YODA: An advanced display for personal computers

by Satish Gupta
David F. Bantz
Paul N. Sholtz
Carlo J. Evangelisti
William R. DeOrazio

YODA (the YOrktown Display Adapter) is an experimental display designed to improve the quality and speed of users' interactions with personal computers. This paper describes the YODA hardware architecture and software design. Special attention is given to techniques used for antialiasing. The various trade-offs and decisions that were made are discussed.

Introduction

Personal computing has revolutionized the computer industry during the past five years. A key reason for this revolution is the high degree of interactivity which results when a processor is dedicated to a single user. The proximity of the input/output system to the processor further contributes to this interactivity. Among the most important elements of the workstation input/output system are the display medium and the architecture of the display controller. The display-controller architecture determines the kinds of images (computer-generated or scanned) that can be displayed, and the speed with which these images can be changed. YODA (the YOrktown Display Adapter) [1] is an experimental display designed to improve the quality and performance of this interaction.

The computer graphics industry has developed workstations and applications that use high-quality color images [2, 3]. Examples of such applications are presentation graphics, VLSI design, computer-aided design, computer art, and satellite-image mapping. Researchers have developed both hardware architectures and software techniques to improve the feasibility of these applications. Franklin Crow, then of the University of Texas, first described a technique known as antialiasing [4], which is a rendering method resulting in higher visual quality through the elimination of sampling artifacts. Antialiasing smooths the "staircases" or "jaggies" commonly found in raster displays. John Warnock first applied this technique to the problem of displaying high-quality text [5].

At IBM Research, we have been working both on architectural techniques to improve the performance of workstation displays and on better algorithms for antialiasing of text and graphical objects [6]. Our initial tool for this work was a host-connected, color raster display of very limited interactivity. We felt a need for a personal display which would permit each of us to work individually, and we needed highly interactive displays to support our productivity as well as applications requiring dynamic images.

YODA is the result of our effort to build a display subsystem for personal computers capable of the dynamic display of images of moderate complexity, and of the static display of images of the type formerly regarded as the exclusive domain of very expensive systems. We developed considerable respect for the capabilities of the IBM Personal Computer for this kind of work, when it is properly complemented by a display subsystem. Another pleasant

surprise was the exceedingly diverse nature of the applications we and others were able to build on this base.

YODA consists of a two-card display controller (the cards plug into a standard IBM PC), microcode to run in the controller's microprocessor, an analog RGB color CRT monitor, and IBM PC programs to create high-quality color images rapidly. The display controller contains a frame buffer and a microprocessor with its control storage. The microprocessor is coupled to the frame buffer through special hardware that allows efficient implementations of display-update algorithms.

YODA differs from other display controllers for the IBM PC primarily in its hardware and software support for antialiasing. Some products can display antialiased pictures, but do little to support their synthesis.

YODA has more bits per pixel in its frame buffer than most display controllers, but, more importantly, more bits in its color specification. This provides the precise color control needed for antialiasing. The choice of a fast, simple display processor provides a combination of programmability and good performance. Programmability is essential to allow experimentation with the pixel combination function, which also demands high performance because it is executed once for each pixel to be modified. YODA's programmability was also necessary to support the experimentation that determined how function should be divided between the PC and the display processor.

For cost reasons, commercially available display controllers for the IBM PC generally omit the display processor and limit the range of displayable colors. Typically, these systems allow eight to sixteen simultaneously displayable colors chosen from a range of 64. The PC processor must do all image synthesis. The IBM Professional Graphics Adaptor is an exception, but its display controller is not fast enough to perform arithmetic operations on each pixel with adequate speed to support interaction. Rather, a primary function of its display controller is the interpretation of a high-level picture-description language.

The YODA hardware design had to address the issue of image quality and update performance in relation to hardware cost. Higher image quality requires greater resolution (more displayed pixels), more choices for the color of each pixel, and more precise specification of colors. Update speed depends on the total number of pixels updated, the complexity of the update, and the speed of the display processor. We had to strike a complex balance to achieve the required speed, quality, and cost. The two most important choices were the use of a television-rate CRT with interlaced scan to get moderately high resolution at low cost, and the use of a simple but very fast microprocessor to update the display. The choice of television-rate refresh had other benefits too. We have been able to take advantage of inexpensive CRT monitors, projection displays, slide-makers, and the like, and we have the potential to integrate our system with television imagery.

In this paper, we present the concepts behind the YODA hardware and software, explaining the various decisions that we faced and the trade-offs that we have made.

The remainder of this paper is divided into three major sections, followed by a discussion and conclusion. The first of these sections discusses the hardware design, with examples of microcode loops illustrating the use of the display processor. These inner loops were the most important factor in determining the hardware architecture. The architecture went through several design iterations before we were convinced that the hardware supported these loops adequately.

Since antialiasing is such an important aspect of our software design, we have dedicated all of the second major section to it. After a brief tutorial on that topic, this section discusses the two issues that we focused on in our software: Our software consistently depicts the display as a device with higher addressability than the number of pixels on the screen, resulting in higher picture quality. We use antialiasing to achieve this effect. Antialiasing in color is also a significant problem, and we describe our solution.

The third major section describes the YODA software. There were two primary issues in the design of the YODA software: the functions the software should provide and their implementation. In deciding what functions to provide, we chose primarily to program functions that would perform efficiently on the hardware. Dividing the work between the PC processor and display microprocessor was difficult because different models of the PC use different processors with significant differences in their performance. In a number of cases, we implemented the inner loop of a function in display-processor microcode, and the outer loop in the PC.

The paper concludes with a discussion of our experience with YODA. We point out its merits and its deficiencies, and discuss how we would improve it in a future design.

YODA hardware

Image quality for a CRT display is determined by three factors. The *resolution* determines the total quantity of information as well as the amount of detail that can be seen on the display. The number of bits used to represent each pixel determines how many *colors* or *intensity levels* can be seen on the display. Intensity levels can be used to enhance the picture at a given resolution by the use of antialiasing. The *refresh rate* (i.e., the number of times per second that the screen is refreshed) together with CRT phosphor characteristics determines the degree to which the picture is free from flicker.

Low-cost CRT monitors and television compatibility led us to use television line rates in refreshing the CRT. Just as in television, we chose to refresh the CRT in an *interlaced* scan, resulting in twice the vertical resolution. An annoying

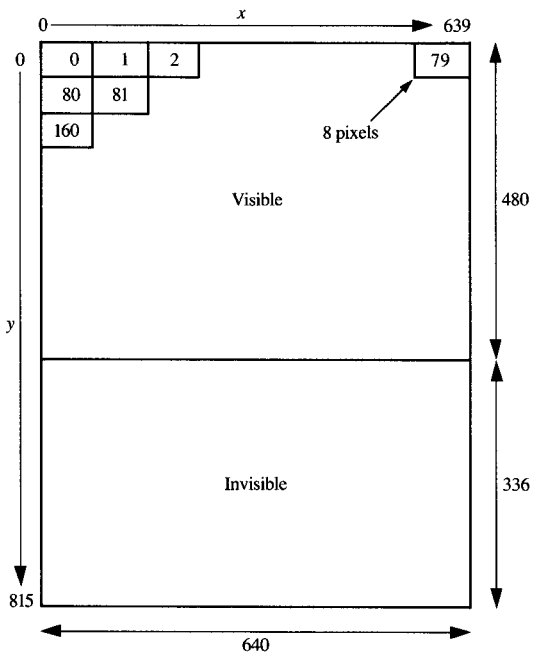

Figure 1

The YODA frame-buffer address map. Eight horizontal pixels map onto one memory address, as indicated.

side effect of this decision is that objects that are only one scan-line high are refreshed too slowly, producing a distracting flicker. We chose a monitor with a long-persistence phosphor in order to minimize this flicker. To maintain the conventional 4:3 CRT aspect ratio, a display resolution of 640 by 480 pixels was selected.

Choosing the number of bits per pixel is a difficult problem. That number determines not only the image quality but also the cost of the frame buffer. Since we wanted to present a large number of colors on the screen as well as retain a few bits for use by antialiasing, using anything less than eight bits per pixel seemed uninteresting. Using more than eight bits per pixel was too expensive—especially since that would have taken more than two PC-sized cards to implement. The compromise was to use eight bits per pixel with a video lookup table.

• *Video lookup table*

The video lookup table (VLT) is a table in the video path from the frame-buffer memory to the CRT. This table translates each pixel value into the red, green, and blue values used by the CRT to determine the color of that pixel. A loadable table allows dynamic choice of color for each pixel value. A larger VLT output (i.e., more bits for the red, green, and blue intensities) allows a finer selection of colors to be assigned to a pixel. We chose 24 bits in the output of the VLT—eight bits for each of the red, green, and blue intensity values. Such a VLT allows YODA to present images which require fine color choices but which do not require more than 256 unique colors, the limit imposed because each pixel is represented by only eight bits. An example that illustrates this is a picture of a sunset where there are fine gradations of red and yellow but almost no blue. The VLT allows all 256 displayable colors to be assigned in the red-yellow region. Given this ability to assign colors to the parts of the spectrum where they are needed, 256 different colors is usually adequate. Experiments based on the theoretical models of the human visual system have shown that eight bits per pixel is sufficient for most color images [7].

The video lookup table is a very powerful mechanism which is useful for a wide variety of applications, including animation and antialiasing.

A commonly used animation method is to provide multiple image buffers which are shown sequentially with successive images. The VLT can be used to divide the frame buffer into several buffers with fewer planes (e.g., four buffers with two bits per pixel). If the VLT is loaded in such a way that the color of each pixel depends only on the bits in one of the buffers, then that buffer is rendered visible and the other buffers become invisible. To furnish the animation, the VLT is reloaded with new values which cause a new buffer to become visible. The invisible planes of the frame buffer can be updated while the visible planes are displayed. The reloading of the VLT is accomplished during a single vertical retrace interval, preventing any visible disturbance during the changeover.

Antialiasing eliminates the staircase sampling artifacts by using intermediate intensities along the edges of objects. The VLT must provide sufficient intensity resolution to permit specification of these intermediate color values. We have found 24 bits in the VLT output to be adequate for this purpose. We describe the exact use of the VLT for antialiasing in the next major section.

• *Frame-buffer organization*

We chose to use eight 64K-bit dynamic RAMs to implement each plane of the frame buffer. The frame buffer of YODA is time-multiplexed between CRT refresh and update. Since the RAMs can cycle every 280 nanoseconds, and video refresh requires eight pixels every 560 nanoseconds (70 nanoseconds per pixel), every other memory cycle is available to the display processor. The choice of television-rate interlaced refresh was important to allow adequate bandwidth for update.

Eight chips per plane gives us a total of 512K pixels for the frame buffer, although only 300K pixels are visible on the screen (640 by 480). The extra invisible frame-buffer memory helps in two ways. It increases the memory bandwidth available, allowing faster update access, and also

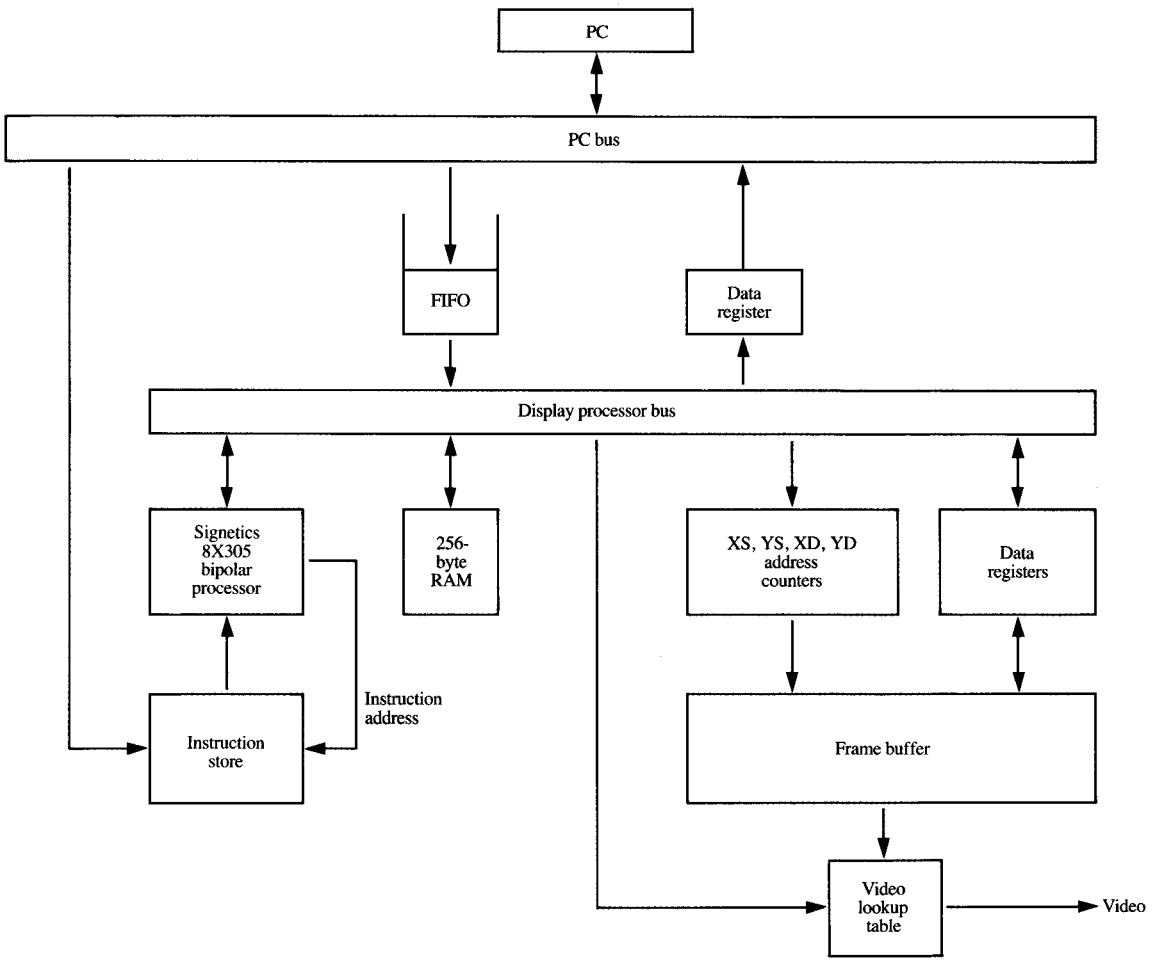

Figure 2

The YODA display architecture.

provides buffer storage for temporary images, fonts, and other tables. **Figure 1** shows the mapping of the (x, y) location to the memory address of the 512K pixels of the frame buffer. Because eight pixels are accessed in parallel, eight pixels map onto the same memory address. A single ROM chip maps the (x, y) coordinates to the memory address using the equation

$$\text{Memory address} = 80y + (x/8).$$

The data-input wires of all eight memory chips forming one plane are tied together. This means that the display processor normally updates one pixel at a time, but a special mode, called *slice mode*, allows eight pixels to be simultaneously updated with the same value. This is useful for writing or clearing large areas. A mask register allows selective suppression of writes to the chips. This is valuable for writing characters onto a fixed background, for example where one wishes to write the *1* bits of the character but to leave the background intact where the character has *0* bits. In this case, the character data are used as the mask, and the pixel value representing the foreground color is used as the value to be stored.

• *Display processor*

The display processor consists of an eight-bit single-chip microprocessor, its data and instruction storage, and its interfaces to the PC and to the frame buffer, as shown in **Figure 2**. The microprocessor is a Signetics 8X305 bipolar

```
LOOP    DEC     COUNTLOW        Decrement the lower byte of the width
        BNC     LOOP1           If positive, move a pixel
        DEC     COUNTHIGH       Decrement the higher byte of the width
        BNC     LOOP1           If positive, move a pixel
        JMP     OUTER           We are done with a horizontal line
LOOP 1  READ    FBXSINC         Read the pixel; post increment XS
        WAIT                    Wait for the read cycle to finish
        LOAD    FBDATAOUT,ACC   Load the pixel into accumulator
        STORE   ACC,FBDATAIN    Store the pixel into FB data register
        WRITE   FBXDINC         Write the pixel; post increment XD
        JMP     LOOP
OUTER   --      --
```

Figure 3

BitBlt loop.

processor having an execution time of 210 nanoseconds per instruction. The design objective of the display processor was to enable efficient implementation of the inner loops of the frequently used raster-graphics algorithms—that is, to minimize the number of frame-buffer-memory update cycles required for these algorithms. This objective was achieved by using a simple and fast microprocessor and augmenting it with an appropriate frame-buffer update interface.

The 8X305 is primarily intended for controller applications. It has an eight-bit internal data path and can be attached to peripheral devices by means of an eight-bit bus. It is designed to operate at a speed of 200 nanoseconds per 16-bit instruction, with instructions fetched on a dedicated bus for higher throughput. It has a powerful but simple instruction set with only eight instruction classes. Most important to us was its ability to perform a variety of bit-manipulation tasks in a single instruction.

The choice of an eight-bit microprocessor made the manipulation of ten-bit frame-buffer addresses rather cumbersome. This motivated the use of external address counters to increase the efficiency of the graphics inner loops. The frame-buffer address interface contains two pairs of (x, y) address counters (increment/decrement only). They are called XS, YS and XD, YD and are used to address the pixels of the frame buffer that are being read or written (source or destination). When the processor initiates a read or a write cycle, the address registers can be independently incremented or decremented, or left alone, after the operation. Thus extra processor cycles are not spent in changing the address registers in the middle of incremental loops. Graphical primitives such as lines, circles, and area fills increment or decrement the address in the inner loop. The BitBlt loop (**Figure 3**), which copies one part of the frame buffer to another, illustrates this. BitBlt is an abbreviation for bit-block transfer, also sometimes called RasterOp.

The frame-buffer operation is asynchronous with the processor operation because of the continuous video refresh. The processor can initiate a memory cycle, which can take anywhere from one to three processor cycles to complete. If another cycle must be initiated or the data must be read immediately, then a WAIT instruction can be used for synchronization. It should also be noted that the use of an eight-bit microprocessor makes such operations as loop counting awkward, because the screen resolution is greater than 256.

The PC interface allows the PC to communicate commands and associated data to the display processor. It also permits the display processor to return data and status to the PC. It is desirable for the PC and the display processor to be able to run asynchronously and synchronize when desired. A 64-byte FIFO (first-in-first-out buffer) is used to route the commands and data from the PC to the display processor. This allows the PC to issue commands to the display processor and continue asynchronously. The PC can test the status of the FIFO to determine whether it is empty. A single register is sufficient for the PC to receive data, because this path is less frequently used.

The display processor can load the video lookup table by setting up the VLT address and the red, green, and blue values, and issuing a VLT write command. The CRT video signal must be disabled before this command is issued. To avoid visible disturbances on the screen, it is desirable to load the VLT during horizontal or vertical retrace times. If the loading of the VLT is allowed to extend over several frame intervals, then video will be generated partly from old VLT data and partly from new data during the transition. Depending on the particular data being used, this inconsistent state may produce distracting intermediate images. The PC, however, is unable to deliver all 768 bytes (256 each for red, green, and blue) of the VLT data during a single vertical retrace interval. The problem was solved by storing the VLT data temporarily in the invisible part of the frame buffer and then activating a microcode routine to reload the VLT in one retrace period. The display processor just makes it!

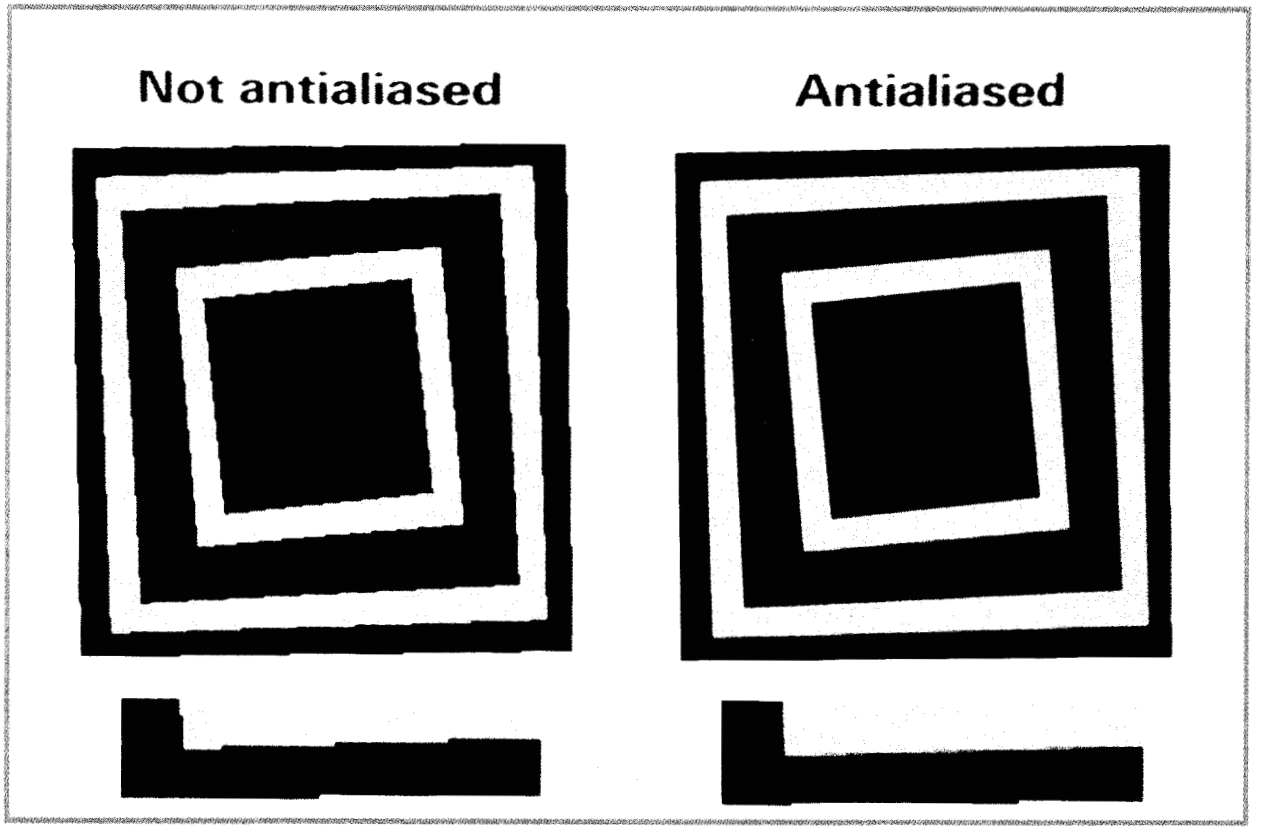

Figure 4

Antialiasing. The use of intermediate intensities can smooth the "jaggies." The bottom of the image shows the results of magnifying the edges from the images above.

Antialiasing

Bitmap images show annoying visual effects in the form of jaggies or staircases when edges or lines cross rows or columns of pixels. On a display device that can present gray-scale images, the effect can be avoided by smoothing the jump through the use of values of intermediate intensity (**Figure 4**). Crow [4] and Catmull [8] were among the first to recognize that this problem is a manifestation of the phenomenon of *aliasing*.

In brief, a computer stores a signal by sampling it and later reproducing the samples. The jagged effect is due to the sampling of a sharp edge over a fixed grid. The original edge contains frequencies higher than those that can be faithfully reproduced by the samples (Nyquist's theorem). This results in higher frequencies appearing as lower ones; the error is called *aliasing*. **Figure 5** shows high- and low-frequency signals which, when sampled, result in the same sample values. Low-pass filtering of the signal prior to sampling prevents the aliasing by removing the offending high-frequency components. As a result, this preprocessing step has become known as *antialiasing*. **Figure 6** illustrates this

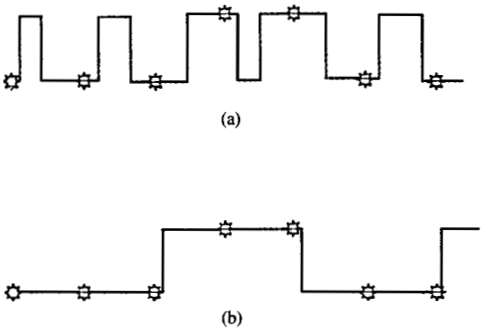

Figure 5

The aliasing phenomenon. The high-frequency signal (a) is sampled at a rate which is too slow. The samples are the same as those from the low-frequency signal (b), resulting in aliasing.

situation in one dimension. Part (a) shows the sampling of the unfiltered signal, and Part (b) shows the sampling of the

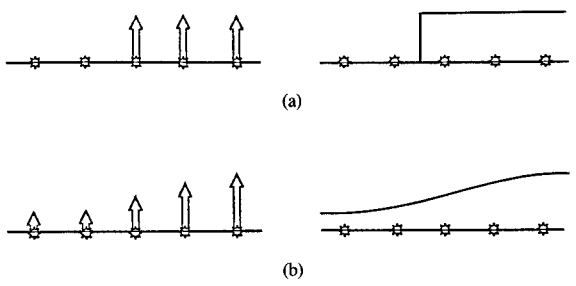

Figure 6

Signal filtering: (a) Binary intensity values which produce the unfiltered signal at right. (b) Varied intensity values which produce the signal at right.

filtered signal. As illustrated by the figure, the filtered samples require the ability to display intermediate intensity values. For the proper appearance of the resulting image, the filter must be appropriately chosen.

Antialiasing removes the visual artifacts of raster display, presenting a more desirable image to the viewer. As a result, small, detailed objects that were previously overshadowed by aliasing artifacts are now presented realistically. **Figure 7** shows magnified small characters (the character "a" is only 6 pixels high!), where one can even tell something about the style of the typeface. To simulate the actual appearance of these characters, the reader should view the page from a distance. The ability to show smaller objects gives us an effective increase in screen content. We can thus show more characters on a lower-resolution display.

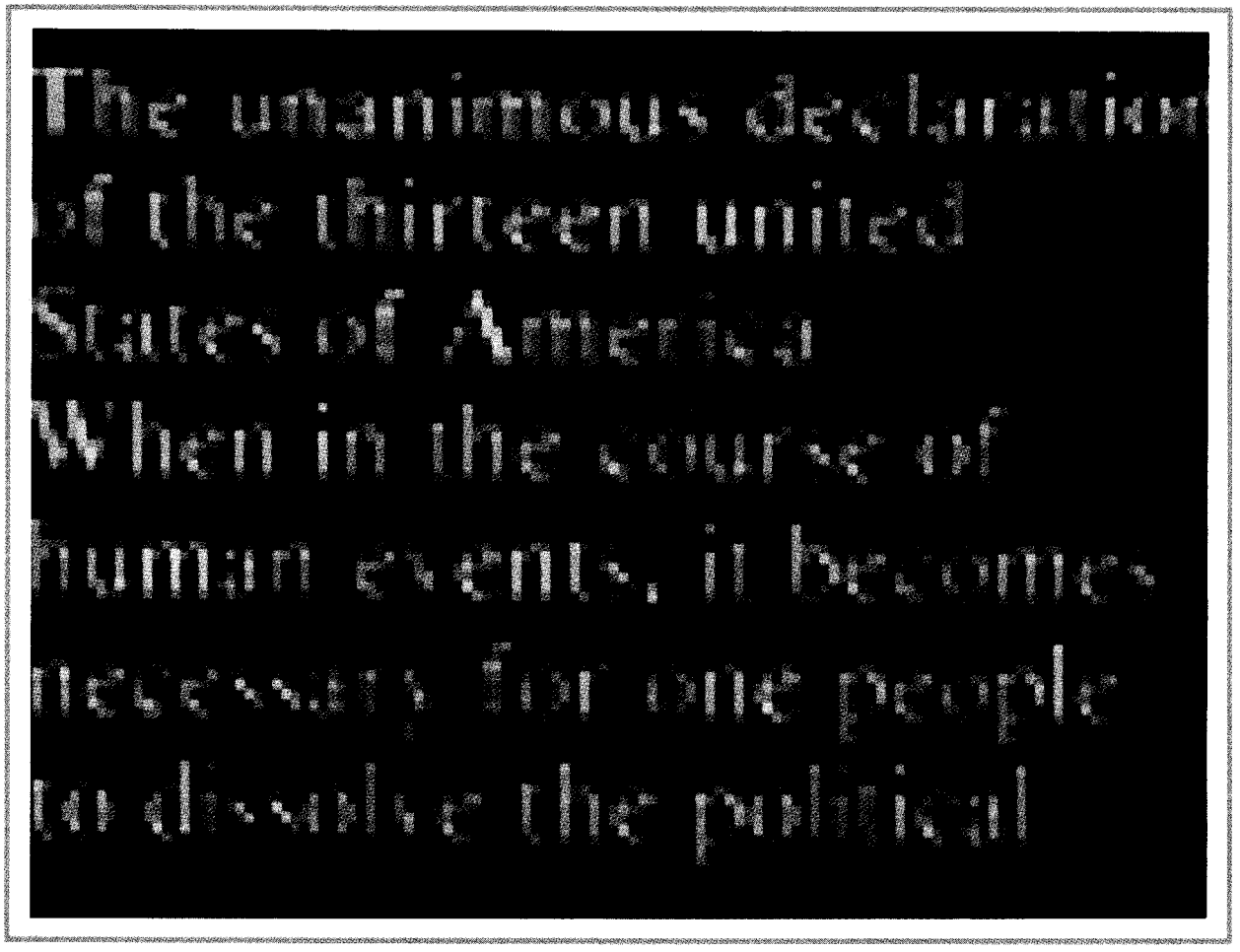

Figure 7

Magnified small antialiased characters. Viewing this image from a distance will show the characters' appearance on the display.

• *Addressability > resolution*

In addition to improving the visual quality of images, antialiasing also increases the effective addressability of the display. **Figure 8** shows how gray scale allows the end point of a line to be positioned at subpixel positions. Other examples that use the improved addressing are lines with noninteger thickness, noninteger intercharacter spacing, and subpixel movements resulting in smoother animation.

Antialiased images in this higher addressability can be created in two different ways. One method is to filter the actual geometric description, computing the intensity value of each pixel. But there are also many situations in which we desire to transform pre-antialiased objects in the higher-resolution coordinate system. Examples of such transformations are translation, scaling, and rotation. In this paper, we illustrate the benefits and the techniques of subpixel translation.

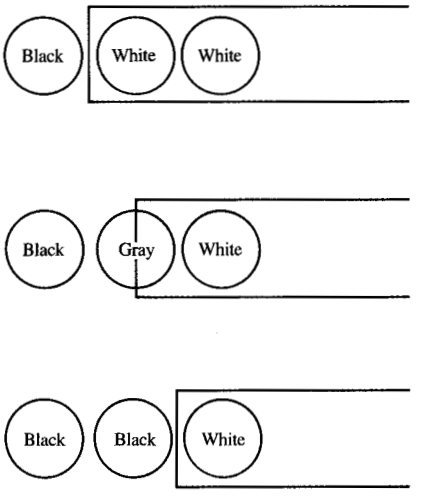

Figure 8

Subpixel positioning of the end of a line. The three horizontal lines have different end points. The use of gray intensity values can show subpixel end points.

• *Subpixel translation*

Antialiasing, as we have seen, allows us to show high-quality characters on low-resolution displays. The antialiased characters can be generated by filtering high-resolution rasterized characters from devices such as phototypesetters. The filtering process assigns an intensity value to each low-resolution pixel which is a weighted average of the high-resolution bilevel pixels underlying the low-resolution pixel. These filtered characters can then be displayed on a low-resolution display.

Such a scheme gives rise to a problem. High-resolution printers use fonts designed for their resolution, and also use width and spacing information in the high-resolution units. These high-resolution quantities do not translate to integer values at the lower resolution. Hence character-writing commands to the display result in attempts to position characters at subpixel positions. A simplistic approach which merely rounds the subpixel position to an integer value is not sufficient. The error caused by rounding one character's position up and the next character's position down can result in a total error of nearly one pixel in the character spacing. This is readily noticeable (**Figure 9**).

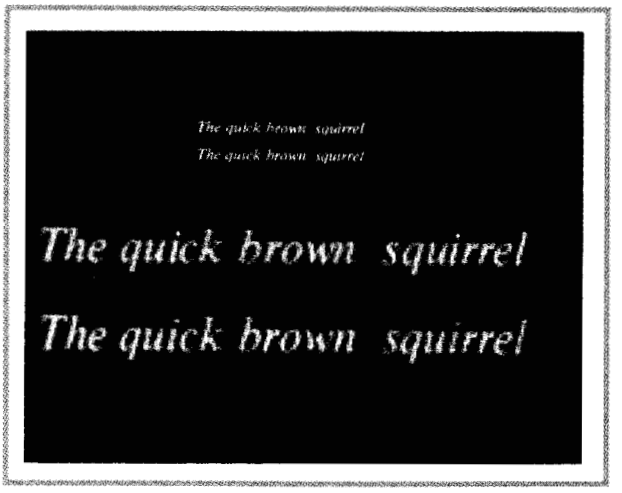

Figure 9

Improved character spacing by the use of subpixel positioning of characters. The top line rounds the character positions to the nearest pixel location. The second line positions each character to subpixel positions. The third and fourth lines are magnifications of the first two.

One possible solution to the subpixel positioning problem is to precompute several different filtered-character definitions and use the one appropriate for the positioning desired. A total of 64 different character definitions would be required to emulate a printer with eight times the resolution in each dimension. A tremendous space saving could be achieved if we could store only one definition for each character and then move it by subpixel distances to obtain the other character definitions.

Translation by subpixel distances is equivalent to shifting the sampled signal by a distance which is not a multiple of the sampling rate. This problem can be solved by reconstructing the original signal and sampling it again at the shifted intervals. If the reconstruction filter, ideally the function $\sin(x)/x$, is approximated by the triangular function, then the resampling at points between the two pixels can be done by linearly interpolating between the

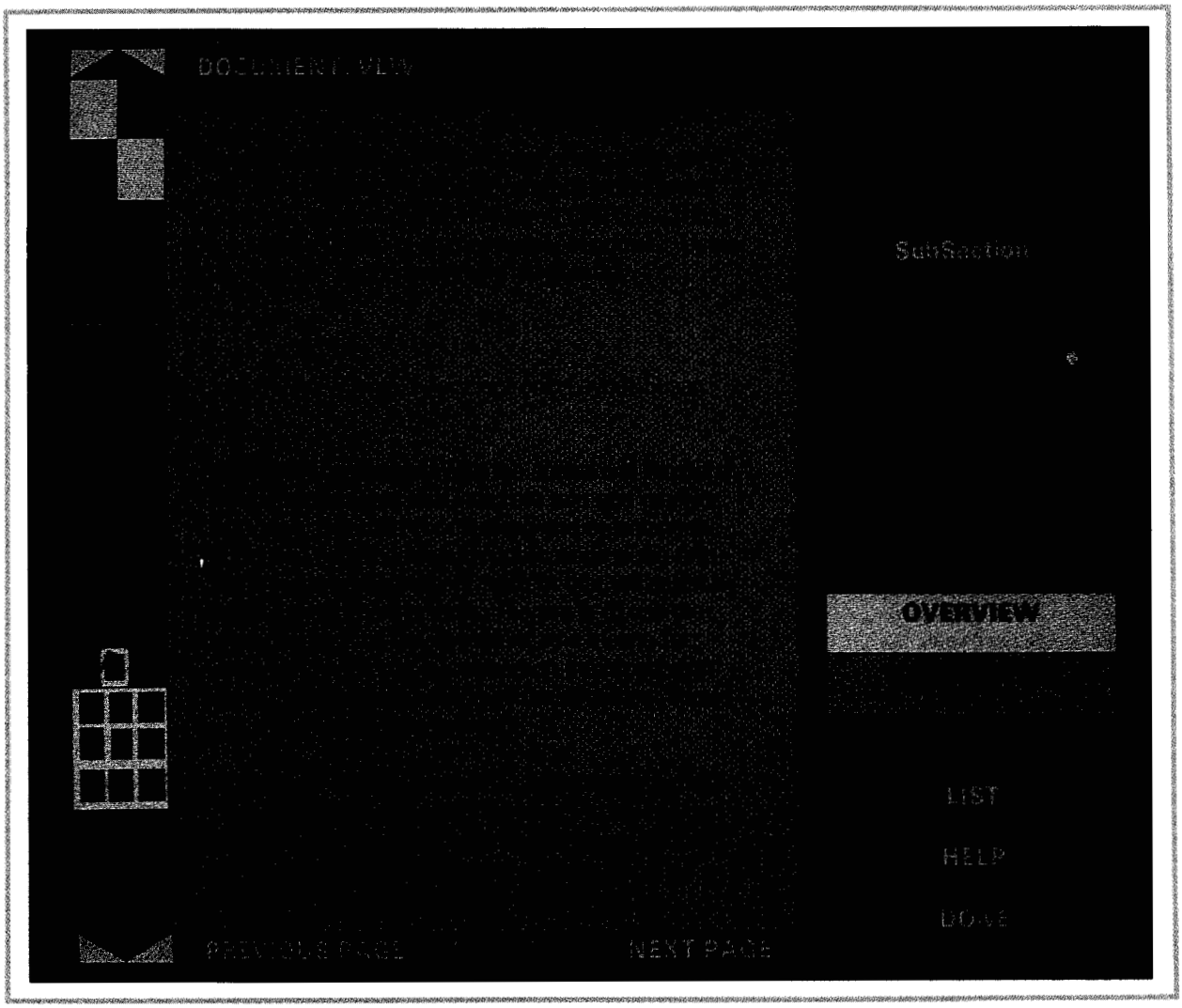

Figure 10

An image from "Conform: A Context for Electronic Layout" [9].

neighboring pixels. In the two-dimensional case, the shifting in both dimensions involves a linear interpolation among four adjacent neighbors. Linear interpolation causes a loss of information, because the triangular filter does not have an ideal low-pass frequency response. Successive interpolations will blur the image because of further loss of high-frequency information. However, the first few successive interpolations give satisfactory results, and in this case we need only one.

• *Antialiasing in color*

Antialiasing in color requires the display to show all the colors in the image, as well as the colors which are intermediate between the background and foreground colors along all edges. A display which uses eight bits to represent each pixel, such as YODA, can show a maximum of 256 colors simultaneously. We experimentally found that 14 intermediate colors in addition to the two edge colors resulted in satisfactorily antialiased edges. No significant improvement was achieved by using more than 16 colors; the use of fewer than 16 resulted in jaggedness. This constrains YODA to display at most 16 antialiased-edge color pairs. **Figure 10** shows a number of color pairs used in antialiasing.

For each color pair, 16 entries of the video lookup table are used. These contain the colors of the two edges at the ends and the 14 interpolated colors between. **Table 1** gives

the VLT entries that make up one such color range (yellow over dark blue). Notice that we have chosen to interpolate colors by independently interpolating the red, green, and blue intensities.

The design described here permits antialiased objects to be drawn in any combination of 16 different colors; these could be used as eight object colors on eight different backgrounds, or 15 different object colors on a single background. In any case, the programmer must know what color pair is being used. The more general case in which antialiased objects are drawn over a mixed-color background of arbitrary complexity cannot be handled by an eight-bit display, because of the potentially large number of intermediate colors required.

The color range between foreground and background colors just described assumes that there is a linear relationship between the color intensity value stored in the VLT and the luminance of the display screen. In most television systems, this relationship is intentionally designed to be nonlinear. In the NTSC system used in the United States and Japan, the relationship is exponential. The YODA software corrects this nonlinearity by means of compensation tables for each of the red, green, and blue signals. This compensation is applied to each entry in the VLT at the time the VLT is loaded. Standard tables embodying the NTSC convention are supplied with the YODA software, but a user may substitute alternative compensation tables if these are desired.

YODA software

The application programming interface (API) [10] for YODA allows applications programmers to create and manipulate high-quality images. This software is motivated by high performance, the highest image quality, and an appropriate functional split between the PC and the display processor. Our API operates entirely in the YODA coordinate system; we believe that normalized device coordinates should be provided by a higher-level software structure if they are needed. We anticipated that applications programmers would want functions other than the ones we have supplied, and we have provided a mechanism for extensibility.

The API consists of three parts. Almost all the performance-sensitive routines are *microcoded* in the display processor. A language-independent *resident extension* of the PC/DOS operating system calls the microcode routines and provides all the functions available to the applications programmer. An application program calls the API functions by linking to one of a set of *language bindings* (C, Pascal, Fortran). The language bindings call the resident DOS extension through software interrupts. This design makes it possible to call the API functions from a variety of different programming languages; when adding a new language, only the language bindings need to be changed.

Table 1 Video lookup table for antialiasing yellow over dark blue.

Pixel value	*VLT output*		
	Red	*Green*	*Blue*
48	0	0	105
49	17	17	98
50	34	34	91
...	...	...	...
61	221	221	14
62	238	238	7
63	255	255	0

- *Pixel map*

A frame-buffered display is sometimes thought of as an array of pixels, each of which has a fixed number of bits to represent its value. Alternatively, such a display may be viewed as a set of planes, each an array of pixels with single-bit values. We sought a general framework for allocating the YODA frame buffer, so that the programmer could handle these different (potentially overlapping) views of the frame buffer in a consistent way. The pixel map concept provides that framework.

All image-update operations are confined to a pixel map, which is a rectangular volume in image space. It is completely defined by the x, y, and z coordinates of its origin, and its dimensions: width, height, and depth. The z coordinate is interpreted to mean the bit-plane number, which in YODA has a range of 0 to 7. It follows from this concept that from one to eight distinct pixel maps may be defined in any given x–y area. The z coordinate gives the plane number of the first plane, and the depth is the number of planes in the pixel map. For example, a pixel map may be defined as four planes beginning at plane 2; this pixel map may contain pixel values from 0 to 15.

A pixel map is analogous to a variable in a programming language. It is created, it is passed as a parameter to functions, and it can have new values assigned to it by functions. A line-drawing function, for example, creates new pixels in the pixel map by drawing the line. A pixel map does not have to be visible on the screen and does not even have to be in the visible part of the YODA frame buffer. It could be in the invisible part of the frame buffer or in the PC memory. Pixel maps can be freely copied into other pixel maps regardless of where they reside. Not all image-update operations are currently implemented for PC-resident pixel maps.

The *OpenDisplay* function returns two pixel maps: one for the entire visible portion and another for the entire invisible portion of the frame buffer. PC-memory pixel maps are created by the *CreateHostPixelMap* function. Lower-level pixel maps can be defined as subsets (in the x, y, and z axes) of existing pixel maps.

A further implication of the pixel-map concept is that all x, y coordinate pairs given as arguments to various update functions are interpreted relative to the pixel-map origin. This allows the user to create an image in a pixel map as if the image had an origin of 0, 0. The pixel map also intrinsically provides a direct method for handling clipping, since pixel maps have explicit extents in all three dimensions. All operations which use pixel maps clip to the destination pixel map.

A good example of operations on pixel maps is the BitBlt function. The term BitBlt was originally used in connection with single-plane displays in which each pixel was represented by a single bit. In YODA the term is used to describe the block transfer of pixels, without regard for the number of bits used for each pixel. The BitBlt functions are the most basic operations in frame-buffer displays and are used to manipulate the image in a variety of ways. In its most general form, the BitBlt operation combines (in accordance with some *combination function*) the pixels of a rectangular source area with the corresponding pixels of a destination area, and stores the result in the destination pixels. This can be simply called by BitBlt(SourcePixelMap, DestinationPixelMap, CombinationFunction).

- *Antialiased text*

Antialiased text was a primary focus of attention in the YODA software effort. YODA can display almost any text that can be printed by printers at IBM Research. Printer fonts are filtered to create the low-resolution YODA fonts. Characters are displayed using 16 intensity levels and positioned to the nearest eighth of a pixel, in an attempt to achieve quality comparable to that of printer output.

All coordinates used in function calls for antialiased primitives use a 32-bit coordinate type called "dblcoord." This coordinate is a 32-bit fixed-point number having a 16-bit fraction. Its units are pixels. The 16-bit integer part defines the pixel offset from the origin of the pixel map. The 16-bit fraction defines a fraction of a pixel. Only eight bits of the fraction are used by the API internally, thus allowing a position to be specified to a precision of 1/256th of a pixel. Coordinates are carried to this precision to prevent the accumulation of roundoff error in character positioning. Using the precision of 1/256th of a pixel yields a maximum error of less than one pixel across a 128-character line. The character-display microcode rounds the 1/256th subpixel position to the nearest 1/8th of a pixel in the linear interpolation algorithm. Interpolation to a higher precision did not affect the visual quality.

Character definitions are stored in the invisible portion of the frame buffer. The user allocates a single rectangular area for font management. Character definitions are brought into this area from the disk-resident fonts as they are needed. When the free space in this area is exhausted, character definitions are eliminated and space is freed on a least-recently-used basis. This is an extremely important aspect of the antialiased text software because we wish to support a very large number of fonts (a user might wish to use a dozen or more fonts simultaneously), and the space required for font storage varies drastically depending on the point size and the number of characters in the font. Each individual character definition is run-length-compressed to save storage space; this feature is especially valuable for large characters.

Following are some of the procedures which are used to manage fonts and display text.

- *TextInit*—Initializes the font storage area. The user supplies a pixel map which becomes the font storage area. The size is quite arbitrary, allowing the user considerable control over the amount of character paging which will occur.
- *OpenFont*—Establishes a control block for the font and opens the font file. The control block contains information about which characters are in the font, the locations of the individual character definitions in the font file and in the internal font storage area, and the character-spacing parameters.
- *CloseFont*—Releases the internal storage space occupied by the characters of a font and closes the font file.
- *LoadFont*—Loads all the characters in an open font into the font storage area. If this procedure is not called, characters are loaded when needed by one of the text display functions.
- *SetSpacePrm*—Specifies character- and word-spacing values which are used while writing strings.
- *SetCurFont*—Establishes a given font as the current font.
- *SetCurPoint*—Sets the current point at which text will be displayed. The parameters are double coordinates.
- *WriteString, EraseString*—Writes (erases) a text string in the current font at the current point. Both *WriteString* and *EraseString* return the updated value of the current point.

In our implementation, the font management is done by the PC, while the rendering of characters is implemented in the display-processor microcode. The PC processes a string to be written by determining whether each character is present in the font, looking up the location of the character definition in the font storage area, and passing that location to the display processor. The microcode fetches and decompresses the antialiased character's definition, calculates the pixel values to be displayed (subpixel positioning), and updates the pixels using the correct color.

- *Other API functions*

While antialiased text is clearly superior to bilevel text in appearance, it is sometimes desirable to display fixed-pitch, bilevel text. For this purpose, we have included a text facility which can display up to 32 rows of 80 characters. Each character occupies a matrix of 8 by 15 pixels. Bilevel text is displayed significantly faster than antialiased text.

Bilevel graphics functions are provided which draw arcs, circles, polygons, lines, polylines (a connected series of lines which is not necessarily closed), and thick lines. Two types of fill operations are provided: boundary fill and flood fill. Boundary fill starts its search from a specified point, filling in all directions until a boundary of a specified color is found. Flood fill changes all connected pixels of a given old color into a new color. The closed objects (circles, polygons) may be hollow or solid. In addition, the solid-object drawing functions have counterparts which can fill with an arbitrary texture pattern supplied by the user as a pixel map.

The only antialiased graphics functions which are provided draw and erase antialiased lines. The algorithm [11] was partitioned such that the inner loop executes in the display-processor microcode, while initialization is done in the PC. The parameters for antialiased lines are double coordinates, permitting end points to be specified at noninteger positions.

A wide variety of BitBlt functions are available. One form of BitBlt rotates the image by multiples of 90 degrees; this function makes it possible to use antialiased text along the vertical axis of a graph, for example. The ZoomBitBlt function enlarges pixel maps by replication of pixels and reduces them by sampling. The MaskBitBlt stores into the destination area only where selected by mask; this facilitates manipulation of irregularly shaped areas.

Although we have tried to provide a comprehensive set of functions, some users may be interested in developing applications which will benefit from specialized microcode. In order to permit a user to develop his own microcode without giving up the benefits of the standard support software, there is a microcode-overlay facility. This set of functions permits loading of a microcode overlay into a reserved transient area of the control store, and execution and data interchange with this transient code. A microcode assembler and debugger are also provided.

Discussion

YODA was successful in achieving its objectives, but there are several areas in which it could be improved. These improvements were not all possible within the realm of technology available at the time YODA was designed, but are feasible now. They would allow for better update performance and a more cost-effective design. We discuss three areas: the frame-buffer design, the display-processor architecture, and the implementation of antialiased text.

The use of standard dynamic RAMs is extremely restrictive in the design of display frame buffers. The refresh of the CRT requires a large memory bandwidth and does not leave much of the available bandwidth for the update of the image. In the case of YODA, use of a noninterlaced CRT would have left only the retrace times available for updates by the display processor. Researchers at IBM [12] have been working to solve this problem by adding a second port to the dynamic RAM. Such parts are now commonly available. An example is the Texas Instruments TI4161. The second port is serial and is dedicated to the video refresh of the CRT. The modified RAM (called a video RAM) contains an internal shift register that can be loaded in one memory cycle and then used to refresh several scan lines. This reduces the memory bandwidth for video refresh to less than 2% of the total memory bandwidth and allows the display processor to achieve better update performance.

Video RAMs also allow the use of higher-density RAMs (256 kilobits, 1 megabit) in the design of frame buffers. Standard RAMs of these densities do not have enough bandwidth for CRT refresh. Use of these higher-density video RAMs also results in a lower-cost display controller.

- *Display processor*

The display processor is well suited for a large number of inner loops and it performs well for them. It also contains several weaknesses that result in less than desirable performance for some other tasks. YODA forces all display-update tasks to be routed through the display microprocessor. We discovered after the design was completed that it would have been more efficient for some tasks to access the frame-buffer update controls directly—for example, those which read and write pixels from the PC.

One of our principal conclusions is that the display processor is best used in the fast execution of tight inner loops. YODA implements BitBlt by using the display processor for the two nested loops; the fill functions are implemented by processing the outer loop in the PC and the inner loop in the display processor; and the circle draw is implemented entirely in the PC, using the display processor only to write single pixels. Optimal performance of a given task is achieved by tuning the display-processor design such that a task's inner loop performs adequately.

The strongest point of the display processor is its speed. The microprocessor we chose has a simple architecture that can execute instructions rapidly. The separate access path to the microprocessor's instruction storage also improves its performance. Its internal bit-manipulation capabilities allow for the concurrent use of two shifters, a masking unit, and the ALU in a single instruction, making pixel manipulation extremely convenient. Augmenting the frame buffer with the address registers was a valuable investment; it pays off in the increased update performance of almost all inner loops.

The eight-bit microprocessor is a handicap for manipulating coordinates. This is further aggravated by the fact that antialiasing requires multiple-precision coordinates. All the inner loops with counts greater than 256 execute more slowly because of the extra instructions required for higher-precision numbers. Coordinates for the antialiased text primitives are three-byte numbers, which are even slower to manipulate. We omitted certain functions because the resulting performance would have been unacceptable.

One such task is the transformation of a world-coordinate system to screen coordinates. Such tasks require a processor with a fast multiply instruction or an outboard multiplier.

The address-counter augmentation to the frame buffer has been extremely valuable, and augmenting the frame buffer with pixel-data manipulation logic would have been equally useful. For example, if the inner loop of BitBlt(Source, Destination, CombinationFunction) could perform the combination functions using a pixel ALU near the frame buffer, pixel values would not need to be brought into the display microprocessor. Examples of such combination functions are the logical functions AND, OR, XOR, etc., and arithmetic functions such as ADD, SUB, MAX, and MIN used in the antialiasing functions.

- *Antialiased text*

A deficiency in the antialiased text mechanism is the inability to show characters in arbitrary sizes at arbitrary orientations. YODA can only show the characters with particular sizes and orientations that have been precomputed and stored on disk. An example is the Press font, where we supply the 10-, 12-, 14-, 18-, 24-, and 36-point sizes horizontally oriented. These can be displayed vertically by using the BitBlt90 function, but rotations and sizes other than those supplied cannot be displayed at all.

If the characters are represented as outline splines [13], then they can be dynamically converted into raster form with any size and orientation. The raster definitions could then be cached for use by the display processor. As Carol Thompson of Yorktown Research has pointed out, the same representation can also be used for printers so that the printed copy and the display have the same appearance.

Another shortcoming of the antialiased text functions is the lack of clipping of characters at pixel-map boundaries. In our implementation, the text-rendering function simply stops, returning the current location whenever a character is encountered which would extend beyond the pixel map. This feature was omitted primarily because of its effect on performance.

Conclusion

YODA has a large and diverse group of users. Seventy YODA displays have been distributed within IBM and to several universities. The applications have varied widely and include image processing, ray-traced rendering, animation, computer-generated art, document processing, and viewing of scientific and engineering data.

YODA has proven to be a viable architecture, enabling a low-cost personal computer to produce high-quality images interactively.

Acknowledgments

We wish to thank Sarah Dolce, Marc Donner, Leon Lumelsky, and Scott Seligman for their time and efforts. Sarah tested and distributed the seventy YODA systems; Marc helped us in upgrading the figures in this presentation; Leon solved a crucial hardware problem; Scott contributed to the software effort. We thank Rick Dill and Archie McKellar for supporting this research.

References and note

1. David Bantz and Satish Gupta, *Yoda 0.2 Principles of Operation*, Internal Publication, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, August 15, 1984, pp. 1–45; available from the authors.
2. William M. Newman and Robert F. Sproull, *Principles of Interactive Computer Graphics*, 2nd Ed., McGraw-Hill Book Co., Inc., New York, 1979.
3. James D. Foley and Andries van Dam, *Fundamentals of Interactive Computer Graphics*, Addison-Wesley Publishing Co., Reading, MA, 1982.
4. Franklin C. Crow, "The Aliasing Problem in Computer-Generated Shaded Images," *Commun. ACM* **20,** No. 11, 799–805 (November 1977).
5. John E. Warnock, "The Display of Characters Using Grey Level Sample Arrays," *Comput. Graph.* **14,** No. 3, 302–307 (July 1980).
6. Paul N. Sholtz, "Making High-Quality Colored Images on Raster Displays," *Research Report RC-9632*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, October 1982.
7. M. F. Cowlishaw, "Fundamental Requirements for Picture Presentation," *Proc. Soc. Info. Display* **26,** No. 2, 101–107 (1985).
8. Edwin Catmull, "A Hidden-Surface Algorithm with Anti-Aliasing," *Comput. Graph.* **12,** No. 3, 6–11 (August 1978).
9. "CONFORM: A Context for Electronic Layout" is a computer program developed by Susan Wascher as part of a joint study by IBM and the Visible Language Workshop, Media Laboratory, Massachusetts Institute of Technology, Cambridge, MA, supported in part by HELL: GmBH. ©VLW/MIT 1985.
10. Paul N. Sholtz, Carlo J. Evangelisti, and Satish Gupta, *Yoda 0.2 Programmers Guide*, Internal Publication, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, May 8, 1985, pp. 1–90; available from the authors.
11. Akira Fujimoto and Kansei Iwata, "Jag-Free Images on Raster Displays," *IEEE Comput. Graph. & Appl.* **3,** No. 12, 26–34 (December 1983).
12. Richard Matick, Daniel T. Ling, Satish Gupta, and Frederick Dill, "All Points Addressable Raster Display Memory," *IBM J. Res. Develop.* **28,** No. 4, 379–392 (July 1984).
13. Takafumi Saito, "Character Generation Using Conic Splines," *Research Report RC-11193*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, June 1985.

Received January 23, 1986; accepted for publication July 10, 1986

Satish Gupta *IBM Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598.* Dr. Gupta is the manager of the display architecture group in the Computer Sciences Department at the Thomas J. Watson Research Center. His research interests are currently focused on computer displays and the study of both software and hardware techniques to provide higher-performance graphics displays. He received a B.Tech. degree from the Indian Institute of Technology, Kanpur, in 1977, and the M.S. and Ph.D. degrees in computer science from Carnegie-Mellon University, Pittsburgh, Pennsylvania, in 1979 and 1981, respectively. Dr. Gupta received an IBM Outstanding Innovation Award for his work on display architectures, including those represented by YODA. He is a member of the Association for Computing Machinery and its Special Interest Group in Graphics.

David F. Bantz *IBM Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598.* Dr. Bantz received his Eng.Sc.D. degree from Columbia University, New York, in 1970. After working in optical character recognition for two years, he joined IBM in 1972. He has worked in office automation, multiple-processor systems, and computer graphics, and most recently in computer displays and printers. He has held positions on both the Research Division and Corporate Technical Committee staffs. He is currently manager of the I/O Systems Department at IBM Research in Yorktown Heights and a lecturer in computer architecture at Columbia University. Dr. Bantz has received five IBM Invention Achievement Awards and an IBM Outstanding Innovation Award for his work on display architectures, including those represented by YODA. He is a member of the Society for Information Display.

Paul N. Sholtz *IBM Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598.* Mr. Sholtz joined IBM Research in 1957 at Poughkeepsie after receiving an M.S. in electrical engineering from the University of Illinois. He worked in speech recognition at Yorktown, and in machine design, design automation, and automated documentation in Poughkeepsie. In 1965 he transferred to the IBM laboratory at Rochester, Minnesota, where he worked jointly with the Mayo Clinic on projects in automated medical history taking and on-line monitoring of postoperative intensive-care patients. In 1980 he returned to Yorktown, where he has worked in display and printer architecture and in computer graphics. Mr. Sholtz is currently a senior programmer in the I/O Systems Department. In 1985 he received an IBM Outstanding Technical Achievement Award for his work in antialiased character fonts.

Carlo J. Evangelisti *IBM Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598.* Since joining IBM in 1958, Mr. Evangelisti has worked in the areas of character recognition, computers in medical diagnosis, computer graphics, signature verification, hardware-description languages, software methodology, and display architecture. At present he works in image processing. Mr. Evangelisti received a B.S. in electrical engineering from Union College, Schenectady, New York, in 1958 and an M.S. in electrical engineering from Syracuse University, New York, in 1965. In 1985 he received his sixth IBM Invention Achievement Award.

William R. DeOrazio *IBM Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598.* Mr. DeOrazio joined IBM and the Research Division in 1961. He was a member of the Computing Systems Department for 19 years until he transferred to the Computer Sciences Department in 1980 to work on the Signature Verification project, after which he joined the display architecture group. He has done engineering design work for large systems, terminal data systems, and Personal Computers. Mr. DeOrazio is currently a senior associate engineer in the Computer Sciences Department. He has received Research Division awards for implementing the Research building wiring plan and for his work on the Advanced Terminal System project.