Check for updates

# VAXstation: A General-Purpose Raster Graphics Architecture

HENRY M. LEVY
Digital Equipment Corporation

A raster graphics architecture and a raster graphics device are described. The graphics architecture is an extension of the RasterOp model and supports operations for rectangle movement, text writing, curve drawing, flood, and fill. The architecture is intended for implementation by both closely and loosely coupled display subsystems. The first implementation of the architecture is a remote raster display connected by fiber optics to a VAX minicomputer. The device contains a separate microprocessor, frame buffer, and additional local memory; it is capable of executing raster commands on operands in local memory or VAX host memory.

Categories and Subject Descriptors: I.3.1 [**Computer Graphics**]: Hardware Architecture—*raster display devices*; I.3.2 [**Computer Graphics**]: Graphics Systems—*remote systems*; *stand-alone systems*; I.3.3 [**Computer Graphics**]: Picture/Image Generation—*display algorithms*

General Terms: Design

Additional Key Words and Phrases: RasterOp, BitBlt, workstation

## 1. INTRODUCTION

High-resolution bit-mapped raster displays, as pioneered on the Xerox PARC Alto computer [17], have now become standard on many personal computers and workstations [1, 3, 14]. These displays are characterized by a relatively large (typically 15 inch to 19 inch diagonal) display surface containing on the order of 1000 × 1000 addressable picture elements at a resolution of 70 to 90 elements per inch. High-resolution raster systems allow the user to create, view, and manipulate images containing graphics, multiple type fonts, and picture images either separately or within a single document or activity [5, 13, 15]. These capabilities are made possible by the high-resolution format. Another powerful concept typically supported by such displays is a window management system [9, 10, 12, 16]. A window management system allows multiple independent processes to share the display screen, the output from each process appearing in a rectangular area known as a window. Although window systems have been implemented on more traditional display terminals [11], the practical application of window systems is aided by the larger screen area provided by newer displays.

Author's present address: Department of Computer Science, FR-35, University of Washington, Seattle, WA 98195.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1984 ACM 0730-0301/84/0100-0070 $00.75

In 1981, a project was started at Digital to provide users of VAX minicomputers with a workstation environment. It was clear at that time that a VLSI VAX implementation suitable for an integrated VAX workstation was still several years away. In the interim, a device was required that would allow experimentation with workstation software while connecting to a standard VAX minicomputer. Two advanced development efforts were then underway at Digital: a project prototyping a single-user VAX system with a raster graphics display (known internally as SUVAX) and an effort to define a display system to support Smalltalk on the VAX [2]. These efforts were combined to form the VAXstation program, with the Smalltalk effort providing the dominant architectural characteristics.

There were several goals for the design of the display system. First, since VAX minicomputers were too large to place in an office, the display would have to operate at a relatively long distance (e.g., 1 kilometer) from the CPU. Second, it should present an architectural interface suitable for use in an integrated workstation when a smaller VAX became available. Third, it should support primitives for window management systems and highly interactive applications (e.g., Smalltalk [7]). Fourth, at least in the initial remote version, it should be possible to connect several of the display systems to one of the larger VAX computers.

Goals one and two were the hardest to combine, since they require a highly integrated yet remote graphics system. Most integrated systems are based on the Xerox PARC Alto model; that is, the display frame buffer memory is shared by the host and display system. The display is an integral part of a single-user computer system, and instructions executed on the CPU of that system directly manipulate the display frame buffer, thus modifying the image on the screen. The ability to directly construct the display image with CPU instructions provides for great flexibility. Not only does the processor have fine-grained control over image production, but images can be stored and manipulated anywhere in the processor's address space, including the frame buffer memory. In contrast are remote raster graphics devices such as the BBN BitGraph [4]. Devices of this type are more loosely coupled; they typically include a microprocessor that receives commands over a serial line and operates on a frame buffer local to the display device.

The VAXstation design lies somewhere in the middle of these two alternatives. VAXstation is similar to remote devices in that it contains a separate microprocessor, memory, and frame buffer. Commands from the host initiate display system graphics operations. However, unlike these devices, the VAXstation can also perform graphics operations in *host* memory. In this way, high-level graphics operations are not restricted to the frame buffer or display-local memory. Host memory can be used for storing images, occluded windows, etc., and operations can be performed by the display processor on those images in host memory. These operations are asynchronous with the operation of the host processor. The VAXstation is thus a coprocessor to the VAX host.

Section 2 describes the structure of the VAXstation 100, the first implementation of the VAXstation architecture. The remainder of the paper presents the VAXstation graphics architecture, that is, the set of text and graphics commands supported by the VAXstation microprocessor.

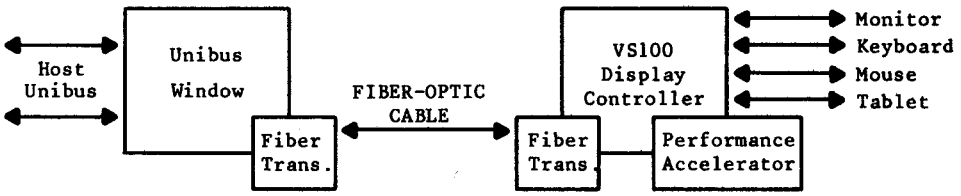


Fig. 1. VAXstation 100 block structure.

## 2. VAXstation 100 HARDWARE STRUCTURE

The VAXstation 100 (VS100) is a display subsystem that connects to a VAX minicomputer. Figure 1 illustrates the basic block structure of the VS100. The right-hand side of this figure shows the device subsystem consisting of a small box housing three modules and connectors to the user I/O devices. This box is placed near the desk on which the devices reside. The monitor is a 19-inch, 60-hertz, horizontal-format, noninterlaced, monochrome device that displays 1088 × 864 picture elements. The display size and format were chosen (over 15 inch vertical format which is used in the Alto) because (1) the larger area gives more flexibility for arranging activities, (2) it was felt that users would rather have additional space for icons and windows on the sides instead of on the top and bottom, and (3) we saw that most second generation systems had gone to larger horizontal format. The system uses a three-button mouse as the standard pointing device, a 105-key unencoded keyboard, and an optional tablet for graphics input.

The left side of Figure 1 shows the VAX connection section of the hardware. A VAX Unibus interface was chosen because of the requirement that the VS100 work on all VAX systems. The Unibus Window module, which plugs into the Unibus backplane of the host, allows the VS100 to read and write VAX memory. It provides the VS100 with an 18-bit (256-Kbyte) virtually contiguous window into VAX primary memory. A data structure maintained by the host operating system specifies the primary memory locations of 512-byte virtual pages within this address space. The Unibus Window also contains control and status registers through which host software commands the VS100. Modification of the control registers by the host causes a special interrupt in the VS100 subsystem.

To control the device, the host VS100 device driver places the address of a command packet in a VS100 control and status register. The VS100 then reads and responds to the command, performing operations on whatever memory is specified. All data transfers between display and host memory are carried out by the display microprocessor.

Because of the requirement that the VS100 run with high bandwidth over relatively long distances, the 15-megahertz fiber optic cable is used between the Unibus Window module on the host and the VS100 display subsystem. Two fiber optic transceiver modules, one attached to the Unibus Window and one in the VS100 subsystem, control the sending and receiving of messages over the fiber optic link.

Within the subsystem itself are three boards: the VS100 display controller, a performance accelerator, and the fiber-optic transceiver. The main VS100 display controller module contains a Motorola 68000 microprocessor, 512 Kbytes of local

image memory, 128 Kbytes of local program memory, and the screen refresh logic. The M68000 is responsible for receiving, interpreting, executing, and responding to all commands from the host. The M68000 also receives keyboard and pointing device events, reports the events to the VAX, and moves the pointer image on the screen as the pointing device is moved.

A ROM in the VS100 contains the initial M68000 program. This program is capable of executing start-up diagnostics, reporting device status to the host, and responding to simple commands from the host driver. When the driver determines that the system is operating properly, it commands the VS100 to down-line load its program into local RAM.

The performance accelerator is a hardware device built with 2901 bit-slice components that aids the M68000 in performing certain operations. These operations include rectangle movement, line and curve drawing, and text writing. The accelerator is simply another processor with a faster cycle time and a faster bus to memory. Its program is stored in ROM. Since the 2901s are somewhat harder to program, most general purpose functions are performed in the M68000, while more specific performance-critical functions are executed by the accelerator. The M68000 interfaces to the accelerator through a shared memory; the microprocessor loads the shared memory with parameters describing the command to execute and then initiates the accelerator. Although the M68000 is capable of executing the entire architecture itself (and will do so should the accelerator be absent) there is up to a fivefold performance improvement with the accelerator.

The VS100 processors, both M68000 and accelerator, view a logical address space that is physically divided into several parts. First are the 512 Kbytes of local storage for images. This half megabyte includes the 128-Kbyte display frame buffer from which the screen is refreshed. The remaining 384 Kbytes of memory are used for storing text fonts, icons, images, and so on, as directed by software in the host. This memory can also be used for building images off screen prior to viewing. Second are 128 Kbytes of local storage for M68000 instructions and data. Although both sections of memory are general purpose and can be used for storing either code or images, a division is made so that the M68000 can execute instructions in parallel with accelerator operations that saturate local image memory. The third section of the logical address space is formed by the 18 bits of VAX memory mapped for the device by the Unibus Window.

The M68000 microprocessor and the accelerator module can both read and write VS100 local memory (frame buffer and off-screen memory) and VAX host memory that has been mapped for the Unibus window. Since raster operations are performed on memory, the device can execute any of its output functions on screen, off screen, or in VAX memory. That is, it is possible for VS100 to construct a complex image involving text and graphics within a segment of VAX memory and then move the image to VS100 to make it visible. The input operands for any VS100 operation can also be stored both locally or in VAX memory. In fact, the display processor cannot tell where its operations are performed—it simply operates on the logical addresses supplied as parameters in the command packet. On a system with multiple VS100s, several VS100s can be reading or writing VAX or local memory, performing in parallel with each other and the VAX CPU.

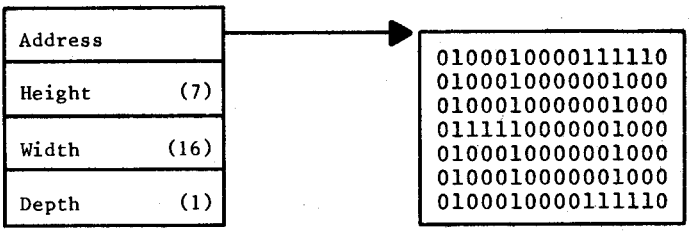


Fig. 2. Bitmap specification.

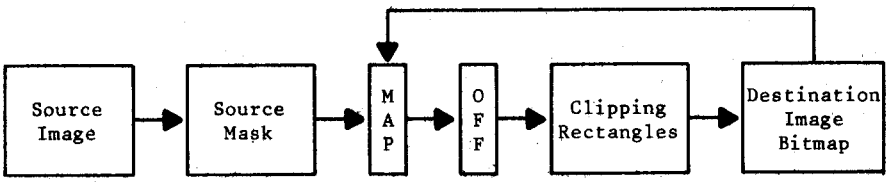


Fig. 3. VAXstation architectural model.

## 3. VAXstation GRAPHICS ARCHITECTURE

The VAXstation architecture defines the set of commands that a VAXstation device must process and the behavior of those commands. Each command is issued by placing the address of the command packet in one of the device's control registers. The message packet contains an operation code, specifying the function to perform, and a set of operands. These operands include addresses of parameters to be read or written by the VAXstation. Command packet addresses are always VAXstation logical addresses, and thus any parameters (and the packet itself) can be stored in either VAX memory or display local memory.

The workstation display executes a set of five basic output commands:

(1) copy area
(2) draw curve
(3) print text
(4) fill area
(5) flood area

Each command is capable of processing a number of different parameter types and formats. In this section we first define the representation of bitmaps on which commands operate, then present a general model of VAXstation operation, and finally describe the commands and their parameters in more detail.

At its most basic level, VAXstation commands operate on *bitmaps*. A bitmap is a memory segment that describes a rectangular image that can be displayed on the monitor. The image is represented by a rectangular array of pixels, where a pixel is a single screen picture element. Each pixel has a *value* that indicates the intensity or color of that picture element when displayed on the screen. For example, Figure 2 shows a bitmap specification and the associated bitmap for a small icon on a 1-bit/pixel machine, such as VS100. The bitmap in Figure 2 is specified by its address, the height and width of the bitmap rectangle, and the depth or number of bits per pixel, which is 1 in this case.

Each VAXstation command generally selects a source bitmap and uses it to modify a destination bitmap in a specified way. However, in the most general case, only certain pixels in the source may be selected to replace destination pixels, and only certain pixels in the destination may be available for modification. The architectural model, shown in Figure 3, indicates how these source and destination pixels are specified. This model is similar to the model used by the Smalltalk graphics system [8].

The architectural model is logically divided into three stages. The first stage, consisting of the source image and source mask, determines a set of source pixels to be used to update the destination. The last stage, consisting of the destination offset (OFF), clipping rectangles, and destination image bitmap, determines what pixels of the destination are available for modification. The map box in the center specifies a function with which source pixel values may be transformed before replacing destination pixels. Alternatively, the destination pixels may be replaced with a function of both source and destination pixels, as illustrated by the arrow leading from the destination to the map. The following subsections describe the parameters more fully as part of the copy area command.

### 3.1 Copy Area Command

The copy area command is the fundamental operation of the display system, and all other output operations are extensions of the basic copy area function. Copy area performs a general BitBlt (bit string block transfer) or RasterOp function, similar to that described in [6–9, 13]. However, copy area differs in the way that the source and mask are defined; it also provides multiple clipping rectangles and two combination function formats. In its simplest form, copy area merely moves a source bitmap to a destination bitmap. Both bitmaps must be in display-addressable memory, which can include the display frame buffer, VAXstation off-screen memory, or VAX host memory, as stated previously.

Assuming that the source and destination bitmaps represent identically sized rectangles, copy area simply replaces destination bitmap pixels with source pixels at corresponding coordinates. In addition to simple replacement, the values moved to the destination can be either (1) a function of the source pixel values, or (2) a function of both source and destination values. Finally, copy area allows for a mask parameter to select some subset of the source to be used and a set of clipping rectangles to restrict the updating of the destination. The following is a list of the copy area parameters and their options:

(1) *Source Image.* The source image parameter specifies the values of pixels used to update the destination bitmap. The source image can be specified in three ways:

(a) In the most general case, the source image can be specified as a bitmap in display-addressable memory. Destination pixels are replaced by source pixels. If the source and destination bitmaps overlap, the copy operation sequences through the pixels so that source pixels are not modified before they are used to update the destination.

(b) A common raster display requirement is to fill a destination area with a single color or intensity. For this purpose, the copy area source parameter can be

a single constant value. The entire destination bitmap, as qualified by the source mask and clipping described below, is set to this pixel value.

(c) Finally, the source can specify a halftone or stipple pattern to be replicated within the destination as specified by the remaining parameters. The halftone is a small (e.g., 16-pixel-square) bitmap. On a monochrome machine, different alternating patterns produce the effect of different grey shades. The halftone bitmap rectangle is replicated horizontally and vertically to fill the destination bitmap. The pattern is always aligned relative to the origin of the destination bitmap.

(2) *Source Mask.* In many cases, only a subset of the source bitmap is used in the copy operation, and the source mask defines the subset. Two mask formats are available, depending on how the source subset is to be determined.

(a) First, the mask can select a rectangular subset of the source bitmap. The subset is specified by its origin (relative to the source bitmap's origin) and a height and width. The selected rectangle is moved to the location in the destination bitmap specified by the destination offset.

(b) Second, the mask can be specified as a *template* of 0 and 1 values (the template is always a binary-valued bitmap independent of the number of bits per pixel in the source). This is the most general form, in which the 1-valued elements of the template select an arbitrary set of pixels from the source to be used in the operation.

Any of the source and mask formats can be combined. For example, combining a constant value source with a rectangle mask generates a bitmap rectangle of the specified size filled with the specified source color. Or, use of a halftone source with a bitmap mask selects a halftone pattern whose shape is determined by the 1 bits in the mask. Thus, if the source is a bitmap, the mask defines a subset of that bitmap to be used; if the source is a constant or halftone, the mask defines the size and shape of an area filled with that pattern. Note that using a template mask, it is possible to extract any arbitrarily shaped image from a bitmap, or to insert any arbitrarily shaped image into a bitmap.

(3) *Destination Image Bitmap and Destination Offset.* The destination image is always a display-addressable bitmap to be modified and is specified as described previously in Figure 2. The destination offset specifies where the source origin should be placed relative to the destination origin. The source can thus be moved to any position within the destination and is automatically clipped to the boundaries of the destination.

(4) *Map.* The map parameter defines a transformation table used to determine the values with which to replace selected destination pixels. This parameter provides what is generally called the RasterOp function [13] on a one-plane machine. The map can have three forms:

(a) First, the map can be null, in which case source pixels directly replace destination pixels (i.e., the identity map).

(b) Second, a *source map* can be specified. On a machine with $n$ bits per pixel, the source map is an array with $2^n$ entries, each entry containing $n$ bits of information. Each source pixel value is used as an index into the map table, the

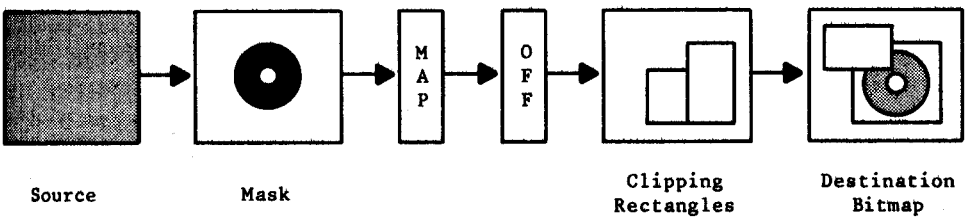


Fig. 4. Copy area example.

value selected being used to replace the corresponding destination pixel. On a 1-bit/pixel machine, for example, the following map would cause the source to be complemented (reverse video) when moved to the destination:

| *source* | *map* |
|---|---|
| 0 | 1 |
| 1 | 0 |

(c) Third, the map can be a *source and destination map*, represented by a two-dimensional array where source and destination pixel values are used as indices to select a new destination pixel value. In the case of a single bitplane machine, for example, this table can be used to express any of the 16 possible logical functions of the source and destination. Some of these functions, however, can be performed more easily using a source-only table or source mask.

For the single bitplane case, the source-destination map is a 4-bit literal. On a machine with $n$ bits per pixel, the map is an array with $2^n \times 2^n$ entries, each entry containing $n$ bits of information.

Although the source-destination map can be expressed as a RasterOp function on the single-plane system, the map concept is in fact more general, since it allows the user to define any meaningful transformation function for multiple bit per pixel machines. For example, the user can define a color transformation for a green source and blue destination. In many cases, however, only a source map is needed. (These maps do not replace the need for a color map to determine the relationship of pixel value to final color.)

(5) *Clipping Rectangles.* A common requirement for the support of a window management system is the ability to constrain output to a subset of the destination. This *clipping* of the output is particularly useful when a program writes to a window that is partially occluded by another. For this purpose, the VAXstation architecture supports multiple clipping rectangles on output commands. The clipping rectangles parameter is the address of a *list* of clipping rectangles (origins plus extents). These clipping rectangles are used to restrict the operation to some subset of the pixels in the destination bitmap. Each rectangle specifies an area of the destination available for modification. Thus, the union of all of the clipping rectangles defines the area of the destination that can be modified. The operation is additionally constrained by the size of the destination bitmap itself.

Figure 4 illustrates a copy area command. In this case, a halftone source is combined with a mask that is a template of a ring. The ring, filled with the halftone, is to be copied to a partially occluded window within the destination

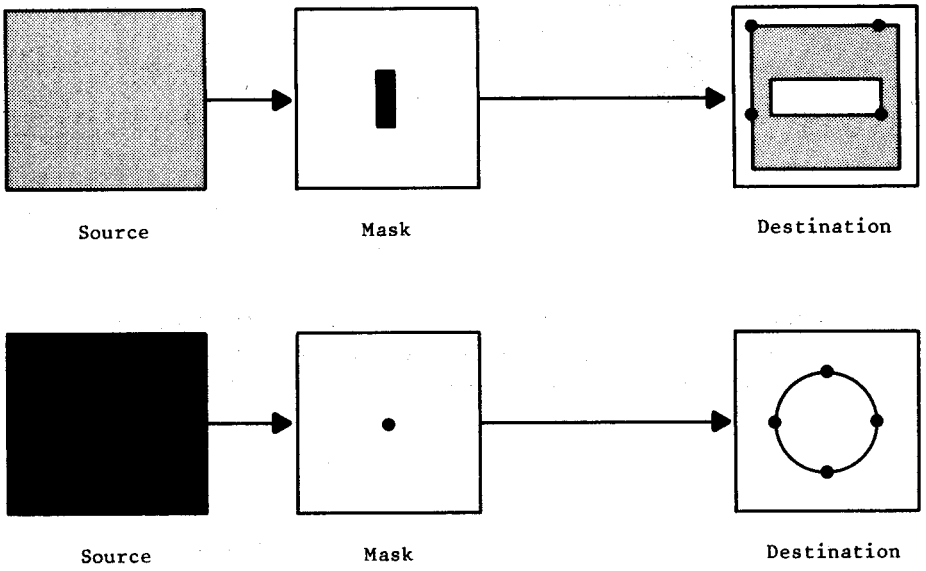


Fig. 5. Draw curve command examples.

bitmap. No map is used. The destination offset for this command would specify the origin of the window to which the ring is being moved, while the clipping rectangles form the nonoccluded partion of that window.

### 3.2 Draw Curve Command

Draw curve is the VAXstation graphics command; it allows drawing of any connected or disconnected graph of curved and/or straight segments. An extension of the basic copy area operation, draw curve accepts all of the parameters specified in the architectural model (Figure 3), plus three additional parameters.

In the draw curve command, the source and source mask parameters specify a shape and color or intensity with which each segment is drawn. To use the standard metaphor, the source mask determines the brush shape and the source image determines the paint with which the curve will be drawn. The curve is then drawn by painting the image, determined by source and source mask, one pixel at a time between the segment endpoints. Conceptually, a copy area of the source is done at each pixel along the path. This allows for drawing of curves with any shaped object in any color. A map parameter can be specified, as in the copy area command, causing the segment to be added to the destination with specified combination function.

The curve to be drawn is described by a list of points, called the path parameter. Each point in the path is specified by a pair of coordinates and a set of flags. The flags contain the following indicators:

(1) A *relative* or *absolute* bit indicates whether the point's coordinates are relative to the previous point in the path list or the destination offset parameter.

(2) A *straight* or *curved* bit indicates whether a straight or curved line should be drawn between the previous and current path points. If a curved line is specified, then the previous point must have a predecessor in the list, and the

current point must have a successor. The curve path is determined by a cubic spline algorithm implemented in the VAXstation display system.

(3) A *draw* or *move* bit indicates whether the segment should be drawn at all, or whether the path should just advance to the next point. This bit allows for specification of additional points needed for curve drawing, and also for construction of disconnected graphs.

(4) A *write* or *skip* endpoint bit indicates whether or not the final point in the segment should be drawn. This is particularly useful when drawing multisegment lines using an XOR map function. In this case, the last point in each segment must be drawn only once; otherwise it will be complemented.

Two additional parameters to the draw curve command allow for drawing of patterned (i.e., dashed or dotted) lines. These parameters specify a pattern string whose bits are scanned to indicate, at each pixel along a segment, whether or not to actually modify the destination. For example, the pattern string 0000111111 will draw a line or curve with alternating 4-pixel spaces and 6-pixel segments.

Figure 5 illustrates the use of the draw curve command. In the first part of the example, a rectangle mask is used to draw straight segments between four points. Since the mask has greater height than width, horizontal lines will be wider than vertical lines. In the second example, the *curved* flag is used to draw a circle from a single-pixel mask (i.e., the circle is drawn with a 1-pixel-wide brush). Although four points define the circle, the first and last points must be specified twice in the path list to provide the tangents for the first and fourth arcs.

Through the use of the draw/move bit, it is possible to draw disconnected figures (for example, several circles) with a single draw curve command. The only restriction is that all lines must be drawn with the same source and source mask (i.e., they must typically be the same color and width).

### 3.3 Print Text Command

Like draw curve, the print text command accepts all of the standard copy area parameters, plus a few additional parameters. An obvious additional parameter is the text string that specifies the symbols to be written. The text string contains indices into a "strike format" [8] font data structure. The font is a constant-height bitmap with variable-width cells; each cell contains a bit pattern representing a character symbol. A table called the "left-$x$" table indicates the starting position, within this bitmap strip, of each of the cells in the font. By looking at the left-$x$ entry for a character and its successor, the VAXstation can determine the width of a particular character cell. As with other parameters, the font can be stored either in VAX host memory or in VAXstation local memory.

Within the framework of the architectural model, the font can be specified in one of two ways: source image or source mask. In the simple case, the font is used as a source image and no source mask is allowed. Each character cell in the font contains the character image plus a surrounding rectangular background. The entire rectangle for each specified character is copied to the destination bitmap. Character and background colors are encoded by pixel values in the font bitmap (although a map can be supplied, as before).

For the second option, the font is specified as the source mask, where each cell is a character template. In this case, each 1-bit/pixel font cell defines the shape

of a character. The source image parameter, generally a constant or halftone, specifies the writing color for the character. Since the character is a mask, no surrounding rectangle is moved to the destination. Use of the font as a mask thus allows overstrikes of other characters and graphics.

The remaining parameters to the print text command are used for justification or special-character string processing and positioning. For simple justification, two parameters, the intercharacter pad and space pad, specify an additional number of pixels to be added following each character or each space, respectively. This allows for a line to be stretched for right justification.

For more complex situations, the command allows a control string to be passed along with the character string. The control string consists of a list of operation codes and parameters indicating how each character in the character string is to be processed. There are four string processing operation codes:

(1) OUT($N$) causes the next $N$ characters of the text string to be output,
(2) OUTALL causes the remainder of the string to be output,
(3) SKIP($N$) skips the next $N$ characters of the string,
(4) ADJUST($X$, $Y$) adds signed $X$ and $Y$ adjustments to the current position before processing continues.

Using a control string, it is possible to output text in any position on the screen with a single command, as long as a single font is used. The SKIP feature allows skipping of special control characters in the text string that may not be represented in the font.

### 3.4 Flood Area and Fill Area

The flood area and fill area commands are used to fill a bounded object with a single color, intensity, or halftone. These commands rely on a somewhat simplified version of the general architectural model. For example, only one clipping rectangle is allowed.

The flood area command operates on a closed region within the destination bitmap. The command specifies a seed point within the destination as the place to start the flood. A boundary map parameter provides a binary-valued table defining the internal and external points of the closed figure in the bitmap. The system begins by examining the seed point and then all of its neighbors. As each point is examined, its pixel value is used as an index into the boundary map. Any pixel value mapping to a 0 entry in the table is an internal point, and any value mapping to a 1 is a boundary point. Internal points are set to the source value and the scan continues. External points cause the scan to stop in the current direction. Processing continues until all internal points have been colored.

Fill area is generally used to construct a source consisting of one or more closed objects, where each object is filled with the source color or halftone. The closed objects are described by a path list; each point in the path list is specified as in the draw curve command. Using the draw/move bit, it is again possible to describe several disjoint objects. The filled shapes are copied to the specified location in the destination bitmap.

The fill command is used when the boundary or boundaries of the area or areas to be filled are known and can be defined by a list of straight or curved segments.

Flood, on the other hand, is used when the boundary is not completely known, but the user can specify one internal point and the pixel values that form the boundary.

## 4. VAXstation MEMORY MANAGEMENT

Management of the VAXstation local memory store is left entirely up to the VAX host processor. When the VAXstation is initialized, ROM-based code in the VAXstation reports its status, including the size and address of its program memory space, the size and address of its frame buffer memory, and the size and address of all additional on-board image memory. All addresses are reported as M68000 virtual addresses. The VAX host then down-line loads the M68000 program into the program memory space and starts it. From then on, the VAXstation simply executes commands. Each command contains some number of addresses that refer to command parameters. The VAXstation is not aware of where those command parameters are located; it merely does 16-bit word reads and writes of memory. Some of those reads and writes occur locally and some require VAX Unibus transfers.

The two memory management commands for the VAX host are the copy area command, which can move bitmaps or parts of bitmaps from memory to memory, and the move object command, which copies a byte string from memory to memory. The move object command is used to transfer fonts, command packets, and other byte-oriented data structures between host and VAXstation memory. It can also be used for moving data structures within VAXstation memory (or for that matter, within VAX memory).

## 5. DISCUSSION

This paper has described the VAXstation graphics architecture and the structure of the VS100, the first implementation of that architecture. The VAXstation 100 is not a stand-alone workstation but a remote graphics device. The architecture, however, is intended for both remote devices and more tightly integrated workstations.

The VAXstation architecture is based on an extended model of the RasterOp function. This model provides a set of general purpose commands for text printing, curve drawing, and rectangle manipulation based on a unified set of hardware primitives. Multiple clipping rectangles are provided to support window management. Flood and fill operations are also included. The general addressing structure allows the device to operate on host memory as well as local memory. This removes some constraints, for example, on the number of supported fonts or windows, that might appear if the device had only a limited local memory.

In developing an architecture, trade-offs are always made between conceptual completeness in the architecture and performance in the final implementation. The VAXstation architecture attempts to provide a consistent model for a set of operations. It can be argued that the overall architecture is somewhat complex in the number of parameters needed to perform operations. When the VAXstation architecture was developed, an intermediate ground was chosen for the level of display system support. For example, although windowing was important, we decided that the first window management system should be built in host

software. The device has primitives for windowing, such as multiple clipping rectangles, but has no concept of window itself.

In retrospect, we should probably have modified the architecture to support at the same time, both lower level and higher level primitives. That is, at the lower level, the device should have extremely simple commands to do the most frequent graphics operations. At a higher level, the device should have some concept of window and be able to move windows, pop windows, and so on. Some experimentation has been done on this and it has been relatively successful.

Of course, within the constraints of the VAXstation architecture, there are many implementation trade-offs. For example, although the draw curve command is very general, it is important to implement the most common options as if they were special purpose. The most typical draw curve command draws a straight 1-pixel-wide segment. If this were implemented as part of the general curve drawing facility, performance of the command would not be satisfactory. In fact, experiments with different codings of the draw curve command showed a sixfold performance range for drawing of simple segments. Therefore, within a very general command, the code must quickly recognize common options and dispatch to special purpose code to handle them.

The most difficult performance problems to solve with VAXstation have been those outside of the device itself, that is, those involving host software. VAXstation device drivers have been built for both the VAX/VMS and Unix® operating systems. In both cases we found that the cost of interfacing to the driver from a user program, which is typically on the order of several milliseconds, quickly surpasses the cost of command processing within the device. We were also surprised to find that the major difference in text performance between fonts in VAXstation memory and fonts in VAX memory was caused, not by the additional access time through the fiber optic and Unibus Window, but by the cost of locking and mapping the font in VAX memory. The conventional device driver approach does not seem to lend itself to displays such as VAXstation because of the ratio of host overhead time to device processing time on small commands.

Since the VS100, several experimental display processors have been built to make integrated workstations from some of the new smaller VAXes. For example, an integrated display has been built for the VAX-11/725, a small VAX in an office workstation package. This display processor is based on the VS100 design but is contained on a single module that plugs directly into the backplane of the VAX-11/725. The fiber optics and performance accelerator have been removed and the Unibus Window logic has been added to the display controller module to meet the single board form factor. The I/O devices connect to a bulkhead on the back of the VAX-11/725 package. Because it supports the same architecture model, support of this device required no change to the VAX-11 host software and only minor changes to the Motorola 68000 software.

ACKNOWLEDGMENTS

As with any effort of this type, a large number of people contributed to the VAXstation design. VAXstation was a joint effort of the Corporate Research Group, the Workstation Group, and the VAX Systems Architecture Group a

Digital Equipment Corporation. The principal contributors to the VAXstation architecture were Stoney Ballard, Kevin Lefebvre, Don North, Craig Schaffert, and Stephen Shirron; the basic structure was motivated by a display device designed for the research group by Stoney Ballard. I would particularly like to thank Stephen Shirron, who implemented several versions of the architecture on the Motorola 68000 and shared his insights. Bob Quinn designed and built the prototype VS100 display hardware.

## REFERENCES

1. Apollo Domain Architecture. Apollo Computer Corporation, North Billerica, Mass., 1981.
2. BALLARD, S., AND SHIRRON, S. The design and implementation of VAX/Smalltalk-80. In G. Krasner, Ed. *Smalltalk-80: Bits of History, Words of Advice.* Addison-Wesley, Reading, Mass., 1983, pp. 127–150.
3. BECHTOLSHEIM, A., AND BASKETT, F. High-performance raster graphics for microcomputer systems. *Comput. Gr. 14*, 3 (July 1980).
4. BitGraph Advanced Graphics Terminal User's Guide and Operating Instructions. BBN Computer Corporation, Cambridge, Mass., 1982.
5. FOLEY, J.D., AND VAN DAM, A. *Fundamentals of Interactive Computer Graphics.* Addison-Wesley, Reading, Mass., 1982.
6. GOLDBERG, A., AND FLEGAL, R. Pixel art. *Commun. ACM 25*, 12 (Dec. 1982), 861–862.
7. GOLDBERG, A., AND ROBSON, D. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, Reading, Mass., 1983.
8. INGALLS, H.H. The Smalltalk graphics kernel. *Byte 6*, 8 (Aug. 1981).
9. KAY, A., AND GOLDBERG, G. Personal dynamic media. *Computer 10*, 3 (March 1977).
10. LANTZ, K.A., AND RASHID, R.F. Virtual terminal management in a multiple process environment. In *Proceedings of the 7th Symposium on Operating Systems Principles* (Pacific Grove, Calif. Dec.), ACM, New York, 1979.
11. MCCROSSIN, J.M., O'HARA, R.P., AND KOSTER, L.R. A time-sharing display terminal session manager. *IBM Syst. J. 17*, 3 (1978).
12. MEYROWITZ, N., AND MOSER, M. BRUWIN: An adaptable design strategy for window manager/virtual terminal systems. In *Proceedings of the 8th Symposium on Operating Systems Principles* (Pacific Grove, Calif. Dec.), ACM, New York, 1981.
13. NEWMAN, W.M., AND SPROULL, R.F. *Principles of Interactive Computer Graphics.* McGraw-Hill, New York, 1979.
14. PERQ. Three Rivers Computer Corporation, Pittsburgh, Pa., 1980.
15. SPROULL, R.F. Raster graphics for interactive programming environments. Tech. Rep. CSL-79-6, Xerox Palo Alto Research Center, Palo Alto, Calif., June 1979.
16. TEITELMAN, W. A display-oriented programmer's assistant. Tech. Rep. CSL-77-3, Xerox Palo Alto Research Center, Palo Alto, Calif., March 1977.
17. THACKER, C.P., MCCREIGHT, E.M., LAMPSON, B.W., SPROULL, R.F., AND BOGGS, D.R. Alto: A personal computer. In *Computer Structures: Principles and Examples*, Siewiork, Bell, and Newall, Eds., McGraw-Hill, New York, 1982, pp. 549–572.

Received April 1983; revised November 1983; accepted March 1984