A Two-Dimensional Frame Buffer Processor

Arie Kaufman

Department of Computer science
State University of New York at Stony Brook
Stony Brook, NY 11794-4400, USA

The two-dimensional *Frame Buffer Processor* (FBP) is part of a proposed raster graphics computer architecture. It is a hardware-oriented organisation of a variation of a *bitblt* engine with a much richer repertoire. In addition, the FBP gives support to window management, transformations, and assists in some image operations ordinarily performed in software. The introduction of the FBP as a co-processor to geometry and video processors would increase efficiency and speed of graphics systems and bitmap workstations. A special skewed frame-buffer organisation, which allows parallel memory access, further improves system performance.

1. Introduction

Recently a considerable attention has been given to *bitblt* operations and *bitblt* engines/workstations [2, 7, 11, 14-16]. *Bitblt*, also termed *RasterOps* [12], is a relatively simple combining operation between two usually rectangular bitmaps. More complicated operations of transforming bitmaps can be surprisingly time-consuming (c.f. [1, 10, 17]). Guibas and Stolfi [5] have presented an abstract algebraic system for raster operations in which *bitblt* is just one such fundamental operation. The Image Prism [10] is a hardware solution for some of these complex operations, the 90° rotation and mirroring. This paper argues for a comprehensive solution and presents the *Frame Buffer Processor* (FBP), which is an extended *bitblt* engine with a much richer repertoire.

The overall system architecture is depicted in Figure 1. In this architecture the new processor, FBP, has been appended to a conventional computer graphics raster system. This system includes two classical processors: a Geometry Processor (GP) and a Video Processor (VP), all accessing the Frame Buffer (FB). The GP, executing its GDL, performs various operations on geometric primitives including geometric transformations, clipping, projection, hidden surface removal, and shading, followed by scan conversion to the FB. The VP scans the FB to refresh the monitor and may also perform, by executing its VDL, some basic operations like table look up, zooming, and cursor manipulation.

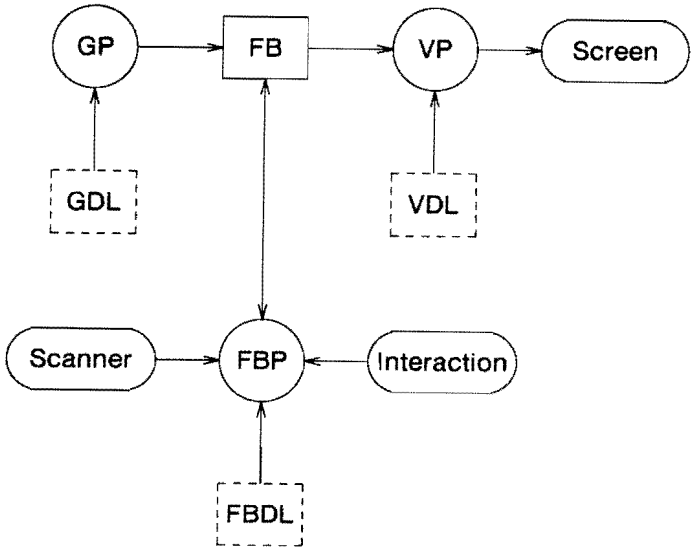

Figure 1: General Overview of System Architecture.

In an effort to speed up such a raster system, direct manipulation of pixel-map data is relegated to the FBP. The FBP, working as a co-processor of the GP and VP, releases the GP to concentrate on its classical geometric tasks and the VP to concentrate on its time bound video scan, while the FBP handles all the pixel-map operations and manipulations. Furthermore, the FBP performs some operations that are traditionally done by the GP (e.g., transformations) and by the VP (e.g., cursor handling).

The FBP operates directly on the images stored in the FB. The primitives of the FBP are cellular rectangular subareas of three types: *windows, cursors*, and *icons*. These primitives are defined and manipulated (transformed, rotated, copied, loaded, etc.) by the FBP, according to the instructions provided in the Frame Buffer Display List (FBDL). Since each FBP primitive is assigned a unique priority, this processor allows the implementation of overlapping primitives in two-and-a-half dimensions, supporting window management. The FBP also acts as a channel for 2D scanned data and as an interface for 2D input devices.

A skewed symmetric FB organisation permits the simultaneous storage and retrieval of all the pixels of a full row/column or parts thereof. The parallel access speeds-up several memory bound operations of the FBP. Furthermore, the VP exploits this organisation for parallel scanline retrieval, and the GP exploits it for parallel storage of runs of pixels.

The FBP external specifications, its primitives, tables, registers, repertoire, and modes, are described in Section 2, while the internal architecture is briefly outlined in Section 3. Section 4 presents the parallel FB access. The significant algorithms

employed by the FB, Clipping and Priority, Transformations, Orientation and Zooming, and Pixel and Cursor operations, are introduced in Sections 5-8. The chapter concludes with brief implementation notes.

2. FBP External Specifications

2.1. FBP Primitives

The FBP handles three types of primitives: *windows, icons,* and *cursors.* A *window* is comprised of a set of pixels, representing a rectangular subarea of the FB. It is defined by five values specifying its origin (X_0, Y_0) in FBP pixel coordinates, the length of each side (D_x, D_y), and its priority which signifies its relative position in between other primitives.

Icons are similar to windows, but are usually smaller, hold constant data, and are subject to simpler transformations. An icon can be bound to a window or it may float independently. If an icon is bound, any instruction that affects the window affects also the icon. If an icon is floating, it behaves like a window. An icon is defined by six parameters: origin, size and priority are defined in the same way as for a window, while the additional parameter identifies the bounding window in case the icon is bounded. The primary reason for introducing icons in addition to windows is to provide rapid and better support for window management and master/cell library graphics systems.

A *cursor* is a small user-defined square window. It usually represents feedback from a locator device and is used to locate and pick primitives from the FB. It is defined by three values specifying its center (X_0, Y_0) in FBP coordinates, and its size. Typical sizes are 8^2 and 16^2. A cursor is physically stored in the FB, and the pixel information covered by a cursor is saved within the processor for later restoration when the cursor moves. Traditionally, 2-D cursor handling has been performed by the VP. Cursor handling by the FBP alleviates the high load already assumed by the VP. Since a cursor might be hidden from the user behind other primitives, it is assigned the highest priority so it is always shown on the screen floating in front of all other primitives as if the cursor is in the third dimension.

2.2. FBP Tables

The FBP supports five indexed tables, the Window, Icon, Cursor, Priority, and Device Tables, which are all designed as an integral part of the FBP. The *Window Table (WT),* indexed by window identifiers, is used by the processor to handle all operations concerning windows. Each entry holds the window attributes and several flags.

The *Icon Table (IT),* indexed by the icon identifier, is used by the processor to handle all operations concerning icons. If an icon is bound to a window, a field in the table holds the identifier of that window.

The *Cursor Table (CT)*, indexed by the cursor identifier, is used by the processor to handle the cursors. In addition to the cursor attributes, each entry holds the loaded cursor pattern or the pixel data it obscures when it resides within the FB.

The *Priority Table (PT)* keeps track of sorted priorities of windows and icons. A PT entry is created every time a new window or icon is defined. The PT is used to speed up the clipping and priority algorithm (see Section 5.1) by allowing the FBP to rapidly determine the current priority order among the primitives.

The *Device Table (DT)* is used by the FBP to interface to the graphical input devices (e.g., joystick locator devices). It contains bookkeeping information about the device and the cursor(s) attached to it for feedback purposes.

2.3. FBP Registers

Internal registers of the FBP can be read, set, or reset, and all subsequent operations are consequently affected. The registers that govern the configuration of the FBP are alterable only during the initial passive mode (see Section 2.5). All the other registers or fields thereof are alterable also during the active mode of the FB.

The FBP registers are divided into four main groups: Address, Mode and Status, System, and Input Registers. *Address Registers* include general purpose address registers, a Stack Pointer (SP), which points to the top of a conventional stack, and a Program Counter (PC) which points to the next executable instruction in the FBDL, which resides in main memory.

Mode and Status Registers are divided into two subgroups. The first subgroup specifies the environment of the system, including the FBP origin (Origin), colour/translucency mode (Colour_Mode), width of components in three colour system mode (Colour_Field), and which colour planes in the FB are enabled (Depth_Plane). In addition, there is a Status register, which indicates which Mode registers (see the following paragraph) are currently enabled.

The second subgroup includes the Mode registers. Each Mode register is enabled by a specific bit in the Status register. This subgroup includes registers which control zooming (Zoom), boolean operations on pixels and cursors (Pixel_Ops and Cursor_Ops), masks for texture and filtering (Texture_Mask and Filter), Background and Foreground colour, and window orientation (Orient), i.e., rotation by 0, 90, 180, or 270 degrees about the window origin. Any operation performed on the primitives that write pixel information into the FB, e.g., copy, is directly affected by these registers. For example, if the x field of the Zoom register, ZoomX, indicates a zoom of factor 2, any pixel written to the FB is automatically duplicated horizontally. If the Orient register specifies that the FB origin is its top right corner all windows written into the FB will be oriented accordingly, i.e., rotated by 180°.

System Registers define the FBP configuration. They include the State register, which defines the Working Mode of the FBP (see section 2.5), and the Init register which determines the configuration of the system (two fields for FB size in X and

Y, and three fields for the maximum number of windows, icons, and locator devices), and is alterable only during the initial Passive mode.

Three additional system registers are used for handling errors and traps. The Trap_Enable register has a one bit field for each type of error. When an error occurs, the Trap_Enable register is used to determine if this error should cause a trap (change Working Mode to Suspended). The Error register also has a one bit field for each type of error. It is used by the **jumperr** instruction to jump to an appropriate error handling routine based upon the kind of error. The Stop register is used to pass the error category back to the CPU. It is only set if an error is detected (a bit is set in the Error register) and trapping is enabled (the corresponding bit is set in the Trap_Enable register).

Input Registers define the 2D input environment, assisting in the user-FBP interaction. Up to four locator devices and one scanner may be defined in the current implementation. The Input_Enable register indicates which input devices are currently enabled. There are four Device$_i$ registers ($0 \leqslant i \leqslant 3$) each contains relative position information. The Device_Status register has two fields for each of the locator devices. One field indicates whether a certain device is ready to pass data to the FBP, and the other whether the data has been read by the FBP.

Two additional input registers are used with the scanner. The Scanner_Status register contains two fields. One indicates whether the scanner is ready to pass data to the FBP, and the other indicates whether the data has been read by the FBP. The Scanner_Buf register holds the data being passed by the scanner to the FBP. The data is a colour value defined by the active Colour_Mode and Colour_Code. These registers together with the **scan** into window command provide a mechanism to input raw pixel data from 2-D scanners directly into the FB.

2.4. FBP Repertoire

The FBP3 instruction set includes operations on windows, cursors, and icons, and general instructions. All primitives can be created (**define**), removed (**delete**), cleared (**erase**), read from FB (**read**), written into FB (**write**), translated (**translate**), copied (**copy**), and swapped with another primitive (**swap**). Windows can also be filtered for noise reduction. Unlike icons and cursors, windows are scaleable (**scalew**) using non-integral factors. Windows can be further rotated and spun (repeatedly) through any angle about either the x or the y axes. Icons and cursors can be rotated only through angles which are multiples of 90°.

Windows and icons, which are assigned priorities, are also subject to a change of priority command which may affect future placements of primitives in the FB. Icons have an exclusive instruction for bounding/unbounding of icons to/from windows. An exclusive instruction for cursors is for loading a cursor pattern from main memory (e.g., from FBDL, from stack) to the CT. The large sets of operations for handling windows, icons and cursors and the underlying FBP registers provide an extremely flexible environment for manipulating the FB images.

The general instructions of the FBP include register manipulation, (**set, setf** - set field, **get, getf, reset**), program flow control (**jump, jumperr, call, return**), stack control (**push, pop**), mode control (**active, halt**) and input device control (**scan, attach** device, **disattach**).

2.5. FBP Modes

The FBP has three working modes: Passive, Active, and Suspended. The *Passive Mode* is the initialisation mode. The FBP powers up in this mode and enters Passive mode whenever the State register is set to 0. In this mode, the FBP loads initial values (either default or user supplied) into the fields Fbx_Size, Fby_Size, Origin, Max_Window_No, Colour_Field and Colour_Code. These registers may only be changed in the Passive Mode. The user may use the **reset** instruction to assign the default values to these registers and to the Trap_Enable register. Other registers of the FBP can be initialised either in the Passive Mode or in the Active Mode. To switch to the Active Mode the **activate** instruction has to be invoked, which also sets the State register to 1.

The *Active Mode* is the mode in which the FBP operations are executed. The FBP performs the FBDL instructions one by one, using the PC to fetch them. There are three possible addressing modes for the instruction data: direct mode in which the data follows the instruction; indirect mode in which the address of the data follows the instruction in the FBDL; and Stack Mode in which the data can be found in the stack.

When the FBP encounters an error while executing the FBDL, it sets the proper bit in the error register and checks the Trap_Enable register. If the error causes a trap, the FBP enters the Suspended Mode by setting the State register to 2. Otherwise, if traps are disabled the user may handle the error using the **jumperr** (jump on error) instruction. To return to the Passive Mode the user ought to use the **halt** instruction, which sets the State register to 0.

In the *Suspended Mode* the CPU reads the Stop register which indicates which error has caused the trap. Thereafter the CPU can return to either the Active Mode or the Passive Mode by setting the State register to 1 or 0, respectively.

3. Internal Architecture

The FBP processor is designed as a pipeline architecture, composed of three units in sequence: the Instruction Unit, the Execution Unit, and Frame Buffer Management Unit. The *Instruction and Decoding Unit* is responsible for fetching the instructions stored in the FBDL, decoding them and passing them to the Execution Unit. It is composed of the following subunits: Fetch & Memory Management Subunit, Decoding Subunit, Data Address Calculation Subunit, and the Init & Error Subunit.

The *Execution Unit* is responsible for the execution of the instruction. It is composed of the following subunits: Control Subunit, Input Subunit, Geometry

Subunit, Clipping & Priority Subunit, Raster Subunit, and the Table Subunit. The microprogrammed Control Subunit holds a number of algorithms, described in Section 5, to be executed by the Execution Unit.

The *Frame Buffer Management Unit* handles the interface to the Frame Buffer Bus, which provides a communication mean to the FB. It handles the address(es) and the data, either a single pixel or a run of pixels, to be stored/retrieved to/from the FB. This unit is internally linked back to the Fetch & Memory Management Unit for instructions like **readw** where the FB data is transferred to the main memory. It is also linked internally to the Geometry Subunit and other units that require data from the FB.

4. Parallel FB Access

Assume a FB of n^2 pixels. It is constructed in such a way as to enable a simultaneous access to a full beam (row or column) of n pixels. The physical memory is divided into n modules, each of which has exactly n pixels, each with its own independent access and its own independent addressing unit, so that no two pixels of a beam reside in the same module. Therefore, all the pixels of a beam can be fetched/stored simultaneously in one memory cycle. This division, however, should occur for all rows and all columns.

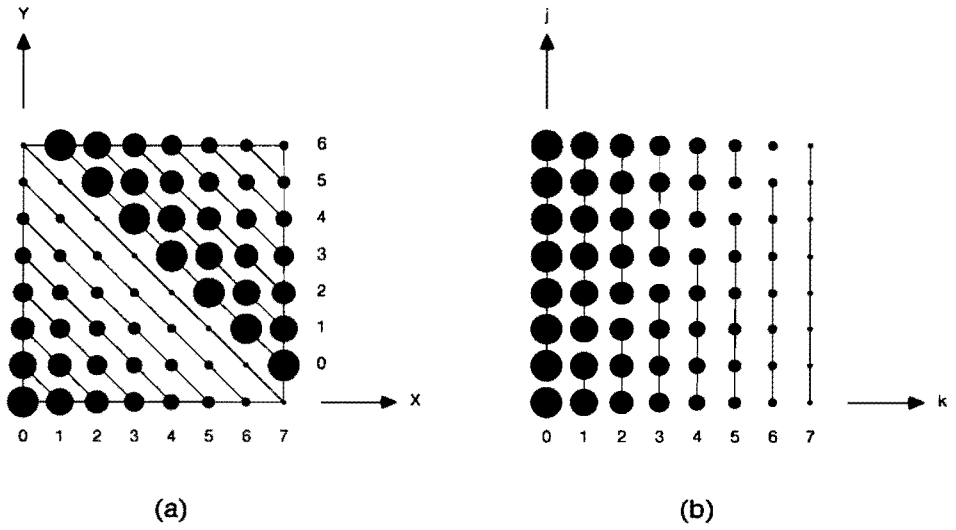

Figure 2: Parallel Memory Organisation for an 8x8 Frame Buffer. The 2-D Frame Buffer (a) is organized as 8 distict modules (b).

These restrictions have suggested a modular memory construction which consists of diagonal parallel sections of the FB having a 45 degree angle with the axes (see Figure 2). The diagonal sections are sequentially numbered and grouped modulo n into n modules indexed 0 through $n-1$. This skewed organisation is formalised as follows. A pixel with raster coordinates (x,y) is being mapped onto the k-th module ($0 \leqslant k \leqslant n-1$) by:

$$k = (x+y) \bmod n \tag{1}$$

Since one coordinate is always constant along any beam, the second coordinate guarantees that one and only one pixel from the beam resides in any one of the modules. The internal mapping j within the memory module is given by:

$$j = y \tag{2}$$

Thus the k-th memory module can be regarded as a 1-D projection of all the diagonal sections in that module (those that are indexed k) onto the y axis. An extension of this skewed organisation to a 3-D cubic frame buffer has also been devised [8].

For a FB of 1024×1024 resolution there are 1024 memory modules, each one contains 1024 pixels. The time needed to retrieve a full beam of 1024 pixels is a single cycle time. Scanning the whole FB in beam by beam (scan-line mode) by the VP will take only about 30 msecs for a memory with 30 nsecs access time. The parallel write can be exploited by the GP in line scan-conversion algorithms, where a run length scan-conversion algorithm [3] can write runs of pixels simultaneously into the FB instead of one pixel at a time. The same applies to polygon scan-conversion in which the pixels comprising the runs between edge intersections are painted simultaneously. Last but not least, this parallel organisation is heavily used by the FBP to perform operations on the entire beam or part thereof, as in translation, 90° rotation and mirroring, erasing with a constant colour, etc.

5. Clipping and Priority Algorithm

The FBP supports several simultaneous windows and icons, each is assigned a unique priority, which is specified at time of definition. If collision occurs, temporal priority is governing. When a window or an icon, w, with two opposite corners $(X_{\min}, Y_{min})$, $(X_{\max}, Y_{\max})$, is written into the FB, by copying it from outside or by moving information within the FB, the priority and clipping algorithm is employed in order to determine which portions of w are not obscured by any of the other windows. The algorithm is a scan-line algorithm [4] adjusted for fast processor-level implementation. For each scan-line, *Yscan*, from minimum Y, $Y_{\min}$, to maximum Y, $Y_{\max}$, of w, the visible segments are written into the FB using the parallel FB access mechanism. This is intrinsicly different from the algorithm proposed by Pike [13] at the window manager level, in which the window in question is recursively subdivided into subrectangles and all the subrectangles have to be accounted for. Here, on the other hand, only the visible segments have to be

handled by the FBP, the invisible sections are handled by the application.

The algorithm employs a temporary table, *InterTable*, which lists all the windows and icons having a priority higher than w and intersecting w, sorted by their minimum Y and minimum X. The algorithm in pseudo-C code follows:

```
For (Yscan = Ymin;   Yscan ≤ Ymax;   Yscan++)
{
    Remove fom InterTab primitives below Yscan
    Sort by X all primitives in InterTable which intersect the Yscan line
    While (LeftBound ≤ Xmax)
    {
        RightBound= Right X of leftmost primitive (called "left primitive")
        LeftBound= Left X of next leftmost primitive ("right primitive")
        While (LeftBound ≤ RightBound and InterTab not empty)
        {
            Get next right primitives and update LeftBound
            If (left and right primitives are adjacent in sorted list)
                Then { Set run from RightBound to LeftBound along
                       Yscan line to the pixel value of w;
                       Get next left and right primitives from list }
                Else   Get the next left primitive and update
                       RightBound if the right edge of the new left
                       primitive is to right of RightBound
        }
    }
}
Restore saved cursors
```

Note that the algorithm has been described assuming that the ZoomX, ZoomY, and Orient registers are set at their default values, 1, 1, and 0°, respectively. However, other values assigned to these registers may drastically affect the way it executes, since the sequence by which w is clipped is not the order of the input data. Consequently, the directions the algorithm progresses depend on the Orient register, and the pixels or rows of pixels have to be replicated according to the ZoomX and ZoomY registers before writing runs of pixels between *RightBound* and *LeftBound*.

6. Geometric Transformations

Special care has been taken to provide simple and fast geometric transformations especially for the large window operands. All transformations, carried out by the FBP, including translation, scaling (using non-integral factors), and rotation (through any angle), are performed using incremental transformation techniques, in which each pixel is transformed, based on the new position assumed by its immediate previous neighbour, with a maximum of 2 additions/increments. The algorithms are based on inter- and intra-scan-line coherency. In other words, after calculating the new location for one pixel, its neighbours relate to it as they did in the old location.

The geometric algorithms also resolve possible overlapping between the source and the destination primitives. This is necessary since there is no scratch FB memory available to temporarily hold old pixel information while performing the transformation.

6.1. Translation

The **translate** function moves a primitive (window, icon, or cursor) relative to its current position by (T_x, T_y). The equations which describe the translation transformation are:

$$X' = X + T_x \qquad Y' = Y + T_y \tag{3}$$

An incremental algorithm specifys the position of the next pixel to be translated $(X+1, Y)$ in terms of its predecessor's new position (X', Y'):

$$\begin{aligned} (X+1)+T_x &= X+T_x+1 = X'+1 \\ Y+T_y &= Y' \end{aligned} \tag{4}$$

The coordinate of the next pixel in the destination primitive can thus be determined by only *incrementing* the appropriate coordinate of the last pixel without performing addition on every pixel. The same also applies to a vertical step in Y. The algorithm may be summarised in pseudo-C as follows (old coordinates refer to the original primitive and new coordinates refer to the translated primitive):

```
Initialise X and Y
X' = X + Tx
Y' = Y + Ty
startX = X

while (Y ≠ Ymax)
{
        while (X ≠ Xmax)
        {
                If ( (X', Y') is visible )
                        Put pixel value of (X, Y) at (X', Y')
                Increment/Decrement X
                Increment/Decrement X'
        }
        X = startX
        Increment/Decrement Y
        Increment/Decrement Y'
}
```

One problem which arises is that if the translated primitive is partially overlapping with the original primitive, then some of the original pixels might be overwritten by the copied pixels before the original pixels are translated, which could distort the destination primitive. The above algorithm therefore has to be modified in a way

that no pixels will be overwritten before they are translated. This requires the algorithm to test for a possible overlap, and if exists to classify it as either a south, north, east, or west overlap, in which case start copying from north, south, west, and east, respectively. This is reflected in the algorithm by incrementing/decrementing X, Y, X', and Y'.

6.2. Scaling

The equations which describe the scaling transformation algorithm for windows about $(X_{\min}, Y_{\min})$ are given in Equation (5). The shifting to and from the origin are accounted for in the FB addressing of the source and the destination.

$$X' = X\,S_x \qquad\qquad Y' = Y\,S_y \tag{5}$$

Icons and cursors are not scaleable. The factors S_x and S_y are not necessarily integers, and the scaling algorithm for windows requires thus two non-integer multiplications and two roundings for each pixel, which can result in heavy computations for large windows. The complexity of the computations can be significantly reduced by using an incremental algorithm based upon:

$$\begin{aligned}(X+1)\,S_x &= X\,S_x + S_x = X' + S_x \\ (Y+1)\,S_y &= Y\,S_y + S_y = Y' + S_y\end{aligned} \tag{6}$$

The coordinates of the next translated pixel can thus be obtained by simply adding the respective scaling factor to the X or Y coordinate of the predecessor pixel and rounding the results to integer coordinates. Precautions must be taken if S_x and S_y are both greater than 1. In this case, applying the above transformation to a primitive may produce holes in the scaled primitive. We can eliminate this problem by *back scaling*, i.e., reversing the incremental transformation process. To do this we transform two opposite corners of the source primitive, which yields two opposite corners in the destination primitive. We then step through the destination primitive. At each point in the destination primitive, we reverse-transform the coordinates to determine the pixel value for that point. This can be done by multiplying the destination primitive's coordinate by $(1/S_x, 1/S_y)$. A more efficient way to do this is to step through the source primitive in parallel with the stepping through the destination primitive. The increments to be used in stepping through the source primitive are $(Xincr/S_x, Yincr/Sy)$, where *Xincr* and *Yincr* are the X and Y increments (1 or -1) used in stepping through the destination primitive.

As with the translation algorithm, precautions must be taken to avoid overwriting pixels in the source primitive before they have been scaled. This is accomplished by properly choosing the corner of the primitive to be used as a starting point for the transformation based on the values of S_x and S_y.

Assume that (X_1, Y_1), (X_2, Y_2) and (X_1', Y_1'), (X_2', Y_2') are the lower left and upper right coordinates of the source and destination primitives, respectively; (X_S, Y_S) and (X_E, Y_E) are the starting and ending corners in the destination

primitive and (X, Y) is the starting corner of the source primitive (the reverse-transformation of (X_S, Y_S)). Then:

1. If $(S_x<1)$ and $(S_y<1)$, then
   $(X_S, Y_S)=(X_1', Y_1')$ $(X_E, Y_E)=(X_2', Y_2')$
   $(X, Y)=(X_1, Y_1)$ $Xincr=1$ $Yincr=1$

2. If $(S_x<1)$ and $(S_y>1)$, then
   $(X_S, Y_S)=(X_1', Y_2')$ $(X_E, Y_E)=(X_2', Y_1')$
   $(X, Y)=(X_1, Y_1+D_y)$ $Xincr=1$ $Yincr=-1$

3. If $(S_x>1)$ and $(S_y<1)$, then
   $(X_S, Y_S)=(X_2', Y_1')$ $(X_E, Y_E)=(X_1', Y_2')$
   $(X, Y)=(X_1+D_x, Y_1)$ $Xincr=-1$ $Yincr=1$

4. If $(S_x>1)$ and $(S_y>1)$, then
   $(X_S, Y_S)=(X_2', Y_2')$ $(X_E, Y_E)=(X_1', Y_1')$
   $(X, Y)=(X_1+D_x, Y_1+D_y)$ $Xincr=-1$ $Yincr=-1$

Once the above variables are determined, the algorithm progresses as follows:

```
X' = X_S;            Y' = Y_S
SXincr = Xincr/S_x; SYincr = Yincr/S_y
while (Y ≤ Y_E)
{
    Round Y
    while (X ≤ X_E)
    {
        Round X
        If new primitive is visible at (X', Y')
            Put pixel value from (X, Y) at (X', Y')
        Increment X by SXincr
        Increment X' by Xincr
    }
    Increment Y by SYincr
    Increment Y' by Yincr
}
```

6.3. Rotation

The equations for rotating a window are as follows:

$$X' = X\cos\theta - Y\sin\theta \tag{7}$$

$$Y' = Y\sin\theta + Y\cos\theta \tag{8}$$

where θ is an arbitrary counterclockwise angle about $(X_{\min}, Y_{\min})$. Like for the scaling, the shifting to and from the origin are accounted for in the FB addressing

of the source and the destination. In raster rotation every pixel of the primitive has to be rotated, however, an incremental algorithm can be derived, determining the new position for a pixel avoiding any multiplication:

$$X_{x+1}' = (X+1)\cos\theta - Y\sin\theta = (X' + \cos\theta) \tag{9}$$

$$Y_{x+1}' = (X+1)\sin\theta + Y\cos\theta = (Y' + \sin\theta) \tag{10}$$

$$X_{y+1}' = X\cos\theta - (Y+1)\sin\theta = (X' - \sin\theta) \tag{11}$$

$$Y_{y+1}' = X\sin\theta + (Y+1)\cos\theta = (Y' + \cos\theta) \tag{12}$$

The values of $\cos\theta$ and $\sin\theta$ are calculated only once, so after transforming the first pixel, subsequent transformations require only 2 additions and two roundings. By adding 0.5 to the initial coordinates, roundings can be reduced to truncation, which is done automatically in fixed point arithmetic. Note that Equations 9 and 10, and Equations 11 and 12, are performed in parallel within the FBP.

Like in scaling *back rotation*, from destination to source [6], is employed to eradicate holes which could be produced from the forward rotation of a window. To accomplish this task the four corners of the original window are rotated to locate the destination window. Once calculated, the minimum coordinates are back rotated by the rotation equation, Equations 7 and 8, by replacing θ by $-\theta$. From this point on, an incremental algorithm is used to back scan the entire window. The constant increments for all pixels are merely $\sin\theta$ and $\cos\theta$. Determining the coordinates of the next pixel when the current ones (X, Y) are known, is a simple matter of applying one of two sets of equations on the current pixel. Equations 13 and 14 give the coordinates of the point (X_{x+1}, Y_{x+1}) back-rotated from $(X'+1, Y')$, while Equations 15 and 16 give the point (X_{y+1}, Y_{y+1}) back-rotated from $(X', Y'+1)$:

$$X_{x+1} = (X' + 1)\cos\theta + Y'\sin\theta = X + \cos\theta \tag{13}$$

$$Y_{x+1} = -(X' + 1)\sin\theta + Y'\cos\theta = Y - \sin\theta \tag{14}$$

$$X_{y+1} = X'\cos\theta + (Y' + 1)\sin\theta = X + \sin\theta \tag{15}$$

$$Y_{y+1} = -X'\sin\theta + (Y' + 1)\cos\theta = Y + \cos\theta \tag{16}$$

Note that either set employs only two fixed-point additions that can be performed in parallel. Note also that the destination window can be clipped first, then only the visible parts need to be back-rotated.

In the rotation operation, we also take precautions that a pixel would not be overwritten until it is rotated. If the rotation angle is between 90° and 270°, and the destination and source primitives do not overlap, then the scan sequence is

irrelevant. However, if the rotation angle is less than 90°, we should scan by columns from left to right, while if the rotation angle is greater than 270°, we can scan by rows from bottom to top.

Rotations of 90°, 180°, and 270° are special cases, and can be directly transformed employing the following equations:

$$\begin{aligned} 90°: &\quad X' = X_0 - Y \qquad Y' = Y_0 + X \\ 180°: &\quad X' = X_0 - X \qquad Y' = Y_0 - Y \\ 270°: &\quad X' = X_0 + Y \qquad Y' = Y_0 - X \end{aligned}$$

7. Orientation and Zooming

The Orient register of the FBP automatically affects the orientation of a window being written into the FB. Let (X_0, Y_0) be the lower left corner of the window, and D_x and D_y be the horizontal an vertical sides of the window, then the equations that describe the effect of the Orient register are as follows:

If Orient = 0, then no effect on the window.

If Orient = 1, then the window rotates by 90°, and the equations are:

$$X_0' = X_0 - D_x \qquad Y_0' = Y_0$$
$$D_x' = D_y \qquad D_y' = D_x$$

If Orient = 2, then the window rotates by 180°, and the equations are:

$$X_0' = X_0 - D_x \qquad Y_0' = Y_0 - D_y$$
$$D_x' = D_x \qquad D_y' = D_y$$

If Orient = 3, then the window rotates by 270°, and the equations are:

$$X_0' = X_0 \qquad Y_0' = Y_0 - D_x$$
$$D_x' = D_y \qquad D_y' = D_x$$

The Zoom register, having two fields ZoomX and ZoomY, automatically affects the pixel replication of all the windows written into the FB. The ZoomX field specifys the pixel replication of each pixel in the X direction, while the ZoomY field specifys the replication of each row of pixels in the Y direction.

8. Pixel- and Cursor-Ops

This algorithm is performed before writing data into the FB. It is governed by two registers: the Pixel_Ops register which applies to windows and icons, and the Cursor_Ops register which applies to cursors. There is a total of 16 pixel-wise boolean operations which operate on a source pixel (S) and/or a destination pixel (D) and place the result in D. These operations are accepted when the processor is working either in the Gray_Scale or in the Binary mode. In the Binary mode, the Background colour represents a 0 and a Foreground colour represents a 1. The

boolean operations are summarised in Table 1. The four bits of the index of the boolean function given in the register are exactly the mask utilised by the FBP to perform the function.

Boolean Function	Index	D=0 S=0	D=0 S=1	D=1 S=0	D=1 S=1
Background	0	0	0	0	0
D and S	1	0	0	0	1
D and (not S)	2	0	0	1	0
D	3	0	0	1	1
(not D) and S	4	0	1	0	0
S	5	0	1	0	1
D xor S	6	0	1	1	0
D or S	7	0	1	1	1
D nor S	8	1	0	0	0
not (D xor S)	9	1	0	0	1
not S	10	1	0	1	0
D or (not S)	11	1	0	1	1
not D	12	1	1	0	0
(not D) or S	13	1	1	0	1
D nand S	14	1	1	1	0
Foreground	15	1	1	1	1

Table 1: Pixel- and Cursor-Ops.

9. Implementation Notes

The FBP has been designed, developed and simulated at both the Ben-Gurion University of the Negev and the State University of New York at Stony Brook. The simulations have been written in C on a VAX-780 and SUN workstations running Unix. At the Ben-Gurion University a RAMTEK-9400 has been used as the output device. This simulation implements most of the FBP instructions, but using only the direct addressing mode [18].

A more complete simulation has been implemented at the University at Stony Brook on a SUN workstation using the SunCore graphics package. The input to FBP is a FBDL which lists the instructions to be executed by the FBP, and the output is shown on the SUN colour monitor. Echo of command execution can be shown on the monitor. Care has been taken to make the FBP system easy to transport to other machines keeping device-dependent routines separate from device-independent ones. This version of the FBP has been used in prototyping the 2-D component of a 3-D voxel-based workstation, the CUBE workstation [9].

In both implementations a Window Manager has been programmed in software on top of the FBP simulation. It employs a major part of the FBP

repertoire. The data structures of the window manager system holding all obscured portions have been handled by the window manager in a way similar to [13]. These implementations have proven the feasibility of the FBP approach especially in a rather complex and critical application of a window manager.

Acknowledgement

This work was supported by the National Science Foundation under grant DCR-86-03603. I wish to thank the many dedicated individuals who have worked on the FBP project, especially Dario S. Zutel, who has contributed the first design and implementation of the FBP on the RAMTEK system.

References

1. H.R. Arabnia and M.A. Oliver, "Arbitrary rotation of Raster Images with SIMD Machine Architecture," *Computer Graphics Forum* **6**, pp. 3-12 (1987).
2. A. Bechtolheim and F. Baskett, "High-Performance Raster Graphics for Microcomputer Systems," *Computer Graphics* **14**(3), pp. 43-47 (July 1980).
3. J.E. Bresenham, "Run Length Slice Algorithm for Incremental Lines," pp. 59-104 in *Fundamental Algorithms for Computer Graphics*, ed. R.A. Earnshaw, Springer-Verlag, Berlin (1985).
4. J. D. Foley and A. Van Dam, *Fundamentals of Interactive Computer Graphics,* Addison-Wesley (1982).
5. L. J. Guibas and J. A. Stolfi, "A Language for Bitmap Manipulation," *ACM Trans. on Graphics* **1**(3), pp. 191-214 (July 1982).
6. R. D. Hersch, "Raster Rotation of Bilevel Bitmap Images," *Proceedings EUROGRAPHICS '85*, Nice, France, pp. 295-308 (September 1985).
7. D. H. H. Ingalls, "The Smaltalk Graphics Kernel," *Byte* **6**(8), pp. 168-194 (August 1981).
8. A. Kaufman, "Memory Organization for a Cubic Frame Buffer," *Proceedings EUROGRAPHICS '86*, Lisbon, Portugal, pp. 93-100 (August 1986).
9. A. Kaufman, "Towards a 3-D Graphics Workstation," in *Advances in Graphics Hardware I*, ed. W. Strasser, Springer-Verlag, Berlin (1987).
10. C. Korenfeld, "The Image Prism: A Device for Rotating and Mirroring Bitmap Images," *IEEE Computer Graphics & Application* **7**(5), pp. 21-30 (May 1987).
11. H. M. Levy, "VAXstation: A General-Purpose Raster Graphics Architecture," pp. 70-83 in *ACM Transactions on Graphics* (January 1984).
12. W. M. Newman and R. F. Sproull, *Principles of Interactive Computer Graphics,* McGraw-hill, (2nd Ed.), New York (1979).
13. R. Pike, "Graphics in Overlapping Bitmap Layers," *ACM Transactions on Graphics* **2**(2), pp. 135-160 (1983).

14. R. Pike, L. Guibas, and D.H.H. Ingalls, "Bitmap Graphics," Course Notes ACM SIGGRAPH'84 (July 1984).

15. R. Pike, "The Blit: A Multiplexed Graphics Terminal," *AT&T Bell Labs Technical Report* **63**(8), pp. 1607-1631 (October 1984).

16. R. Pike, B. Locanth, and J. Reiser, "Hardware/Software Tradeoffs for Bitmap Graphics on the Blit," *Software-Practice & Experience* **15**(2), pp. 131-151 (February 1985).

17. R. F. Sproull, I. E. Sutherland, A. Thompson, and C. Minter, "The 8 by 8 Display," *ACM Transactions on Graphics* **2**(1), pp. 1-31 (January 1983).

18. D. S. Zutel, "A Frame Buffer Processor," MSc Thesis, Ben-Gurion University, Beer-Sheeva (June 1985).