Chapter 19
Graphics Toolkit

Overview

These words were designed to give a simple starting point for interfacing to the Amiga graphics library. The words are fairly generic. An application written using them can (and has been) ported to or from other machines with different graphics environments.

If need be, you can also access any of the advanced features of the Amiga by using the CALL and :STRUCT facilities. So while these GR words do not access all of the Amiga facilities, you are not restricted from doing so. By looking at the source code, you will see how to extend this toolkit. You should also look at the Chapter on miscellaneous Amiga tools for more graphics words. For information on using the Amiga libraries, please refer to the *Amiga ROM Kernel Reference Manual*.

The majority of the demos in the JD: directory use this GR system so please refer to them for more examples of its use.

Graphics Tutorial

These words are designed to work with the concept of a current rastport. All drawing commands will act on that rastport. To distinguish these words from other Forth words all of these routines will be prefaced with a 'GR' for GRaphics.

To demonstrate how this system works, let's open a window and do some drawing. The graphics subsystem is not normally loaded in the JForth dictionary. To load this code, enter:

```
INCLUDE? GR.INIT JU:AMIGA_GRAPH
```

The above will load the graphics code from the file AMIGA_GRAPH only if GR.INIT has not already been defined. Before any use is made of the graphics system, we should make sure the graphics system is initialized properly. To do this enter:

```
GR.INIT
```

This sets all of the variables to their proper state and opens the Graphics and Intuition libraries. Now we need to load the information needed to talk to the Amiga libraries. This normally resides in the ".J" include files. These files contain the definitions of structures and constants. Structures are simply a collection of variable organized in a specific way. This organization is defined in the include files. JForth provides the most commonly needed information in precompiled modules. To load the main module, enter:

```
GETMODULE INCLUDES    ( link in precompiled includes )
```

Now we should open a window for drawing. First we need to declare a NewWindow structure. This is a template that gives Intuition some information about the window we desire including its width, height, location, input event flags, etc. This structure will occupy some memory space in the JForth dictionary. We refer to structures by the address of their first byte. Every byte in the computer has an address which is simply a number which uniquely identifies it. When we say the we are "passing a structure" or "passing a pointer to a structure" we mean that we are passing the address of the structure. (Pointer means address.)

JForth provides a word called NEWWINDOW.SETUP that will set the values in a NewWindow structure to some reasonable values.

```
NEWWINDOW MY-WINDOW       ( define a structure )
```

```
MY-WINDOW NEWWINDOW.SETUP ( set default values in structure)
```

We can use FILE? to see how the NewWindow structure is defined. Enter:

```
FILE?  NEWWINDOW
```

and hit 'Y' when asked if you want to see the source code. Let's examine and change some of the default settings. Enter:

```
NEWWINDOW .  ( print address of first byte just for fun )
NEWWINDOW S@ NW_WIDTH .  ( set by NEWWINDOW.SETUP )
500 NEWWINDOW S! NW_WIDTH  ( let's change it to 500 )
NEWWINDOW S@ NW_WIDTH .  ( did it work )
```

The 500 in the above example was the width of the window in pixels or video dots. We can now pass that structure to the Amiga Intuition Library asking for a window.

```
MY-WINDOW GR.OPENCURW  .  ( open the window )
```

Notice that a window has been opened. You may also have noticed that a value was left on the stack That was the window pointer. You should always check to make sure that this pointer is non-zero. If it is zero, it means that the window did **not** open and your program will probably crash if you pretend that it did.

To draw in this window we can set the current color, and then issue move and draw commands. These commands are based on using an imaginary color pen. To pick up the pen and reposition it you use GR.MOVE . To put the pen down and drag it in a straight line you use GR.DRAW .

```
2 GR.COLOR!
30 30 GR.MOVE
200 55 GR.DRAW   ( draw a line from  30,30 to 200,55 )
300 100 GR.DRAW  ( draw a line from 200,55 to 300,100 )
```

Now let's try drawing a filled rectangle in another color.

```
1 GR.COLOR!
20 20 100 120 GR.RECT   ( draw filled rectangle )
```

We can also output a text message, try entering:

```
3 GR.COLOR!
250 60 " Hello!" GR.XYTEXT
```

The Amiga uses numbers to reference different colors. The Amiga video circuitry displays a picture by reading these numbers from memory and converting them to an actual color. It uses a *color table*, or *pallette*, to figure out what color should be displayed for a given number. These colors can be changed using the Preferences program described in your Amiga user manual.

To finish this session we should close the window and terminate the graphics system.

```
GR.CLOSECURW
GR.TERM
```

A Simple Graphics Program

Let's experiment with defining a Forth word that draws random vectors. (*Vector* is the graphics industry word for a single line segment.) We can use the Forth word CHOOSE which selects a random value between 0 and whatever value is on the stack. You may want to enter these in a file so that you can modify the program when done.

```
\ Load necessary code.
INCLUDE? GR.INIT JU:AMIGA_GRAPH
INCLUDE? ?CLOSEBOX JU:AMIGA_EVENTS
```

```
INCLUDE? CHOOSE JU:RANDOM

ANEW TASK-RANDOM_LINES

NEWWINDOW  MYNW
: DRAW.RAND  ( -- , draw a random vector )
   300 CHOOSE   ( generate random X )
   120 CHOOSE   ( generate random Y )
   GR.DRAW      ( draw line )
;

: MANY.RAND ( -- , draw many random vectors )
   GR.INIT
   MYNW NEWWINDOW.SETUP
   MYNW  GR.OPENCURW
   IF  ( Check to make sure window opened !!!!)
      BEGIN
         DRAW.RAND
         ?CLOSEBOX   ( has closebox been hit )
      UNTIL
      GR.CLOSECURW
   THEN
   GR.TERM
;
```

?CLOSEBOX will return a TRUE if the closebox is ever hit. This gives you a way out of the program.

The word GR.OPENTEST gives you an easy way to open a window for testing. It will automatically ABORT if the window did not open.

Extending the Graphics Toolbox

You may want to add new words that act on the "current RastPort". If so, just remember that the RastPort is stored in ABSOLUTE mode to save us having to convert it before calling the Amiga Libraries. Here is an example of a new routine to draw a single pixel in the current RastPort.

```
: GR.POINT  ( x y -- , draw single point )
    GR-CURRPORT @ -ROT  ( -- rp x y )
    CALLVOID GRAPHICS_LIB WritePixel
;
```

Generic Graphics Glossary

There are three main types of routines in this toolkit. The Control Routines initialize and terminate data structures, and are involved with opening and closing windows. The Output Primitives produce actual graphics output in the current rastport. The Output Attribute involve the appearance, colors, modes, etc. of the Output Primitives.

Control Routines

GR.INIT (-- , Initialize the Graphics Subsystem)

Set attributes to their default values. It also opens the Graphics and Intuition libraries. This word

MUST be called before using any of the other words. Calling this routine twice will NOT result in an error.

GR.TERM (-- , Terminate the Graphics Subsystem)

These words are Amiga specific words for setting up NewWindow structures and opening windows.

GR.CLOSECURW (-- , Close CURRENT window if open)

This command looks in the variable GR-CURWINDOW for a window pointer and closes it if one is there. It then clears GR-CURWINDOW and GR-CURRPORT.

GR.CLOSEWINDOW (Window -- , Close an Intuition Window)

If this window is the same as the window stored in the variable GR-CURWINDOW then both GR-CURRPORT and GR-CURWINDOW will be cleared. This is to prevent a window from being closed twice, a fatal error, when working with multiple open windows.

GR-CURWINDOW (-- addr , Variable with rel addr of Window)

GR-CURRPORT (-- addr , Variable with abs addr of RastPort)

This variable contains the ABSOLUTE address of the Rastport. This is used instead of the relative address because most of the Amiga graphics routines require an absolute rastport address. See >REL.

GR.OPENCURW (NewWindow -- Window | 0)

Open a window based on the requested values set in the NewWindow structure. Returns the relative address of a window structure or 0 if it could not open. Be sure to check this return value. If a window does open, this word calls GR.SET.CURWINDOW to make this the current window for drawing.

GR.OPENTEST (-- , Open a window for experimentation.)

This opens a window for testing. It can be used in place of the code in the first example. It uses a NEWWINDOW structure called WINDOWTEMPLATE . This structure can be reused by other programs.

GR.SET.CURWINDOW (Window -- , Sets the current Window)

This sets the current Rastport to this windows Rastport. Subsequent drawing operations, therefore, will take place in this window. If you are using multiple windows, you should save your own window pointers to each of them. You can call this word to determine which of your windows will be drawn into by the GR.xxx routines.

The variables GR-CURRPORT and GR-CURWINDOW are set by this routine.

NEWWINDOW (<name> --INPUT-- , define a NewWindow structure)

NEWWINDOW.SETUP (newwindow -- , set defaults for a new window)

This loads a request structure for a 640 by 200 window that will open in the Workbench screen. It will have a Close gadget and a Sizing gadget. You can modify any of these defaults before calling GR.OPENWINDOW to get different kinds of windows.

Output Primitives

GR.CLEAR (-- , Clear the current drawing surface.)

The rectangle will be based on the current windows drawing surface.

GR.DEHIGHLIGHT (x1 y1 x2 y2 -- , DEHighlight a Rect Region)

Reverse the effect of GR.HIGHLIGHT

GR.DRAW (xpix ypix -- , Draw a line.)

This will draw a line from the current position to xpix, ypix using the current attributes.

GR.FONT! (font -- , calls SetFont() for current RastPort)

See the file JD:DEMO_FONTS for an example.

GR.FONT@ (-- font , get font from current RastPort)

GR.HIGHLIGHT (x1 y1 x2 y2 -- , Highlight a Rectangular Region)

This will highlight a region to bring attention to it. On the Amiga this will be done by XORing with color = 3.

GR.MOVE (xpix ypix -- , Move the current position to xpix, ypix)

GR.NUMBER (n -- , Display number at current position.)

If you need special formatting, you can format the number separately and draw using GR.TYPE .

GR.RECT (x1 y1 x2 y2 -- , Draw a rectangle)

This will draw a rectangle using the current color. The current position will be left at x1, y1.

Warning! Make sure that X2 >= X1 and that Y2 >= Y1. Otherwise the Amiga library routine will overwrite a huge part of chip memory and you will crash. See JD:DEMO_BOXES.

GR.TEXT ($string -- , Draw a text string)

Draw text at the current position. The string must have a byte count at the given address. The current position will be left at the end of the text.

```
" Hello" GR.TEXT
```

GR.TEXTLEN (addr count -- xpixels , x size of string)

Returns length of string in pixels if drawn in current font. This can be used to right justify text by moving XPIXELS to the left of your right margin and drawing from there.

```
100 \ right margin
" Hello" COUNT GR.TEXTLEN -  \ calc x position
50 ( -- x y ) GR.MOVE  \ to start
" Hello" GR.TEXT  \ will end at x=100
```

GR.TYPE (addr count -- , Draw text, like Forth TYPE)

GR.XYTEXT (xpix ypix string -- , Draw a text string)

Draw text at the given position. This is essentially GR.TEXT that does a move first.

Output Attributes

The appearance of these output primitives can be controlled by the setting of output attributes. These attributes remain in effect until changed. The setting words are balanced by query words so that an environment can be saved, changed, and then restored by low level code. The setting words end in ! and the query words end in @ .

GR.COLOR! (color-index -- , Set the color for drawing.)

A color-index is a number that references a color defined in the current pallette. See

JD:DEMO_RGB for an instructive example.

```
3 GR.COLOR!
```

GR.BCOLOR! (color-index -- , Set the background color.)

The background color is used when clearing the screen and for filling in around text.

GR.MODE! (mode -- , Set the drawing mode.)

This controls the logic mode that is used to modify the pixels when drawing. You can use the Amiga constants JAM1 , JAM2 , and COMPLEMENT. We have also defined two constants GR_INSERT_MODE and GR_XOR_MODE to promote portability. There is an official bug in the Amiga Library that forces the color to index 3 when in COMPLEMENT mode.

GR.COLOR@ (-- color-index , Fetch the color for drawing.)

GR.BCOLOR@ (-- color-index , Fetch the background color.)

GR.MODE@ (-- mode , Fetch the drawing mode.)

Graphics Input

In event driven systems, all input events should be routed through a top level routine that can handle any event that is generated. These include mouse movement, button presses, menu picks, window close box hits, etc. The type of events generated and the way that they are handled is different in every application. For this reason, we have not included a routine that only handles mouse location input. For information on how to access input events, see the documentation on the EV routines for event processing. You can also examine the DEMO_PAINT file for an example of graphics input.

Event Driven Programming.

A new style of programming is evolving to meet the needs of highly interactive systems. In modern user interfaces, the user is normally free to use any input that the program offers. These might include picking from menus, moving windows around, entering graphic information, hitting the keyboard, or poking at other gadgets on the screen. The program must be ready to respond to any of these input events. A typical program is structured with a loop at the top of the program that gets input events and processes them with a case statement. The loop is exited when, for example, a CLOSEBOX is hit, or QUIT is selected from a menu. Since the user's input events control the flow of the program, this style is referred to as 'Event Driven Programming'.

The Amiga provides support for this style of programming. When a window is opened, the programmer can select which types of events can be generated by setting the IDCMP flags. Messages can then be received from that window that contain one of the selected events or a NULL event if nothing happened. Please refer to the Intuition manual, Chapter 8, that discusses IDCMP and IntuiMessages for details.

Please also examine the files JD:DEMO_MENUS and JD:DEMO_PAINT for examples of how to use this system.

Routines in JU:AMIGA_EVENTS - EV.xxxx

This part of the system normally has to be developed from scratch to meet the needs of individual programs. We have included some simple routines, however, to get people started.

?CLOSEBOX (-- flag)

Checks for a CLOSEBOX hit in GR-CURWINDOW. This is handy if the only kinds of events you want are CLOSEBOX events. All other classes of events are lost. Used in the demos.

EV.2CLICK? (-- flag)

Returns TRUE if last mouse click was the second click of a double click. See JD:DEMO_CLICK

EV.FLUSH (-- , flush all events from queue)

This does EV.GETCLASS with GR-CURWINDOW until a NULL event is received.

EV.GETCLASS (window -- class , get message class or NULL)

This word calls GET.PORT.MSG on the USERPORT associated with the window. If an input event has occurred, the relevant information is extracted and placed in variables for easy access. See PARSE.PORT.MSG below. The message is then replied to using ReplyMsg.

EV.GETXY (-- x y , get last reported mouse position)

This just fetches the values in EV-LAST-MOUSEX and EV-LAST-MOUSEY .

EV.GETXY00 (-- x y , gets x,y corrected for GIMMEZEROZERO)

This can be used if you are getting x,y from a GIMMEZEROZERO window. It corrects for the border of the window.

EV.WAIT (window -- , wait for message, leave in queue)

If you are just waiting for an event, you should call this instead of sitting in a polling loop. Otherwise you will eat up all the CPU time and leave very little for other tasks. When it returns, you can call EV.GETCLASS.

GET.PORT.MSG (port -- class | 0)

This is called by EV.GETCLASS.

PARSE.PORT.MSG (message -- , extract info, place in variables)

The variables are as follows:

```
EV-LAST-CODE
EV-LAST-IADDRESS
EV-LAST-MOUSEX     EV-LAST-MOUSEY
EV-LAST-MICROS     EV-PREV-MICROS
```

This code is in the file JU:AMIGA_EVENTS.