1. Introduction

This manual is for the programmer interested in developing an application on the Pixel-planes 4 system. It will help the reader decide whether the Pixel-planes 4 Prototype will fulfill the needs of his application, and if so, will help him implement that application. The reader is assumed to have a basic knowledge of computer graphics on a raster-graphics frame buffer and a working knowledge of the C programming language.

1.1 Example Pixel-planes Algorithm

The Pixel-planes polygon algorithm provides an excellent introduction to using the Pixel-planes Frame Buffer. In this algorithm, a polygon is rendered in three stages: scan conversion, hidden surface removal, and shading.

Scan conversion determines which pixels comprise the polygon. A polygon is scan-converted in the Pixel-planes system by sending a bilinear expression (Ax+By+C) to the Frame Buffer which evaluates to zero on the edge of the polygon and positive within the polygon. For each edge, pixels turn themselves 'off' if the bilinear expression evaluates negative. After each edge has been defined in this manner, the polygon has been scan converted -- only the pixels within the polygon remain 'on'.

Hidden surface removal is performed next by sending the Frame Buffer a bilinear expression describing the 'z' plane of the polygon. Each pixel compares its new z value with its current z value; pixels for which the polygon is hidden turn themselves off. The new z plane is loaded at the remaining pixels in the bits of pixel-memory designated for the z buffer. At this point, only visible pixels of the polygon remain turned on.

To **shade** the polygon, bilinear expressions describing red, green, and blue are loaded at the remaining (visible) pixels of the polygon. This allows the polygon's color to be linearly interpolated across it's vertices, for an extremely efficient implementation of the Gouraud polygon shading algorithm.

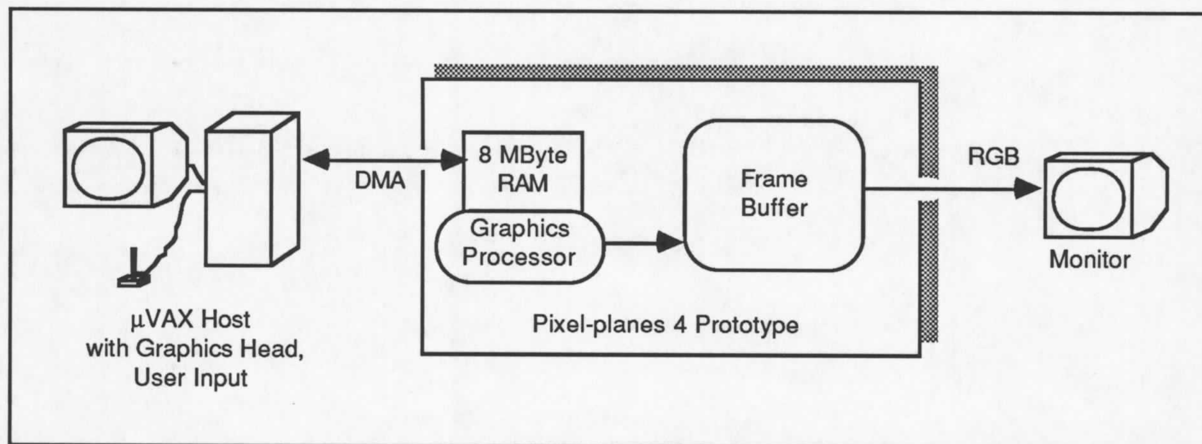
See the references section at the end of this chapter for more detailed descriptions of Pixel-planes algorithms.

1.2 Polygon Library

The polygon algorithm described above has been implemented in a ready to use package, along with spheres, stereo, and anti-aliasing algorithms. See *Intro(3p)* in the *Pxpl4 Programmer's Reference* for details. The remainder of this manual is concerned with developing custom applications.

1.3 Pixel-planes 4 System Overview

In developing a custom application on Pixel-planes, the programmer will work with the three major components of the system: the Pixel-planes Frame Buffer, the Graphics Processor (GP), and the MicroVAX workstation Host (hostname: pxpl4).



Pixel-planes Overview

The Pixel-planes Frame Buffer is the rendering engine of the system. Provided with linear equations describing graphics primitives, the Frame Buffer renders them into 3-D graphics scenes and displays them on a conventional color monitor.

The Frame Buffer is a Single-Instruction Multiple-Datapath (SIMD) machine with a 512x512 array of *pixel processors* which execute instructions in parallel. Each pixel processor consists of an Arithmetic and Logic Unit (ALU) and 72 bits of *pixel memory*. Data arrives at the pixels from the leaves of a bilinear expression evaluation tree in the form of bilinear equations in x and y (Ax + By + C), where the Graphics Processor provides the linear coefficients A, B, and C, and x and y are the pixels' screen coordinates. The pixels operate on data from the tree and pixel memory, storing their results back in pixel memory. The Frame Buffer continuously feeds bytes 0-2 of pixel

memory (the video bytes) into three color lookup tables which drive the RGB inputs to the Pixel-planes monitor. Chapter 2 covers the Frame Buffer and its instruction set.

The Graphics Processor is a floating point engine based on the Weitek 8032 floating point chip set. The GP's 8 Megabytes of RAM is used to store graphics databases. The GP processes these databases in response to Host commands, writing instructions to the Frame Buffer via memory-mapped control registers.

The GP is easily programmed. The programmer compiles C code for the GP on the Host using a cross-compiler supplied and supported by Weitek. The entire UNIX math library has been ported to the GP, along with many C runtime library functions such as printf and malloc. Chapter 3 covers developing C code for the GP.

The Host is a Digital Equipment Corporation MicroVAX II running Ultrix. An application's user interface software runs on the Host and sends graphics commands to the Graphics Processor via a DMA facility and bidirectional interrupts. Host programs communicate with GP programs by reading and writing external variables declared in the GP program, as described in Chapter 3.

1.4 References

References useful to the programmer interested in developing an algorithm on Pixel-planes include

Fuchs, H., J. Goldfeather, J.P. Hultquist, S. Spach, J. Austin, F.P. Brooks, Jr., J. Eyles, and J. Poulton. "Fast Spheres, Textures, Transparencies, and Image Enhancements in Pixel-Planes" *Computer Graphics,* Vol. 19, No. 3. pp. 111-120, 1985. (Proceedings of SIGGRAPH '85)

Discusses the polygon algorithm of Section 1.2, along with shadows, spheres, and anti-aliasing. Also describes an implementation of Adaptive Histogram Equalization.

Goldfeather, J., J.P.M. Hultquist, and H. Fuchs. "Fast Constructive Solid Geometry Display in the Pixel-Powers Graphics System" *Computer Graphics,* Vol. 20, No. 4. pp. 107-116, 1986. (Proceedings of SIGGRAPH '86)

Describes a method for rendering CSG (Constructive Solid Geometry) objects directly on Pixel-planes.

The Host and GP library routines described in this manual are detailed with manual entries in the *Pixel-planes Programmer's Reference Manual*, available through the *pman* online documentation utility. Execute the command *pman pman* for an introduction.

The last chapter in this manual presents several brief code examples for the Host and GP.

2 The Pixel-planes Frame Buffer

The Frame Buffer is the rendering engine of the Pixel-planes system. Provided with bilinear expressions describing graphics primitives, the Frame Buffer renders them into 3-D graphics scenes and displays them on a conventional color monitor. This chapter describes the Frame Buffer, the interface by which the GP controls it, and the process by which the Host configures the Frame Buffer.

2.1 Frame Buffer Description

The Frame Buffer can be divided logically into two components: the bilinear expression evaluator (called the Tree) and the pixel processors.

The 'Tree' derives its nickname from its structure: it is a actually a pair of multiplier trees used to evaluate a bilinear expression Ax+By+C at each pixel in parallel. Here x and y are the screen coordinates of each pixel; and A,B, and C are constants, called the *linear coefficients*, which are the same for all pixels. The GP passes the linear coefficients to the Frame Buffer as floats. These values are *truncated* to fixed-radix numbers with 12 fractional bits, then the Tree computes the values of the bilinear expression at each pixel and *truncates* the results to two's complement integers.

Don't let the truncation of linear coefficients and tree results catch you unawares. For example, the linear expression x+y+1.99 will evaluate to the integer '1' at x=0, y=0. Also, since the linear coefficients are converted to fixed-radix form with 12 fractional bits, coefficients smaller in magnitude than 1.0 begin to suffer from loss of precision.

The array of pixel processors execute the same instruction stream in parallel. Pixels cannot communicate with each other; each can write results only into its own memory. Each pixel processor consists of the following:

Memory: 72 bits of random access memory. The programmer usually divides this memory into segments of various lengths, each of which contains an unsigned or two's complement integer. This segmentation is the same at each pixel.

ALU: (Arithmetic and Logic Unit) performs operations on memory segments, the enable register, the carry register, and tree results. Results are written into memory segments or the enable register (and the carry register is

set).

Enable Resister: Used to condition certain memory writes. In particular, arithmetic instructions write results into memory only at *enabled* pixels: those pixels where the enable register contains a '1'.

Carry Register: Contains the carry result of the last arithmetic operation, regardless of the state of the enable register when the arithmetic operation was performed. NO, NO

The pixel-ALU's execute the same instruction on each cycle, but memory writes on arithmetic operations are conditioned by each pixel's enable register. A pixel whose enable register has been cleared can be thought of as 'turned off.' For example, the standard polygon algorithm scan-converts a polygon by 'turning off' pixels not in that polygon before loading colors from the tree to shade that polygon.

2.2 Interface to the Frame Buffer

The Graphics Processor controls the Frame Buffer by writing to a set of memory-mapped control registers, including an instruction opcode register and three coefficient registers for the A,B,C Tree inputs. A GP program writes these registers by calling instruction macros (described in the next section) and the following macros for loading the bilinear coefficient registers:

```
#include <gpigc.h>
/* These three macros work on floats in the range -2^32 to +2^32   */
FB_A (x)        /* load A coefficient register with float x        */
FB_B (x)        /* load B coefficient register with float x        */
FB_C (x)        /* load C coefficient register with float x        */

/* These macros work on positive integers in the range 0 to 2^23-1 */
FB_IA (x)       /* load A coefficient register with integer x      */
FB_IB (x)       /* load B coefficient register with integer x      */
FB_IC (x)       /* load C coefficient register with integer x      */
```

The integer macros, FB_IA, FB_IB, and FB_IC, cannot be used for instructions for which the sign-bit of the Tree result is used, and since the values are only 23 bits unsigned, they also cannot be used for instructions which use more than 23 bits of the Tree result.

The following GP code fragment causes the Frame Buffer to evaluate the bilinear expression

x+2y-3, and places the result in 8 bits of memory beginning at bit 24, at each enabled pixel.

```
#include <gpigc.h>
#include <commands.h>

FB_A  (1.0);        /* write 'A' register                          */
FB_B (2.0);         /* write 'B' register                          */
FB_C (-3.0);        /* write 'C' register                          */
FB_LOAD(24, 8);     /* send 'load' instruction --  this instruction */
                    /* is conditioned by the enable register        */
```

The first three macro calls load the coefficient registers on the Frame Buffer. These registers are not modified by executing Frame Buffer instructions, so if one of these coefficients were the same as the last one loaded, the GP would not need to reload it for this instruction. The last macro call loads the tree result into pixel memory at enabled pixels.

2.3 Frame Buffer Instruction Set

This section describes the instruction macros for controlling the Frame Buffer. See the frame buffer microcode source in *~pxpl/src/libpxgp/igc/ucode.txt* for complete details.

Integers defined in pixel memory have their LSB at their lowest address. Memory segments are identified with the notation **mem[lsb, len]**. Thus an 8 bit memory segment at bits 24 through 31 (with its LSB at bit 24) is denoted **mem[24 : 8]**. Contents of memory segments may represent unsigned or two's-complement signed integers. For many instructions, it does not matter if the contents of the memory segment are considered to be signed or unsigned; for others it does, and these are noted.

The tree result is always two's complement signed and is always truncated to integer form; negative values are truncated downwards, rather than towards zero. The tree result may either be fully computed to its sign-bit (identified with the notation **tree**), or only some fixed number of bits of the tree result may be computed (identified with the notation **tree[n]**); when **tree[n]** is specified the tree result is always sign-extended to n bits, regardless of its actual length. The integer coefficient macros (FB_IA, FB_IB, and FB_IC) must not be used for instructions which use **tree** or which use **tree[n]** for **n** > 23.

The instructions can be divided into four categories: instructions to modify the enable register, instructions to store the enable register, arithmetic instructions, and miscellaneous

instructions. There is also an initialization macro, described below.

Initialization Macro

Any function which calls FB macros must first call the initialization macro **FB_INIT()** before the first statement of the function.

instruction	*synopsis*
FB_INIT( )	initializes pointers for the FB macros

Configuration Macro

When the floating point coefficients are converted to fixed-point form, some number of fractional bits are generated. This number is 12 by default. However, it can be set in the range 0-15, on-the-fly, using the macro:

instruction	*synopsis*
FB_FBITS(fbits)	changes number of fractional bits to fbits, 0 <= fbits <= 16.

This macro must be used after one instruction opcode is loaded, and before any of the Coefficient Registers are loaded for the next instruction. Also, after and **FB_BITS** macro, the Coefficient Registers must be reloaded even if previous coefficient values are to be re-used.

The user may decrease fbits to speed up an application or increase fbits to provide greater precision. Do not forget that the coeficients are truncated to fbits fractional bits, and therefore precision can be lost, depending on the magnitude of the coefficient and the setting for fbits.

Enable Instructions

Enable instructions alter the contents of the enable register. Remember that instructions which use the tree result (denoted **tree** below) must load the Coefficient Registers before executing the instruction.

instruction		*synopsis*			
FB_CLRENABS	( )	enable	=	0	
FB_SETENABS	( )	enable	=	1	
FB_ENABINV	()	enable	=	! enable	
FB_MEMintoENAB	(src)	enable	=	mem [src : 1]	
FB_TREEgeZERO	( )	enable	&&=	(tree >= 0)	
FB_FEDGE	( )	enable	=	(tree >= 0)	
FB_SEDGE	( )	enable	=	(mem[src:1] && (tree >= 0)	
FB_EDGE2	( )	enable	&&=	(tree >= 0)	
		carry	&&=	(tree < 0)	
FB_STRIPEDGE	(src, dst)	enable	&&=	(tree >= 0)	
		mem[dst:1]	=	mem[src:1] && (tree < 0)	10
FB_TREEeqZERO	( )	enable	&&=	(tree == 0)	
FB_MESH	(n)	enable	&&=	(tree[n] == 0)	
FB_GRID	(n)	enable	&&=	(~tree[n] == 0)	
FB_FTECT	( )	enable	=	((tree[1]) == 1)	
FB_MEMeqZERO	(src, len)	enable	&&=	(mem[src : len] == 0)	
FB_MEMneZERO	(src, len)	enable	&&=	(mem[src : len] != 0)	
FB_MEMeqMEM	(dst, src, len)	enable	&&=	(mem [dst : len] == mem [src : len])	
FB_MEMgeMEM	(dst, src, len)	enable	&&=	(mem [dst : len] >= mem [src : len])	1
FB_MEMgtMEM	(dst, src, len)	enable	&&=	(mem [dst : len] > mem [src : len])	1
FB_MEM2geMEM2	(dst, src, len)	enable	&&=	(mem [dst : len] >= mem [src : len])	2
FB_MEM2gtMEM2	(dst, src, len)	enable	&&=	(mem [dst : len] > mem [src : len])	2
FB_MEMeqTREE	(src, len)	enable	&&=	(mem[src : len] == tree[len])	
FB_MEMleTREE	(src, len)	enable	&&=	(mem [src : len] <= tree)	3
FB_MEMltTREE	(src, len)	enable	&&=	(mem [src : len] < tree)	3
FB_MEMgeTREE	(src, len)	enable	&&=	(mem [src : len] >= tree)	3
FB_MEMgtTREE	(src, len)	enable	&&=	(mem [src : len] > tree)	3
FB_FCMEMA	(src, len)	enable	=	(mem [src : len] <= tree)	3
FB_SCMEMA	(src, len, src1)	enable	=	mem[src1:1] && (mem [src:len]<=tree)	3
FB_ENABandeqMEM	(src)	enable	&&=	mem[src:1]	

FB_ENABandeqMEMBAR	(src)	enable	&&=	~mem[src:1]
FB_ENABoreqMEM	(src)	enable	||=	mem[src:1]
FB_ENABxoreqMEM	(src)	enable	^^=	mem[src:1]
FB_ENABoreqCRY	()	enable	||=	carry

Enable Store Instructions

These instructions are used to store the enable register into memory: These writes occur regardless of the contents of the enable register.

instruction		*synopsis*		
FB_ENABintoMEM	(dst)	mem [dst: 1]	=	enable;
FB_MEMoreqENAB	(dst)	mem [dst: 1]	||=	enable;
FB_MEMandeqENAB	(dst)	mem [dst: 1]	&&=	enable;
FB_ENABintoCRY	( )	carry	=	enable;

Arithmetic Instructions

These arithmetic instructions operate on signed tree results and signed or unsigned integers in pixel memory segments.

These instructions all write their integer results into **mem[dst : dlen]** (which represents **dlen** bits of pixel memory with LSB at address **dst**). Addition overflow and subtraction underflow are truncated. Some instructions have a separate memory source as well as destination, in which case the source and destination must either be the same memory segment, or non-overlapping. The following rules apply to instructions which have separate length arguments for the source and destination (slen and dlen respectively):

dlen	==	slen	:	overflow is discarded
dlen	>	slen	:	carry/borrow is rippled through all bits of destination source may be considered unsigned or signed
dlen	<	slen	:	higher order bits of source are ignored

Segment lengths must all be at least one (or two, if noted), and memory segments must be contained within the 72 bits of pixel memory.

The arithmetic instruction writes are all conditioned by the enable register, though the ALU actually carries out the instruction at each pixel, affecting the carry register in the process.

instruction		*synopsis*			
FB_CLEAR	(dst, dlen)	mem [dst : dlen]	=	0	
FB_SET	(dst, dlen)	mem [dst : dlen]	=	~0	
FB_LOAD	(dst, dlen)	mem [dst : dlen]	=	tree[dlen];	
FB_CPY	(dst, src, dlen)	mem [dst : dlen]	=	mem [src : dlen]	
FB_SWAP	(dst, src, dlen)	mem [dst : dlen]	<—>	mem [src : dlen]	6
FB_INC	(dst, src, dlen)	mem [dst : dlen]	+=	1;	
FB_DEC	(dst, src, dlen)	mem [dst : dlen]	-=	1;	
FB_SHIFTL	(dst, src, dlen, n)	mem [dst : dlen]	=	mem [src : dlen] << n	4
FB_SHIFTR	(dst, src, dlen, slen, n)	mem [dst : dlen]	=	mem [src : slen] >> n	5
FB_INVERT	(dst, src, dlen)	mem [dst : dlen]	=	~mem [src : dlen]	
FB_NEGATE	(dst, src, dlen)	mem [dst : dlen]	=	- mem [src : dlen]	2
FB_MEMpluseqMEM	(dst, src, dlen, slen)	mem [dst : dlen]	+=	mem [src : slen]	7
FB_MEMminuseqMEM	(dst, src, dlen, slen)	mem [dst : dlen]	-=	mem [src : slen]	7
FB_MEMpluseqMEM2	(dst, src, dlen, slen)	mem [dst : dlen]	+=	mem [src : slen]	8
FB_MEMminuseqMEM2	(dst, src, dlen, slen)	mem [dst : dlen]	-=	mem [src : slen]	8
FB_MEMandeqMEM	(dst, src, dlen)	mem [dst : dlen]	&=	mem [src : dlen]	
FB_MEMoreqMEM	(dst, src, dlen)	mem [dst : dlen]	|=	mem [src : dlen]	
FB_MEMxoreqMEM	(dst, src, dlen)	mem [dst : dlen]	^=	mem [src : dlen]	
FB_MEMpluseqTREE	(dst, src, dlen)	mem [dst : dlen]	=	mem [src : dlen] + tree[dlen]	
FB_TREEminusMEM	(dst, src, dlen)	mem [dst : dlen]	=	tree[dlen] - mem [src : dlen]	
FB_MEMandTREE	(dst, src, dlen)	mem [dst : dlen]	=	mem [src : dlen] & tree[dlen]	
FB_MEMorTREE	(dst, src, dlen)	mem [dst : dlen]	=	mem [src : dlen] | tree[dlen]	
FB_MEMxorTREE	(dst, src, dlen)	mem [dst : dlen]	=	mem [src : dlen] ^ tree[dlen]	
FB_CRYintoMEM	(dst)	mem [dst : 1]	=	carry	
FB_OVFIX	(dst, len, tmp)	if (carry) mem [dst : len]	=	~0	

Notes:

1) The contents of the memory segments are assumed to represent unsigned integers.

2) The contents of the memory segment(s) are assumed to represent two's-complement signed integers.

3) The contents of the memory segment is assumed to represent an unsigned integer, and it is zero-extended to the full length of the tree result.

4) These restrictions apply: dlen >= 2, 0 < n < dlen

5) These restrictions apply: dlen >= 2, n > slen - dlen, 0 <= n < slen. The contents of the memory segment are assumed to represent an unsigned integer, so no sign-extension is done.

6) The contents of the two memory segemnts are interchanged.

7) The contents of the 'src' memory segment is assumed to represent an unsigned integers; it is zero-extended if dlen > slen. These restrictins apply: dlen >= 2, slen >= 2.

8) The contents of the 'src' memory segment is assumed to represent a signed integers; it is sign-extended if dlen > slen. These restrictions apply: dlen >= 2, slen >= 2.

9) 'Tmp' is a memory location for temporary use. Its contents are destroyed.

10) Mem[dst:1] is written for all pixels, regardless of the value of the enable register. Dst and src may point to the same memory location.

Miscellaneous Instructions

None of the following commands are affected by the contents of the enable register.

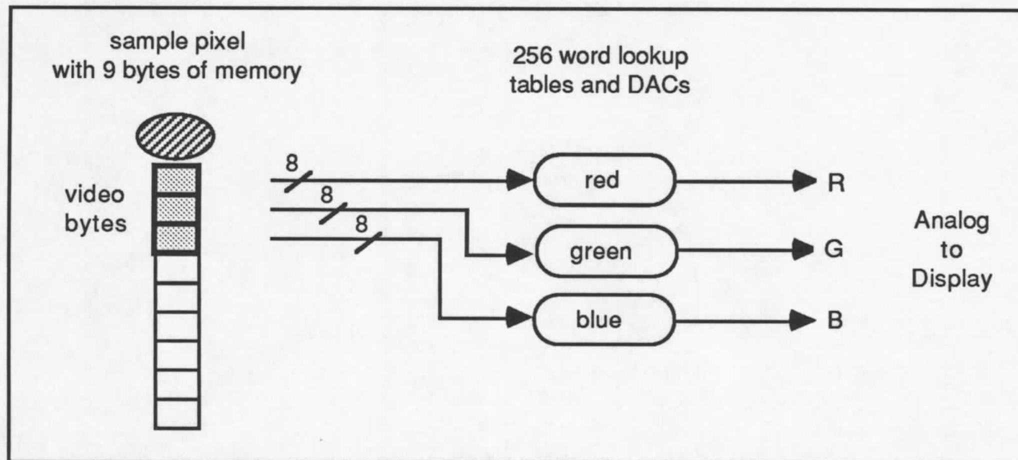
instruction	*synopsis*
FB_NOOP ( )	/* no operation */
FB_NOOP2 ( )	/* no operation (causes supplementary opcode to be sent) */
FB_HANG ( )	/* hangs the IGC in a tight loop (for debugging purposes) */
FB_VRWAIT ( )	/* wait until the monitor is in video retrace (between frames). */
	/* this is used to prevent video 'tear' before copying a new */
	/* image into the video bytes. See section below on */
	/* double buffering images */
FB_FRWAIT (count)	/* wait *count* video frames before proceding. */
	/* this is useful for timing purposes. */
FB_REFRESH (dst, dlen)	/* refresh the memory bits mem [dst : dlen], dlen >= 2 */
	/* explicit refresh of the video bytes may be required by some */
	/* memory intensive algorithms. If some lines of pixels seem */
	/* to be placed on several lines of the monitor, try refreshing */
	/* the video bytes with this command. */

Any **FB_VRWAIT** or **FB_FRWAIT** instruction must be followed by an instruction which does not use the Tree; to be safe, simply follow with an **FB_NOOP()**. Also, if the *count* argument to **FB_FRWAIT** is too large, the GP may generate a "hung igc" or "bus timeout" error.

2.4 Display via the Color Lookup Tables

The Frame Buffer can display its results on the display in two different modes: three-byte and one-byte. Since the two are very similar, the three-byte scheme is described in detail, followed by a brief description of the one-byte mode.

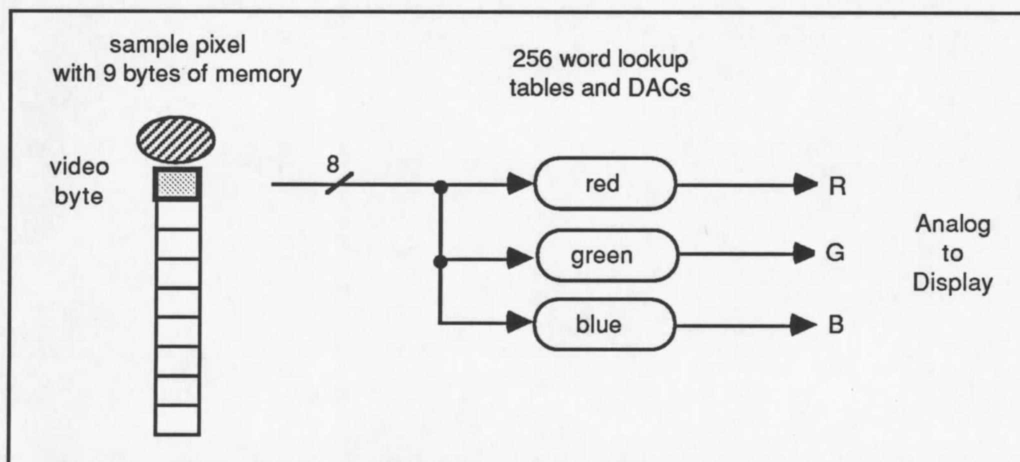
In the three-byte display scheme, the Frame Buffer continually reads the three low-order bytes of pixel memory into three separate color lookup tables -- one each for red, green, and blue.



Three-Byte Display Mode

The color lookup tables are 256 word tables which map the video data from the frame buffer video bytes into red, green, and blue levels for the pixels. The host library function *px_vcload,* given the parameter PX_THREEBYTE, initiates three-byte scanout and loads the color lookup tables with a gamma corrected color map. See the pman doc for *px_vcload*(3h). Non-default color maps can be loaded with subsequent calls to *px_loadcmaps*.

In one-byte display mode, the Frame Buffer drives all three lookup tables from the low-order byte of pixel memory, leaving the other two video bytes free for other uses, e.g., an integer segment storing a constant quadratic term for sphere rendering.



One-Byte Display Mode

This scheme gives the programmer a choice of 256 colors from the palette of 256^3 colors. The host library function *px_vcload,* given the parameter PX_ONEBYTE, initiates one-byte scanout and loads the color lookup tables with a simple hue-shade color map, with 2 bits of hue (red, green, blue, and white) and 6 bits of shade. See the pman doc for *px_vcload*(3h). Again, non-default color maps can be loaded with subsequent calls to *px_loadcmaps*.

2.5 Double Buffering Images

For double buffering, the Frame Buffer constructs each image in three bytes of pixel memory other than the video bytes, called the image bytes. When the image is complete, the Frame Buffer simply copies the image from the image bytes to the video bytes, and the monitor screen is updated with the new image. If bytes 3-5 are used as the image bytes, the Graphics Processor requests the double buffer copy with the Frame Buffer instructions

```
FB_VRWAIT ( );       /* wait for video retrace              */
FB_CPY (0, 24, 24);  /* copy image bytes into video bytes   */
```

The 'wait for video retrace' instruction is used here to prevent the double-buffer copy from creating a 'tear' on the display.

3 Developing an Application on the Host & GP

3.0 Introduction

This chapter describes how to develop an application on the Host - GP combination. The sections cover:

- **3.1** The programmer's environment.
- **3.2** The locations of libraries and include files.
- **3.3** Compiling and Linking GP & Host code.
- **3.4** Executing the GP program.
- **3.5** Transferring data between the Host and GP
- **3.6** Interrupting the Host from the GP
- **3.7** GP Memory Structure

If you read nothing else in this chapter, don't miss the floating point conversion discussion in section 3.5! Of course, all of the library functions mentioned here are documented in the *pman* facility.

3.1 Programmer's Environment

All Pixel-planes programmer applications (and quite a few user applications) reside in two directories, both of which should be in your path:

/usr/proj/pxpl/bin	local applications
/usr/proj/pxpl/xl/bin	GP compiler, linker, etc. from Weitek

In addition, the following environment variable assignments should be in your c-shell startup file (.cshrc):

setenv XLBIN	*/usr/proj/pxpl/xl/bin*
setenv XLDEFCHIP	*8032*

3.2 Library & Include File Locations

All Pixel-planes Host and GP libraries are in the directory */usr/proj/pxpl/lib*. Since this is not a UNIX standard location, these libraries must be specified with a full pathname when linking the GP or Host code. These libraries include:

libpxhost.a	(Host)	All host library functions covered in this document.
libpxgp.lib	(GP)	All GP library functions covered in this document.
libm.lib	(GP)	The GP math library. Includes all functions in the UNIX

		math library.
libc.lib	(GP)	The GP C runtime-support library. Includes printf, malloc, and free. This library is automatically linked in by the GP linker *gpln*.

All Pixel-planes include files are in the directory */usr/proj/pxpl/include*. In particular:

px_defs.h	Constants used in *libpxhost* and *libpxgp*.
gpigc.h	Frame Buffer initialization and coefficient loading macros.
commands.h	Frame Buffer command macros.

Since these include files are not in a standard UNIX location, try placing the option *-I/usr/proj/pxpl/include* in the command line of all compiler invocations, both for Host and GP code. This allows you to include the above files with the angle bracket notation, .e.g.

```
#include <px_defs.h>
```

3.3 Compiling & Linking

The Weitek C compiler supports portable C for the GP with the following types:

type	description
char	8 bits, signed or unsigned
short	16 bits, signed or unsigned
int	32 bits, signed or unsigned
long	32 bits, signed or unsigned
float	32 bits, IEEE format

Weitek C is a superset of the C supported under BSD UNIX. For a list of the minor additions, see the *ACP C Compiler User's Guide* in the *ACCEL Software Documentation.*Compile GP C Code with the Weitek compiler *acc:*

```
acc -c -I/usr/proj/pxpl/include foo.c
```

This creates the GP object file *foo.o* which can be linked with other GP objects and GP libraries into a Host object *gp.o* with the GP linker *gpln:*

```
gpln foo.o bar.o /usr/proj/pxpl/lib/libpxgp.lib
```

Gpln creates a Host object (default *gp.o*) which is linked with other Host objects and Host libraries to form a Pixel-planes application:

```
cc -o prog main.o gp.o /usr/proj/pxpl/lib/libpxhost.a
```

The *acc* and *gpln* options are very similar to those of the UNIX utilities *cc* and *ld*. Note that GP sources must have the suffix *.c*. For more details on *acc* command line options, see its chapter in the *ACCEL Software Documentation.*

The *gpln* utility is a shellscript interface to the ACCEL linker *aln*; most of its option flags are passed to *aln*. See the *pman* entry for *gpln*, and the entry for *aln* in the *ACCEL Software Documentation.*

3.4 Executing the GP Program

To begin GP execution in a Pixel-planes application, the Host program calls the library routine *px_gpexec:*

```
main()                  /* Host main routine */
{
        int gpargc;
        char gpargv[10][80];
        ...
        px_gpexec(gpargc, gpargv);          /* GP execution begins here */
        ...
}
```

GP execution begins with a call to *main(gpargc, gpargv)* in the GP code.

3.5 Transferring Data between Host and GP

A Host program transfers data to and from GP data memory by calling library routines which initiate DMA transfers between Host and GP memory. These routines do not return until the requested transfer is complete. The two routines are

```
px_gpwrite (gpaddress, hostaddress, nbytes);    /* write to GP memory     */
px_gpread  (gpaddress, hostaddress, nbytes);    /* read from GP memory    */

char   *gpaddress;        /* GP address (generic pointer)       */
char   *hostaddress;      /* Host address (generic pointer)     */
int    nbytes;            /* number of bytes to transfer        */
```

GP addresses can be bound at link time by referring to gp external variables in the following manner: a gp external variable *var* is referenced in host code as *gp_var*. For example, this fragment of Host code transfers four integers to the GP external integer array 'dest':

```
extern int gp_dest[];
...
px_gpwrite (gp_dest, intarray, 4 * sizeof (int) );
```

Buffers are transferred to and from the GP starting at the beginning of the buffer, in 16-bit transfers. This means that any variables which are used as synchronization flags should be 16-bit (short) integers.

The MicroVAX Host uses the DEC floating point standard, while the GP and the Frame Buffer use the IEEE floating point standard. Consequently, any floating point values transferred from the Host to the GP or back must be converted before use. The following library routines are provided for doing this in both directions on both machines:

```
px_DECtoIEEE (buf, num);        /* converts num floats in buf from DEC to IEEE */
px_IEEEtoDEC (buf, num);        /* converts num floats in buf from IEEE to DEC */
```

Note that large databases can often be stored on the Host's disk in IEEE format and transferred directly to the GP without conversion. These conversion routines do not always handle numerical exceptions correctly.

3.6 Interrupting Host from GP

The GP program can interrupt the Host program, allowing asynchronous operation on the two machines without the Host polling the GP. This facility is supported with the following two procedures:

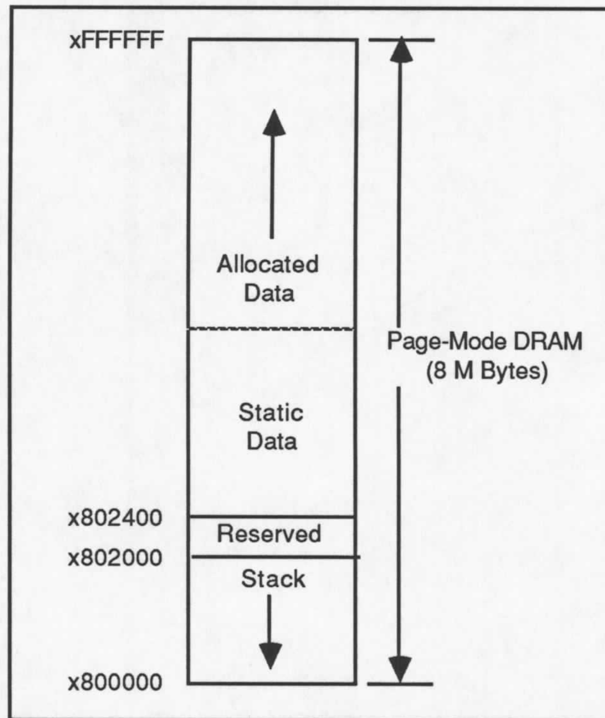
```
(On the GP)     px_hostint ( );     /* library routine to interrupt Host            */
(On the Host)   px_gpwaitint ( );   /* library routine to wait for GP interrupt     */
```

3.7 GP Memory Structure

The GP data memory is implemented with page-mode dynamic RAM. The programmer interested in writing optimal code should be aware of the efficiency considerations this imposes. The GP can normally write any Data RAM in two cycles and read it in one. However, A memory access with crosses a 8K byte page boundary will cause the GP to lose 2 cycles. For this reason, memory

should be accessed in a fashion which avoids frequent page changes.

The linker *gpln* allocates data memory as follows:



Graphics Processor Default Memory Map

The default stack size of 4K bytes can be easily modified-- see the *pman gpln* documentation. *Malloc* and *free* are supported on the GP, allocating space in the *allocated data* area.

4 Examples and Conclusion

This chapter presents several Pixel-planes examples. Study these examples, but don't type them in; they can be found on the Host in directories under */usr/proj/pxpl/src*.

4.1 First Example -- The Blazing Red Edge

The first example is a small GP program to draw a moving red edge. The sources are in the directory */usr/proj/pxpl/src/rededge*.

<u>GP Code</u>

The GP program runs in a continuous loop, drawing a single red edge which moves back and forth across the screen. The algorithm to draw an edge is simple: turn all the pixels on, set the video bits to black, turn the pixels on the back side of the edge 'off', and set the red byte of the remaining pixels to ones. Here is a listing of the file gp_main.c:

```
/* gp_main.c -- GP program to draw  moving red edge on          */
/*  Pixel-planes 4                                               */

#include <gpigc.h>
#include <commands.h>

main ()
{
        int pos = 0;            /*  gives position of edge on screen    */
        int dir = 1;            /* 1 if stripe moving right else -1     */

        FB_INIT();              /* initiliazation for the FB macros     */

        /*  program loops continuously  */
        while (1) {
                /* compute new edge position       */
                pos += dir;
                if (pos <= 0)           dir = 1;
                if (pos >= 511)         dir = -1;

                FB_SETENABS ();                 /* turn all pixels on           */
                FB_FRWAIT(1);                   /* wait until next video frame  */
                FB_CLEAR(0,24);                 /* clear  video bytes           */

                FB_A(1.0);                      /*  set A to 1.0                */
                FB_B(0.0);                      /*  set B to  0.0               */
                FB_C (-pos);                    /*  set C to -pos               */
                FB_TREEgeZERO  ( );             /* define edge   x - pos        */
                FB_SET(0,8);                    /* turn remaining pixels red    */
                                                /*   by setting all bits of red byte */
        }
}
```

Host Code

For this example, the host code is simple-- it loads the GP program and starts it up. Here is a listing of the file *main.c*:

```
/* Main.c -- Host program  host program simply boots GP     */

#include <px_defs.h>

main(argc, argv)
int argc;
char **argv;
{
        px_open();                      /* open the library                    */
        px_vcload(PX_THREEBYTE);        /* initiate frame buffer scan-out      */
        px_gpexec (argc, argv);         /* execute 'gp' on the GP              */
}
```

<u>Compilation and Linking</u>

Make sure your environment is set up as in section 3.1, then execute the following commands to compile, link, and execute *rededge:*

```
acc -c -I/usr/proj/pxpl/include gp_main.c               # compile gp_main.c
gpln gp_main.o                                          # link GP code into gp.o
cc -c -I/usr/proj/pxpl/include main.c                   # compile main.c
cc -o rededge main.o gp.o /usr/proj/pxpl/lib/libpxhost.a        # final link
rededge                                                 # execute!
```

4.2 Second Example -- Red Edge under Host Control

In the second example, the Host program determines the position of the edge and DMA's it to the GP. The sources are in the directory */usr/proj/pxpl/src/hostedge*.

<u>GP Code</u>

The GP program runs in a continuous loop, drawing a single red edge at the position indicated by the Host program. The Host DMA's a structure for each frame which has two fields: the edge position and a flag, always set to 1. The GP waits for the flag to become non-zero before reading the new edge position from the structure. The GP draws the edge, then notifies the Host that it is ready for a new one by sending an interrupt. Here are listings of the files *packet.h* and *gp_main.c:*

```
/* packet.h:  structure for DMA packet   */
struct packet {
        fload pos;     /* edge position */
        short flag;
}

/*  gp_main.c --  draws a red edge in position indicated by packet     */
/*       DMA'd from Host                                               */

#include <px_defs.h>
#include <gpigc.h>
#include <commands.h>
#include "packet.h"

struct packet packetslot;

main ()
{
        FB_INIT();
        /* program loops continuously */
        while(1) {
                /* clear flag in packet slot */
                packetslot.flag = 0;

                /* interrupt host to signal ready */
                px_hostint();

                /* wait for packet to arrive */
                while(packetslot.flag != 1)
                        ;

                /* draw edge */
                FB_SETENABS ();             /* turn all pixels on              */
                FB_FRWAIT(1);               /* wait until next video frame     */
                FB_CLEAR(0,24);             /* clear  video bytes              */

                FB_A(1);                    /*  set A to 1.0                   */
                FB_B(0);                    /*  set B to  0.0                  */
                FB_C (-packetslot.pos);     /*  set C to -pos                  */
                FB_TREEgeZERO  ( );         /* define edge   x - pos           */
                FB_SET(0,8);                /* turn remaining pixels red       */
                                            /*   by setting all bits of red byte  */
        }
}
```

Host Code

The Host code polls the keyboard and sends the appropriate DMA structure to the GP. Here is a listing of the file *main.c*:

```
/* Main.c -- host portion of program to poll keyboard and move red edge on
        on Pixel-planes accordingly.

        'h':    move edge left
        'l':    move edge right
*/

#include <px_defs.h>
#include <stdio.h>
#include "packet.h"

extern struct packet gp_packetslot;     /* slot for packet on GP            */

main(argc, argv)
int argc;
char **argv;
{
        float pos = 255.0;              /* computed position of edge        */
        struct packet packet;           /* DMA packet                       */
        char c;

        px_open();                      /* open the library                 */
        px_vcload(PX_THREEBYTE);        /* initiate frame buffer scan-out   */
        px_gpexec (argc, argv);         /* execute 'gp' on the GP           */

        /* main loop */
        while ((c = getchar()) != EOF) {
                switch (c) {
                case 'h':
                        pos -= 5.0; break;
                case 'l':
                        pos += 5.0; break;
                }

                /* Build DMA packet -- convert edge position to IEEE float */
                packet.pos = pos;
                px_DECtoIEEE(&packet.pos, 1);
                packet.flag = 1;

                /* Wait for GP to signal packet slot available */
                px_waitint();

                /* DMA packet to GP */
                px_gpwrite(&gp_packetslot, &packet, sizeof(struct packet));
        }
}
```

Compilation and Linking

Make sure your environment is set up as in section 3.1, then execute the following commands to compile, link, and execute *hostedge:*

```
acc -c -I/usr/proj/pxpl/include gp_main.c          # compile gp_main.c
gpln gp_main.o  /usr/proj/pxpl/lib/libpxgp.lib     # link GP code into gp.o
cc -c -I/usr/proj/pxpl/include main.c              # compile main.c
cc -o hostedge main.c gp.o /usr/proj/pxpl/lib/libpxhost.a    # final link
hostedge                                           # execute!
```

Note: you must hit the return key after a sequence of *h*'s and *l*'s before anything will happen.

4.3 Conclusion

The preceding examples provide a reasonable framework for developing a custom Pixel-planes application. With the exception of the Frame Buffer instructions, the concepts are all simple and should be quite familiar. Develop your code in a modular a fashion to allow individual testing of the modules which exercise the Frame Buffer.

When your application is completely debugged, you may wish to optimize some GP routines by rewriting them in assembler. The Weitek software development package includes a high level assembler for the patient and stout of heart. See its chapter in the *ACCEL Software Documentation.*