NOVEMBER 1980 Volume 5, Number 11 $2.50 in USA/$2.95 in Canada

BYTE®

A McGRAW-HILL PUBLICATION

the small systems journal

HIGH-RESOLUTION GRAPHICS

Language Control Structures for Easy Electronic Visualization

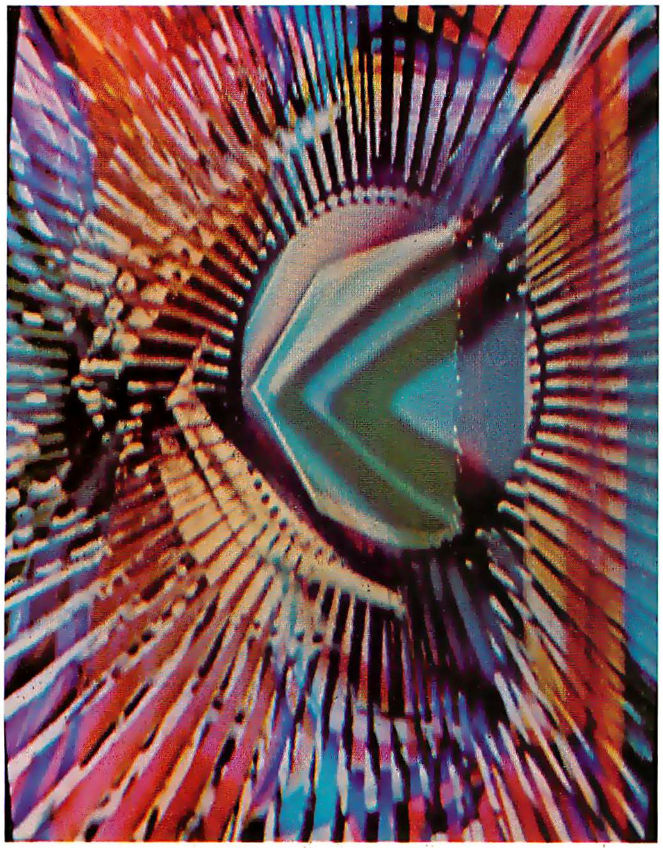
Dr Thomas DeFanti
Electronic Visualization Laboratory
University of Illinois at Chicago Circle
POB 4348
Chicago IL 60680

Control structures are the program- flow manipulation features of the language that you use to beat your computer into submission. BASIC's control structures are embodied in the RUN, GOTO, GOSUB, and RETURN keywords and a few functions, certainly an impoverished set. Highly structured languages like Pascal are rigidly limited to the control structure of subroutines. Lowly structured approaches like assembly language are necessary to implement higher-level languages and real-time systems, because the lack of enforced structure allows an infinite variety of control structures to be used at a cost of great human effort. The execution-speed gain in using assembly language is more due to the efficient building of customized tables and linked lists than to efficiency in adding, subtracting, multiplying, and dividing numbers.

Assembler coding is by no means easy. Note the word "easy": it's important because in one sense it means *"accessible."* In this case, it's your access to complex electronic visualizations.

Electronic visualizations are important because producing and manipulating images, especially animated ones, is a truly multidimensional task which reflects our real-world interactions much more than maintaining an accurate laundry list or printing payroll checks. Producing them demands a lot from software,

1a



1b

Photos 1a and 1b: *Sample output from the GRASS/Image Processor. Photo 1a was made by Guenther Tetz, and photo 1b by Dan Sandin and the author.*

Circle 55 on inquiry card.

and making their access easy requires paying attention to the provision of rich control structures in a language.

Electronic Visualization is an intentionally broad term meant to conjure thoughts of computer graphics, animation, image processing, video synthesis, and even advanced word-processing. Anyone successfully producing images for communication is unlikely to reject a technique for reasons of algorithmic purity (as a computer scientist might feel forced to do). Computer hobbyists use the tools at hand, and electronic visualization is the means to the end and the end product of using these tools. Simultaneously, it can be *both* because we are seeing the vast increase of real-time imaging systems, even in microcomputer-based configurations; and controlling these real-time systems can be as feedback-intensive as playing a musical instrument or driving a racing car.

Just to unify the concepts so far, think about this question: what besides the cosmetic packaging governs our choice of a musical instrument or an automobile? It is a combination of capability and user control, of course: having one without the other is useless. So why are the programming languages currently available so impoverished on the control-structure side?

The most successful approaches to date are basically highly developed, beautifully evolved kluges.

Perhaps it is because computers were invented to process payrolls, not images. Television, on the other hand, is image-oriented and currently uses a host of presently emerging real-time digital techniques and increasingly flexible control structures. As a matter of fact, just about all the television you see these days is digitally processed for purposes of synchronization.

Television is a high-speed medium conducive to parallel and pipeline processing. You are driving television rather than generating it. TV cameras are on all the time and you, as director, are fading, switching, adding titles and constantly throwing away images that you don't want. Control is the name of the game.

The television folk are not about to give up rich, real-time control structures and the computer folk won't give up language. How to get them together is the essence of the task at hand.

Getting Computers and Television Technology Together

Looking at the history of control structures for computer graphics and for television, we see that most computer-graphics usage, with the obvious and exciting exception of video games, is some variety of non-real-time plotting. This is where the money is and where the language development for computer-aided design has been focused. No manufacturer of equipment for computer graphics (excepting the video-game people) now depends on animation for solvency. Plotting is slow and often merely the side output of a large FORTRAN finite-element analysis program. Visual aesthetics are rarely the primary concern, if any concern at all. People who use such systems are highly skilled and highly paid technicians who became that way by having to deal with plotting packages as a condition of employment. If the job were easy, they wouldn't get paid so much.

We are just reaching the point of electronically generating and manipulating images, in real time, under program control. How do we design languages to deal with real time? Or, more important, why do we want such a language, an alphanumeric string-oriented language, at all? Why not use picture-based languages with symbols for motions and timing?

How Can You Control Images Easily?

After about ten years of living with this obvious and nagging question, some conclusions became clear. First, purist approaches to electronic visualization are hopeless. Image control employs a hybrid of languages, several input devices, picture-oriented commands, custom hardware, and a smattering of idiosyncrasies. The most successful approaches to date are basically

A CREATION OF COMPUTER HEADWARE

WHATSIT?

(Wow! How'd All That Stuff get In There?)

A sophisticated, self-indexing filing system—flexible, infinitely useful and easy to use, that adapts to your needs.

WHATSIT's unique capabilities:

Multiple Entries *allowed per field: For example, a bibliographic file can associate each work with any number of authors. WHATSIT allocates file space as needed for each.*

New Data Fields *added "on the fly": You're not confined to a particular "record layout" that must be declared in advance. Your file evolves to fit your needs.*

Immediate Response: *Even in the largest files, WHATSIT responds in seconds, thanks to pointer linkages and hash coding.*

Conversational Dialogue: *Query and update requests may be intermixed in any order, without returning to a "menu selector."*

NEW
Apple II Plus WHATSIT at special introductory price: $95

(Regular price, $150 after December 31, 1980).

WHATSIT comes ready to run on your Apple, Apple II Plus, AlpaMicro NorthStar, or CP/M computer. See your dealer for a full demonstration. . . or write or call:

HARDHAT Software

P.O. Box 14815 • San Francisco, CA 94114 • Tel: (415) 621-2106

Circle 56 on inquiry card.

highly developed, beautifully evolved kluges. We know what "purism" in coding FORTRAN and BASIC does to image production. Purism in television technique eliminates computer graphics as we know it. So how about using graphic symbols to save the day?

Using symbols in a menu and some sort of manual-selection mechanism is an approach taken by many FORTRAN graphics systems. This limits the number of symbols to those defined in the menu and there is no user-level extensibility in that you cannot create new symbols out of sequences of old symbols, which eliminates the one truly unique feature of computers. To state it bluntly, you can't program with a menu.

What happens, however, if you do find a system that provides for the combination of nonalphanumeric symbols in meaningful ways? In an extremely advanced case, it should look something like Japanese, and you might note that the language used to program computers in Japan is a *phonetic alphanumeric transcription* of their language. They do not program in their extremely beautiful and rich symbol set. Eliminating alphanumeric languages is not such a hot idea, except in turnkey systems.

The second conclusion gestating for the past ten years is that complete parallelism is necessary for controlling images in meaningful ways. You simply must be able to develop sequences independently and merge them in ways that do not necessitate rewriting the programs. Xerox's Smalltalk and certain other languages have this capability, as do television technology and everyday life: making this parallelism easily accessible takes real care.

The third conclusion is that a flexible priority scheme is needed. Some tasks are more important than others, just as in real life and computer operating systems. It is essential to give this capability to the user of an electronic visualization system.

Fourth, providing for user extensibility at several levels is the only way people will easily be able to use a system for applications not envisioned by the designer. I will discuss this later.

Fifth, the system must be software-fault tolerant. Fault-tolerant hardware has been a research area of great importance to real-time control systems, yet language purists still think people should solve problems in structured, orthodox, algorithmic ways. A computer language should provide as many paths to a given communication as possible, as natural languages do, and the kind of error handling that a friend would offer. Allowing nonstructured, nonprocedural, "seat-of-the-pants" programming is often the only salvation when the final goal is aesthetically defined, and is, perhaps, not at all clear. It has been called "fuzzy programming," and it's easy to throw in the recursive, value-returning, clever structured-programming capabilities as well, but limiting yourself to these latter approaches stifles human creativity, problem-solving, and sideways thinking.

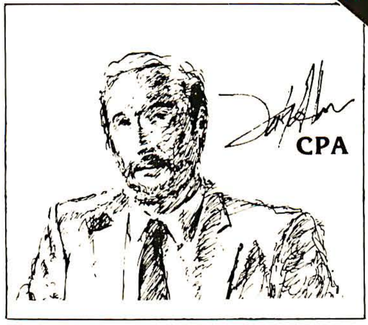
Zgrass — A Language for Easy Electronic Visualization

Zgrass is a programming language and operating system written in assembly language for the Z80 microprocessor by Nola Donato, Jay Fenton, and me. Not surprisingly, it embodies all the control structures mentioned so far in this article and

Tax Practitioners / CPAs:
Successful Tax Professional Reveals Effective Practice Development Formula....

NEW!

"Every tax professional...whether he is desirous of expanding his practice, or just better serving his existing clientele (or both)...can profit from this system."



Are you satisfied with the growth of your tax practice?

Successful practice development is a perplexing problem facing all tax professionals today. The field tested MICROTAX professional tax preparation software package can provide you with an innovative solution...and at the same time enable you to serve your existing clientele in a more professional and timely manner.

Consider the advantages this state-of-the-art package can bring to your practice:

- **Complete System...**
  ...containing Federal Individual, Corporate AND State Individual returns. The system is designed to accept information, summerize the data, compute the tax, and print the returns; including **all** forms and schedules required by the I.R.S.
- **Versatile**
  MicroTax is cost-efficient for practices preparing as few as 20 returns per tax season...but comes with a fast mode, capable of processing 2,000 returns—or more!
- **Best Of All...**
  ...and this is really exciting...MicroTax allows you to offer **preyear-end tax planning** to your clients—enabling you to predict the potential liability and take steps to minimize the tax in a timely manner.
- **Complete In-Office Security**
  —no risk of sensitive client information falling into the wrong hands.
- **Saves Time**
  Compared with conventional service bureaus, MicroTax offers virtually instant turn-around time.
- **More Professional**
  Greater range of services allows you to present a more professional appearance to your clients.
- **Requirements:**
  CP/M, Microsoft Basic & 48K Memory
- **Introductory Price ('til Dec. 1, 1980)**
  MicroTax is available as a three part system:

Federal Individual	$750.00	Annual Updates
State Individual	250.00	Are Available
Corporate	250.00	

MicroTax is a versatile and money-saving tool that **NO** tax professional should be without (it's been selected for use by COMPU-TAX of Utah).

Call or write today for additional information, or your nearest dealer. Problems that can't wait? Call Don White our V.P. of Research and Development, direct, at (213) 668-0238.

Circle 272 for microTax

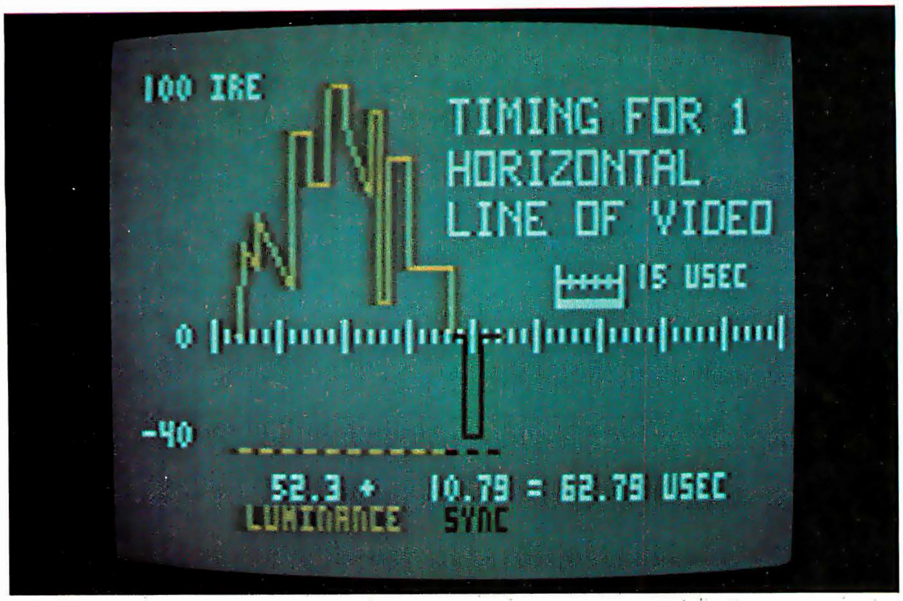
Circle 58 for SOFTWARE Inc.

microTax
Income Tax Software Specialists

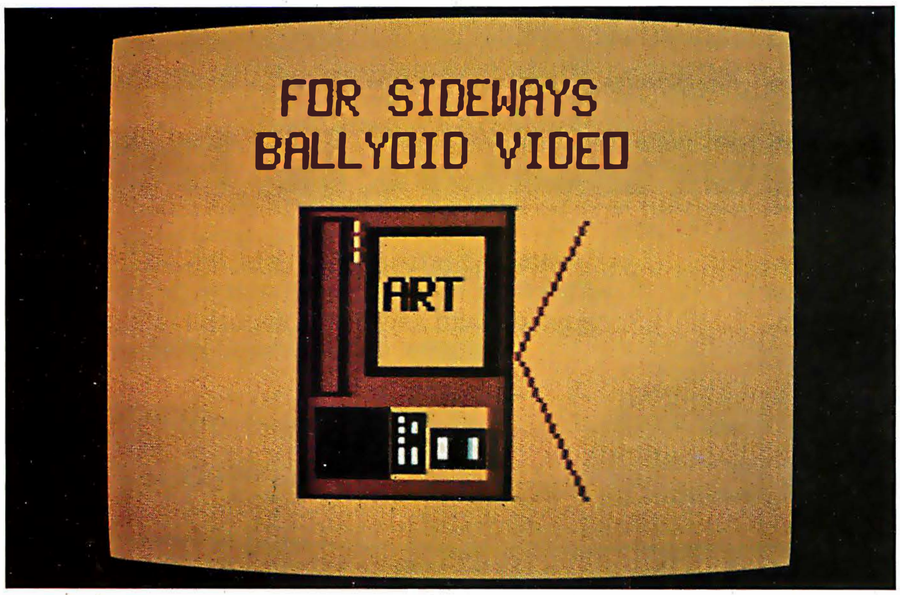
Available at most Professional computer retailers

Exclusive Distributor
S.O.F.T.W.A.R.E.+, Inc.
3600 WILSHIRE BOULEVARD, #1510, LOS ANGELES, CALIFORNIA 90010 • **(213) 738-9972**

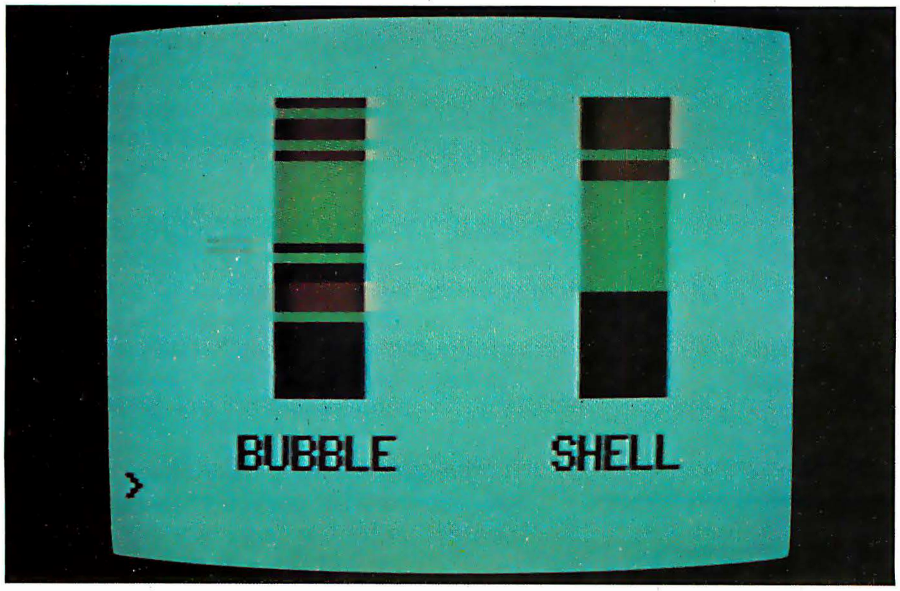
2a



2b



2c



Photos 2a, 2b, and 2c: *Sample output from the first Zgrass system, with a resolution of 160 by 102 pixels, with 2 bits per pixel. Photo 2a was made by Copper Giloth, and photos 2b and 2c by Nola Donato.*

has been in development for ten years.

Zgrass started out as GRASS (Graphics Symbiosis System), a language designed to bring the immense complexity of a Digital Equipment Corporation PDP-11/45 and as Vector General 3DR Display system within the grasp of artists and educators at Ohio State University. It has high levels of interaction, parallelism, priority, and tree-structured manipulations of vector-defined objects. Photos from this system can be seen in "About the Cover... And Some More of the Same," in the October 1977 BYTE, page 22.

GRASS depends on $120,000 of equipment to run — rather expensive for a single-user system — but it is one of the first highly developed non-FORTRAN interactive graphics systems for use by artists.

In 1973, Dan Sandin, inventor of the Image Processor, brought color television usage to our computer graphics work at the University of Illinois at Chicago Circle. Dan and I developed most of the ideas about control structures presented here. Photos 1a and 1b show some output from the GRASS/Image Processor system.

Generating a complete programming language with parsers, compilers, and graphics takes a lot of human effort. More than ten person-years of programming were devoted to GRASS, aided by generous support from the National Science Foundation, National Endowment for the Arts, and others.

GRASS is totally oriented toward real-time generation and control of images for the simple reason that television cannot easily be slowed down for long and/or time-lapse exposures as can be done with film. The control structures for GRASS were developed ad hoc and became increasingly idiosyncratic. Nola Donato, a postgraduate student of mine, decided to teach me how to generalize many of the programming-language concepts. The result was GRASS3, which later became Zgrass.

In 1977, I was led to Jeff Frederiksen at Dave Nutting Associates, who was developing a deluxe home computer for Bally Corporation using the custom integrated circuits they had developed for the Bally Arcade video game. The pros-

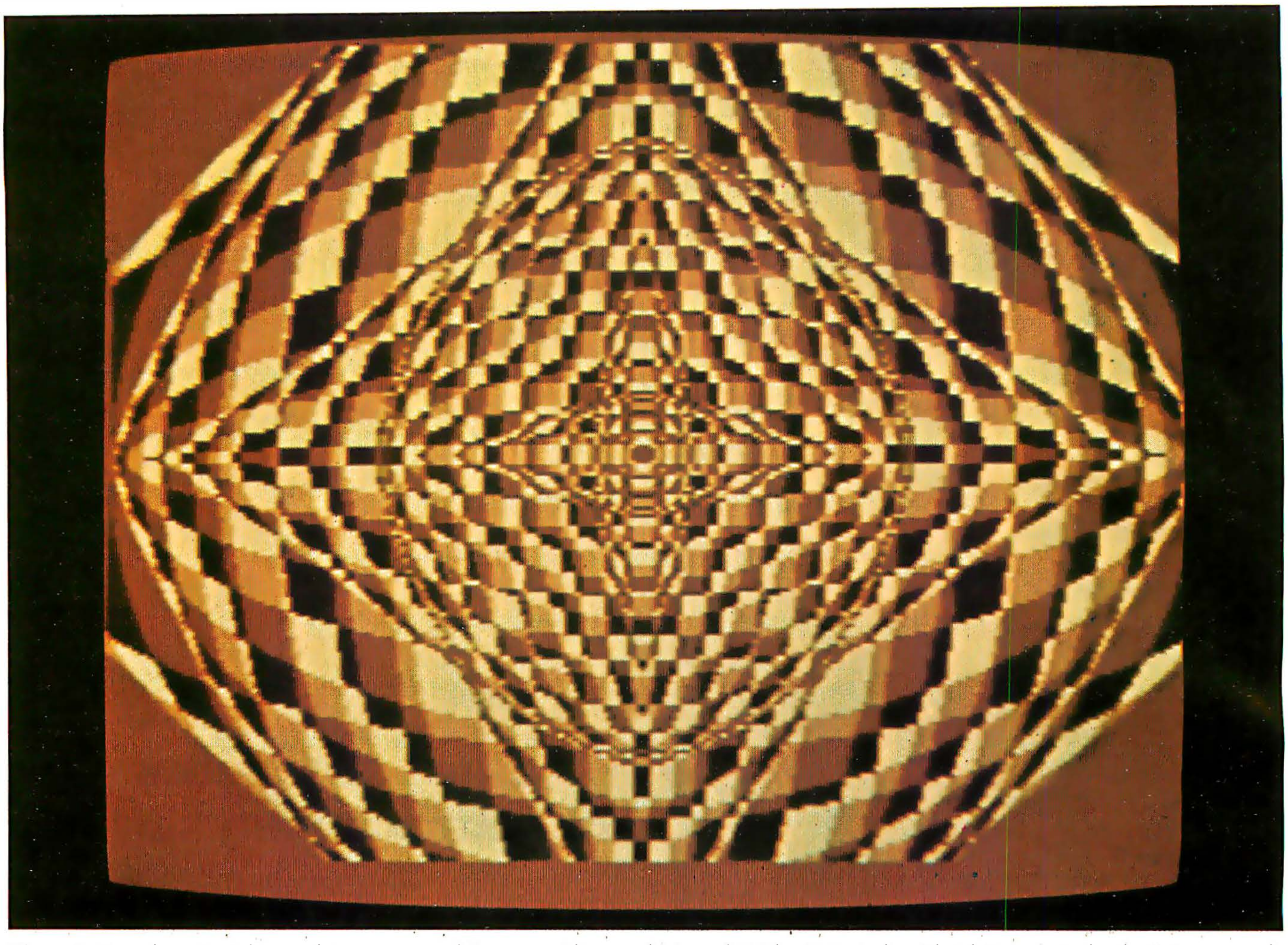


Photo 3: *Sample output from a later version of Zgrass, with a resolution of 320 by 204 pixels with 2 bits per pixel. Photo 3 was made by Frank Dietrich.*

pect of developing a language for fun, one that had user-orientation as the benchmark rather than how many FOR-NEXT loops you could execute per unit time was too good to pass up. I was contracted to produce Zgrass, and in a year, Nola Donato, Jay Fenton (a legendary wizard of video games and pinball-machine operating systems), and I had generated 9000 lines of code. (Much of the work was done not in a lab but in a cabin in the woods of Wisconsin!) Examples of output from this system are seen in photos 2a, 2b, and 2c. Note that the resolution of this first Zgrass machine is 160 by 102 pixels (ie: picture elements), with 2 bits per pixel.

Some confusion arose about whether we were producing a hobbyist machine or a home computer for consumers, so the project was suspended. Even now nobody really knows what a "consumer computer" is supposed to be.

From consulting with less enlightened would-be consumer computer manufacturers, I have perceived that they follow the rather negative view of consumerism. (Few people reading this article would be considered only consumers — I assume that BYTE readers are mostly hobbyists or professionals.) Consumerism is based on great market penetration, and the big question is: "How do you get 90% market penetration like color TV?"

It is also based on consuming, that is, wearing out or getting sick of hardware and software so you go buy more and consume it. The user is expected to supply no creativity, just assume a passive, susceptible-to-entertainment pose — this reminds you of television watching, doesn't it? Well, anything requiring creative energy is akin to hobbyism.

Consumer computers do exist in the form of video games that you can get bored with and buy more — even the advertisements invariably cite the number of new games to be available each month. I don't know how to write a programming language that wears out, though. User-extensibility is planned "nonobsolescence." Zgrass is not a consumer language by current standards.

The project is on active status again, but this time with a hobbyist/professional orientation. We believe there are many people who want a recordable image-producing system for around $3000. The current configuration includes:

- Z80 processor with 16 K bytes of EPROM and 48 K bytes of programmable memory
- custom graphics integrated circuits and floating-point hardware
- dual UARTs (universal asynchronous receiver/transmitters) for connection to larger computers, printers, etc
- RBG (red, blue, green) monitor for best color resolution

- alphanumeric terminal (which the user provides)
- provision for floppy disks, tablet, other I/O (input/output) devices

Eight Zgrass units in this configuration have been alive and well and tied into the Bell-Laboratory-developed UNIX operating system since January 1980. Although I have only discussed software design, I must mention that the hardware to test the concepts really exists! See photo 3 and note that the resolution is now 320 by 204 pixels, with 2 bits used per pixel.

Details of Zgrass Control Structures

Programs in Zgrass are called *macros*. Macros are stored as ASCII (American Standard Code for Information Interchange) character strings and normally contain executable Zgrass commands. The fundamental unit of execution in Zgrass is a command, which is either an assignment statement or a function call.

Zgrass does not require declaration of variable types (with the exception of array dimensioning). The software automatically does all conversions that make sense based on the context. Any argument can be a function call whose returned value is converted to whatever is needed, if at all possible. Literals, indirect references, variables, built-in commands, user-defined commands, and user-defined macros are all handled by the same parser, so the syntax is very predictable. The fact that there are no restrictions on name length helps to produce easily read code.

User-Level Extensibility

Extensibility in Zgrass is achieved in two major ways. First, you can write macros which return values, produce graphics, or ask questions; or, through string-manipulation primitives written by Barb Wilson, you can generate other macros. Macros use arguments in exactly the same way as system commands, and are even named and called like system commands.

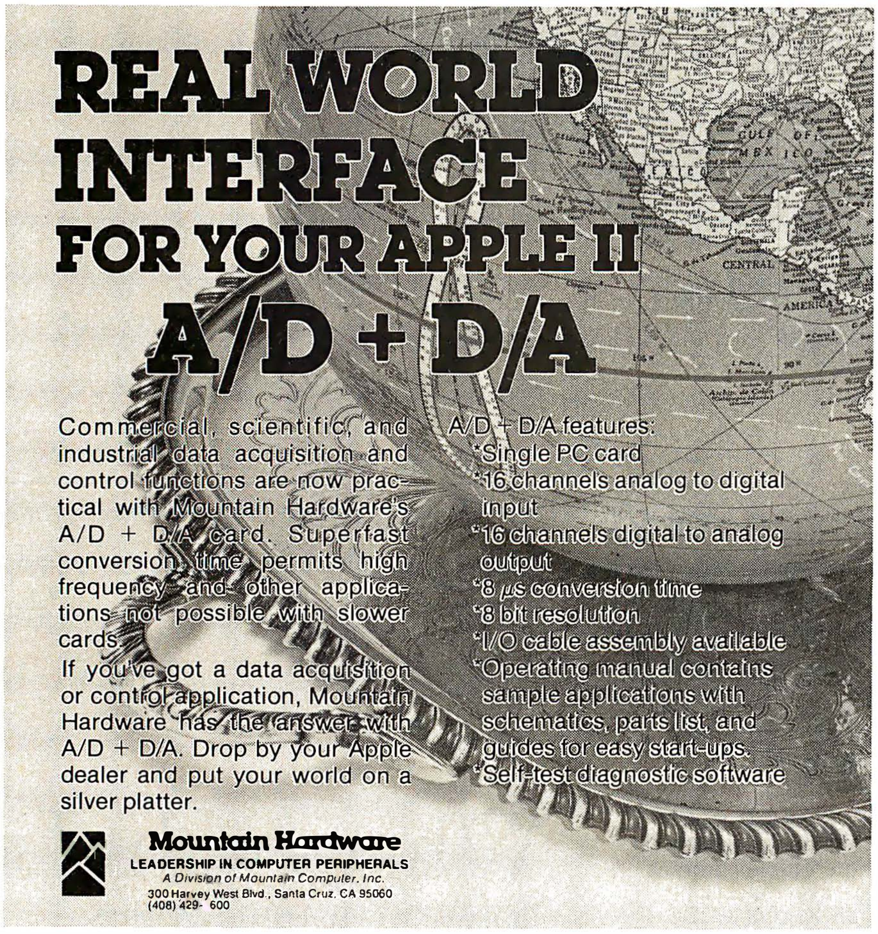
To reiterate, macros are simply strings of ASCII characters. When a macro is called, an MIB (Macro Invocation Block) is automatically built. It gives information on the invoking function call, the passed-argument list, and pointers to local variables, and provides room for the returned value. MIBs form a stack which implements the subroutining and block structuring of the language. When the macro returns, the MIB is deleted along with the local variables and unused literal arguments, if any, and control is passed back to the caller.

If arguments are to be passed to a macro, they are read by the normal input command, and print statements are suppressed as long as there are arguments left. If no arguments are present or an insufficient number are passed, the print statements function normally and the macro asks for input from the terminal. This allows macros to be used whether or not you know the arguments wanted, with no extra code by the author of the macro.

Macros can also be executed in parallel as background jobs. When called and suffixed by a ".B", the Macro Invocation Block is added to a background linked list. After that, the macro will run forever (it restarts at the beginning when it tries to return) until Control-C or the stop command selectively kills it. Photo 2c shows two sorting algorithms being compared for execution speed in real time, a tricky task in most languages, easy in Zgrass.

The background parallelism is achieved by interleaving execution of the macro statements. The MIB contains all relevant context for execution, including a pointer to the next command to execute, so switching MIBs after each line has been completed is simple and gives the functional parallelism. If there are five background macros, each one gets a line executed, in turn, round-robin fashion. This construct is simple and straightforward with no bizarre side-effects except that unusually time-consuming commands will make the parallelism temporally step somewhat. Background interleaving is easily understood and used even by the most naive users.

Meanwhile, the keyboard is still active. When the user types a command line, it is executed at a higher priority than the background macros. If the user initiates a macro at keyboard level, it will finish before the background macros continue. In any event, the keyboard overrides the background, again in an obvious, predictable way.



REAL WORLD INTERFACE FOR YOUR APPLE II

A/D + D/A

Commercial, scientific, and industrial data acquisition and control functions are now practical with Mountain Hardware's A/D + D/A card. Superfast conversion time permits high frequency and other applications not possible with slower cards.

If you've got a data acquisition or control application, Mountain Hardware has the answer with A/D + D/A. Drop by your Apple dealer and put your world on a silver platter.

A/D + D/A features:
*Single PC card
*16 channels analog to digital input
*16 channels digital to analog output
*8 μs conversion time
*8 bit resolution
*I/O cable assembly available
*Operating manual contains sample applications with schematics, parts list, and guides for easy start-ups.
*Self-test diagnostic software

Mountain Hardware
LEADERSHIP IN COMPUTER PERIPHERALS
A Division of Mountain Computer, Inc.
300 Harvey West Blvd., Santa Cruz, CA 95060
(408) 429-8600

Circle 61 on inquiry card.

The user may also specify programs to run as the result of a clock interrupt. When a macro call is suffixed by a ".F", the Macro Invocation Block is chained into a list that is polled every 1/60 second. The user sets the frequency of execution from 1 to 32,768 sixtieths of a second. These foreground macros execute on a higher priority level than the keyboard and background macros so they will start up just about on time (again, delayed only by a time-consuming graphics command). Foreground macros allow a keyboard command to be slipped in during context switching.

Zgrass, then, has three effective levels of priority with parallelism at two of the three levels. Since the Macro Invocation Block maintains all context information, even recursive programming is possible at any level.

One of the severe problems in interpretive, extensible languages like Zgrass is the overhead of parsing and looking up names in name tables. For this reason, Zgrass has a compiler which eliminates the overhead and dramatically increases speed. All the automatic conversions, priority, and parallelism continue to work. Compiling does eliminate some of the interactive debugging features, so you usually debug on the noncompiled version first.

Zgrass System Extensibility

Zgrass also allows extensibility at the system-command level. A system such as this should allow an experienced programmer to write new commands in assembler and interface them to the system easily, certainly without changing the EPROMs (erasable, programmable read-only memories). A transfer vector in low memory and a series of Z80 RST (special restart subroutine-call) instructions allow communication with about one hundred system routines which do parsing, type conversion, graphics primitives, and so on.

Documentation explains what these routines do, and anyone with a cross assembler (or patience for hand assembly) can write new commands of which the system has no prior knowledge. Such extensibility allows virtually infinite variety of specialty graphics commands, device drivers, and so forth to be written and distributed to others on audio tape, disk, or over telephone lines. Terry Disz wrote a debugging program used as a disk-resident command for setting break-points, dumping memory and registers and so on. This capability is not for everyone, but it's there.

The maximum size of one of these user-written nonresident commands is 4 K bytes. Since the typical Zgrass machine has 30 K bytes of programmable memory, the amount of potential custom code is immense. All housekeeping for storage allocation and deletion, maintenance of temporary scratch-pad areas and general cleanup is done by system routines. You only concentrate on the details, obeying a few rules for writing position-independent code.

One further type of extensibility is easy to get. Zgrass has an extra UART which talks to other computers quite nicely. Larger computers can send graphics and character data to your Zgrass machine. Zgrass units can even talk to one another at up to 19.2 k bps!

Error Handling, Debugging and Automated Instruction

Zgrass was designed from the beginning to be a language for writing CAI (computer-aided instruction) programs. In particular, it was designed to be self-teaching to a fairly high degree. When Zgrass is used as a CAI system, the result of providing parallelism, string manipulation, and good error handling is that the student always has the power of the whole language to explore while the author of the CAI programs is also in control.

Since macros are character strings, they can be built and executed. You can take student input, make it into a program (before the student even knows how to edit), let parameters be changed, show the results, and verify certain classes of results both during execution and after. The approaches we have taken to Zgrass CAI are beyond the scope of this article, so I will just mention the system features which make CAI possible.

Error-handling routines normally generate error-message numbers on the terminal. There are about sixty of them and they are quite specific. During regular programming, they are used in conjunction with single stepping, variable printing and other debugging techniques to identify

SOFTWARE ENGINEERS

WE'RE KNOWN BY THE COMPANY WE KEEP.

Wescom, Inc. has joined with Rockwell International. This move will strengthen our already envious industry standing. We are seeking bright professionals interested in a career at the leading edge of the telecommunications industry.

These positions require a BSEE or BSCS with a minimum of 1-3 years software design experience. Experience in control processing, real time, and/or 8080 ASSEMBLY language is considered a plus.

SYSTEMS SOFTWARE
Provide the design for new call processing features and systems, modular functional descriptions and interfaces, software architectural and data base design.

OPERATING SYSTEMS SOFTWARE
Design and develop software that maintains a system that's operated by stored program control. Involvement also includes design and implementation of administrative diagnostic and fault recovery programs.

SOFTWARE METHODOLOGY
Explore and intitate software techniques for future switching systems development. Areas of interest include structured design techniques, high-level languages, documentation schemes and software testing mechanisms.

We offer a competitive salary, generous benefits and the assurance that you'll be in good company with an innovative leader in the design, development and manufacture of sophisticated telephone equipment and systems.

We are an equal opportunity employer. Minorities, women and handicapped are encouraged to apply. For more information, send a letter or resume, or call:

Dennis Kebrdle
Wescom, Inc.
Rockwell International
8245 S. Lemont Rd.
Downers Grove, IL 60515
(312) 985-9000, ext. 2349

Rockwell International

...where science gets down to business

Circle 62 on inquiry card.

problems. When teaching, however, the CAI program must trap errors. These fall into three types: syntax, nontermination, and logic.

To trap syntax errors, you should use the ONERROR command which transfers the control to a diagnostic section of the program that you, as a CAI author, will have provided. There you can get the error number, the erroneous argument, and even the entire ASCII text of the line in error with the GETERROR command. You can then explain the problem to the user in whatever level of detail you wish.

Indefinite loops are caught with the LOOPMAX command which sets a limit to the number of control transfers (ie: skips and GOTOs). Once the limit is exceeded, an error is generated and trapped as explained earlier. So, you can catch nonterminating programs or be very meticulous and require efficiency from advanced students by lowering the LOOPMAX appropriately.

Logic errors are trickier and the general case is impossible. However, if you choose suitable problems to solve, you can do some very nice verification. For graphic tasks, the CMPARA command can check a student's building of an image against a prototype. The CAI author can tell if the student's image is a proper subset of the prototype and let it continue. Once a stray pixel is written, CMPARA returns a value of −2 which means the image is "mixed up," and you inform the student immediately. This approach clearly falls short of genuine artificial intelligence, but it is nevertheless quite useful. Several classes at the University of Illinois at Chicago Circle have been taught with great success using a GRASS-coded prototype (called GAIN, by Tom Towle).

Conclusions

Zgrass is a language/system designed to provide easy access to computer graphics and, in general, to computing. It has sophisticated real-time structures and control capability, and it's friendly, extensible, and fun. The language is more efficient than BASIC, more user-oriented than FORTRAN or Pascal, and it has the kind of language-control structures that will help you create your mind's fantastic visualizations on your video screen with more ease than ever before. ■

Glossary

Color: *The 256 colors available in Zgrass form an abbreviated spectrum. You can get four colors on the screen at any one time. The default colors are white, red, green, and blue. They are also known as color 0, color 1, color 2, and color 3. The values are stored in $L0, $L1, $L2, and $L3 unless you modify $HB to use the right-side colors $R0, $R1, $R2, and $R3.*

Color Map: *The color map is the way Zgrass translates color 0 thru color 3 to the 256 available colors. The hardware looks at the values of $L0 thru $L3 before it writes a pixel to the screen. If it is writing a 0, it uses the color stored in $L0; if it is writing a 1, it uses the color stored in $L1, and so on. To change the color map so 1 refers to yellow instead of red, set $L1 to 127. There are actually two color maps, the $Ls and the $Rs. You get to the $Rs by setting $HB.*

Color Option: *The possible values for color option are 0 thru 15. You may need to study your truth tables for inclusive-OR and exclusive-OR (XOR) logical operations to really understand what's going on. The following is functionally true, however:*

Color Option	Meaning
0	*replace with color 0 (white)*
1	*replace with color 1 (red)*
2	*replace with color 2 (green)*
3	*replace with color 3 (blue)*
4	*don't draw (actually XOR with 00)*
5	*XOR screen with color 1 (01 binary)*
6	*XOR screen with color 2 (10 binary)*
7	*XOR screen with color 3 (11 binary)*
8	*change red to white, blue to green (clear bit 0)*
9	*change green to white, blue to red (clear bit 1)*
10	*OR with 01 (if red or white, stay red; if blue or green, stay blue)*
11	*OR with 10 (if green or white, stay green; if red or blue, stay blue)*
12	*replace with red only if white were there*
13	*replace with green only if white or red were there*
14	*increment the color there by 1 (white to red, red to green, green to blue, and blue to white)*
15	*decrement the color there by 1 (white to blue, red to white, green to red, and blue to green)*

Macro: *A string that is supposed to contain legal Zgrass commands. Most programming languages call such things "programs" or "subroutines," but we call them macros. Macros are effectively user-defined commands. Macros can behave just like commands in the sense that you can pass arguments to macros with the INPUT command and return values with the RETURN command. You define a macro just like you define a string, with an assignment to a name or by using EDIT.*

String: *A collection of characters (ie: numbers, letters, punctuation) delimited (ie: enclosed) by single or double quotes or balanced (ie: enclosed) by brackets or braces. If you have to use a string delimiter in a string, make sure that it is delimited by a different string delimiter or things will get very confused. Most likely it will consider the rest of your macro as part of the string. Examples:*

```
"THIS IS A LONGER STRING"
"PRINT A*B*C
SKIP  -1  ;.THIS  STRING
COULD BE A MACRO TOO"
[THIS IS HOW TO PUT A
QUOTE IN A STRING: " ' "]
[1234]
[ ]
```

Circle 65 on Inquiry card. ⟶