four values for the prefix are:
1. '00'B Control Information: If the count is '111111'B=63, it means that a new picture is being started. The plotter should be moved to the start of a new page and the "previous-scan-line" buffer should be set to zeros. If 0<count<62 it means that a scan line has been completely specified. The "scan-line-differences" buffer should be exclusive or'ed into the "previous-scan-line" buffer and it should be written (count+1) times. The "scan-line-differences" buffer is then cleared to zeros. A scan line is always ended by one of these latter control bytes.
2. '01'B Skip and Plot: Set (count+1) bytes to zero in the "scan-line-differences" buffer and then insert the
next byte into the buffer.

3. '10'B Repeat Plot: Repeat the next byte (count+1) times in the "scan-line-differences" buffer.

4. '11'B Plot Raw Data: Insert the next (count+1) bytes of data into the "scan-line-differences" buffer exactly as they are given.

A "device-driver" for the VERSATEC had to be written at SLAC and installed in the VAX operating system to unpack and plot the graphic data. This work was originally done by Richard A. Moyer of the Data Analysis Group. When Release 4 of the VMS operating system arrived, it was found that extensive changes were necessary to keep things running. The device-dependent code in the Unified Graphics System did not change but the device-driver and the "print-symbiont" in the operating system both had to be modified. This work was done by Michael E. Huffer of the Data Analysis Group.

On the IBM computers, the printer/plotter is connected to the CPU through a special controller supplied by VERSATEC. This controller has two functions: First, it simulates the essential functions of an IBM 1403 or IBM 3211 printer and its controller. Second, the controller recognizes special bit patterns within the print line which signal the controller to assume that the print line contains graphic data. This graphic data is essentially the X and Y coordinates of the end points of line segments. The controller converts this line data into the raster data required by the printer/plotter. The line segments are organized into "bands". The zero-th band is at the top of a fan-fold page and consists of LINEMX scan lines; the first band is below the zero-th band. Line segments must be ordered so that the controller receives all line segments starting in the I-th band before it receives the line segments starting in band (I+1). A complete description of the format of the graphic data will be found in [VER76 and VER77].

The device-dependent code doing the sort works as follows:

1. The line segments for an entire picture are saved in blocks of length MAXNSA words. The data for each line segment includes its starting band number.

2. The line segments in each block are sorted by band number, and finally,

3. The line segments in all of the blocks are merged together, reformatted, and written to the output data set.

Each line segment uses 12 bytes of data in one of these internal blocks and 8 bytes of data in the output data set.

The sorting method used is the "Quicksort" algorithm. Quicksort is described in detail in [Knu73 and Ric72]. The specific algorithm used is described in Section 62 of [Ric72]. This algorithm is a modified Quicksort which includes an improvement which speeds up the algorithm when equal keys are present. This
enhancement is very important in this case because it is normal to have many line segments starting in the same band.

The differences between the Model 1200 and the Model V80 are infuriating. Since these units are constantly breaking down, it is impossible for a user to produce a correct plot, especially on the IBM computers, because of the difficulty of knowing which model will be connected to the computer when the file is finally sent to the printer/plotter. Yet, as usual, I seem to be the only one that is concerned about this problem.

SECTION 9.21: THE X-WINDOWS PROTOCOL

The DECwindows system is described in [DEC88a, DEC88b]. Unfortunately, these manuals, and all other X-Windows manuals, leave much to be desired. For example, the DECwindows programming examples in [DEC88a] are almost all incorrect. Most of the calls to subroutines with names like *_OF_SCREEN are shown with an incorrect argument. Another major problem with X-Windows is that there is no place to start when reading the documents. The best introduction seems to be a slightly underground DEC document [McG88].

X-Windows can also be condemned for trying to force all users to do thing its way and to forget about other more natural ways of doing things. A glaring example of this is again the examples which universally are written as a large loop with a call to X$NEXT_EVENT within the loop. In the Unified Graphics System, this mode of doing things is not possible. If a Unified Graphics System subroutine were to call X$NEXT_EVENT, and no event were available, the program would be locked in the wait state. In addition, especially in the slave-display mode, an event such as an exposure will usually occur when control is outside of the Unified Graphics System. Fortunately, DECwindows allows a user to specify Asynchronous System Traps (AST's) to process events from X-Windows. However, the use of AST's means that the code is not transportable to the IBM computer.

In spite of all of these problems, the code on the VAX is relatively simple and straightforward when all of the problems have been solved. The real objection is the massive amount of time that was necessary to achieve that end.

SECTION 9.22: GRAPHIC PSEUDO-DEVICES

The device-dependent modules for the pseudo-devices are exceptionally simple and straightforward.
SECTION 9.23: A GENERIC WORKSTATION

The device-dependent code for a generic workstation is relatively straightforward. However, it is also very complex because it has to be ready to support a wide variety of devices.

The following material will describe the format of the records exchanged by the workstation and the server, how the workstation is to respond to the records, and the mathematics needed for the three-dimensions to two-dimensions transformation.

The Format of the Records Exchanged by the Workstation and Server:

The records contain ASCII character strings, 16 bit integers (for all integer values except three-dimensional coordinates), and 32 bit integers (for three-dimensional coordinates). Floating point numbers are never exchanged. When character strings are exchanged, they are always preceded by an integer count and padded out to a multiple of 2 characters. The general form of all of the records is that the first word is the number of words in the record and the second word is an identifying integer. The workstation must always be ready to receive a record from the host computer and respond to it. In some cases, that response will include sending an acknowledgment to the host computer. The workstation never has to send unsolicited records to the host computer. The maximum sized record that will ever be exchanged will be 1024 16 bit words.

There are 12 different types of records that the device-dependent code can send to the workstation and they will now be described.

1. Open Command:

This is always the first record transmitted in a graphic session and an acknowledgment record is always sent back to the device-dependent code. The acknowledgment tells the device-dependent code what operations the workstation is capable of performing.

<table>
<thead>
<tr>
<th>Word</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>variable</td>
<td>The number of words in the record.</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>Open command identification.</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>The number of characters in the string that follows. The string is the UGOOPEN options list. At most 1024 characters of this string will be sent.</td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td>The first 2 characters of the string.</td>
</tr>
<tr>
<td>5</td>
<td>-</td>
<td>The second 2 characters of the string.</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Word</td>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>-------</td>
<td>-------------</td>
</tr>
<tr>
<td>1</td>
<td>variable</td>
<td>The number of words in the record.</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>Open response identification.</td>
</tr>
<tr>
<td>3</td>
<td>1,2,3</td>
<td>Interaction level (1 means non-interactive, 2 means slave-display, and 3 means interactive).</td>
</tr>
<tr>
<td>4</td>
<td>1,2,3</td>
<td>Drawing medium (1 means non-erasable, 2 means raster-scan, and 3 means re-fresh).</td>
</tr>
<tr>
<td>5</td>
<td>2,3</td>
<td>Dimension (2 means 2-D only and 3 means 3-D).</td>
</tr>
<tr>
<td>6,...,9</td>
<td>-</td>
<td>Limits of the 2-D space (for example: 0,...,1023 in X and 0,...,780 in Y). The order of the limits should be X low, Y low, X high, Y high. It is not necessary that the spacing of the raster units be the same in X and Y.</td>
</tr>
<tr>
<td>10,11</td>
<td>-</td>
<td>Size of 2-D space in millimeters.</td>
</tr>
<tr>
<td>12,...,23</td>
<td>-</td>
<td>Limits of the 3-D space (for example: 0,...,2**24-1 in X, Y, and Z). The order of the limits should be X low, Y low, Z low, X high, Y high, Z high. The limits in X, Y, and Z should normally be the same. These values are given as 32 bit integers.</td>
</tr>
<tr>
<td>24</td>
<td>0,1</td>
<td>Keyboard flag (1 means available).</td>
</tr>
<tr>
<td>25</td>
<td>0,1</td>
<td>Pick flag (1 means available).</td>
</tr>
<tr>
<td>26</td>
<td>0,...,n</td>
<td>Button flag (the number is the button count; the maximum value is 64).</td>
</tr>
<tr>
<td>27</td>
<td>0,1</td>
<td>Stroke flag (1 means available).</td>
</tr>
<tr>
<td>28</td>
<td>0,1</td>
<td>Locator flag (1 means available).</td>
</tr>
<tr>
<td>29</td>
<td>0,...,n</td>
<td>Valuator flag (the number is the valuator count).</td>
</tr>
<tr>
<td>30</td>
<td>0,1</td>
<td>Line structure flag (1 means dashed, dotted, and dot-dashed lines can be drawn).</td>
</tr>
<tr>
<td>31</td>
<td>0,1</td>
<td>Scissoring flag (0 means 3-D data should be scissored to the world volume and 1 means it should be scissored to the object volume). This value should normally be 0. The value of 1 is only needed on devices that cannot perform the zoom or pan operation and have limited 3-D scissoring ability. This item is ignored on a 2-D device.</td>
</tr>
<tr>
<td>32</td>
<td>0,1</td>
<td>3-D viewing transformation response flag (0 means the workstation cannot return the 3-D viewing parameters and a 1 means it can). This item is ignored on a 2-D device.</td>
</tr>
</tbody>
</table>
| 33   | -1,...,n | Number of different hardware generated character sizes that are
available. A maximum of 8 different sizes may be specified. They must be ordered from the smallest to the largest. A value of -1 means all sizes are possible. If -1 is given, one group of three words giving the relative offsets must be given below. This value cannot be 0 if the device is to be used as an interactive device.

The index of the character size to be used for the keyboard input buffer. If the previous word is -1 or 0, this value should be 0. If this is an interactive device and the previous word is -1, then the following group of three words should give the size of the keyboard input buffer to be used.

Three words for the first character size. The first word is the distance between centers of consecutive characters, the second word is the distance from the center of the character to the location point in the X direction, and the third word is the distance in the Y direction.

Three words for the second character size.

Number of different orientations for the hardware characters. A maximum of 8 different angles may be specified. They must be ordered from the smallest to the largest. A value of -1 means all orientations are possible. If -1 is given, no data follows.

Degrees in the first orientation.

Degrees in the second orientation.

While this record is large and complicated, it is essentially a constant for a given server program.

2. **Close Command:**

This is always the last record transmitted in a graphic session.

<table>
<thead>
<tr>
<th>Word</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>The number of words in the record.</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>Close command identification.</td>
</tr>
</tbody>
</table>
If possible, the server program should respond by displaying such things as the high water marks in local memory for the session and other critical limits. This will make it easier for a user to know when they are about to run into implementation limits.

3. **Clear Screen or Window Command:**

<table>
<thead>
<tr>
<th>Word</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.7</td>
<td>The number of words in the record.</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>Clear screen or window command identification.</td>
</tr>
<tr>
<td>3</td>
<td>0.1</td>
<td>A 0 indicates that the entire screen is to be cleared and all stored segments deleted. A 1 indicates that a window is to be cleared. In this case, the record contains the coordinates of the window.</td>
</tr>
<tr>
<td>4,...,7</td>
<td>-</td>
<td>The coordinates of the window (X low, Y low, X high, Y high).</td>
</tr>
</tbody>
</table>

The clear-window operation will only be requested of those workstations that have identified themselves as raster-scan devices.

4. **Manipulate Screen Command:**

<table>
<thead>
<tr>
<th>Word</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>The number of words in the record.</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>Manipulate screen command identification.</td>
</tr>
<tr>
<td>3</td>
<td>1,2</td>
<td>A 1 means turn the display unit on and a 2 means turn it off without discarding the contents of the picture.</td>
</tr>
</tbody>
</table>

If the workstation cannot do this operation, it may simply ignore the record and always leave the display on. The display should initially be on.

5. **Partial Segment Command:**

A graphic segment is transmitted as a sequence of these records.

<table>
<thead>
<tr>
<th>Word</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>variable</td>
<td>The number of words in the record.</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>Partial segment identification.</td>
</tr>
<tr>
<td>3</td>
<td>0,...,3</td>
<td>Record type (0 means this is the first record of a segment, 1 means it is an intermediate record, 2 means it is the last record, and 3 means it is both the first and last record of a segment).</td>
</tr>
</tbody>
</table>
In the case that this is the first record of a segment (that is, if word 3 is 0 or 3) then the record continues as follows:

<table>
<thead>
<tr>
<th>Word</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>-</td>
<td>The segment identification.</td>
</tr>
<tr>
<td>5</td>
<td>0,1</td>
<td>Pick state (0 means NOPICK and 1 means PICK).</td>
</tr>
<tr>
<td>6</td>
<td>0,1</td>
<td>Include state (0 means INCLUDE and 1 means OMIT).</td>
</tr>
</tbody>
</table>

The PICK and OMIT states will only be requested of re-fresh display devices, but this space is reserved for other types of devices.

Following this initial information (either 3 or 6 words) is the graphic data itself. Data blocks containing three-dimensional data will only be sent to a workstation if it has identified itself as a three-dimensional device in the open response. These graphic data blocks have the following format:

Whenever the intensity has changed, the following data block is inserted. The workstation should use this data block and the color data block to the greatest extent possible. Intensity level can be interpreted as line width.

<table>
<thead>
<tr>
<th>Word</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>k</td>
<td>1</td>
<td>Intensity data block identification.</td>
</tr>
<tr>
<td>k+1</td>
<td>1,...,5</td>
<td>Intensity indicator (1 means VDIM, 2 means DIM, 3 means MEDIUM, 4 means BRIGHT, and 5 means VBRIGHT).</td>
</tr>
</tbody>
</table>

Whenever the color has changed, the following data block is inserted. The workstation should use this data block and the intensity data block to the greatest extent possible.

<table>
<thead>
<tr>
<th>Word</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>k</td>
<td>2</td>
<td>Color data block identification.</td>
</tr>
<tr>
<td>k+1</td>
<td>1,...,8</td>
<td>Color indicator (1 means WHITE or normal, 2 means RED, 3 means GREEN, 4 means blue, 5 means YELLOW, 6 means MAGENTA, 7 means CYAN, and 8 means BLACK or background).</td>
</tr>
</tbody>
</table>

Whenever the blink mode has changed, the following data block is inserted. If blinking cannot be done, the workstation may ignore this data block.

<table>
<thead>
<tr>
<th>Word</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>k</td>
<td>3</td>
<td>Blink mode data block identification.</td>
</tr>
<tr>
<td>k+1</td>
<td>1,2</td>
<td>Blink mode indicator (1 means STEADY and 2 means BLINK).</td>
</tr>
</tbody>
</table>
Whenever the line structure for a two-dimensional line has changed, the following data block is inserted. This item only applies to two-dimensional lines; three-dimensional lines are always solid. This item will never be inserted if the open response indicated that line structure cannot be generated.

<table>
<thead>
<tr>
<th>Word</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>k</td>
<td>4</td>
<td>Line structure data block identification.</td>
</tr>
<tr>
<td>k+1</td>
<td>1,...,4</td>
<td>Line structure indicator (1 means SOLID, 2 means DASHED, 3 means DOTTED, and 4 means DOTDASH).</td>
</tr>
</tbody>
</table>

Whenever the PICK control unit is available and the pick identifier has changed, the following data block is inserted:

<table>
<thead>
<tr>
<th>Word</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>k</td>
<td>5</td>
<td>Pick data block identification.</td>
</tr>
<tr>
<td>k+1</td>
<td>-</td>
<td>Pick value.</td>
</tr>
</tbody>
</table>

The following data block is inserted whenever a two-dimensional point must be drawn.

<table>
<thead>
<tr>
<th>Word</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>k</td>
<td>6</td>
<td>2-D point data block identification.</td>
</tr>
<tr>
<td>k+1</td>
<td>-</td>
<td>X coordinate of the point.</td>
</tr>
<tr>
<td>k+2</td>
<td>-</td>
<td>Y coordinate of the point.</td>
</tr>
</tbody>
</table>

The following data block is inserted whenever a move must be done within a group of two-dimensional lines.

<table>
<thead>
<tr>
<th>Word</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>k</td>
<td>7</td>
<td>2-D move data block identification.</td>
</tr>
<tr>
<td>k+1</td>
<td>-</td>
<td>X coordinate of the line end point.</td>
</tr>
<tr>
<td>k+2</td>
<td>-</td>
<td>Y coordinate of the line end point.</td>
</tr>
</tbody>
</table>

The following data block is inserted whenever a draw must be done within a group of two-dimensional lines.

<table>
<thead>
<tr>
<th>Word</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>k</td>
<td>8</td>
<td>2-D draw data block identification.</td>
</tr>
<tr>
<td>k+1</td>
<td>-</td>
<td>X coordinate of the line end point.</td>
</tr>
<tr>
<td>k+2</td>
<td>-</td>
<td>Y coordinate of the line end point.</td>
</tr>
</tbody>
</table>

The following data block is inserted whenever two-dimensional text must be drawn. The strings transmitted by this block will always be fully scissored by the device-independent part of the Unified Graphics System.

<table>
<thead>
<tr>
<th>Word</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>k</td>
<td>9</td>
<td>2-D text data block identification.</td>
</tr>
<tr>
<td>k+1</td>
<td>-</td>
<td>X coordinate of the location point.</td>
</tr>
<tr>
<td>k+2</td>
<td>-</td>
<td>Y coordinate of the location point.</td>
</tr>
</tbody>
</table>
The following data block is inserted whenever a three-dimensional point must be added to the display file.

<table>
<thead>
<tr>
<th>Word</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>k</td>
<td>11</td>
<td>3-D point data block identification.</td>
</tr>
<tr>
<td>k+1,k+2</td>
<td>-</td>
<td>X coordinate of the point. These values are given as 32 bit integers.</td>
</tr>
<tr>
<td>k+3,k+4</td>
<td>-</td>
<td>Y coordinate of the point.</td>
</tr>
<tr>
<td>k+5,k+6</td>
<td>-</td>
<td>Z coordinate of the point.</td>
</tr>
</tbody>
</table>

The following data block is inserted whenever a move must be done within a group of three-dimensional lines.

<table>
<thead>
<tr>
<th>Word</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>k</td>
<td>12</td>
<td>3-D move data block identification.</td>
</tr>
<tr>
<td>k+1,k+2</td>
<td>-</td>
<td>X coordinate of the line end point. These values are given as 32 bit integers.</td>
</tr>
<tr>
<td>k+3,k+4</td>
<td>-</td>
<td>Y coordinate of the line end point.</td>
</tr>
<tr>
<td>k+5,k+6</td>
<td>-</td>
<td>Z coordinate of the line end point.</td>
</tr>
</tbody>
</table>

The following data block is inserted whenever a draw must be done within a group of three-dimensional lines.

<table>
<thead>
<tr>
<th>Word</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>k</td>
<td>13</td>
<td>3-D draw data block identification.</td>
</tr>
<tr>
<td>k+1,k+2</td>
<td>-</td>
<td>X coordinate of the line end point. These values are given as 32 bit integers.</td>
</tr>
</tbody>
</table>
The following data block is inserted whenever three-dimensional text must be added to the display file. The strings transmitted by this block have not been scissored. The three-dimensional string is positioned by first projecting the three-dimensional coordinates into the three-dimensional viewport and then offsetting the text with the two-dimensional offsets. If necessary, the workstation can ignore the offsets.

<table>
<thead>
<tr>
<th>Word</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>k+3,k+4</td>
<td>-</td>
<td>Y coordinate of the line end point.</td>
</tr>
<tr>
<td>k+5,k+6</td>
<td>-</td>
<td>Z coordinate of the line end point.</td>
</tr>
</tbody>
</table>

The size of the characters.

The angle of the characters.

The number of characters in the string (the largest value ever transmitted will be 1024).

The first 2 characters of the string.

The second 2 characters of the string.

The following data block is inserted whenever device-dependent data is sent to the workstation. This data block can be ignored, in fact, it usually should be ignored.

<table>
<thead>
<tr>
<th>Word</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>k+1</td>
<td>-</td>
<td>X coordinate associated with the data.</td>
</tr>
<tr>
<td>k+2</td>
<td>-</td>
<td>Y coordinate associated with the data.</td>
</tr>
<tr>
<td>k+3</td>
<td>-</td>
<td>The number of characters in the string (the largest value ever transmitted will be 1024).</td>
</tr>
<tr>
<td>k+4</td>
<td>-</td>
<td>The first 2 characters of the string.</td>
</tr>
<tr>
<td>k+5</td>
<td>-</td>
<td>The second 2 characters of the string.</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Partial Segment Response:

This response is only returned for the final record in a segment (that is, when word 3 in the partial segment command is 2 or 3).

<table>
<thead>
<tr>
<th>Word</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>The number of words in the record.</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>Partial segment response identification.</td>
</tr>
<tr>
<td>3</td>
<td>0,1</td>
<td>Response value (0 means the segment was accepted and 1 means the segment was too large to fit into the display file).</td>
</tr>
</tbody>
</table>

6. Segment Manipulation Command:

<table>
<thead>
<tr>
<th>Word</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
<td>The number of words in the record.</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>Segment manipulation command identification.</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>The segment identification.</td>
</tr>
<tr>
<td>4</td>
<td>0,1</td>
<td>Delete flag (0 means do not delete the segment and 1 means delete the segment).</td>
</tr>
<tr>
<td>5</td>
<td>0,1,2</td>
<td>Pick status flag (0 means no change, 1 means set status to NOPICK, and 2 means set status to PICK).</td>
</tr>
<tr>
<td>6</td>
<td>0,1,2</td>
<td>Include status flag (0 means no change, 1 means set status to INCLUDE, and 2 means set status to OMIT).</td>
</tr>
</tbody>
</table>

7. Miscellaneous Command:

<table>
<thead>
<tr>
<th>Word</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>The number of words in the record.</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>Miscellaneous command identification.</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>Sound the audible alarm.</td>
</tr>
</tbody>
</table>

If the workstation cannot do this operation, it may ignore the record.

8. Set Status of Control Unit Command:

These records may be sent to the server even when the control unit is not enabled.

<table>
<thead>
<tr>
<th>Word</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>variable</td>
<td>The number of words in the record.</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>Set Status of Control Unit command identification.</td>
</tr>
<tr>
<td>3</td>
<td>1,3,4</td>
<td>Control unit status being modified (1 means KEYBOARD status has changed, 3 means BUTTON status has changed, and 2 means PICK or INCLUDE status has changed).</td>
</tr>
</tbody>
</table>
If the KEYBOARD status has changed (that is, if word 3 of the command is 1), then the record contains:

<table>
<thead>
<tr>
<th>Word</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>-</td>
<td>The X coordinate of the position of the keyboard input buffer.</td>
</tr>
<tr>
<td>5</td>
<td>-</td>
<td>The Y coordinate of the position of the keyboard input buffer.</td>
</tr>
<tr>
<td>6</td>
<td>-</td>
<td>The number of characters in the keyboard input buffer (the largest value ever transmitted will be 128).</td>
</tr>
<tr>
<td>7</td>
<td>-</td>
<td>The first 2 characters of the keyboard input buffer.</td>
</tr>
<tr>
<td>8</td>
<td>-</td>
<td>The second 2 characters of the keyboard input buffer.</td>
</tr>
</tbody>
</table>

The server program should put the keyboard input buffer on the screen at, or close to, the given position. The character string has not been scissored; the server should do the best it can while maintaining its length. The keyboard cursor should be put into the first character and any use of the keyboard should be echoed in the keyboard input buffer. When the keyboard event key (CARRIAGE RETURN, ENTER, etc.) is hit, the KEYBOARD event is enabled, and the server is in the event wait state, then an event record should be sent to the device-dependent code. The characters sent to the device-dependent code should be those from the beginning of the keyboard input buffer to the cursor position. If the KEYBOARD is not enabled, then the event should be discarded. If the server is not in the event wait state, then the server should save the event and transmit it when the event wait state is entered. After the record has been processed, the keyboard input buffer should be restored to the state specified in the record described here.

If the BUTTON status has changed (that is, if word 3 of the command is 3), then the record contains:

<table>
<thead>
<tr>
<th>Word</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>-</td>
<td>The first 16 bits indicating the status of the lights on the buttons (0 means off and 1 means on).</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>7</td>
<td>-</td>
<td>The fourth 16 bits indicating the status of the lights on the buttons.</td>
</tr>
</tbody>
</table>

This record may be ignored if there are no lights associated with the buttons.

If the STROKE status has changed (that is, if word 3 of the command is 4), then the record contains:
# MAINTENANCE MANUAL

## 9. Enable or Disable Control Unit Command:

<table>
<thead>
<tr>
<th>Word</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4,5</td>
<td>-</td>
<td>Maximum stroke length as a ratio of full screen (precision is (32,30)).</td>
</tr>
<tr>
<td>6</td>
<td>-</td>
<td>Maximum stroke time in hundredths of a second.</td>
</tr>
<tr>
<td>7</td>
<td>-</td>
<td>Maximum number of strokes to be permitted (the largest value ever transmitted will be 128).</td>
</tr>
</tbody>
</table>

The maximum stroke length and/or maximum stroke time may be ignored if the workstation cannot honor them.

## 10. Obtain an Event Command:

When this record is received, the workstation should go into a timed or indefinite wait state. Then, either a time-out will occur or the workstation operator will generate a KEYBOARD, PICK, BUTTON, or STROKE event. At that time a response record is sent to the host computer.

<table>
<thead>
<tr>
<th>Word</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>The number of words in the record.</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>Obtain an event command identification.</td>
</tr>
<tr>
<td>3,4</td>
<td>-</td>
<td>The time-out value. A positive or zero value specifies the time interval (in hundredths of a second) that the workstation is to wait for the operator to signal an event. A negative value specifies that the workstation is to wait indefinitely and never indicate that a time out has occurred. If the workstation cannot time an event, it may always go into an indefinite wait. This</td>
</tr>
</tbody>
</table>
Obtain an Event Response:

If the workstation has created an event, or if a time-out has occurred, then one of the following records is transmitted.

<table>
<thead>
<tr>
<th>Word</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>variable</td>
<td>The number of words in the record.</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>Obtain an event response identification.</td>
</tr>
<tr>
<td>3</td>
<td>-1,...,4</td>
<td>The type of event being reported (-1 means emergency termination, 0 means NONE, that is, a time-out has occurred, 1 means KEYBOARD, 2 means PICK, 3 means BUTTON, and 4 means STROKE).</td>
</tr>
</tbody>
</table>

An emergency termination or a time-out event does not contain any additional information.

If a KEYBOARD event is being reported (that is, if word 3 of the response is 1), then the record contains:

<table>
<thead>
<tr>
<th>Word</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>variable</td>
<td>The number of characters in the string being returned. This value may range from zero to the length of the keyboard input buffer.</td>
</tr>
<tr>
<td>5</td>
<td>-</td>
<td>The first 2 characters of the string being returned.</td>
</tr>
<tr>
<td>6</td>
<td>-</td>
<td>The second 2 characters of the string being returned.</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

If a PICK event is being reported (that is, if word 3 of the response is 2), then the record contains:

<table>
<thead>
<tr>
<th>Word</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>-</td>
<td>The identification of the graphic segment.</td>
</tr>
<tr>
<td>5</td>
<td>-</td>
<td>The pick identification of the graphic item.</td>
</tr>
</tbody>
</table>

If a BUTTON event is being reported (that is, if word 3 of the response is 3), then the record contains:

<table>
<thead>
<tr>
<th>Word</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>-</td>
<td>The button index.</td>
</tr>
</tbody>
</table>

If a STROKE event is being reported (that is, if word 3 of the response is 4), then the record contains:
If the LOCATOR is being reported (that is, if word 3 of the command is 5), then the record contains:

- Word 4: X coordinate of the locator.
- Word 5: Y coordinate of the locator.

If a VALUATOR is being reported (that is, if word 3 of the command is 6), then the record contains:

- Word 4: Index of the valuator being reported.
- Word 5: Valuator value.

An emergency termination does not contain any additional information.

If a VALUATOR is being reported (that is, if word 3 of the command is 6), then the record will be 516 words long.

When this record is received, the workstation should go into a wait state and wait for the workstation operator to signal that the LOCATOR or VALUATOR data is available.

No time-out can occur with this operation.

It is this record that dictates the minimum number of 16 bit integers in the I/O buffers. If a full 128 words are returned, the record will be 516 words long.

---

### Table: Description of Words in a Control Unit Command

<table>
<thead>
<tr>
<th>Word</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>Sample a control unit command:</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>Sample a control unit response:</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>Sample a control unit response:</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>Sample a control unit command:</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>Sample a control unit response:</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>Sample a control unit response:</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>Sample a control unit response:</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>Sample a control unit response:</td>
</tr>
</tbody>
</table>
12. **Three-Dimensional Viewing Transformation Command:**

This record will only be sent to the workstation only if it has identified itself as a three-dimensional device in the open response. The request for the current three-dimensional parameters will only be sent if the open response has indicated that the data is available.

<table>
<thead>
<tr>
<th>Word</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3,33</td>
<td>The number of words in the record.</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
<td>Set 3-D Viewing transformation command identification.</td>
</tr>
<tr>
<td>3</td>
<td>0,1</td>
<td>Action flag (0 means the transformation data is being supplied and 1 means the current data should be returned in a response record).</td>
</tr>
</tbody>
</table>

If the data is being supplied (that is, if word 3 of the command is 0), then the record also contains:

<table>
<thead>
<tr>
<th>Word</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4,...,7</td>
<td>-</td>
<td>The viewport in 2-D coordinates (X low, Y low, X high, Y high).</td>
</tr>
<tr>
<td>8,...,19</td>
<td>-</td>
<td>The object volume. These values are given as 32 bit integers.</td>
</tr>
<tr>
<td>20,...,25</td>
<td>-</td>
<td>The eye point. These values are given as 32 bit integers.</td>
</tr>
<tr>
<td>26,...,31</td>
<td>-</td>
<td>The upward direction. The coordinates are given as integers with a precision of (32,30). These values are given as 32 bit integers.</td>
</tr>
<tr>
<td>32,33</td>
<td>-</td>
<td>The projection flag (0 means a parallel projection and a positive value means a point projection with the value giving the position of the near scissoring plane at precision (32,30)). This value is given as a 32 bit integer.</td>
</tr>
</tbody>
</table>

**Three-Dimensional Viewing Transformation Response:**

This record is sent to the device-dependent code only when the transformation data is being requested, that is, when word 3 of the viewing transformation command is 1.

<table>
<thead>
<tr>
<th>Word</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>28</td>
<td>The number of words in the record.</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
<td>Set 3-D viewing transformation response identification.</td>
</tr>
<tr>
<td>3,...,14</td>
<td>-</td>
<td>The object volume. These values are given as 32 bit integers.</td>
</tr>
<tr>
<td>15,...,20</td>
<td>-</td>
<td>The eye point. These values are given as 32 bit integers.</td>
</tr>
</tbody>
</table>
The Mathematics of the Three-Dimensions to Two-Dimensions Transformation:

This section describes the mathematics involved in projecting the three-dimensional data onto the three-dimensional viewport of the screen. The section describes how the transformation is performed when three-dimensional data is sent to a two-dimensional device. A real three-dimensional device should act as closely as possible to this. The section also describes the seven recommended transformations for rotating, panning, and zooming the three-dimensional data.

First, the notation used in this section will be described.

1. Scalar quantities are shown in lower case. Vector and matrix quantities are shown in upper case.
2. Points and vectors are represented as column vectors.
3. If X is a square matrix, the notation \( X^T \) means the transpose of X.

The given items for the three-dimensional to two-dimensional transformation are:

1. The limits of the three-dimensional viewport on the screen: \( xvl, yvl, xv h, yvh \).
2. The limits of the object volume: \( xol, yol, zol, xoh, yoh, zoh \).
3. The eye point:
   \[
   E = \begin{bmatrix} xe & ye & ze \end{bmatrix}^T.
   \]
4. The upward direction:
   \[
   U = \begin{bmatrix} xu & yu & zu \end{bmatrix}^T.
   \]

The object is to find a transformation that transforms a three-dimensional point,

\[
P = \begin{bmatrix} x & y & z \end{bmatrix}^T,
\]

in three-dimensional world coordinates into the corresponding point,
\[ Q = \begin{bmatrix} t & u \end{bmatrix}^{T}, \]

in the coordinates of the three-dimensional viewport. The transformation will be represented as a 3x4 matrix, \( A \), such that:

\[ \begin{bmatrix} k \end{bmatrix} = \begin{bmatrix} A \end{bmatrix} \begin{bmatrix} x \end{bmatrix}, \]

\[ \begin{bmatrix} k 1 \end{bmatrix} = \begin{bmatrix} A \end{bmatrix} \begin{bmatrix} y \end{bmatrix}, \]

\[ \begin{bmatrix} k 2 \end{bmatrix} = \begin{bmatrix} A \end{bmatrix} \begin{bmatrix} z \end{bmatrix}. \]

This matrix equation is to be interpreted in the following manner. First, expand it into 3 scalar equations to get:

- \( k t = a_{11} x + a_{12} y + a_{13} z + a_{14} \),
- \( k u = a_{21} x + a_{22} y + a_{23} z + a_{24} \),
- \( k = a_{31} x + a_{32} y + a_{33} z + a_{34} \).

The first two equations are then divided by the third to obtain \( t \) and \( u \) as rational functions of \( x, y, \) and \( z \).

The first step in computing the matrix \( A \) is to compute the largest square area that will fit in the three-dimensional viewport. This square should be centered in the three-dimensional viewport. We shall use the notation \( x_l, y_l, x_h, y_h \) to refer to the limits of this square.

Next, the center of the object volume, \( O = [x_0 \ y_0 \ z_0] \), may be computed by:

- \( x_0 = 0.5 \ (x_h + x_l) \),
- \( y_0 = 0.5 \ (y_h + y_l) \),
- \( z_0 = 0.5 \ (z_h + z_l) \).

Now compute the distance, \( d_1 \), from the center of the object volume, \( O \), to the eye point, \( E \), and the half size, \( d_2 \), of the projection screen in three-dimensional world coordinates.

\[ d_1 = \sqrt{(x_0-x_e)^2 + (y_0-y_e)^2 + (z_0-z_e)^2} \]
\[ d_2 = 0.5 \sqrt{(x_h-x_l)^2 + (y_h-y_l)^2 + (z_h-z_l)^2} \]

By using this rather large value for the screen size, it is assured that the entire object volume will be displayed within the three-dimensional viewport in almost all cases. If the screen is too large, the workstation operator can always zoom in on the object.

The next step is to compute the normalized trihedral for the projection. This requires computing a unit vector defining the horizontal direction of the screen, \( V_1 \), the vertical direction of the screen, \( V_2 \), and a unit vector in the view direction, \( V_3 \). These three unit vectors form a mutually orthogonal set.
\[ V_3 = \text{unit}(0 - E), \]
\[ V_1 = \text{unit}(V_3 \times U), \]
\[ V_2 = \text{unit}(V_1 \times V_3). \]

The items involved in computing the transformation are shown in Figure 9.23.1.

![Figure 9.23.1: The Items Used in Computing the Three-Dimensions to Two-Dimensions Transformation.](image)

The transformation matrix for a point transformation is:

\[
A = \begin{bmatrix}
&xh-xl&0&xh+xl \\
&0&y_l-yh&y_l+yh \\
&0&0&2 \\
\end{bmatrix} \begin{bmatrix}
d_1&0&0 \\
d_1&0&0 \\
0&0&d_2 \\
\end{bmatrix} \begin{bmatrix}
V_1 & V_2 & V_3 \\
\end{bmatrix}^{\text{T}} \begin{bmatrix}
1&0&0&-x_l \\
0&1&0&-y_l \\
0&0&1&-z_e \\
\end{bmatrix}
\]

The transformation for a parallel transformation is computed in the following manner. First let the first two rows of the matrix \( B \) be

\[
\begin{bmatrix}
1&0&0&-x_e \\
V_1 & V_2 \\
\end{bmatrix}^{\text{T}} \begin{bmatrix}
1&0&0&-x_e \\
0&1&0&-y_e \\
0&0&1&-z_e \\
\end{bmatrix}
\]

and let the third row be
Then the matrix of the parallel transformation is:
\[
\begin{bmatrix}
0 & 0 & 1 \\
0 & 0 & 1 \\
1 & 0 & 0
\end{bmatrix} \begin{bmatrix}
xh-xl & 0 & xh+xl \\
yh-yl & 0 & yh+yl \\
0 & 0 & 0
\end{bmatrix} \begin{bmatrix}
1 & 0 & 0 \\
1 & 0 & 0 \\
0 & 0 & 1
\end{bmatrix}B.
\]

After the lines and points have been projected by this transformation, the resulting lines and points should be scissored at the limits of the three-dimensional viewport. If this is impossible on the graphic device, the scissoring can be made at the screen boundary.

The transformations that should be able to be performed locally on the three-dimensional data on the workstation can be thought of as follows:

1. **Horizontal Rotations**: The upward direction, \(V_2\), and center of object, \(O\), remain fixed. The eye point, \(E\), horizontal direction, \(V_1\), and view direction, \(V_3\), rotate about the center of the object, \(O\), in the plane determined by them. The distance from the eye point, \(E\), to the center of object, \(O\), remains constant.

2. **Vertical Rotation**: The horizontal direction, \(V_1\), and center of object, \(O\), remain fixed and \(E\), \(V_2\), and \(V_3\), rotate about \(O\) in the plane determined by them.

3. **Two-Dimensional Rotation**: The view direction, \(V_3\), and center of object, \(O\), remain fixed. \(V_1\) and \(V_2\) rotate about \(O\) in the plane determined by them.

4. **Horizontal Pan**: The eye point, \(E\), and center of object, \(O\), move together parallel to the horizontal vector, \(V_1\).

5. **Vertical Pan**: The eye point, \(E\), and center of object, \(O\), move together parallel to \(V_2\).

6. **In-Out Pan**: The eye point, \(E\), and center of object, \(O\), move together parallel to \(V_3\). This motion has very little affect on the picture for a point projection, and no effect at all for a parallel projection. However, it is important for positioning the center of motion for a rotation.

7. **Zooming In and Out**: The center of object, \(O\), remains fixed. The distance from the eye point, \(E\), to \(O\) and the size of the object volume are simultaneously increased or decreased.

The transformation matrix and the near scissoring plane should be recomputed after each of these incremental transformations.

The code running on the Silicon Graphics was written by Lennox E. Sweeney of the SLAC Computing Services. Part of the code on the VAX itself had to be written in the C Language. This C code was written by Michael F. Gravina. The code on the IBM computer contains an Assembler Language module that could be used to drive the ETHERNET link. This was never done because people seem to
have lost interest in using a workstation in this way and because it appears to be a difficult job with very little helpful documentation.
SECTION 10: COMPARISONS WITH OTHER SYSTEMS

This section will describe the history of the Unified Graphics System and its relation to other similar systems. As much as possible, the origins of the ideas and concepts used in the Unified Graphics System will be described.

SECTION 10.1: A HISTORY OF THE UNIFIED GRAPHICS SYSTEM

Work on the Unified Graphics System began during the middle of 1972. At that time SLAC had access to a few graphic devices (a CALCOMP Drum Plotter, IBM 2250's, and an IDIIOM Graphic System). There were also plans to obtain additional graphic devices (a microfilm recorder, and TEKTRONIX Display Terminals). As was common at that time, each graphic device had its own subroutine package that was used to drive it. It was clear that this situation could not go on indefinitely unless SLAC was willing to absorb ever increasing programming costs for its computer graphic applications. Development was therefore begun in an effort to alleviate these problems.

Some of the initial design requirements were:

1. A wide variety of graphic devices must be able to be supported. Interactive devices must be fully supported.
2. There must be a subset of operations such that a user who restricts himself to this subset will find it very easy to switch from one graphic device to another. At the same time, it must be possible to take advantage of some of the more unusual features of a graphic device; the IDIIOM had a number of advanced accessories for its time including an animation camera and a rotary stereo viewer.
3. A program must be able to control more than one graphic device. At a minimum, it should be able to write to one interactive device and one non-interactive device.
4. The number of subroutines must be kept small, and the number of arguments in the subroutines must also be kept small.
5. The system must be available for both FORTRAN and PL/1.

A notable oversight was that transportability of the system from one computer to another was not considered important.

The first version of the Unified Graphics System became available in December of 1972 and ran on an IBM/360, Model 91 computer. That version was described in [Bea73a, Bea73b, Bea75a, and Bea75b]. This system was written almost exclusively in Assembler Language and supported a CALCOMP Drum Plotter (Model 564/565), the IBM 2250, and the IDIIOM Display Console. The system was developed over the years by adding more graphic devices to the system and by adding a few new subroutines.
By the middle of 1975, a few problems were apparent. The principal problem was that the source code was becoming unwieldy. Every time a change or addition was required, it meant that both the FORTRAN and PL/1 version had to be changed. In addition, SLAC was soon to get the PL/1 Optimizing Compiler, and the intent was for SLAC to continue support of the PL/1 F-Level Compiler.

The principal changes in the second version of the Unified Graphics System were therefore internal. The subroutines that were coded in Assembler Language were rewritten in a language-independent manner and macro libraries for each language were prepared. When a FORTRAN object deck was to be generated from its Assembler Language source module, the source module was assembled with the FORTRAN macro library made available to the Assembler. If a PL/1 Optimizing Compiler object deck was required, the only difference was that a different macro library was made available to the Assembler. This scheme was versatile enough that the FORTRAN and PL/1 calling sequences for some subroutines were actually different.

There were also a few minor external changes to the Unified Graphics System at this time. Mostly, this consisted of adding the options argument to a few subroutines that did not already have it. This version of the system was described in [Bea76a and Bea76b]. A short note describing the differences between the new and old systems was available in [Bea76c].

By 1977, the need for a portable version of the system was apparent. Therefore, a version of the Unified Graphics System coded almost entirely in FORTRAN was prepared [Bea78]. This version was tested on the IBM computers; only enough graphic devices were supported to check out the device-independent parts of the system. A few restrictions applied to this version; the principal one was that a program could only support a single graphic device. It was this version of the Unified Graphics System that was transported to the VAX-11/780 computers where a moderately large number of graphic devices were supported. Other people at SLAC moved this system to a PDP-11 and a SIGMA/5 computer. The differences between the VAX version of the Unified Graphics System and the IBM version were described in [Bea80b].

For some time, the group at SLAC charged with maintaining the IBM computers had been pushing the VM/CMS operating system. VM is an abysmally deficient operating system. Nevertheless, when SLAC got an IBM 3081 computer at the beginning of 1981, it was decided to support only VM on that machine. This forced many people to move their work to VM and a version of the Unified Graphics System was prepared for VM. This version was described in [Bea80a].

Distribution tapes of this version of the Unified Graphics System were sent to quite a few other computer installations. The IBM Assembler Language version was sent to 28 other computer centers, the IBM FORTRAN version was sent to 13 places, and the VAX version was sent to 19 installations. In most cases, I do not
know if the recipient ever made any use of the system or not. In
the case of the IBM version, I would estimate that no more than
half actually used the Unified Graphics System. The difficulty,
in part, is that there is no standard way to transfer data from
an IBM computer to a graphic device. For example, the TEKTRONIX
terminals at SLAC were connected to the computer through the
WYLBUR-MILLEN interface [Faj73], but this was of little use to
most recipients. On the VAX computers, this problem is much less
severe, and I believe most of the installations receiving the VAX
version made use of it.

Once again, by the end of 1980, it became clear that problems
existed. Some of these problems were:

1. The inability to control more than one graphic device
   from a program was causing trouble on the VAX
   computers.
2. Portability was recognized as being much more important
   than originally thought. It would be much preferable
   if most of the source code was common among the various
   computers.
3. The EXTENDED Character Sets were limited to 256
   distinct characters. Nearly this many had been
   defined, so expansion in this area was impossible.
4. The scaling and scissoring scheme allowed users to
   write device-dependent programs. There was a way to do
   almost anything in a device-independent manner, but the
   device-dependent way was often simpler and many users
   chose the easy way.
5. It was difficult to save a picture in a data set in a
   device-independent form. As long as the scaling and
   scissoring manipulations were not too complicated, the
   job could be done. However, it was, I believe,
   impossible to solve this problem in a completely
   general way.
6. There was strong compatibility between the Assembler
   Language and FORTRAN versions. In particular, the
   graphic elements had an identical format. This caused
   problems because different data types can be packed
   together very easily in Assembler Language modules, but
   the same operations result in very bad FORTRAN.
7. The internal organization made it difficult to add a
   new graphic primitive. The problem was that a new
   graphic primitive required extensive changes to all
   existing device-dependent modules.
8. While it was not difficult to add a new graphic device
   to the system, different organization could make it
   even simpler and also reduce the size of the device-
   dependent code.
9. Other similar systems had introduced interesting and
   useful ideas but extensive changes would be required to
   incorporate them into the system. Also, a terminology
   has arisen in computer graphics that is almost
   standard, but which is different from that used in the
   earlier versions of the Unified Graphics System.

For all of these reasons, a totally new version of the Unified
Graphics System was prepared. This new version is described in [Bea81a and Bea81b] and, of course, this document. A short note describing the differences between the new and old systems is available in [Bea84].

It was decided that compatibility with the old version would not be a strong constraint. The overall flavor of the system would be preserved, but any changes that appeared to be improvements would be made. Subroutine names were changed, picture generation, especially for points and text, was changed, scaling and scissoring was changed, interactive control was changed, and numerous other external and internal changes were made.

It was also decided that the requirements for a PL/1 version could be dropped for two reasons. First, the PL/1 F-Level Compiler was no longer supported at SLAC and the PL/1 Optimizing Compiler allowed FORTRAN modules to be mixed with PL/1 modules. The second reason for dropping PL/1 was that FORTRAN-77 Compilers were becoming available. The big advantage that PL/1 had was the availability of character string operations and the presence of statements (principally the IF-THEN-ELSE) which allowed legible programs to be written. With the coming of FORTRAN-77, both of these features were available in FORTRAN.

One of the areas where major changes have been made is in the manipulation of interactive control units (keyboards, picks, buttons, etc.). In this area the Unified Graphics System has adopted the ideas of Foley and Wallace [Fol74] which were also used in the ACM-SIGGRAPH Standards Proposal [ACM77, ACM79] and the GKS Standards Proposal [ACM84].

In late 1984, SLAC acquired some IBM 5080 graphic devices. For the first time, SLAC now had a graphic device for general use that was capable of doing some three-dimensional manipulation within the device itself. Thus, during the latter part of 1984 and the beginning of 1985, the three-dimensional primitives and the code to process them were added to the Unified Graphics System. At this time the polygon fill and device-dependent data primitives were also added. In spite of all of the "standardization" efforts going on in the world, there was very little guidance in this project. The GKS proposal was strictly two-dimensional so no guidance for the three-dimensional manipulations was available there. The ACM-SIGGRAPH proposal contained three-dimensional primitives and a large structure to control the projection of three-dimensional picture data into two-dimensions. It did not, however, have much to say about high performance graphic devices which could themselves manipulate and control the projection. The additions were therefore made solely with the intention to preserve the basic principles that were used in originally designing the Unified Graphics System. Ease of use while minimizing the number of subroutines were the most important considerations. It was also important that existing programs continued to work as they had before.
The addition of three-dimensional primitives to the Unified Graphics System has been an abysmal disaster. There are a number of causes of this situation. First, the IBM 5080's were purchased without any idea of what they would be used for. The purchase was made in secret, so no advance planning could be done. When its purchase finally became known, additional months were squandered as people tried to decide if any Unified Graphics System support was wanted. When it was finally decided that the Unified Graphics System should provide some support for the IBM 5080, we were at least eight months behind in any reasonable schedule. The addition of three-dimensional primitives was a major project because it affected so much of the device-independent code. This resulted in additional people, with no commitment to the "unified" part of the Unified Graphics System, being brought into the project. Before coding actually started, a document was prepared that described a proposal for a three-dimensional addition to the Unified Graphics System. That document clearly described what-could-be-done and what-could-not-be-done under the proposal. The proposal was supposedly agreed to by all of the people involved. When coding began, the addition of items from the can-not-be-done list to the device-dependent code of the IBM 5080 was one of the first things to be done. These items cause serious problems with the transportability of programs from one graphic device to another, in particular, it becomes difficult to do preliminary checkout of an IBM 5080 application on a simpler device. These device-dependent items are not described in [Bea81a]. Had anyone had any idea of what these units would be used for and if these additions were really necessary, many of them could have been done in a device-independent manner.

The addition of the polygon fill primitive is also unsatisfactory. It was added because most color graphic devices have this ability and the GKS standardization proposal included it. It is however, an extremely difficult primitive to process in a reasonable manner on a graphic device that does not support it in hardware. The problem is that when the primitive is supported in hardware, a filled polygon will usually erase any primitives that were drawn earlier in the polygon's area. To simulate this correctly on a simple device like a pen plotter would require that the entire picture be retained and nothing written out until it was certain that no polygons would wipe out parts of the picture. This would turn the device-dependent code for the simplest graphic device into the most complicated of modules. If one wants to get an idea of the problems this has caused other people, one should read what the GKS proposal says about the subject. That proposal engages in all sorts of deliberate ambiguities when trying to say what the system should do on graphic devices which do not support this feature. In the final analysis, the GKS document avoids saying anything about the problem.

During the end of 1989 and beginning of 1990, SLAC moved from the VM/HPO operating system on the IBM computers to the VM/XA system. While these two systems are superficially similar, IBM really
pulled the rug out from under modules coded in Assembler Language. It appears that IBM replicated all of their earlier mistakes in the interests of "compatibility" while screwing up everything else. A new version of the Unified Graphics System was therefore prepared. Little change was made to the FORTRAN modules but the Assembler Language modules required extensive changes. Only parts of the earlier version of the Unified Graphics System would run under VM/XA and that part would only run in "370 compatibility" mode in a small machine. The new VM/XA version runs in any mode in any size machine; unfortunately, it will not run under VM/HPO. There were no external changes in this version.

At the same time that the VM/HPO to VM/XA conversion was taking place, SLAC acquired some Silicon Graphics workstations for use from the VAX over an ETHERNET connection. The device is capable of three-dimensional manipulations and it came as no surprise when I was pushed aside on the interesting work on this device also. The use of a three-dimensional graphic device at SLAC appears to be completely dependent on the extraneous device-dependent extensions to the Unified Graphics System that were inserted into the IBM 5080 device-dependent code without my knowledge. Since these extensions have no counterpart in any of the standardization efforts such as GKS-3D, this will make these programs very difficult to convert to another system when the Unified Graphics System finally dies.

On a few occasions, the Unified Graphics System has been criticized for the small number of alternatives supplied for such attributes as intensity level and color. It has been suggested, for example, that intensity level could be specified by a floating point value between zero and one, and that color could be specified by three values giving the red-green-blue mix. In the first place, graphic devices used for line drawing do not usually have that much flexibility in this area. Secondly, even if they have great flexibility in this area, this flexibility is wasted for line drawing devices. A number of studies have been made which show that only a few intensity levels or colors can be reliably distinguished in line drawn pictures. For example, a table is shown in [Barm66] which indicates that, with 95 percent accuracy, the average person can distinguish only 4 different intensity levels and 11 different colors. In addition, to get results as high as 4 and 11, the intensity levels and colors must be very carefully chosen. A similar table in [Fol74] gives even smaller numbers for easily distinguished attributes (2 intensity levels and 6 colors). However, it must be pointed out that the above conclusions only apply to graphic devices used to display line drawn pictures. When continuous tone pictures, for example LANDSAT images, are displayed, a large number of intensity levels and colors are required.

As of the date of this document, I have sent the VAX version of this system to 129 different installations while I have sent the IBM version to 36 different installations. These installations have included computing centers in Australia, Austria, Brazil,
Canada, Peoples Republic of China, Republic of China, Denmark, Finland, France, Germany, Israel, Italy, Japan, Russia, South Korea, Sweden, and Switzerland. In addition, many of these people have passed their copy on to other installations and some people have taken it away from SLAC without my knowledge.

SECTION 10.2: THE GRAPHICS COMPATIBILITY INTERFACE

GCI, the Graphics Compatibility Interface [GCI72] was developed by Programming Methods Incorporated, of Palo Alto, California, under contract to the Ames Research Center of NASA at Moffett Field, California. This is one of the very few device-independent graphic packages that was in existence and publicly documented when the Unified Graphics System was started. GCI is coded principally in FORTRAN and was available on NASA's IBM 360/Model 67 computer.

GCI provides the user with a moderate number (about 40) of FORTRAN subroutines which may be used to generate pictures on a number of different devices. The subroutines make it reasonably easy to produce graphs with labels and titles. Although one of the early devices to be supported was an ADAGE, the system did not provide a means of writing interactive programs; any interaction had to be performed by device-dependent subroutines which were outside of GCI.

A few of GCI's subroutines have arguments which are very similar to the OPTIONS argument in Unified Graphics subroutines. This idea was adopted from GCI and expanded in scope for inclusion in the Unified Graphics System.

SECTION 10.3: DISSPLA

DISSPLA, Display Integrated Software System and Plotting Language [DIS74, Hir75] is a proprietary product of Integrated Software Systems Corporation of San Diego, California. DISSPLA consists of a very large number (over 150) of FORTRAN subroutines to drive a wide variety of non-interactive graphic devices. The large number of subroutines clearly makes this a difficult system to learn. This is ameliorated somewhat by the fact that only a few subroutines (30 or 40) are needed for simple pictures; many of the subroutines are of a fairly high level like those in the Unified Graphics System Algorithm Manual. Since the system is largely coded in FORTRAN, it is essentially independent of the host computer and is available on the IBM 360/370, CDC 6000/7000, and UNIVAC 1100 and a number of other computers.
One of DISSPLA’s features is the ease with which new devices can be added. In DISSPLA, all plotting data is broken down into the operations of "pen up", "pen down", and "move pen", and these operations are passed to an interface module. The interface module then calls the device-dependent code which is usually supplied by the manufacturer of the graphic device. While this does mean that the addition of a new device is a simple matter, it also poses a problem; it is very difficult for DISSPLA to take advantage of the special features of a device. If the graphic device has a hardware character generator, for example, DISSPLA cannot use it easily in a device-independent manner. While DISSPLA has this narrow interface between the device-independent and device-dependent code, the Unified Graphics System has a much broader interface and can more easily utilize the special feature of a graphic device. The broader interface of the Unified Graphics System allows much greater flexibility, but does require more work when a new device is to be added.

The concept of a rectangular shielded area was adopted from DISSPLA.

SECTION 10.4: GINO-F

GINO-F, the Graphical Input and Output for FORTRAN System [Woo--, Woo71] was written at Cambridge University and at the Computer Aided Design Centre in Cambridge, England. Various versions of this system have been written, some principally in Assembler Language, and others principally in FORTRAN. GINO-F is widely distributed in England.

It was GINO-F which first developed the handshaking scheme between device-independent and device-dependent code wherein the device-independent code asks the device-dependent code if it can perform an operation. The Unified Graphics System adopted this idea from GINO-F.

SECTION 10.5: THE GRAPHICS COMPATIBILITY SYSTEM

GCS, the Graphics Compatibility System [GCS74a, GCS74b] was developed at the United States Military Academy at West Point, New York. GCS is largely coded in FORTRAN and has been distributed to a few other installations. GCS consists of a moderately large number (the primer describes 68) of subroutines which can drive a number of graphic devices including interactive devices.
GCS was one of the few early device-independent graphic systems which fully supported interactive graphic devices. GCS also uses character string arguments similar to the Unified Graphics System's OPTIONS argument to select non-default options for a graphic device.

SECTION 10.6: THE GRAPHICS DISPLAY SYSTEM

The Graphic Display System, also known as GD3 [Mil76], was written at CERN in Geneva, Switzerland. The system runs on CERN's CDC 7600/6000 and IBM/370 computers and supports both non-interactive and interactive graphic devices. GD3 is written principally in FORTRAN.

For most non-interactive devices, a user's program does not directly generate orders for the graphic device. Instead, the actual output is a display file which contains a complete device-independent description of the picture. The GD3 system then provides a group of "interpreters" which can transform such a display file into the actual orders for any of the supported graphic devices.

The programming manual lists about 60 subroutines, but most applications will not need all of these subroutines. The number of arguments is small in most cases, so GD3 should be quite easy to learn and use.

The algorithm used in subroutine UGCHTR was strongly influenced by a similar subroutine in GD3.

SECTION 10.7: THE INTEGRATED GRAPHICS SYSTEM

The Integrated Graphics System [Goo76, Bli76] was developed at the University of Michigan in Ann Arbor, Michigan. This system was a later entry into the growing field of device-independent graphics packages. The Integrated Graphics System runs under the Michigan Terminal System on their Amdahl 470V/6 computer and seems to be fairly strongly coupled to that system. The system has been distributed to a few other installations.

For simple pictures, there is almost a one-to-one correspondence between the Integrated Graphics System and the Unified Graphics System; however, certain areas of the Integrated Graphics System are more fully developed than the related areas in the Unified Graphics system. In the Integrated Graphics System, the creation of a picture is also a two-step process. The picture is first added to an internal data structure and later this data structure
is converted to device-dependent orders and transmitted to the graphic device. In the Integrated Graphics System, the data structure is not directly accessible by the programmer, as a graphic segment in the Unified Graphics System is, but this data structure has more versatility than the Unified Graphic System's graphic segments. The Integrated Graphics System's data structure contains a flexible transformation scheme and subpictures may be defined and utilized. The data structure may be either two-dimensional or three-dimensional. In the three-dimensional case, polyhedra may be defined and built-in hidden line removal algorithms may be applied to the polyhedra when the picture is generated.

The Integrated Graphics System also tries to simulate pick and locator control units on interactive graphic devices which do not have them. Thus, the programming task is simplified because the programmer can always assume they are available. However, it is not clear how well this works because the simulation usually requires that the console operator type in such things as the pick identifier or the screen coordinates for the locator.

SECTION 10.8: THE GENERAL PURPOSE GRAPHICS SYSTEM

GPGS, the General Purpose Graphic System [Gro75] was developed jointly by the Delft University of Technology and the Catholic University of Nijmegen both in the Netherlands, and the University of Cambridge in England. Two Assembler Language versions of the system were developed; one for an IBM 370 with a satellite PDP-11/45 and another for a stand-alone PDP-11/45. A third version, coded principally in FORTRAN, was developed at the University of Trondheim in Norway. Much of the design of GPGS was based on GINO-F. Development of the Assembler Language versions of the system began in 1972 while the FORTRAN version was started in 1974.

This system consists of a large number of subroutines, nearly 110. However, many of these subroutines will not normally be needed and many of the necessary subroutines fall into classes with very similar calling sequences. The result is a system that may be easy to learn and use.

GPGS supports both non-interactive and interactive graphic devices. No attempt is made to simulate items like a light pen if it is not present in the hardware. Multiple graphic devices may be utilized in a single program; the selection of a "current" or active device is done in a similar fashion as in the Unified Graphics System. The data given to this system may be either two-dimensional or three-dimensional. A number of subroutines are provided to generate transformations of the three-dimensional data into the two-dimensional screen. There is, however, no provision for hidden line algorithms to work on the three-
SECTION 10.9: THE GRAPHICAL DATA DISPLAY MANAGER

The Graphical Data Display Manager, GDDM [IBM83a, IBM83b], is IBM's rather late entry into this field. Actually, there are some reasons to question whether this system should be listed here. In the first place, I suspect its principal use may be on non-graphic display stations where it can be used to design data entry panels. Also, the system only works on some of the graphic devices made by IBM. In addition, it is not clear how device-dependent the system really is because the documentation contains numerous warnings about a given feature not working on certain devices.

The system works on some models of the IBM 327x Display Stations, the IBM 8775 Display Station, the IBM 3290 Information Panel Display, and the IBM 3287 and IBM 4250 Printers. The subroutines in the system may be called from PL/I, COBOL, FORTRAN, APL, and BASIC. It is clear from the Programming Reference Manual that the system is written in Assembler Language. A large number of fancy character fonts are available, but they do not work on all devices. Multiple graphic devices may be used at one time; the procedure is identical with the Unified Graphics System's active device concept.

The manuals list a total of 197 user-callable subroutines in GDDM. The examples are not particularly easy to read because almost all arguments are numeric including the selection of such items as color. In addition to GDDM, the manuals describe the Presentation Graphics Feature, PGF, which uses GDDM to produce line graphs, histograms, bar charts, and pie charts. PGF consists of an additional 66 subroutines.

SECTION 10.10: THE ACM-SIGGRAPH STANDARD PROPOSAL

This proposal is an outgrowth of a Workshop on Machine Independent Graphics [ACM74]. The conference was sponsored by the National Bureau of Standards and the Special Interest Group on Graphics (SIGGRAPH) of the Association for Computing Machinery (ACM). It was held in Gaithersburgh, Maryland on 22-23 April 1974. Device-independent graphics was one of a number of topics scheduled to be covered at the conference. One of the results of the conference was a call for some sort of standards in graphics software, and a committee was set up.
Unfortunately, the committee got off to a very bad start. The conference included a session where existing device-independent graphic systems were to be discussed, and it was this session which was canceled to begin setting up the committee. The committee also included few people who had first hand knowledge of the problems and compromises invoked in writing a device-dependent system. The result was another committee trying to re-invent the wheel.

The first report of the committee appeared in late 1977 [ACM77], and a revised report appeared two years later [ACM79]. The second report was submitted to the American National Standards Institute (ANSI) but was eventually rejected. ACM has talked of making this an "ACM Standard" similar to the IEEE Standards. The final effect of these reports is still unclear at this time.

The proposal prescribes a large number of graphic functions that the authors of the documents think a graphic package should provide. Since the proposal does not describe how it is to be implemented in a given computer language, it could be a substantial job to convert programs between two systems, both of which adhered to the proposal.

The relation between this proposal and the Unified Graphics System is that a subset of the proposal is compatible with a subset of the Unified Graphics System. Picture generation is similar and interactive control of input devices is very similar. Major differences are in the way a picture is accumulated and sent to a graphic device and in that the proposal calls for a full set of interactive controls to be simulated if they are not actually available on the device.

The proposal is three-dimensional but there is no means to link the three-dimensions to two-dimensions transformation to the interactive control units. This means that a user can produce projected images of three-dimensional stick figures, without hidden line removal, but little else.

SECTION 10.11: THE GKS STANDARD

GKS, the Graphical Kernal System [ACM84, ANSI85] has now been adopted by the International Standards Organization (ISO) and the American National Standards Institute (ANSI). GKS was initially a German project and was started around 1974. The original design was for a two-dimensional graphics system and was reasonably simple and straightforward. Then GKS began its growth. Discussions spread beyond Germany, it became three-dimensional, and evolved into something very similar to the ACM-SIGGRAPH proposal. In still later versions, GKS again became two-dimensional. The version submitted to the ISO was Version 7.0, which is at least the twentieth version.
GKS does correct some of the problems of the ACM-SIGGRAPH proposal. The proliferation of different FORTRAN versions cannot occur because GKS specifies the calling sequences to be used. Bindings for other languages are also available. By only specifying a two-dimensional system, the world coordinate system to screen transformations are much simpler. However, a three-dimensional extension is promised for the future.

A major problem is that the FORTRAN-77 binding calls for a total of 212 subroutines. Even the lowest level of the standard, level ma, consists of 56 subroutines. I would estimate that an active user would have to be very aware of at least 100 subroutines. Such a system cannot be easy to learn and is probably only appropriate for people who intend to make a career of computer graphics. Also, items like color and line structure may be selected by integer arguments in these subroutines. For these reasons, programs written using GKS can become unreadable unless the programmer takes special care to keep this from happening. In spite of this being a "standard", there can be severe transportability problems between different implementations of GKS. The standard simply contains too many items that are "implementation dependent". Another problem is that there are minor differences between the ISO version of GKS and the ANSI version.

Nevertheless, since GKS is an American and international standard, it cannot be ignored. People and installations will be forced to move to it, or become isolated from the rest of the world. However, I believe a subset of GKS along with additional subroutines that call GKS subroutines can be put together. Such a group of subroutines could be easy to learn and use, highly transportable, and alleviate the readability problem. A proposal for such a project was made in [Bea87] and carried out in [Bea92].
This section contains a list of all of the publications that have been referenced in this document.


REFERENCES


October 1988, and April 1993).


[Ber69] G. M. Berns, Description of FORMAT, a Text-Processing Program, Communications of the Association for Computing Machinery, Volume 12, Number 3 (March 1969), pages 141-146.


REFERENCES


[IBM--] Numerical Surface Techniques and Contour Map Plotting, International Business Machines Corporation, Form Number E20-0117 (undated).


REFERENCES


REFERENCES


