Personal systems image application architecture: Lessons learned from the ImagEdit program

by A. Ryman

Image applications require complex processing on large amounts of data. The application designer is presented with difficult challenges that are exacerbated on personal systems which have limited processor speed and constrained memory. This paper discusses the problems relevant to personal systems image application architecture and how these problems were solved in the ImagEdit® program. A virtual array manager (VAM) consisting of a virtual memory manager (VMM) and an access scheduler was used to solve the data management problem. The VAM divided each image into segments and transferred them to the VMM for storage. These segments were swapped between memory and disk in response to a sequence of access requests, controlled by the access scheduler using performance-maximizing heuristics. Object-oriented design was used to address the functional complexity problem. The processing functions were divided into two classes. The data-stream class included scanning, printing, and filing, with each data-stream function decomposed into a series of demand-driven pipe objects. The editing class included cut and paste, textual and graphical annotation, and freehand drawing.

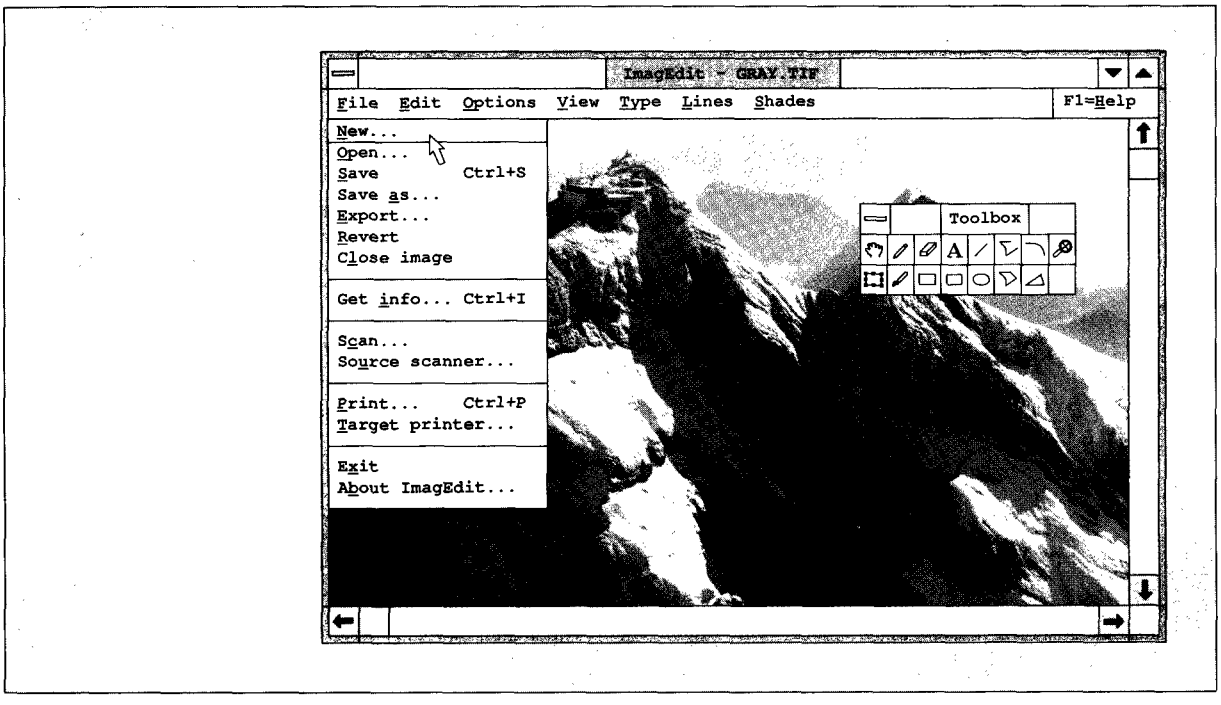
This paper discusses software design issues that pertain to personal systems image applications. The first section defines the category of image applications that can be controlled effectively by current hardware and describes the main problems faced by software designers in creating efficient and functional applications in this domain. The next section describes the image types under consideration and breaks down their processing into high-level and low-level functions. The virtual array manager, which is the main architectural component, is then discussed. The final section describes the means by which object-oriented design (OOD) can be used as a guiding principle to organize the diverse collection of image processing functions.

Image applications on personal systems

The discussion is based on experience gained from developing the ImagEdit® program, a personal computer image editing application, at the Image Systems Centre, IBM Canada Laboratory. Frequent references are made to that experience for purposes of illustration. ImagEdit V1.0 was designed for office applications and was shipped in 1987. ImagEdit V2.0 included enhancements to support desktop publishing and was shipped in 1988. The user interface of ImagEdit V2.0 is illustrated in Figure 1.

Characteristics. For purposes of this paper, personal systems image applications are those that have general-purpose personal computer hardware, medium image size, and low transaction rate.

Figure 1 ImagEdit V2.0 user interface



General-purpose personal computer hardware. Systems that have general-purpose hardware can integrate the image application with other functions, so that the user has a multitask workstation. Special-purpose image peripheral equipment, such as scanners and video cameras, are included if they allow the workstation to be used for (nonimage) purposes. The image application should work adequately with standard displays and printers but take advantage of any higher-resolution equipment that is available.

Medium image size. A medium image size is defined as approximately one megabyte. Typical images in this range would be letter-size pages scanned in *bilevel* at up to 300 pixels per inch (ppi), or 8- by 10-inch photographs scanned as gray *halftones* up to 120 ppi. A bilevel image uses just two levels, black or white, as in the display of line drawings. Halftone images display various shades of gray, as in a photo of a person's face. From a software design point of view, the most important thing about the medium-size range is that it requires more bytes than the amount of available memory, and less storage than the amount of available disk space.

Low transaction rate. The expected transaction rate, meaning capturing, viewing, or editing, is under 10 images per hour. Although the system may be capable of higher rates, the user is probably performing an unstructured task under these circumstances, and so is not pushing the hardware to its limit.

Examples. Some typical examples of personal systems image applications include office applications, desktop publishing, and low-end technical records handling.

Office applications include image notes and composite documents. Image notes, which are widely used in Japan, give users the ability to send and receive bilevel images from their desktops as if they had a personal facsimile (FAX) machine. A composite document contains a combination of scanned images with computer-coded word processing documents. ImagEdit V1.0 was designed to address this domain, using IBM's Mixed Object Document Content Architecture[1] (MO:DCA) as the carrier data stream for both notes and composite documents.

Desktop publishing allows the user to lay out text, images, and graphics in newsletters, brochures, and other common in-house publications. Here, both bilevel and halftone images are used. Line drawings are scanned as bilevel images, photographs as half-

tone images. The resulting images have to be modified in size and edited before inclusion in a publication. ImagEdit V2.0 was designed for this application domain, using as data streams IBM's Image Object Content Architecture[2] (IOCA) and the industry standards of Tag Image File Format[3] (TIFF) and Encapsulated PostScript® Format[4] (EPSF). While IOCA and TIFF are used for document interchange, EPSF is used only for exporting documents; its greater complexity requires a PostScript interpreter (typically a dedicated processor in a printer).

Image processing is functionally complex.

Technical records handling is potentially a new application area. Engineering drawings are typically scanned as bilevel images at 200 ppi; thus A-size (8 1/2- by 11-inch) and B-size (11- by 17-inch) drawings fall within our definition of medium-size images.

Software design challenges. The job of any software designer is to create applications that provide timely, usable, and cost-effective solutions to problems. Some of the challenges faced are especially acute when designing image applications for personal systems. They include large amounts of data, complex processing functions, evolving industry standards, new applications, and the technical constraints of personal computers. These are discussed below.

Large amounts of data. The most obvious challenge is to efficiently manage the large amount of data contained by images. Not only does a single image contain a great deal of data, but many applications require several images to be in use at the same time. (For example, editing may involve cut-and-paste operations between images.) This challenge will increase in severity as new technologies, such as color scanning and color printing, become commonplace.

Complexity of processing functions. Image processing is functionally complex. A typical application must support capture, display, and print for a range of peripheral equipment, as well as build and parse a variety of data streams—each with its own compression and decompression algorithms. The range of possible editing functions is even more extensive (these are described later). Organizing this complexity into a coherent software design is a difficult challenge.

Evolving industry standards. Image applications are relatively new, and standards are still evolving. For example, there is no standard method for compressing gray images, and image interchange formats are also in a state of flux. Consequently, designers must be prepared to plan for change.

New application domains. As new technologies become available, more power is put on the desktop and new application domains are made possible. The challenge is to design an application for reuse, so that new opportunities can be realized in a timely fashion.

Memory and speed constraints of personal computers. Even as personal computers become more powerful, memory and speed constraints remain major obstacles. This is because with more power come more demands and more layers of system software between the application and the hardware. Applications will continue to compete with each other and the operating system for resources. Careful attention to efficient application design is still required.

Functional specifications

This section describes the type of images under consideration and breaks down their processing into high-level and low-level functions.

Image types. Images can be described by the characteristics of pixel depth, resolution, and extent.

Pixel depth is defined as the number of bits assigned to each pixel. Two depths, namely 1 and 8 bits per pixel are used. Line art and text are normally captured at 1 bit per pixel, giving two colors (usually black and white). Photographs and video images are normally captured at 8 bits per pixel, giving 256 shades of gray. (Although the capture and display hardware is usually only accurate to 6 or 7 bits, 8 bits are stored to simplify processing.)

Resolution assigns a physical size to the pixels. The upper end of the resolution range, which is attained by some phototypesetters, is 2540 ppi. The lower

end of the range is set to be 100 times smaller than this, namely 25 ppi. Typical bilevel resolutions are 200 ppi for FAX, 240 ppi for office printers, 300 ppi for desktop publishing, and 1270 or 2450 ppi for phototypesetters. Common gray resolutions are 60 ppi for desktop publishing, and 100 or 120 ppi for phototypesetters.

The *extent* of an image is the number of pixels along each dimension. Extents are typically on the order of 1000 pixels. Although an image could consist of a single pixel, a useful minimum extent is 8×8 pixels. This size is handy for defining repeating patterns used to fill areas. Other useful examples of small images are 16×16 cursors, and 32×32 icons. Images having such small extents are usually handled by special-purpose applications such as icon editors. A convenient upper limit for extents is 5100 pixels; this value was chosen to accommodate a bilevel tabloid image (17 inches wide) at 300 ppi.

The choice of a maximum extent influences the design of an application in two ways. First, some compression and decompression algorithms allocate working buffers whose size depends on an image's horizontal extent: the wider the image, the more temporary storage is required for compression or decompression. Second, the data storage scheme may assume that all pixels in the same row are stored together. The storage unit size therefore limits the horizontal extent.

High-level functions. Most image applications will include some combination of high-level functions, named file, scan, print, view, and edit. Each is described in a subsequent paragraph.

File. Filing consists of moving images between the temporary working storage used by the application, and the permanent storage of a file system. Several file formats exist to ensure that applications can interchange images. A typical file format contains the image data and descriptive information about it such as pixel depth, resolution, and extent. As an option, the image data may be compressed to reduce storage requirements.

IBM office applications use IOCA, while industry applications in the desktop publishing domain (such as Aldus PageMaker® and IBM InterLeaf™) often use TIFF. Both these formats are easily interpreted by applications and maintain the image in a form that can be readily edited. EPSF is used as an image interchange format when the image is in final form, such as when it is intended to be sent to a printer or display. An EPSF file may contain text and graphics in addition to images.

Filing may also involve a number of image conversions. An image in one format may be saved in another; or the pixel depth, resolution, extent, and compression may be changed when the image is saved.

Scan. Scanning consists of capturing an image from an external source. Scanning hardware includes video digitizers and flatbed and feed-through document scanners. A workstation may have more than one scanning device; for example, creating a parts catalog may require a video camera for capturing actual parts, and a scanner for capturing photographs of parts. The user must be able to select the source device.

The user must also be able to control a number of parameters that govern the capture. For example, with the IBM 3119 PageScanner™ one can specify the pixel depth, resolution, extent, gray response curve, enhancement, and halftoning. The gray response curve permits compensation for source documents that are too dark or too light. Enhancement can bring out detail (sharpen) or eliminate noise (smooth). Halftoning affects the way a bilevel image is created: the user can specify *thresholding* or a *dither matrix.*

Print. Printing provides a hard copy of the image. The user can specify the portion of the image to be printed, the number of copies, and how the image is scaled to fit the paper. Most printers can actually only print black and white; they simulate shades of gray by halftoning.

The majority of office printers offer a simple interface for printing images. The image is sent as raster data, which require the application to convert the resolution and pixel depth of the image to match that of the printer. In contrast, desktop publishing printers usually offer a high-function interface. For example, the PostScript processor transfers an image device-independent form; the printer microprocessor converts the pixel depth and resolution. This has the advantage that a document can be proofed on a low resolution printer and then printed as camera-ready copy on a high resolution printer without losing image quality.

View. Viewing lets the user see all or portions of the image at various magnifications. It must be possible

to move to the area of interest (scroll) and select the desired magnification (zoom), or see several images at once.

Scrolling can be handled in a number of ways. ImagEdit V2.0 offers three: scroll bar controls, a grabber tool, and an overview window. Scroll bar controls are standard in most window managers. Here, two bars are placed along a vertical and a horizontal edge of the image window, and the relative position of the window over the image is indicated by movable boxes in the bars. The image can be moved horizontally or vertically by dragging one of these scroll boxes. The grabber tool lets the user reposition any visible part of the image within the window by simply dragging the part to where it is wanted. (This is an example of direct manipulation in user interface design.) The overview window displays a reduced view of the image. Here, the user can center the image window over any point in the image by clicking the mouse on the corresponding point in the overview window.

Several images can be viewed at once.

Zooming is controlled by menu commands. (Most of these have keyboard shortcuts.) High magnification views are useful for detailed pixel editing, while low magnification views are useful for large-scale cut-and-paste editing.

Several images can be viewed at once. ImagEdit V1.0 directly supports multiple image windows, while ImagEdit V2.0 uses the multitasking capabilities of the Microsoft Windows® [5] operating environment.

Edit. Editing is the most complex application area. Editing operations can be classified as either annotation or block operations. Annotation is associated with the change and entry of data. This includes text, graphics, and freehand. Block operations are associated with the manipulation of portions of pages or images.

Text annotation allows paragraphs to be overlaid on the image, which is useful for labeling diagrams. The user can select the font, size, leading, alignment, color, and transparency of the text.

Graphical annotation is useful for diagramming. The user can draw lines, polylines, rectangles, ellipses, rounded rectangles, and polygons; have closed shapes either filled or outlined; and select line color, width, and style, and fill color.

Freehand editing consists of drawing, painting, and erasing. Drawing is done with a pencil tool that creates a stroke one pixel in width. Painting is done with a paintbrush tool. The user can specify the shape and size of the brush tip, as well as the paint color. Erasing is the same as painting except that the paint color is only black or white. The eraser normally paints white. This can be changed to black via a menu command.

Block operations include the following operations: cut, copy, paste, clear, duplicate, reverse color, gray response curve, flip, rotate, position, and size. Most block operations can be applied to the entire image, to a selected portion of it, or to a clipping.

Cut, copy, and paste functions are handled through a standard temporary storage area called the clipboard, which is maintained by the window manager. The user selects a rectangular area and can clear (erase) it, or cut or copy it to the clipboard. (Cutting is a combination of copying and clearing.) The contents of the clipboard can then be pasted anywhere on the image, or moved between applications. The user controls how the pasted clipping combines with the image to produce a variety of special effects. Images with nonrectangular outlines can be handled in this way.

With color reversal, the user can interchange the color of every selected pixel, reversing black and white or dark gray and light gray. Using the gray response curve, every pixel can also be mapped to a new color.

A selection can be rotated by any multiple of 90° or flipped along a horizontal, vertical, or diagonal axis. It can also be positioned and sized anywhere over the image. The user controls target position and size numerically or interactively and can also control how the clipping is converted prior to pasting. This may involve smoothing if the resolution needs to be changed, or halftoning if a gray image is pasted on a bilevel image.

All high-level functions described above are built up from combinations of low-level functions. These are described below.

Low-level functions. The low-level functions are the basic building blocks used to create, modify, and specify new high-level functions.

Building and parsing data streams. Image data are interchanged in standard formats, which contain the data along with descriptive information about the image such as resolution, extent, and pixel depth. The image data may be optionally compressed to reduce storage requirements. ImagEdit V2.0 supports the formats IOCA, TIFF, and EPSF. These formats are mainly used to interchange image files between applications, but may also be used to transmit images from scanners to applications, and from applications to printers.

The resolution and extent of images are often changed in viewing, printing, and editing.

Compression and decompression. Image interchange formats often support a variety of compression algorithms, especially for bilevel images. ImagEdit V2.0 supports Modified Modified READ (MMR) in IOCA and Modified Huffman (MH) in TIFF. (Gray compression standards are currently being defined.)

Magnification and reduction. The resolution and extent of images are often changed in viewing, printing, and editing. The fastest techniques for doing this are pixel replication for magnification, and pixel skipping for reduction. However, while these techniques are usually adequate for viewing, they may not be satisfactory for editing or printing since they can create artifacts in the images—replication may cause staircasing (also referred to as "jaggies") on diagonal lines, while skipping may cause thin lines to disappear. Thus, when image quality is critical, pixel interpolation is used for magnification and pixel averaging for reduction.

Rotations and flips. The rotation and flip functions are mainly used in editing, but are also useful for viewing and printing images with landscape orientation. ImagEdit V2.0 supports rotations by right angles and flips along vertical, horizontal, and diagonal axes. Rotation by small angles is useful for correction of images that are scanned slightly out of alignment, while rotation by general angles is useful for advanced editing.

Text, image, and graphics annotation. Annotation functions are normally provided by an operating environment kernel such as the Microsoft Windows Graphical Device Interface. In practice, these kernels are not tuned for handling large images, so the application typically has to divide editing tasks into small pieces. Also, the general-purpose image magnification and reduction algorithms may lack the necessary performance, and may require replacement by application-supplied routines. This is a symptom of the relative newness of the image as a data type. We can expect future operating environment graphical kernels, such as the OS/2® Graphics Programming Interface, to provide much higher image function.

Halftoning and anti-aliasing. Halftoning changes gray images to bilevel images, while anti-aliasing is a technique for preserving image quality under reduction by converting a high-resolution bilevel image into a low-resolution gray one. These operations are used for viewing and displaying gray images on bilevel devices, and for combining images with different pixel depths when editing.

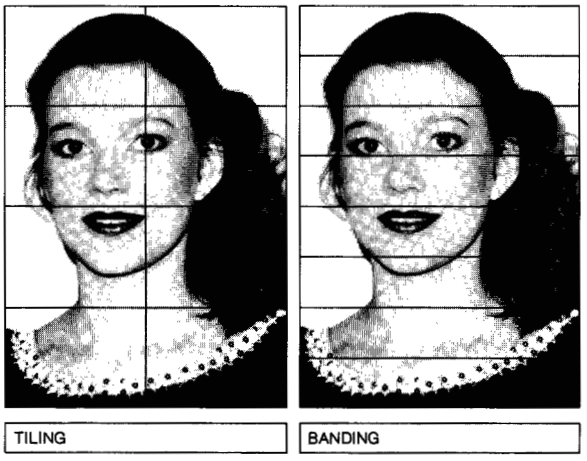
Intensity mapping. Intensity mapping is a useful technique for enhancing gray images. It can be used to bring out detail in poorly exposed photographs. ImagEdit V2.0 supports it in scanning, editing, and printing.

Digital filtering. Digital filtering is another flexible image enhancement technique. For example, graininess or noise can be removed with a smoothing filter, while blurriness can be removed with a sharpening filter. ImagEdit V2.0 only supports digital filtering when scanning.

The virtual array manager

As stated previously, one of the main challenges in the design of image applications is to efficiently manage large amounts of data within a small amount of memory. This section describes the main archi-

Figure 2 Image segmentation



tectural component for achieving this, the virtual array manager (VAM).

Virtual arrays. Image data may be regarded as defining a two-dimensional array of pixel values. These values may be packed eight to a byte for bilevel images, or one to a byte for gray images. The arrays are typically large, and the application may require access to several arrays simultaneously. Therefore, when the data requirements of the images are combined with the code requirements of the application, the operating environment, and other resident applications, the designer must make a decision whether or not to support configurations in which all the image data cannot be stored in memory.

For personal systems image applications, it is normally a requirement to support small memory configurations. This means that the application must support a disk-based array management architecture. We refer to disk-based arrays as *virtual arrays* since they make use of the same concepts as operating system (OS) virtual memory.[6] However, some of these concepts need to be altered in order to achieve peak performance. This will be discussed in more detail below.

Image segmentation. The VAM is responsible for dividing an image into segments. The general method for doing this is two-dimensional *tiling*. It is also appropriate for handling images that are already tiled, such as large technical drawings. Tiling is very efficient for random access operations such as viewing, editing, and especially rotations, and this method was used in ImagEdit V1.0. However, there is significant processing overhead when opening or saving an image that is not already tiled. For this reason, a limiting form of tiling called *banding* was used in ImagEdit V2.0. In this method the tiles are full width, which improves essentially sequential operations such as opening and saving. It reduces the performance of viewing and editing slightly, and that of rotations to a greater extent. Tiling and banding are illustrated in Figure 2.

The virtual memory manager. The VAM passes image segments to the virtual memory manager (VMM) for storage. The VMM swaps segments between a region of memory called the *cache*, and a disk file called the *spill file* in response to access requests. This architecture is illustrated in Figure 3. When the application performs an operation on a virtual array, the VAM must determine which segments are involved and schedule access to these segments. This algorithm can greatly affect performance.

If the cache is large, the segments stay in memory and the operations run at full speed. Both ImagEdit V1.0 and V2.0 let the user increase the cache size by installing expanded memory. When the cache becomes full, segments must be swapped from memory to disk to make room for new segments. In this situation, performance depends on disk access speed.

The VMM is based on the same concepts as OS virtual memory management. Image segments correspond to memory pages, the cache corresponds to real memory, and the spill file corresponds to virtual memory. The strategy used to control swapping between real and virtual memory affects system performance. The goal here is to minimize the average number of swaps required to perform typical image processing operations.

In OS virtual memory, swapping between real and virtual memory is usually controlled by the least recently used (LRU) algorithm. When a page has to be swapped from real to virtual memory, the least recently used page is selected. This means that the most recently used pages are preferentially retained in real memory. This algorithm assumes locality of reference. Roughly, this means that the probability that a page will be accessed is proportional to how recently it has been accessed. This often makes sense. For example, in code, statements in a loop are repeatedly accessed. However, it often does not make sense for image processing.

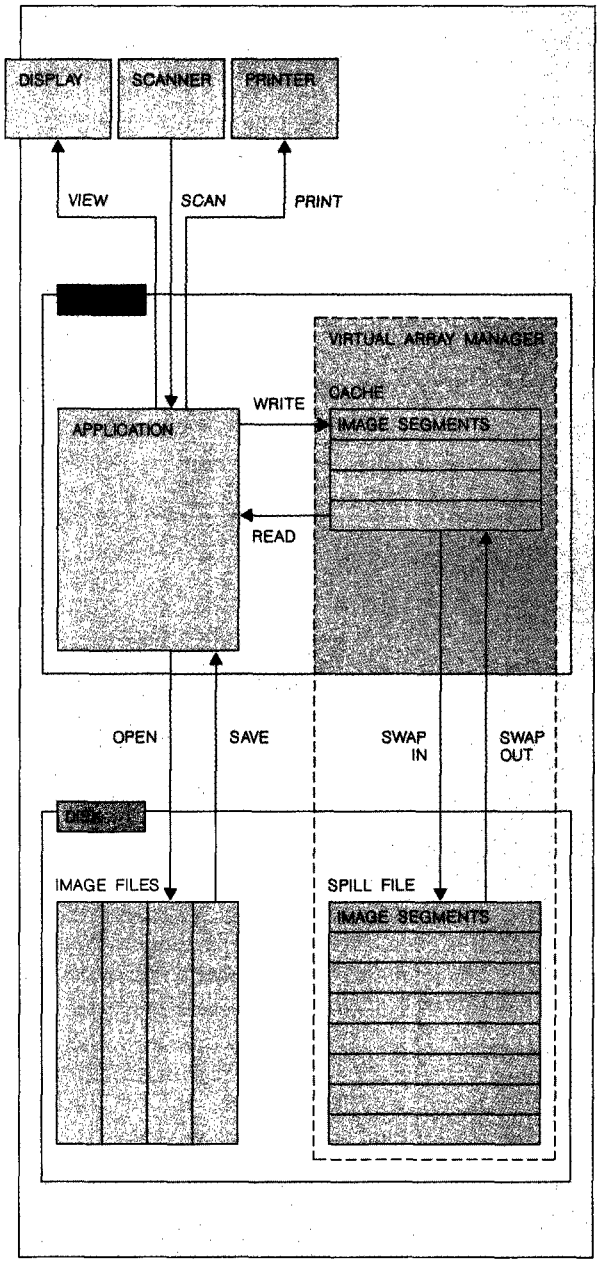
In image processing, the assumption of locality of reference is often violated. In some operations, every pixel in the entire image is accessed precisely once. For example, consider the operation for displaying an entire image. In this case, the most recently used pixel is the least likely to be accessed next. If the entire image is too large to fit in real memory, the LRU algorithm will lead to *disk thrashing*, meaning that every image segment must be swapped each time the operation is performed. In fact, using the LRU algorithm leads to worse performance than a disk-based array management scheme that has no cache, since the memory devoted to the cache is unavailable for usefully storing code or data.

Access scheduling. There is a better swapping algorithm than LRU, one that schedules swapping based on access requests. We refer to this as the Modified LRU (MLRU) algorithm. (This technique is sometimes also called *clairvoyant caching* because the access scheduler is given information about what will happen in the future.) In this approach, the application makes access requests to the VAM that specify a region of a virtual array and whether it will be read or written. The VAM maintains information about the contents of the cache, including how recently the segments have been accessed, whether they have been modified since the last time they were swapped out to the spill file, and whether or not they are involved in any pending read or write access requests.

After the application notifies the VAM that it is about to make a request, it can ask it to recommend which segment in a given list is the best to process next. The VAM uses its knowledge of the cache contents to select the best segment. For example, segments currently in the cache are given the highest preference, and of these, modified segments are given preference to unmodified ones. In general, the VAM assigns a heuristic "cost" to each element in the list and recommends the cheapest one. The cost is conceptually the expected cost to access the segment. The use of this feature improves the performance of random access operations (operations that can process the segments in any order). For example, annotation is a random access operation since each segment can be annotated independently. Other operations, such as writing an image to a disk file, require sequential access, and so cannot alter the order in which they process the segments.

If the application requests access to a segment that is not in the cache, the VAM must swap it in from the spill file. If the cache is full, it first must select a segment to swap out. The VAM also maintains a heuristic cost for swapping out segments, and again selects the cheapest one. For example, segments involved in pending operations are preferentially retained in the cache.

Figure 3 Virtual array manager



The main difference between the VAM and OS virtual memory is that the VAM is not transparent to the application, whereas the OS is. This means that applications must be written to explicitly take advantage of VAM features. However, this may be a necessary price to pay if performance is a critical requirement. Using an LRU-based OS virtual memory will lead to disk thrashing in some circumstances; this can be avoided by using an MLRU-based VAM.

Object-oriented design

While the VAM solves the data management problem, object-oriented design (OOD) solves that of functional complexity. This section describes the principles of OOD and how they were used in ImagEdit.

Dynamic binding and inheritance. Software design is mainly a process of decomposition. There are several principles that guide the designer in this process, of which OOD has recently been recognized as an important set. OOD is based on the proven concepts of data abstraction, information hiding, and encapsulation, which are embodied in such conventional programming languages as Ada, and extends these with two additional concepts: *dynamic binding* and *inheritance.*

Dynamic binding and inheritance are generalizations of programming techniques that are often used in the design of operating systems. The following discussion illustrates these concepts by casting an example that should be familiar to readers with a programming background, namely that of *file systems*, into the language of OOD. Later it will be shown how OOD can be applied to image processing.

Dynamic binding is a technique for improving the generality and reusability of code, by deferring to run-time the binding of functions to requests. This is implemented by linking functions to the data on which they act. For example, when an application program opens a file, the operating system assigns an identifier to it. This identifier is sometimes called a *file handle*, and all future references to the file are through its handle. The handle points to a device driver that implements requests to read, seek, and write. The actual device is transparent to the application. For example, the file could be stored on a floppy disk, a fixed disk, or a virtual disk, but the application code is the same in all cases. If a new file device is added to the computer, the application program will not have to be changed.

In the language of OOD, the file device drivers are called *classes*. When a file is opened, an object (i.e., the file handle) belonging to the appropriate class is created. The requests to read, seek, and write are called *messages*, and the functions that implement these requests are called *methods.*

Inheritance is a technique for basing new classes on existing ones. For example, a file device driver to handle 1.44 MB diskettes might be based on one that handles 720 KB diskettes. The new class is said to inherit from its *superclass*. The inheritance relation defines a hierarchy on the classes. The complete class inheritance hierarchy for ImagEdit V2.0 is illustrated in Figure 4, and is described in further detail below.

Object-oriented programming (OOP) languages, such as Smalltalk-80™,[7] C++®,[8] and Objective-C™,[9] support dynamic binding and inheritance. Smalltalk-80 is a pure OOP language; here, all data items are objects. This can lead to performance problems for some classes of applications. C++ and Objective-C are hybrid languages; they both add programming language support for objects to standard C,[10] and are usually implemented by preprocessors that generate standard C. ImagEdit V2.0 is based on the OOD model of Objective-C, but was coded directly in standard C because appropriate language support was unavailable when development started.

Data-stream classes. Although images are conceptually two-dimensional arrays, they can also be regarded as one-dimensional, by considering each row as a one-dimensional stream of bytes and then stringing all the rows together. This is how images are stored in files and created by scanners.

The data-stream (DS) classes take this view of images, which lets many image processing functions be decomposed as a sequence of pipes and filters in the sense of the UNIX® operating system.[11] For example, consider the operation of reading a compressed bilevel IOCA image from a disk file into a virtual array. Each pixel of image data passes through the following sequence of processing functions:

1. Read the IOCA data stream from the disk file and parse it to extract the compressed image data.
2. Decompress the image data.
3. Reverse the color of the image data, since bilevel compression algorithms use 1 for black and 0 for white, while displays use 0 for black and 1 for white.

Figure 4 ImagEdit V2.0 class hierarchy

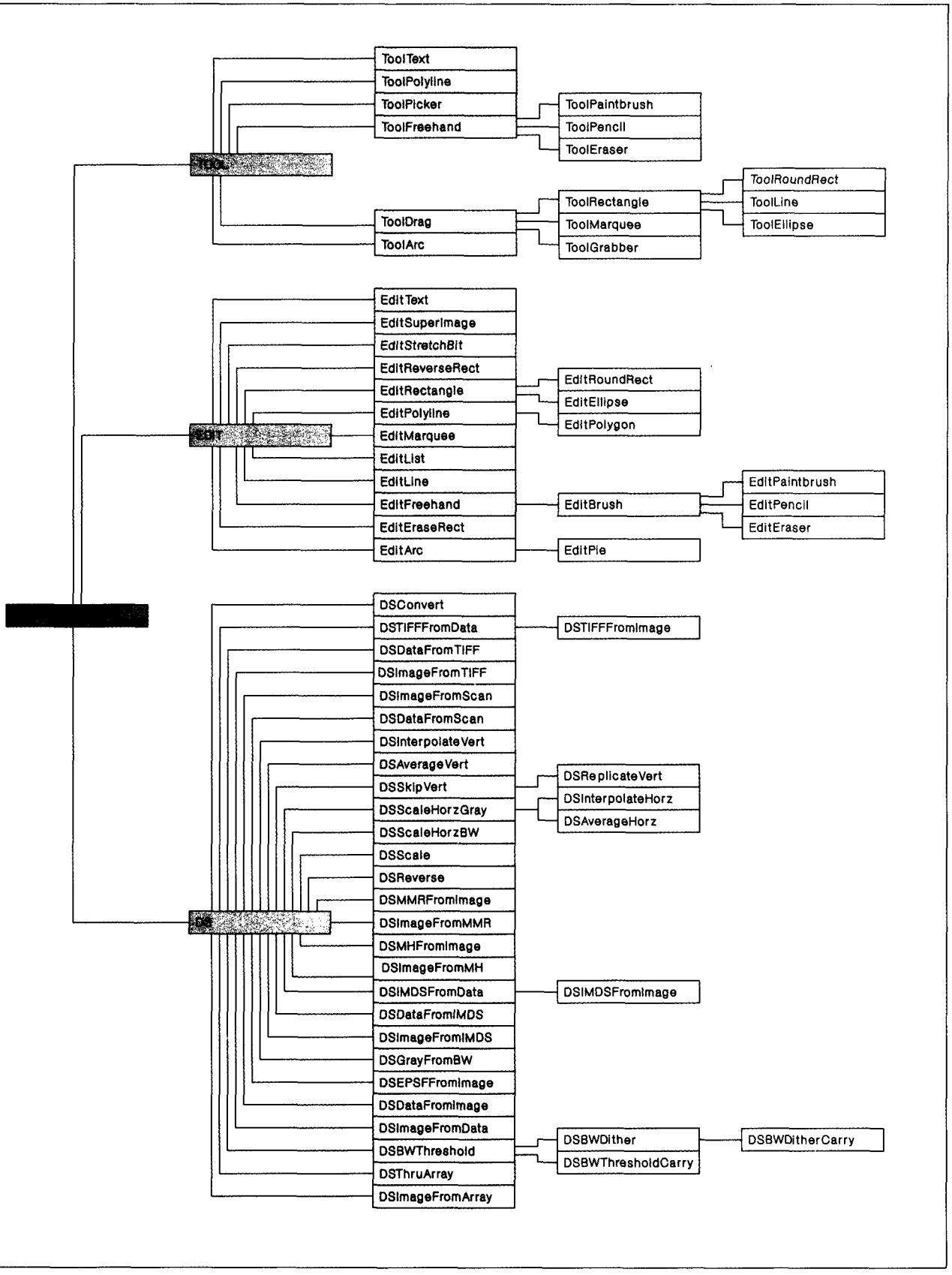
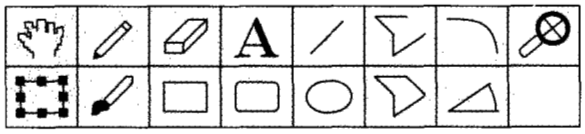


Figure 5 ImagEdit V2.0 toolbox



4. Word-align each row of image data, since graphic kernels require rows to start on word boundaries.
5. Write the image data into the virtual array.

Functions that generate a data stream as output are called *sources*, while those that consume one as input are called *sinks*. Functions that are both sources and sinks are called *filters*. The communication channel for passing a data stream between functions is called a *pipe*. In the above example, reading the disk file is a source, writing the virtual array is a sink, and all the other functions are filters.

In UNIX, which is a multitasking operating system, each function could be implemented as a separate program that reads from its input pipe and writes to its output pipe. The operating system would control the sequence of execution, giving each program a slice of processing time. Each program is able to execute write requests freely since these can be buffered by the operating system. However, when a program makes a read request, its execution may get suspended if not enough input data are yet available.

This scheme gets modified when it is implemented by a single-tasking application. In this approach, the processing is organized as a sequence of demand-driven functions. Each source or filter becomes a class that can create a data-stream object, and the data-stream objects can only respond to read requests. The sink is implemented as a loop that repeatedly reads its input data stream and disposes of the data until the data stream is exhausted.

In ImagEdit V2.0, the above example would use the following data-stream classes:

- DSImageFromIMDS—This class transforms an IOCA file into an image data stream that has the alignment and color interpretation expected by a virtual array. It links together the required sequence of data-stream objects from classes that perform the elementary operations of parsing, decompression, color reversal, and word alignment.
- DSDataFromIMDS—This class parses and extracts an image data stream from an IOCA file. The image data may be either bilevel or gray. They may require further decompression, color reversal, and/or word-alignment.
- DSImageFromMMR—This class decompresses a bilevel MMR data stream.
- DSReverse—This class reverses the color of an image data stream.
- DSImageFromData—This class word-aligns an image data stream.

The remaining data-stream classes handle the following functions:

- Other image sources (TIFF files, scanners, virtual arrays)
- Other image sinks (IOCA, TIFF, and EPFS files)
- Scaling (replicate, skip, interpolate, average)
- Color conversion (bilevel to gray, halftoning)

Refer to Figure 4 for the complete set of data-stream classes in ImagEdit V2.0.

Editing classes. Interactive image editing is implemented with two more or less parallel families of classes. Tool classes handle user interaction, while Edit classes represent the editing operations created by the tools. The user selects a tool from a toolbox (see Figure 5), and uses it to create an edit operation. Edit classes describe actual edit operations; for example, the paintbrush tool is used to draw a stroke. Here, the paintbrush is an object in the ToolPaintbrush class, while the stroke is an object in the EditPaintbrush class. This application of OOD is well documented in the literature[7,9] and is often used as an example to illustrate the benefits of object-oriented design.

The main benefit of OOD here derives from dynamic binding, which allows general-purpose functions to be written for handling generic tools and edit operations. For example, several edit operations may be stored together in a queue; then when the queue is drawn on the display, a draw message is sent to each object in the queue. Similarly, the queue is used to control the "Undo" and "Redo" commands. The queue itself is independent of the type of objects it contains. If new types of edit operations are added at a later point in development, the queue management functions will not be affected. This feature makes software more adaptable to changing requirements and also provides a sound basis for decomposing the system into independently constructible parts.

An additional benefit of OOD comes from inheritance: allowing similar classes to be based on common pieces of code. The result is less code and more assured consistency of behavior. For example, consider the user interaction required to draw a rectangle and an ellipse. In both cases the user presses the mouse button down at one point, drags the mouse to another point, and releases the button. The two points are used to define opposite corners of a box that is either the border of the rectangle or the bounding box of the ellipse. This commonality of user interaction is reflected in the class hierarchy by making ToolRectangle the superclass of ToolEllipse. ToolEllipse inherits "MouseButtonDown" and "MouseMove" methods from ToolRectangle. It overrides the "MouseButtonUp" method by creating an EditEllipse object instead of an EditRectangle object. Similarly, ToolRoundRect (rectangles with rounded corners) and ToolLine (straight lines) also inherit from ToolRectangle. (Refer to Figure 4 for the complete set of Tool and Edit classes in ImagEdit V2.0.)

Conclusion

The move to operating systems with virtual memory, such as OS/2 and AIX®, may solve data management problems in the short term. However, as capture and printing technology for new application domains (such as color and high-end technical records) becomes available, the limitations of LRU-based memory management may become apparent: disk thrashing will occur once image size becomes large enough. To overcome performance problems, either operating systems must provide access scheduling functions, or applications must implement their own virtual array managers.

Our experience with OOD has been positive. It appears to be a very appropriate paradigm for decomposing functionally complex systems, and it should scale well as applications become larger. The widely reported benefits of OOD were realized in our use of it for interactive image editing. In addition, the use of demand-driven pipe objects greatly simplified data-stream handling. Finally, we confirmed that the slight additional overhead incurred by dynamic binding had no measurable effect on system performance; memory management continued to be the most important issue.

Acknowledgments

I am indebted to my former function manager, Ray Douglas, for supporting the ImagEdit program and for creating an effective work environment. I would also like to thank the following developers who made significant technical contributions to the project: Alan Adamson, Ian Ameline, Jack Botner, Alexis Cheng, Brian Farn, Isao Furukawa, Johann Gurnell, Dave Ings, Jim Lemke, Jary Martin, John McFall, Alan Mineault, Howard Nasgaard, Luciano Pedron, and Ichiro Sayama.

Cited references

1. *Mixed Object Document Content Architecture,* SC31-6802, IBM Corporation; available through IBM branch offices.
2. Y. Hakeda "The Image Object Content Architecture," *IBM Systems Journal* **29,** No. 3, 333–342 (1990, this issue).
3. *Tag Image File Format Specification Revision 5.0,* Aldus/Microsoft Technical Memorandum, Aldus Corporation, Seattle, WA (1988).
4. *PostScript Language Reference Manual,* Adobe Systems Incorporated, Addison-Wesley Publishing Co., Reading, MA (1986).
5. *Microsoft Windows Software Development Kit Version 2.0,* Microsoft Corporation, Redmond, WA (1987).
6. *Introduction to Virtual Storage in System/370,* GR20-4260-1, IBM Corporation (1973); available through IBM branch offices.
7. A. Goldberg and D. Robinson, *Smalltalk-80: The Language and Its Implementation,* Addison-Wesley Publishing Co., Reading, MA (1983).
8. B. Stroustrup, *The C++ Programming Language,* Addison-Wesley Publishing Co., Reading, MA (1986).
9. B. J. Cox, *Object-Oriented Programming: An Evolutionary Approach,* Productivity Products International, Addison-Wesley Publishing Co., Reading, MA (1986).
10. B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall Inc., Englewood Cliffs, NJ (1978).
11. D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," *The Bell System Technical Journal* **57,** No. 6, Part 2, 1905–1929 (July–August 1987).

General references

IBM ImagEdit Version 1.0 User's Guide, 6476113, IBM Corporation (1987); available from IBM branch offices.

IBM ImagEdit Version 2.0 User's Guide, 75X3255, IBM Corporation (1988); available from IBM branch offices.

R. M. Helms, "Introduction to Image Technology," *IBM Systems Journal* **29,** No. 3, 313–332 (1990, this issue).

Arthur G. Ryman *IBM Canada Centre for Advanced Studies, 844 Don Mills Road, North York, Ontario, Canada, M3C 1V7.* Dr. Ryman is currently the associate head of the IBM Canada Centre for Advanced Studies where he is also the principal investigator of the Advanced Software Design Technology program. He received a B.Sc. in physics from York University, Toronto, in 1972, an M.Sc. in mathematics from the University of London in 1973, and a Ph.D. in mathematics from Oxford University in 1975. After performing postdoctoral research in computational atomic and nuclear physics at York University in Toronto, Memorial University in St. John's, and the University of Toronto, he became a mathematician at the W. P. Dobson Research Laboratory, Ontario Hydro, in 1979. He joined the IBM Canada Laboratory, Toronto, in 1982, where he worked on office systems and image processing. His last development assignment was as the designer and manager of the ImagEdit program. Dr. Ryman's current research interests are in the applications of logic programming and graphical visualization to systems software design. He is a member of the Institute of Electrical and Electronics Engineers and the Association for Computing Machinery.

Reprint Order No. G321-5409.