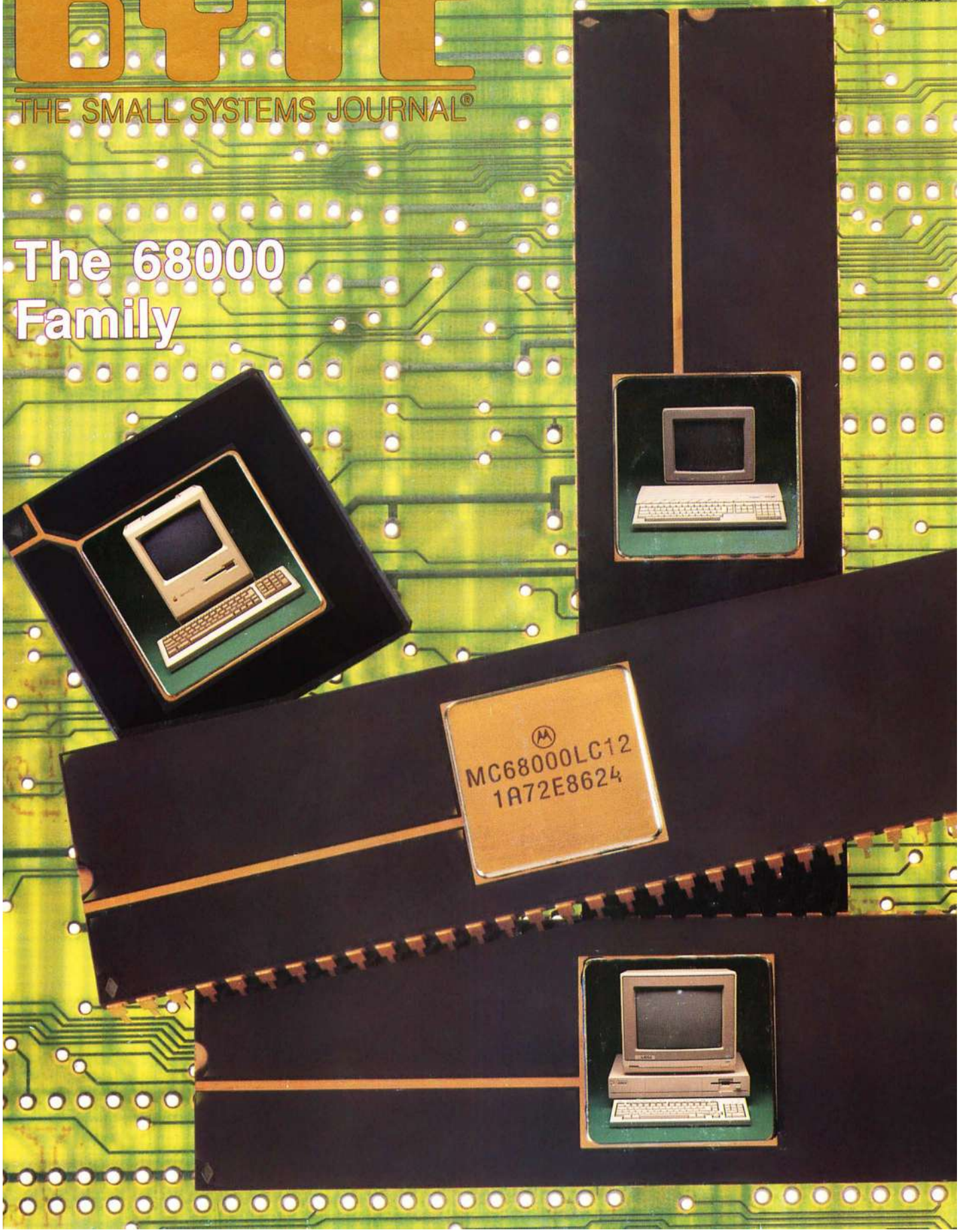
SEPTEMBER 1986 VOL. 11, NO. 9

$3.50 IN UNITED STATES
$4.25 IN CANADA / £1.75 IN U.K.
A McGRAW-HILL PUBLICATION
0360-5280

BYTE

THE SMALL SYSTEMS JOURNAL®

The 68000 Family



AMIGA ANIMATION

BY ELAINE A. DITTON AND RICHARD A. DITTON

Bringing graphics to life on the Amiga

COMPUTER ANIMATION is the process of displaying a series of images on a video screen. The images can be displayed in the same spot on the screen for static animation or moved about the screen for dynamic animation. Animation in general consumes a large portion of a system's available processing power and memory space. Usually, though, trade-offs can be made between the amount of memory space and the amount of processing power required.

The Commodore Amiga has specific hardware that makes the task of animation consume less CPU processing. The sprite DMA (direct memory access) channels allow a relatively small image to be moved and altered by changing just a few locations in memory. The Blitter is a high-speed hardware device used for copying or merging image data and drawing lines. Because the colors in the Amiga are stored in registers, a special form of animation called color animation is possible.

In this article we will briefly discuss the various aspects of animation on the Amiga and the facilities provided in the Amiga ROM Kernel to generate graphic images. In conclusion, we will describe our methods of programming animation on the Amiga.

THE DISPLAY

The first thing that must be specified is the background on which the animation will take place. This is done by defining a View structure, which describes the characteristics of the display (see figure 1). The View consists of one or more ViewPorts, each with specified height, width, display mode, image data, colors, and position on the screen. ViewPorts must be vertically stacked and separated by at least one blank line. The width should be specified as either 320 or 640 pixels. Two or more ViewPorts of different horizontal resolution can exist on the screen at the same time. The ViewPort points via RasInfo to the BitMap structure, which in turn points to the actual bit planes of image data (figure 2). The number of bit planes determines the maximum number of colors. The ViewPort also points to the ColorMap, which is interpreted depending on the mode.

SPRITE ANIMATION

Sprites are hardware "objects" that are independent of the background display. The Amiga can have eight sprites, each 16 pixels wide and any number of lines high (figure 3). Even though there are only eight sprites, each one can be reused after its horizontal endpoint has been reached on the screen. Each sprite can have 3 colors plus transparent, or you can attach two sprites to have 15 colors plus transparent.

A sprite is displayed on the screen by specifying its *x,y* coordinates and a pointer to the memory area that describes the formed image. To animate a sprite you only have to change either of the coordinates or the pointer to the image data. Since you only have to change a few bytes to move or alter the image, sprite animation takes very little CPU processing.

The Amiga ROM Kernel provides several routines for manipulating the hardware sprites. GetSprite allocates a hardware sprite for exclusive use of the requesting task. ChangeSprite

(continued)

Elaine A. Ditton and Richard A. Ditton (Free-Radical Software, 1323 South Yale Ave., Arlington Heights, IL 60005) are president and vice president of Free-Radical Software, a company specializing in consumer software and computer graphics consulting.

changes the pointer to the image bit map of a reserved sprite. MoveSprite modifies the *x,y* coordinates of the sprite. FreeSprite returns a reserved sprite to the system.

To use the hardware sprites in the animation system, the Amiga ROM Kernel defines a structure known as the VSprite (for "virtual sprite"). Information about the sprite such as color data, collision detection, and double buffering is contained in the VSprite structure.

Sprites are extremely easy to animate but have limitations that you must consider. A limited number of sprites are available, and each sprite is of limited size. Therefore, if you have to animate a large area, sprites will not be appropriate. An individual sprite can have only 3 colors and attached sprites can have 15 colors, whereas the background display can have up to 32 colors. The SPRITES mode bit in the ViewPort structure must be set if you are using VSprites or hardware sprites.

Background Animation

The simplest type of background animation uses the XOR trick. For instance, if you XOR an area of the screen with a pattern, the pattern appears on the screen with a different color from the background (figure 4). The original background can be restored by XORing in the same position with the same pattern. This method is fast because no data is actually being moved. The Amiga drawing mode COMPLEMENT supports this idea. It is limited because all the bit planes are complemented so that the resulting color is always determined by the background color. You can obtain more color control if you selectively XOR bit planes. Unless you choose the colors in the registers carefully, the overlapped portion of XOR images will be a different color than either of the images.

The method most often used to animate complex background images is to move the actual blocks of data. This method takes up the most CPU time, but the Amiga assists with the hardware Blitter. The Blitter uses up to four DMA channels to move data 4 to 10 times faster than the 68000 microprocessor.

The routines BltBitMap and ClipBlit copy rectangular areas from one sec-

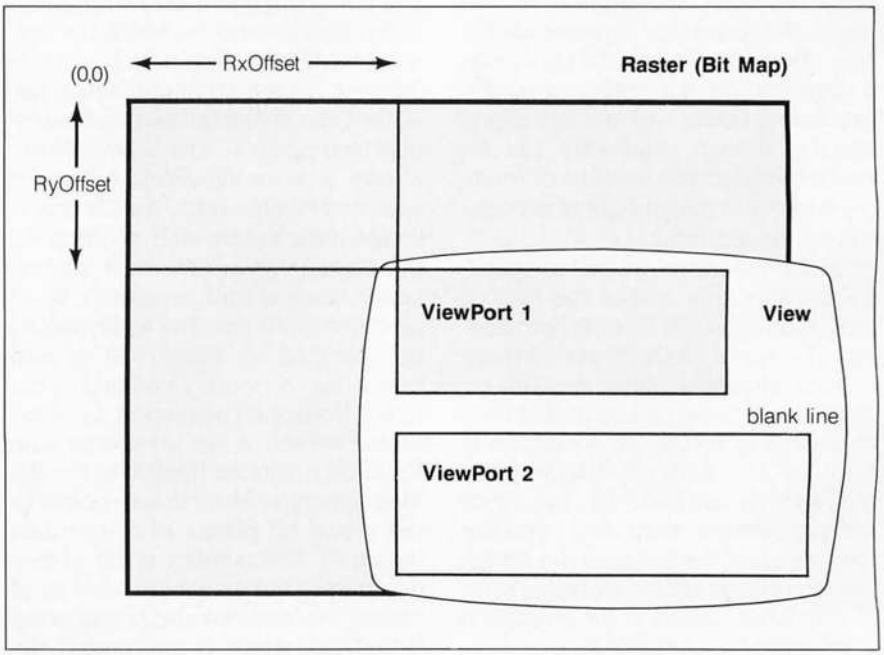


Figure 1: *The* View *structure, which defines the display characteristics of the Amiga, consists of one or more* ViewPorts, *which must be separated by at least one blank line. The values* RxOffset *and* RyOffset *determine which portion of the background bit map is displayed in a* ViewPort.

tion of chip memory to another. BltBitMap takes bit maps as arguments and will blit only specified bit planes. ClipBlit works with the RastPort structure and within the multitasking system. It will not destroy data of another task's overlapping window. Both routines use minterms, 8-bit values that determine how the source rectangle is moved into the destination area.

If you want the object to do more than just animate in a stationary position, you must save the background underneath the object so that it can be restored. If more than one object is moving across another, the data moves must be processed in last-in/

(continued)

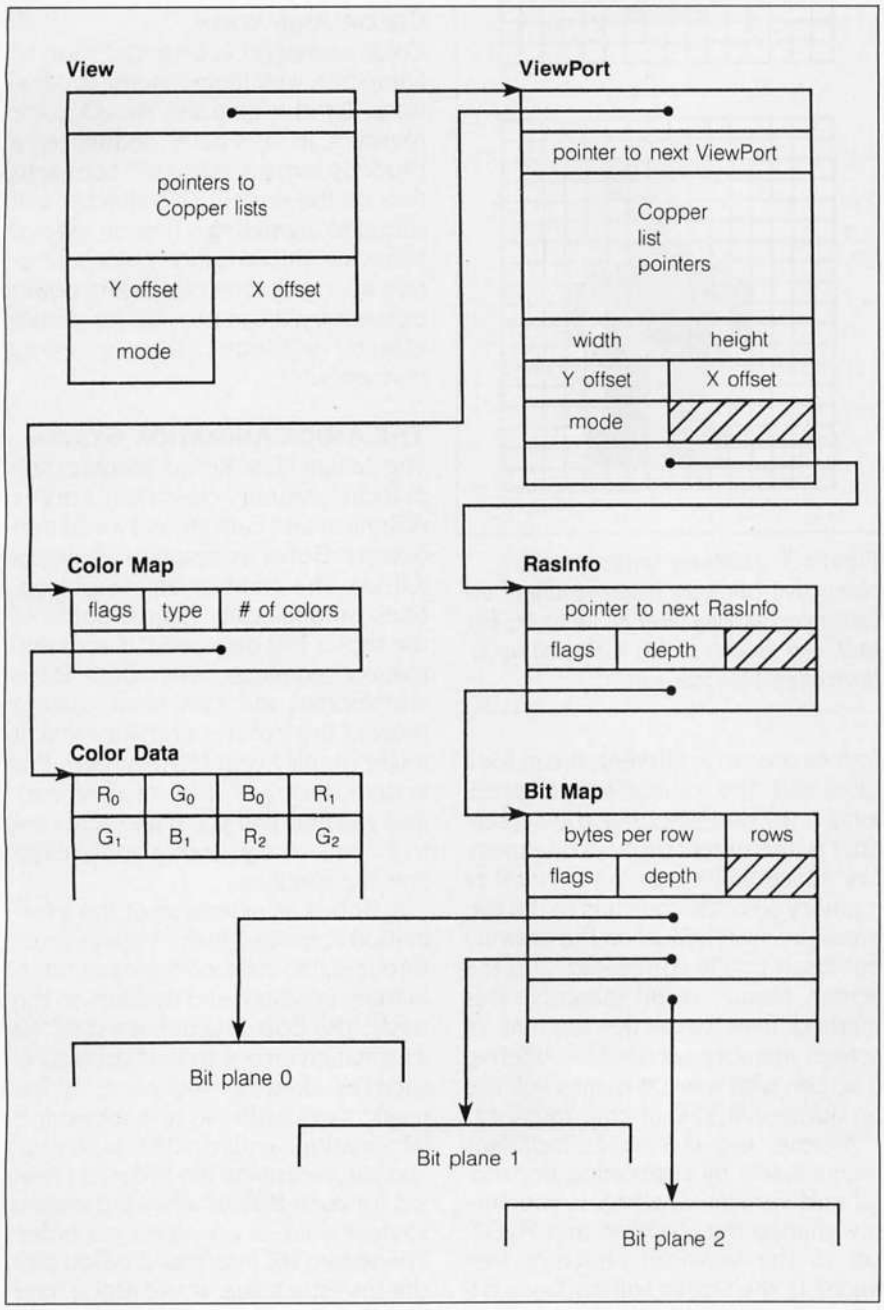


Figure 2: *The* ViewPort *structure contains pointers to the* ColorMap, *which tells the system which colors to use in the* ViewPort, *and to* RasInfo, *which tells the system where the* BitMap *is located. The* BitMap *structure contains pointers to from one to six bit planes representing from 2 to 64 colors, respectively.*

first-out order. Two moving and overlapping objects would be processed in this order: (a) save background 1, (b) place object 1, (c) save background 2, (d) place object 2, (e) restore background 2, (f) restore background 1.

If the animation object is not rectangular and you want the background to move behind the actual shape of the object, you can accomplish this by using the Blitter. First, the Blitter will OR all the bit planes of the object together to form a mask describing the shape of the object. Any of the colors can be chosen as the background color to be ignored in the mask. The Blitter will then AND the mask with each bit plane of the object to create an image of the object with a background of zero. It then inverts the mask and ANDs it with the background bit planes, creating an object-shaped hole. Now the Blitter will OR the object into the background in the hole. This procedure can be written as

$$D = AB + \overline{A}C$$

where A is the object mask, B is the object, C is the background, and D is the new animation frame. That is, the new frame is replaced with the object wherever the object mask is true, and with the background wherever the object mask is not true.

To implement this "cookie-cutter" operation, object and background data addresses are loaded into the Blitter source data registers BLTxDAT (where x equals A, B, or C, as above) and the minterm resulting from the above equation is placed in the BLTCON0 hardware register. The same thing can be accomplished by dividing the operation into two parts and using the BltBitMap function, which will take two sources at a time.

When the Amiga changes an image in memory, it does so by altering one bit plane at a time. Because of the finite period of time it takes to modify each bit plane, a moving object will tend to have its bit planes separate across the screen. Another problem occurs if the program starts drawing new information where the video beam is passing. This results in a screen consisting partly of old material and partly of new. If the two frames are very different, it can look quite bad. The solution to both problems is to double-buffer the screen. That is, the system displays one memory space while drawing in another memory area. The pointers to the two areas are switched when the drawing has been totally completed, and the screen retains visual integrity. This method uses twice the amount of screen memory, so double-buffering a screen with four bit planes will use an additional 32K of chip memory.

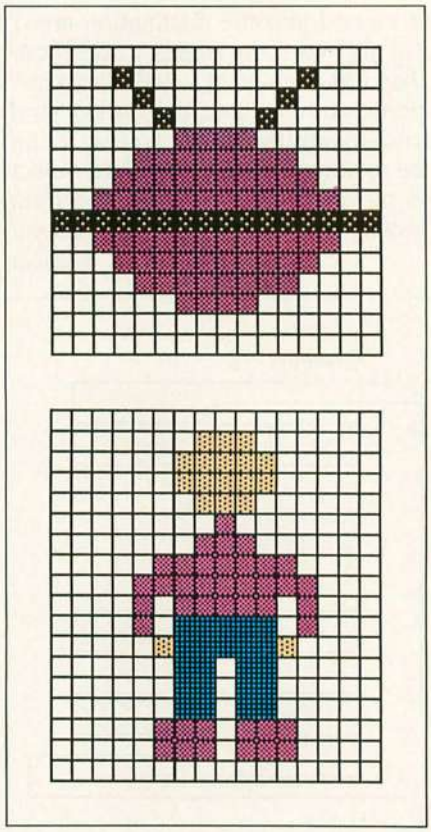


Figure 3: *Hardware sprites, graphic objects that can move independently of the background display, may be up to 16 bits wide, any number of bits high, and up to three colors plus transpartent.*

Another way the Amiga facilitates animation is by supporting horizontal and vertical scrolling. If you simply change the RxOffset and RyOffset of the ViewPort structure (see figure 1), the screen will display a different portion of the background data. No memory is moved, so the scrolling is fast and smooth. One quirk of the scrolling, however, is that it disables hardware sprites 6 and 7.

If you set the ViewPort to DualPlayfield mode, you will have two independently controllable playfields, one with a higher priority than the other. Each playfield can have up to seven colors plus transparent. A moving background outside an airplane window is the type of effect possible with this technique.

Color Animation

Color animation is a special form of animation with fairly limited applications. By changing the Amiga color registers in a regular sequence, a properly formed image will appear to flow on the screen. This effect is well suited to animating a flowing river or billowing smoke. Since color animation uses very little processing power or memory, it can provide for simple effects without utilizing many resources.

The Amiga Animation System

The Amiga ROM Kernel graphics animation system classifies sprites (VSprites) and background (or Blitter) objects (Bobs) as graphics elements (GELs). The graphics animation routines automatically handle some of the topics just discussed. If your animation sequence needs both Bobs and VSprites, and if you need to utilize most of the features of this system, it might simplify your job. However, this system does add a lot of overhead, and you may find you have more control if you write your own application-specific routines.

A Bob is an extension of the information contained in the VSprite structure (height, collision-handling information, position, and pointers to the data). The Bob structure handles the information unique to the background such as drawing sequence, image mask, save and restore background information, and double buffering. You can determine the order of drawing for each Bob or allow the system to draw them in y, x positional order. The system will first draw the Bob with the lowest y value. If two Bobs have the same y value, then the Bob with the lowest x value is drawn first. Objects drawn later overlap objects drawn earlier.

In order to have the system auto-

matically save the background to be restored after moving the Bob to a new location, you must set the SAVEBACK bit in the variable sprFlag of the VSprite structure and set the variable SaveBuffer to the address of a memory location. To "cookie-cut" the Bob into the background, set the OVERLAY bit and define the ImageShadow mask. To double-buffer, a Bob pointer is set in the Bob structure to a structure called DBufPacket. This structure contains information that helps keep track of the background in the current drawing buffer for correct restoration. If any of the Bobs are double-buffered, all the Bobs must be double-buffered.

Four variables describe the boundaries of a rectangle that will clip the Bob. If the GEL has passed completely outside the clipping region, the GELGONE flag will be set. If this GEL is no longer needed, you may delete it from the GEL list to speed up the overall processing. The SAVEBOB bit can be set to tell the system not to erase the old image of the Bob, which gives a "paintbrush" effect as the Bob moves. Once all of the GELs are moved or changed, they must be sorted with the routine SortGList and finally displayed with the routine DrawGList. DrawGList makes up a Copper instruction list.

The Amiga supports a set of structures and routines that will animate Bobs. The AnimOb (animation object) is the top-level data structure that organizes the AnimComps (animation components) and contains the registration point in the display relative to which of its component Bobs are drawn. The AnimComp is a component of the animation that contains the actual imagery, such as an arm, leg, or other part of the complete object. The AnimOb structure contains the initial position of the object, its velocity and acceleration in the *x* and *y* directions, a pointer to the first of a linked list of AnimComps, and a pointer to a special animation routine. The AnimComp contains a pointer to the next AnimComp in the sequence and a timer to tell the system when to switch. After the two structures are set up, calling the Animate function sets the animation in motion.

The Amiga animation software supports sequenced drawing and motion control, which can be used separately and together. In sequenced drawing, each view is a modification of the preceding view. This is particularly useful with an animation that is cyclical in nature, like walking. One step in a walking sequence would be a sequenced drawing. To make the object look like it's moving, each new view is positioned farther from a common reference point. After the animation has completed one cycle, the AnimOb must be moved a certain

(continued)

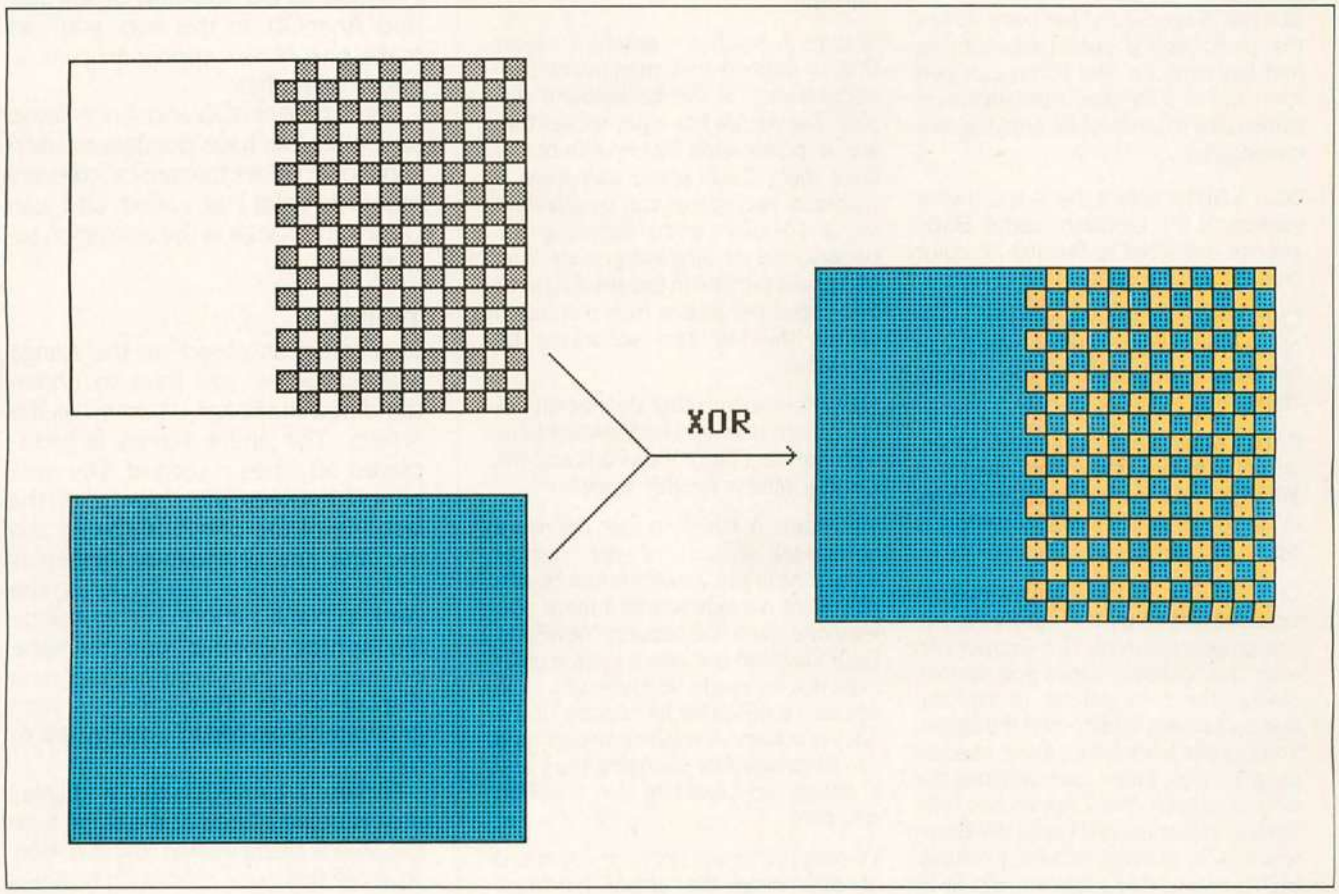


Figure 4: *One form of background animation on the Amiga involves using a pattern to* XOR *an area of the background. The pattern will appear on the screen with a different color than the background.*

AMIGA ANIMATION GLOSSARY

ANIMOB: A data structure that brings many AnimComps together in a whole object.

ANIMCOMP: An expansion of a Bob to allow the Bob to function as part of an AnimOb.

BIT MAP: A structure that contains pointers to bit planes and defines the width and depth of the bit-plane data.

BIT PLANE: An area of memory that defines the color of the pixels displayed on the screen. Each bit plane multiplies the number of possible colors in the display by 2. If one bit plane is used for a display, only two colors are possible. If two bit planes are used, four colors are possible. A maximum of five bit planes can be used in a low-resolution display and four bit planes in a high-resolution display.

BLITTER: A specialized hardware device that performs high-speed data copying and line drawing. The Blitter can perform up to 256 logic operations on three data sources while copying to a destination.

BOB: A Blitter object that is a software version of the hardware sprite. Bobs are not restricted to the size or color limitations of a sprite.

CHIP MEMORY: The lower 512K bytes of memory on the Amiga, which can be directly accessed by the custom chips.

COLOR MAP: A list of the red, green, and blue values that are attached to a ViewPort and loaded into the Amiga color registers when that ViewPort is being displayed.

COPPER: One of the Amiga's custom coprocessor chips, it controls the entire graphics system. The Copper can alter the registers, reposition sprites, change the color palette, update the audio channels, and control the Blitter. The Copper frees the 68000 to execute program logic rather than updating the display screen. The Copper has only three commands: WAIT until the beam reaches a specific screen position, MOVE a value into a register, and SKIP the next instruction if the beam is past a specified screen position.

GEL: A graphic element that can be manipulated by the graphic animation routines in the Amiga ROM Kernel. VSprites, Bobs, AnimComps, and AnimObs are all GELs.

IFF: Interchange Format File, the standard format for data written to files on the Amiga. This standard allows for data to be easily exchanged among development tools and products.

MINTERM: An 8-bit value that determines the logic operations to be performed by the Blitter during a data transfer operation.

RASTPORT: A data structure that contains information needed for manipulating the graphics display with the Amiga ROM Kernel routines. The drawing pen colors, drawing mode, area fill pattern, text attributes, font, pen position, and line pattern are stored in the RastPort.

SPRITE: A hardware graphics object that is defined and manipulated independently of the background display. The Amiga has eight sprites that are 16 pixels wide by any number of lines high. Each sprite can have 4 colors, or two sprites can be attached for a 16-color sprite. Scrolling the background or displaying more than 320 pixels per line in low resolution or 640 pixels per line in high resolution causes the last two sprites to be unusable.

VIEW: A structure that defines an entire screen display. The View contains a pointer to a list of ViewPorts and the *x* and *y* offsets for this screen.

VIEWPORT: A structure that defines a horizontal section of the display screen. Multiple ViewPorts can be displayed on a single screen if there is at least one blank line between ViewPorts. Each ViewPort contains a unique color map, display mode, width, height, *x* offset, and *y* offset for its portion of the display screen. A scrolling background can be created by changing the *x* and *y* offsets contained in the ViewPort structure.

VSPRITE: A virtual sprite, the method of describing the actual hardware sprites for use in the Amiga animation system.

distance to keep the apparent motion smooth. This distance is contained in the AnimOb structure.

Motion-control animation specifies objects that have independently controllable velocities and accelerations. The velocity and acceleration values are treated as 16-bit fixed-point binary fractions that have the form vvvvvvvvvv.ffffff. The slowest possible speed is one pixel every 64 frames. Each call to Animate causes the acceleration values to be added to the velocities.

The drawing precedence for AnimObs objects is determined by the precedence of the Bobs that make them up. The animation system automatically updates the precedence in the Bob structures for each frame to reflect the order of the first sequence. If more than one AnimOb is on the screen, one complete object can have precedence over another by linking the last Bob of the first AnimOb to the first Bob of the second AnimOb. In this way, you can make one object appear to pass in front of another.

Both the AnimOb and AnimComp structures can have pointers to user-supplied routines that are called every time Animate( ) is called and can cause any change in the animation sequence.

TIMING

To animate an object on the Amiga without flicker, you have to understand how the image is formed on the screen. The entire screen is redisplayed 60 times a second. The time period between the drawing of the last line of the previous screen and the first line of the next screen is called the vertical blank. During the vertical blank period on the Amiga, the sprites, Copper, and bit-plane pointers are initialized for the next display screen. The screen is then generated from top to bottom, left to right.

Flicker is caused when a display area is being altered at the same time the area is being written to the screen. To avoid this, your animation routines must complete all updates before the beam reaches the display area or after the beam has left the display

area. The current beam position can be determined by calling the routine VBeamPos.

Under the multitasking system of the Amiga, each task is given 4/60 second to execute before the task is preempted to execute the next task of the same priority. This can play havoc with any attempt to perform smooth animation, since your animation routines may not be executed in a regular time frame. One method of getting around this problem is to increase the priority of your animation task to a high level or use the Forbid( ) function to defeat the multitasking system and allow the animation to execute exclusively in the Amiga.

Another method is to attach a subroutine onto the vertical blank interrupt chain. This increments a counter, which can then be checked to determine the number of frames that have been displayed since the last animation update. The animation routines can then use this value to correctly position the animation objects.

Memory Usage

In one of our applications we have a scrolling background that is four screens wide and five bit planes deep, requiring a total of 160K bytes. We use six sprites to form one moving image. Each animation step uses approximately 1K byte. If the moving image has 40 steps, just the image and background data would consume 200K bytes of memory.

As you can see, it is practically imperative to add the additional 256K-byte memory expansion to the internal 256K on the Amiga to accommodate the operating system and any significant piece of animation. Currently, only the lower 512K bytes of memory are available to the custom chip hardware. If additional RAM is added via the 68000 bus, accessing image data from there will be slower than accessing data stored in the lower 512K bytes because all image and sound data must still be transferred to the lower 512K bytes to be used.

Materials and Methods

We would like to briefly describe how we have developed animation on the Amiga. We are using the Lattice IBM-PC-to-Amiga cross-compiler and the Metacomco assembler and linker to develop our programs. We wrote our own PC-to-Amiga parallel transfer program because we found the one supplied with the Amiga developer's kit to be too slow. Developing large programs on the Amiga itself will be difficult until additional RAM and a good hard disk are available. Adequate debuggers for the Amiga are only now becoming commercially available.

The mechanics of creating animated graphics are difficult, and good animation results only when artist and programmer take the system to its limits to produce visual effects.

The art for our animation was drawn using Deluxe Paint by Electronic Arts. Once again, we wrote our own routines to extract the data from the IFF file generated by Deluxe Paint and convert it directly into an assembly file. Then we used the structures and functions described earlier to program the animations.

Conclusion

The mechanics of creating animated graphics can be difficult enough, and the result is only as good as what you see in the end. If a car wheel in an animated sequence is rotating too fast for the speed of the background passing by, the animation will not reflect what you intended it to reflect. Good animation is created by both artist and programmer taking the system to its limits to produce visual effects. In this regard, the possibilities for animation on the Amiga are very exciting. ■

AMIGA VS. MACINTOSH

BY ADAM BROOKS WEBBER

A *programmer's comparison of the system calls on two 68000-based machines*

I HELPED IMPLEMENT the True BASIC language system on the Macintosh. The project has been completed for some time now and we've all recovered our perspective and good humor, but at the time I felt that if I saw that smiling "Welcome to Macintosh" message one more time I would scream. It was in this frame of mind, on my way to see the Amiga for the first time, that I wrote out a list entitled "Why I Hate the Macintosh."

That incident led to this article. I have now completed the same project on the Amiga and have many gripes about that machine, too. In this article I compare the system software of the two machines.

Implementing a language system is a good way to get to know a machine. True BASIC is not just a compiler and interpreter, it's a screen editor, a graphics program, and a number cruncher. It makes sounds, it prints, it manipulates files—in short, it uses most of a system's software. My comparison of the Macintosh and the Amiga is necessarily limited, but I have tried to choose areas that are of general interest and that are representative of the differences between the two machines. These are the user interface, graphics primitives, printers and other devices, multitasking, and memory management.

THE USER INTERFACE

The user interface of a system is usually examined from the user's point of view. You ask, "Is it intuitive? Is it powerful? Is it forgiving of error?" as you consider how to communicate with a program. I am interested in another point of view: What does a program have to do to communicate with the user? For some machines, communicating with the user means reading characters from the keyboard and writing text to the screen. Adding windows, menus, and mouse operations makes things more complicated.

Just how complicated is shown by the Macintosh user interface software, which determines the structure of every Mac program more or less completely. The program must have a main loop that is executed as often as possible, usually 60 times a second. (One rule of thumb: The quality of system software is inversely proportional to the number of things a program must do "as often as possible.") The main loop checks for events like keystrokes, mouse clicks, and disk insertions and responds to them.

Responding to an event is usually a lot of work because the Mac's user interface software provides little assistance. Figure 1 shows, in pseudocode, a simple example of what a program must do to allow a window with a scroll bar to change in size—something that most Mac programs have to deal with. The system software does offer significant help in two areas of user interface: text editing and file access. TextEdit is a subsystem that provides a mechanism for displaying and editing text in a window. (True BASIC doesn't use TextEdit, but only because we have our own internal string-handling procedures.) For

(continued)

Adam Brooks Webber (39 South Main St., Hanover, NH 03755) is a design engineer for True BASIC Inc. He has a B.A. in mathematics from Dartmouth College and was involved with operating system and language system development there before going to work for True BASIC.

file access, the system software provides the Standard File Dialog, which gives a graphic display of the contents of a disk and allows you to choose a file. It's an excellent tool and almost all Mac programs (including True BASIC) use it.

Intuition, the Amiga user-interface software, supports two different mechanisms for communicating with the user. The console device is the simpler of the two. From your point of view the console device looks like a no-frills window, but from the program's side it looks like a terminal. The program gets a sequence of characters as input and sends off a sequence of characters as output. Mouse clicks and other nontextual input come through the console device as special ANSI-defined character sequences. The console device is a very easy way for text-only applications to communicate with the user. It is especially useful for prompt-based applications like those typical of UNIX and the IBM PC.

The Amiga's other, more powerful user interface mechanism is the Intuition Direct Communication Port, or IDCMP. The Amiga Executive allows tasks to communicate with each other through message ports; an IDCMP is a message port that has a program at one end and Intuition at the other end. Events are communicated to the program through the port. This is a very flexible scheme: The program can poll the IDCMP, or it can arrange to run only when there is some event for it to work on, or it can spawn a separate task to handle events. And the events themselves are much more straightforward to handle—figure 1 shows in pseudocode what an Amiga program using an IDCMP must do to allow a window with a scroll bar to change in size. The program can get more control of the resizing process if it wants to, but for most programs (including True BASIC) it works almost automatically.

In future revisions of Intuition I'd like to see something like the Mac's TextEdit for displaying and editing text in a non-console-device window and some kind of file interface like the Standard File Dialog. But in spite of these deficiencies, I prefer the Amiga user interface. It is more flexible and usually requires less work on the part of the program.

Amiga	**Macintosh**
Anything happen? *Yes, your window is a new size.*	Anything happen? *Yes, the mouse button was pressed.*
Fine.	Where was it pressed? *In the resizing box.*
	If the window isn't frontmost, make it frontmost. *OK.*
	Otherwise, what size does the user want? *This size.*
	Hide the scroll bar. *OK.*
	Change window sizes. *OK.*
	Change scroll bar to the location and size I've calculated. *OK.*
	Redraw the scroll bar. *OK.*
	Remember to redraw the part of the window whose size I've calculated. *OK.*

Figure 1: *Comparisons (in pseudocode) of a program's actions and the operating system's responses to a user event for both the Amiga and Macintosh. The user attempts to resize a window that contains a scroll bar.*

GRAPHICS PRIMITIVES

The Amiga has more in the way of graphics special effects than the Macintosh. It has color, many different screen depths and resolutions, and special-purpose animation hardware. I suppose there's no comparison when it comes to games, simulations, art programs, and that crowd. But for a language system what I really want is a complete, logical set of basic graphics primitives. I want the system software to handle straightforward drawing on the screen. In this area the Macintosh system software excels.

The Mac knows how to draw rectangles, rounded rectangles, polygons, ellipses and arcs, and general closed figures. These basic objects can be outlined, filled in with any black-and-white pattern, or inverted (turning black pixels in the area to white and vice versa). The Mac can also draw lines and write text in software-definable fonts. The Macintosh graphics subsystem, called QuickDraw, supports a mechanism for recording graphics operations in a compact way and then playing them back.

The Macintosh doesn't support flood-fill. Flood-fill is what the "paint can" does in MacPaint: Starting at some point, the program colors in the screen, stopping at whatever boundary lines it comes across. True BASIC and several other programs accomplish this with custom software. (Note: The new 128K ROMs from Apple do support flood-fill.) Another weakness is the ellipse-drawing algorithm used by the Mac: An ellipse that is substantially longer on one axis than on the other is drawn unconnected, with frequent perforations. This means that such an ellipse cannot be flood-

filled—it leaks! Plenty of published ellipse-drawing algorithms don't have this undesirable property.

The Amiga's graphics primitives are less well developed than the Mac's. The Amiga knows how to draw rectangles and polygons. Objects can be outlined, filled with any multicolored pattern, or inverted. Sometimes the machine can perform a combination of these operations, like outlining with one color and filling with another, in one pass. The Amiga does support flood-fill. It draws lines and supports software-definable fonts. However, it does not draw ellipses or any other curved figures at all.

Figure 2 summarizes the graphics primitives of the two systems. The Amiga graphics software strongly reflects the Amiga graphics hardware. The hardware helps the system draw lines and rectangles, so lines and rectangles are well represented; other figures are not.

Another example of this kind of hardware-oriented design is the software interface to the blitter. The Amiga's blitter is a custom chip that does bit-aligned data manipulation. One of its many duties is moving screen images around. The strictest hardware limitation is imposed on the blitter: It can access only the lowest 512K bytes of memory. That's fine, that's what hardware is all about; but the software that uses this hardware should relax that restriction. It doesn't. One of the things you might want to do with a lot of memory is to store lots of images in it. This is possible on the Amiga, but you have to copy those images down into the lowest 512K of memory before any of the graphics routines will touch them.

For these reasons I prefer the Macintosh graphics primitives. They're well chosen, and they have natural interfaces. The Amiga graphics software is very powerful, in its idiomatic way, and it cooperates well with the hardware. Is it too much to ask that it cooperate with the programmer too?

DEVICES

A printer is the programmer's bane. Printers are so similar on the outside that it seems ridiculous not to support a large variety of them—but they're all annoyingly different on the inside. Both the Amiga and the Macintosh try to get around the problem of printing by offering device independence. This means that the system software deals with the problem by offering a single device interface, regardless of what is connected to the computer.

To a program running on the Macintosh, the printer looks like a video screen. The program draws onto this screen just as if it were the video display, and the system software takes care of translating those operations into something the printer will understand. This makes good sense, but in practice it doesn't work out well at all.

The different Apple printers (and there are really only two) accept different subsets of the graphics primitives, so an operation that prints on the Imagewriter may produce no effect on the LaserWriter. The program is substantially involved in setting up, adjusting, paginating, and disposing of the special printer "screen." All this overhead is acceptable for a program that is printing complicated graphics, but for an ordinary text-only printing program like True BASIC, it's a lot of work.

An especially odd aspect of printing on the Macintosh is printing "in the background." Programs can allow you to do other things while the printer is busy. To do this, the program gives the Printing Manager (the software to handle printing operations) a procedure that it can call *as often as possible*. The program calls the Printing Manager, and the Printing Manager calls the program over and over to lend it processor time, and then when printing is done the Printing Manager returns to the program. This is ridiculous, but it's the Mac's lack of multitasking that is at fault, not the printing software—and multitasking is discussed in more detail below.

The Amiga offers several ways to use the printer. One is specifically designed for programs that print text only. Using this method, the program deals with the printer as if it were a file. The printer can be opened, written to, and closed just like a part of the file system. Special fonts and other features of the printer can be activated by the Amiga's printing software, which generates the required control sequences for the specified printer. Programs that need even more control over printing can access the printer driver directly. For printing graphics, the program passes the printer driver the same image used by the graphics primitives, rather like the Mac's scheme but a bit less automatic. [*Editor's note: The Amiga uses printer-specific drivers that perform the conversion from generic primitives to hardware-specific control sequences. You select the driver that fits your printer from a variety of printer drivers supplied with the system disk.*]

For other devices, like the serial port, the Amiga uses the same two-level scheme: The device can be

(continued)

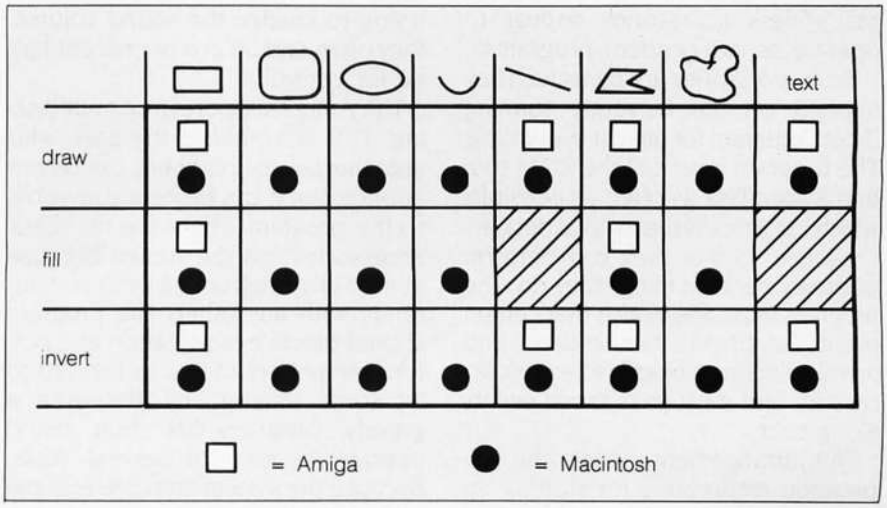


Figure 2: A *comparison of the graphics primitives supported by each machine.*

Consistent, high-level access to a variety of devices is a strong point of the Amiga that has almost no counterpart in the Macintosh.

opened as a file for simple access, or it can be manipulated directly for more complicated operations. The Amiga's parallel port and RAM disk device also use this method. Consistent, high-level access to a variety of devices is a strong point of the Amiga that has almost no counterpart in the Macintosh. For example, the Mac's serial port is accessed only through a low-level driver. (Actually, it's accessed through two low-level drivers, one in ROM and one in RAM; but the latter seems hardly more than a debugged version of the former.)

MULTITASKING

A multitasking system is one that runs several independent programs at once, giving each the impression that it is the one and only. The Macintosh system software does not support multitasking. Given this, how do the Mac's desk accessories appear to operate as independent programs?

Desk accessories are parasitic. They depend on the currently running "host" program for almost everything. The program must call the ROM routine SystemTask as often as possible, which in turn calls the various desk accessories so that they can perform whatever periodic things they do. The program must also notice that certain events (keystrokes, mouse clicks, and menu selections) belong to a desk accessory and must pass those events along to it.

This arrangement makes the application responsible for starting up desk accessories. A program that allows the use of desk accessories must find out which ones are available, create a menu listing them, and start up any desk accessory selected by the user. This means that desk accessories interact badly with the rest of the system and waste more programmer time than anything else I can think of on the Mac. Every program on the Mac becomes a frantic conspirator in the multitasking cover-up. Well, almost every program—some just give up on desk accessories, and others (like MacPaint) sharply restrict their use.

The only thing worse than supporting a desk accessory is being one. Desk accessories are hard to write because they're constructed so differently from the host programs they depend on. They're written as device drivers—which means, among other things, that they are table-driven, that they have to be small (about 8K bytes at the most), and that they have to be very careful not to alter the environment they work in.

One thing a multitasking system does is to mediate between programs that want to use the same device at the same time. If one program is using, say, the printer, another program will be told that the printer cannot be used right now. No such orderliness exists on the Macintosh. The Control Panel is a desk accessory that allows you to change (among other things) the volume of the Macintosh's sound. If a program is using the Sound Driver to make sound at the same time that the Control Panel is trying to change the sound volume, they often end up in a permanent battle for control.

The Amiga supports true multitasking. This is visible to the user, who sees that several programs can be run at once, but it can be nearly invisible to the program. There are no "desk accessories" on the Amiga because almost any program can run concurrently with any other. The program should practice moderation and not, for example, count to a million just to let some time go by. But even a greedy program like that could operate as one of several tasks, because the system software ensures that every task gets processor time. The system also mediates device access, so that problems cannot be caused by having two programs accidentally access a device at the same time. The only thing a program really needs to be polite about is memory usage, because any program could take up all available memory if it wanted to. A multitasking system really needs a more sophisticated plan for memory management than this, and, as you'll see, I have one in mind.

The Macintosh should have had multitasking. I can't stress enough what a big contribution it makes to the elegant design of system software. The Amiga has an excellent multitasking system, and I think it will have twice the product life of the Macintosh because of it.

MEMORY MANAGEMENT

The architecture of the 68000 processor requires a stack, which is an area of memory used for subroutine addresses and other last-in/first-out data. Since the Mac and the Amiga both use the 68000, both have stacks; but the multitasking Amiga needs a separate stack for each task, and this results in a memory arrangement very different from that of the Macintosh.

The Macintosh has one stack, which grows from the top of memory down to a preset limit. It has two *heaps*, the system heap and the application heap. (Heaps are areas of memory for blocks that, unlike stack blocks, need not be freed in any particular order.) The system heap is small and cannot grow; it is used mostly by system software and is preserved across runs of a program. The application heap can grow up to a preset limit (the same limit toward which the stack is growing, from the other side). The application heap is reinitialized each time a program is run, so programs need not remember to deallocate blocks when they are shutting down.

Many heap blocks on the Macintosh *float*; that is, the system software may decide to move a data block without consulting the program that allocated that block. Often it will move all allocated blocks to one end of the heap so that all free memory is in a

(continued)

continuous piece at the other. This helps reduce fragmentation and so conserves memory. Figure 3 shows a sample Macintosh memory layout.

One problem with the Mac's memory management scheme is the preset limit that divides the stack from the heap. The program has to decide where that limit is going to be—once set, it cannot easily be altered. For many programs it is difficult, if not unnatural, to set such a permanent fence. True BASIC can't tell what kind of memory demands the user is going to make: Running a BASIC program that uses big strings would use a lot of heap memory, but running one that's heavily recursive would use a lot of stack. Another problem is the complexity introduced by the floating blocks. Naturally, there's a way for programs to find the current location of a floating block, but the current location isn't current for very long and it's easy to make mistakes.

The most irritating problem with the Mac's memory management is that the stack can easily grow down past the limit and wipe out part of the heap. The system software uses some unspecified amount of stack space in addition to the amount used by the program. This makes it doubly difficult to determine when the stack is growing too low. There is virtually no way to guarantee that the stack won't overwrite the heap, and most programs merely try to make it as unlikely as possible.

The Amiga's memory management places everything at fixed locations in one heap: Data blocks do not float. Some memory is not accessible to the custom hardware, so programs need to explicitly request chip-accessible memory (that is, available to the graphics chips) if they want it. Each task has a stack, which is usually rather small (less than 8K bytes), and each task has to cope with the same problem of noticing when the stack is full. The operating system does not use the task's stack at interrupt time, which makes it somewhat easier to avoid stack disasters.

One drawback of the Amiga's memory management is that a program must remember exactly what heap blocks it allocated and must deallocate them before shutting down. If it doesn't, the memory cannot be reused until the system is reinitialized (as by turning the machine off and then back on). The small, fixed maximum size for task stacks is also a nuisance. Most annoying is that other tasks may have allocated immovable blocks at any location in the heap, so a program cannot count on finding large continuous pieces of memory free. This is part of the reason why task stacks are usually so small—they have to be small enough so the program can be pretty sure of being able to allocate one.

Considering the design constraints of the Amiga, I don't see how the memory management software could have been much better. It's simple and fast, unlike the Mac's memory

(continued)

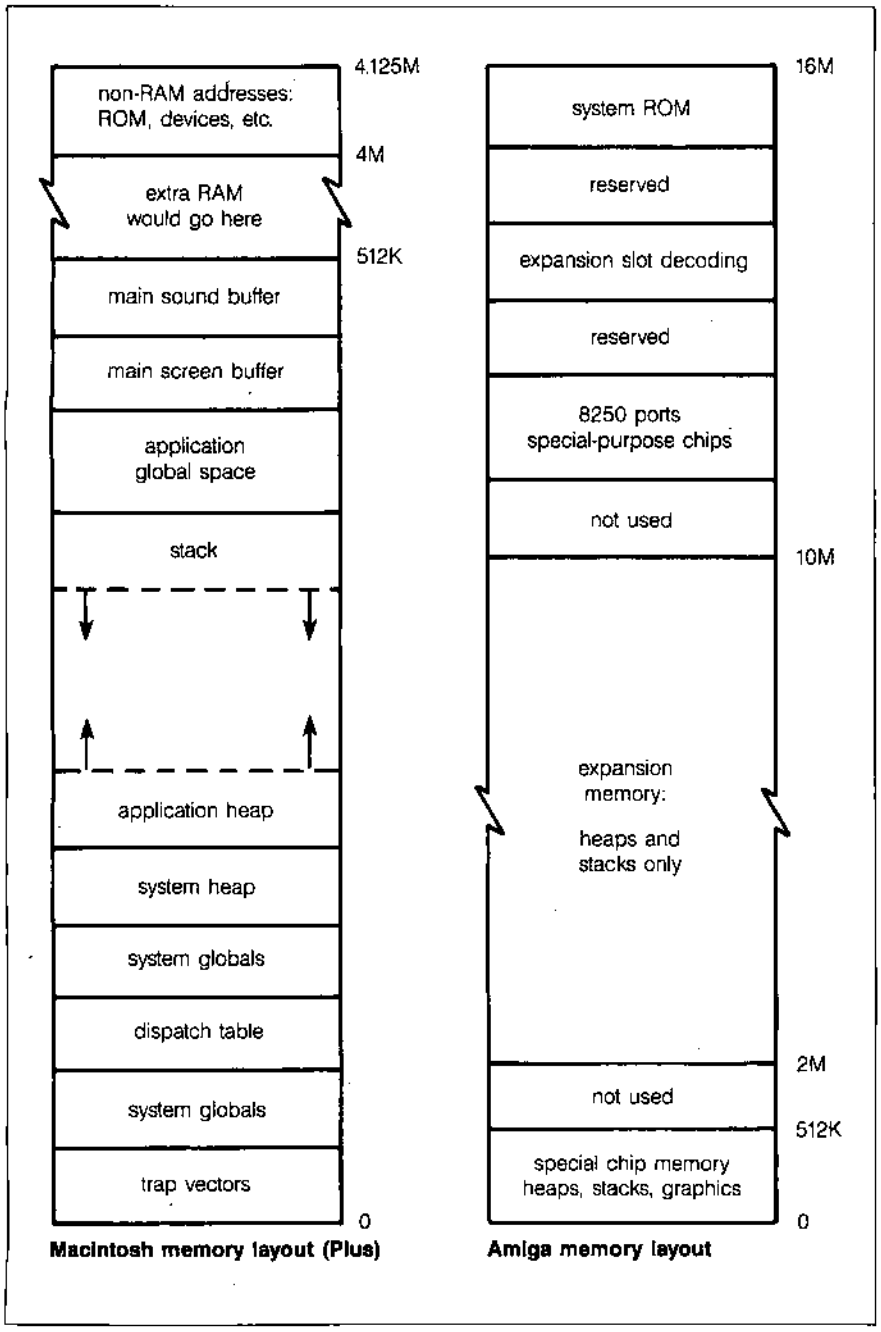


Figure 3: A *comparison of the Amiga and Macintosh memory maps.*

management software, which, though it reduces memory fragmentation, is complex and slow. What I'd like to see in future machines is some memory management hardware. Mainframes have had the benefit of virtual memory systems for the last 20 years, and it's time this technology made it to micros. When people think of virtual memory systems they often think of *paging*, the automatic use of disk space as an extension of RAM. But with memory so inexpensive, who wants to page to a floppy disk? The real benefits of memory management hardware are *segmentation* and *access control*.

Segmentation means that memory is grouped into logical areas, not physical ones. There might be a segment for the stack and a segment for the heap. Hardware would take care of mapping these continuous logical areas into possibly noncontiguous pieces of RAM and would prevent them from ever running into each other. Access control means that the segments can be restricted for certain kinds of operations: A segment containing scratch space could be examined and altered but not executed, while a segment containing instructions could be examined and executed but not altered. Memory management hardware like this may never become an absolute necessity for single-user systems, but it would certainly make software development easier and more fun. I'd like to see it in future machines from Apple and Commodore.

Conclusion

Both of the 68000-based machines have excellent system software in places, but the Amiga is the winner in five of my six areas of comparison. The Amiga software is a bit thin: For example, it could use a more complete set of graphics primitives. But the Amiga's shortcomings are minor in comparison with some of the Macintosh's deep-rooted problems. The Macintosh lacks multitasking but tries to fake it, and it insists on a complicated user interface but leaves much of the work up to the application. These are serious drawbacks, and it is difficult to imagine elegant repairs for them. ■