Language Control Structures for Easy Electronic Visualization

Tom DeFanti
University of Illinois at Chicago Circle
Box 4348 Chicago IL 60680

Perhaps the best way to start this article is to explain the title. Control structures are the program flow and 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, RETURN, DATA, INPUT and READ commands, an impoverished set, to be sure. Highly structured languages like PASCAL are rigidly limited to the control structure of subroutines. Lowly structured approaches like assembler language coding 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 assembler is more due to the efficient building of customized tables and linked lists than adding, subtracting, multiplying and dividing numbers.

Assembler coding is by no means easy, though, and the title does contain that word. In fact, it's the most important word in the title because "easy," in reality, means "access." In this case, it's your access to complex visuals in short order.

Why does this article concentrate on visuals? Because producing and manipulating images, especially animated ones, is a truly multi-dimensional task which reflects our real-world interactions much more than maintaining an accurate laundry list or printing payroll checks. Producing visuals demands a lot from software and making the access easy requires paying attention to providing rich control structures in a language.

"Electronic Visualization" is an intentionally broad term meant to conjure up thoughts of computer graphics, animation, image processing, video synthesis and even advanced word processing. Anyone sucessfully producing images for communication is unlikely to reject a techinque for reasons of algorithmic purity (as a computer scientist might feel forced to). One uses the tools at hand, and electronic visualization is the act and end product of using these tools. It can be both simultaneously because we are seeing the vast increase of real-time imaging systems, even in micro-based configurations. And controlling these real-time systems is like playing a musical instrument or driving a race car.

Just to tie the whole concept together, 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. One without the other is unacceptable to the thinking person. So why are programming languages currently available so impoverished on the control structure side?

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 newly emerging real-time digital techniques and increasingly flexible control structures. As a matter of fact, just about all the TV you see these days is

digitally processed for synchronization purposes.

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

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 this article.

Getting Computers and TV Technology Together

Let's look at the history of control structures for computer graphics and for television. Most computer graphics usage, with the obvious and exciting exception of TV 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 focused. No manufacturer of equipment for computer graphics (excepting the video game people, again) now depends on animation for solvency. Plotting is slow and often merely the side output of a large FORTRAN finite element analysis program. Visual esthetics 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 it were easy, they wouldn't get paid so much.

We are just recently to the point of electronically generating and manipulating images in real-time under program control. How do we design languages to deal with real time? But first, why do we want language, particularly alpha-numeric string-oriented language? 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, nagging question, some conclusions became clear. First, purist approaches to electronic visualization are hopeless. You need a hybrid combination of language, several input devices, picture-oriented commands, custom hardware and a smattering of idiosyncrasies. The most successful approaches to date are basically highly developed, beautifully evolved kludges. We know what purism in FORTRAN and BASIC does to image production. Purism in TV technique eliminates computer graphics as we know it. How about using graphic symbols to save the day?

Using symbols in a menu and some sort of manual picking mechanism is an approach taken by many FORTRAN graphic systems. This limits you to the number of symbols predefined in the menu and there is no user-level extensibility in that you cannot create new symbols to use 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 combination of non-alphanumeric 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 in Japan is a phonetic alphanumeric transcription of their language. They do not program in their extremely beautiful and rich symbol set. Eliminating alpha-numeric languages is not such a hot idea,

except in wholly 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 have to be able to develop sequences independently and merge them in ways that do not necessitate rewriting the programs. SMALLTALK and certain other languages have this capability, as do television technology and everyday life. How to make 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 like 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. We'll get further into this later.

Fifth, the system must be software fault-tolerant. Fault-tolerant hardware has been an active area of research of great importance to real-time control systems, yet language purists still think people should solve problems in structured, orthodox, algorithmic ways. A 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 non-structured, non-procedural, seat-of-the-pants programming is often the only salvation when the final goal is esthetically defined, and maybe not at all clear. You might call it "fuzzy programming." 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 Z-80 assembler by myself, Nola Donato and Jay Fenton. It embodies, not surprisingly, all the control structures put forth so far in this article. It has been in development for ten years.

Zgrass started out as the Graphics Symbiosis System (GRASS), a language designed to bring the immense complexity of a PDP-11/45--Vector General 3DR Display System within the grasp of artists and educators at The 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 the October 1977 issue of Byte.

GRASS depends on $120,000.00 of equipment to run, kind of expensive for a single-user system. But it is one of the first highly-developed non-FORTRAN interactive graphics systems for use by artists.

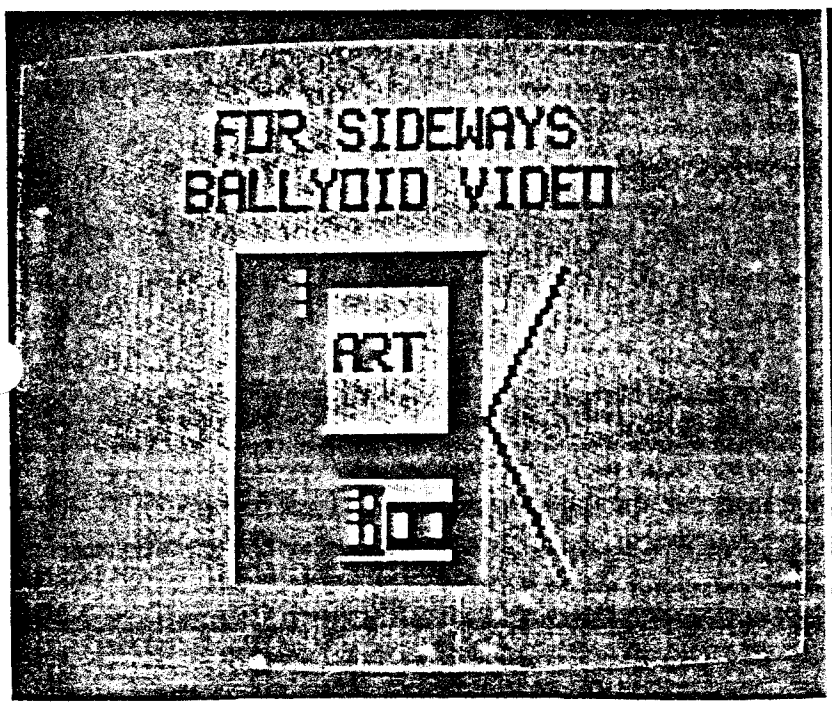
In 1973, Dan Sandin, inventor of the Image Processor, brought color TV into our computer graphics at the University of Illinois at Chicago Circle. Together, Dan and I developed most of the ideas about control structures presented here.

Generating a complete programming language with parsers,

compilers, and graphics takes a lot of human effort. Easily ten person-years of programming were devoted to GRASS aided by generous support from the National Science Foundation, National Endowment of the Arts, and others.

GRASS is totally oriented toward real-time generation and control of images for the simple reason that TV cannot easily be slowed down for time-lapse and time-exposure as film can. The control structures for GRASS were developed ad hoc and became increasingly idiosyncratic. Nola Donato, a Master's 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 told that Jeff Frederiksen at Dave Nutting Associates was developing a deluxe home computer for Bally Corporation using the chipset they had developed for the Bally Arcade. The prospect 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 contracted to produce Zgrass, and in a year, Nola Donato, Jay Fenton (a legendary wizard of videogames and pinball machine operating systems), and I had generated 9,000 lines of code, much of it in a cabin in the woods of Wisconsin. Examples of output from this system are seen in photos 1-5. Note the resolution of this first Zgrass machine is 160x102x2 bits/pixel.



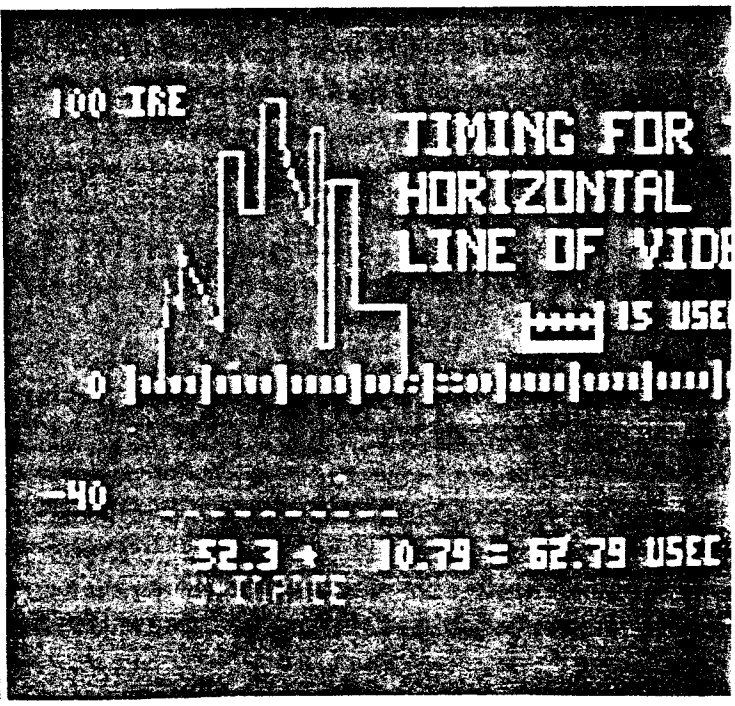


Photo 1 and 2: Zgrass for producing images. Graphics by Copper Giloth.

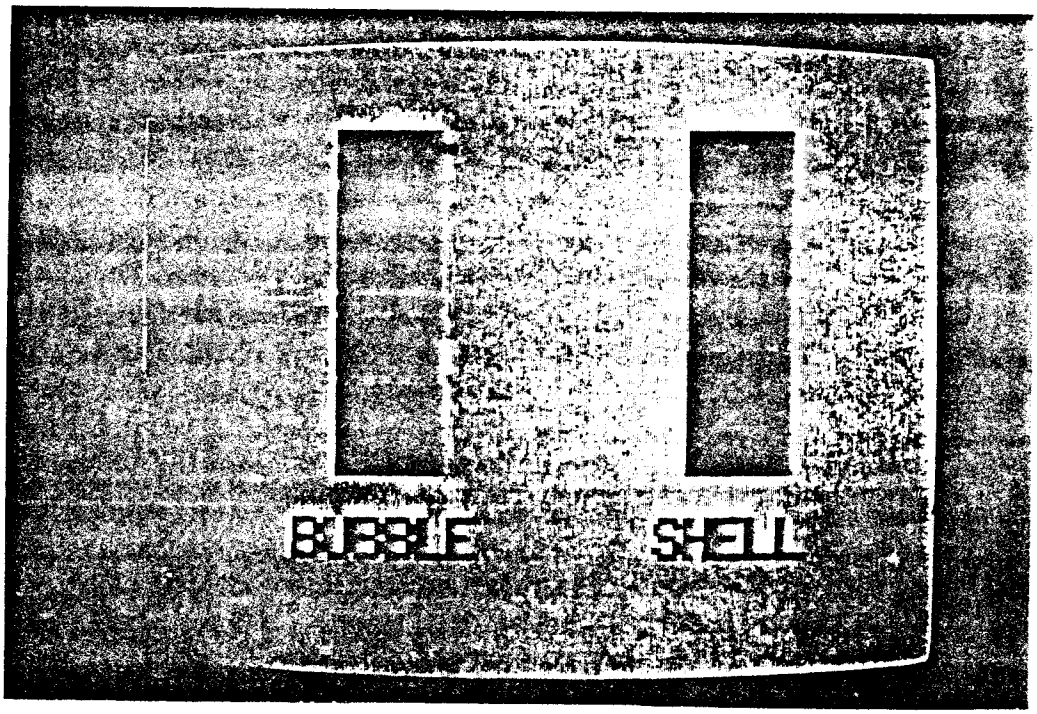
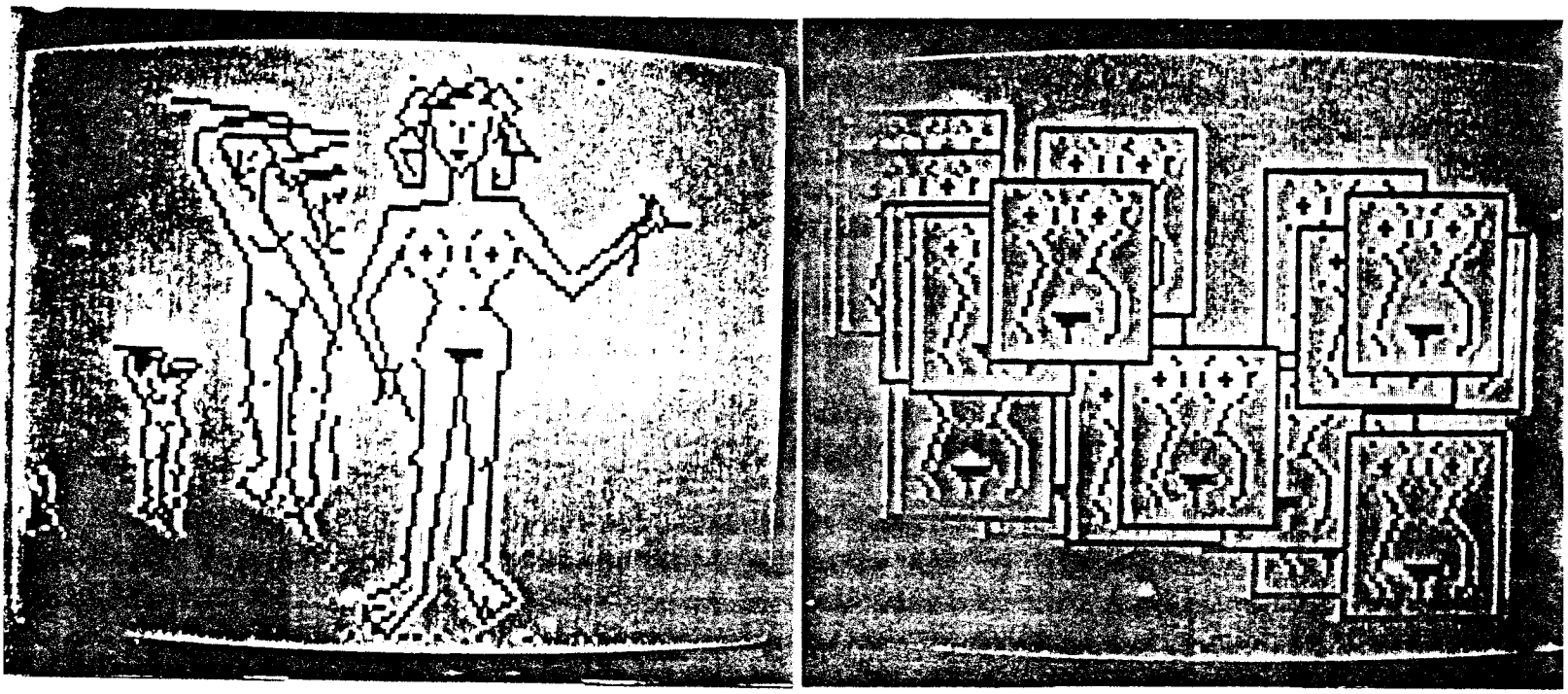


Photo 3: Using parallelism to show shell sorts beat bubble sorts. Program and graphics by Nola Donato.



Photos 4 and 5: Example of a hand-drawn image using a joystick and the duplication of one element by the "snap"/"display" command sequence. Graphics and programming by Copper Giloth.

Some confusion arose as to whether we were producing a hobbyist machine or a home computer for consumers, so the project was

suspended. To this date, nobody really knows what a consumer machine is supposed to be.

From consulting with less enlightened would-be consumer computer corporations, I have perceived a rather negative view of comsumerism (few people reading this article would be considered consumers--you 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? Consumerism is 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, succeptive-to-entertainment pose. Sounds like TV, doesn't it? Anything requiring creative energy is hobbyism.

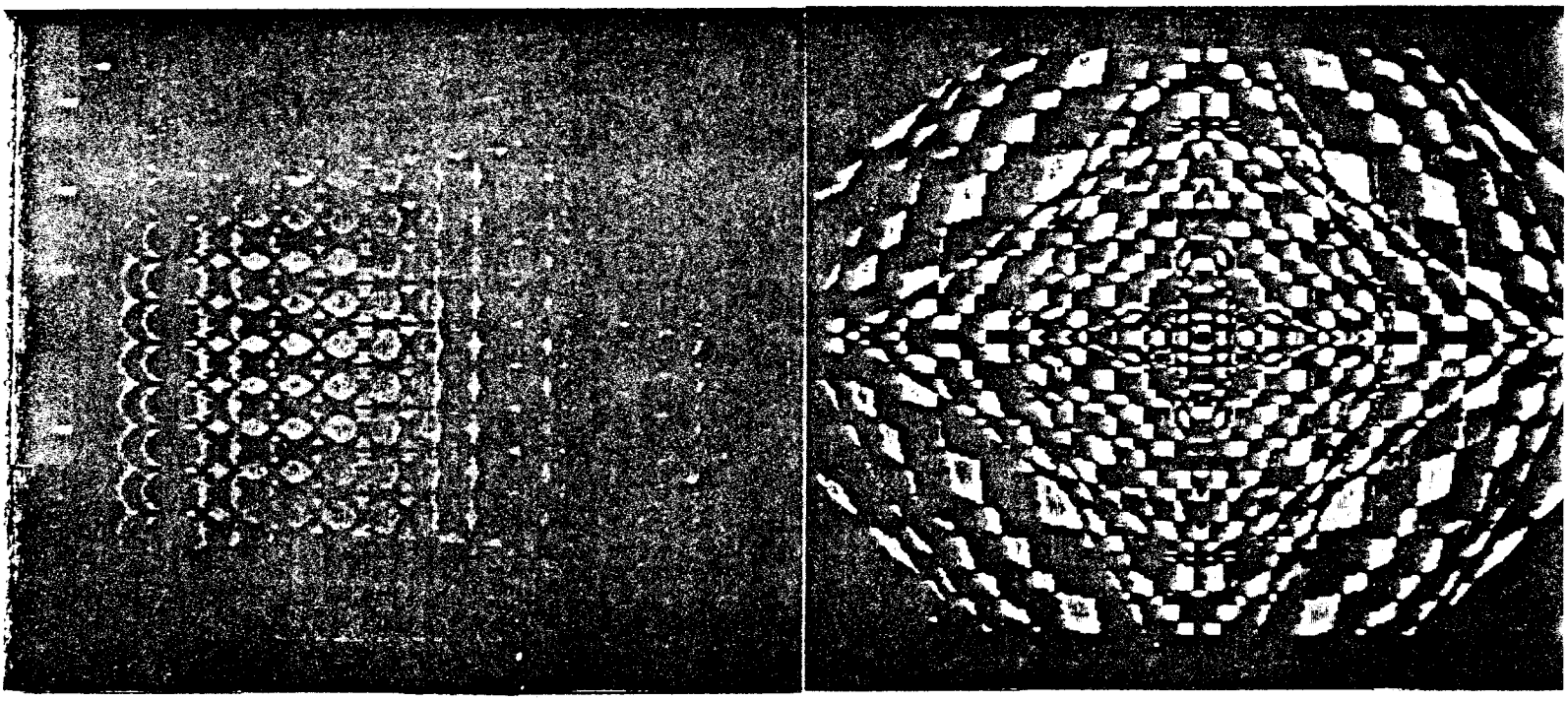
Consumer computers do exist in the form of video games that you can get bored with and buy more--even the ads invariably cite the number of new games to be available each month.

I don't know how to write a programming language that wears out, though. User-extensibility is planned un-obsolescence. 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 is a goodly number of people who want a recordable image producing system for around $3,000.00. It's current configuration looks like:

1. Z-80 processor with 16K EPROM, 48K RAM
2. Custom graphics chip and floating point hardware
3. Dual UART for connection to larger computers, printers, etc.
4. RBG monitor for best color resolution
5. Alphanumeric terminal (which you provide)
6. Provision for floppy disks, tablet, other IO devices

Eight Zgrass units in this configuration have been alive and well and tied into UNIX (TM-Bell Labs) since January 1980. Although this is an article on software design, the hardware to test the concepts really exists! See photos 6-8 and note the resolution is now 320x204x2 bits/pixel.



Photos 6 and 7: Two images by Frank Dietrich from the 320x204x2 bits/pixel Zgrass units.

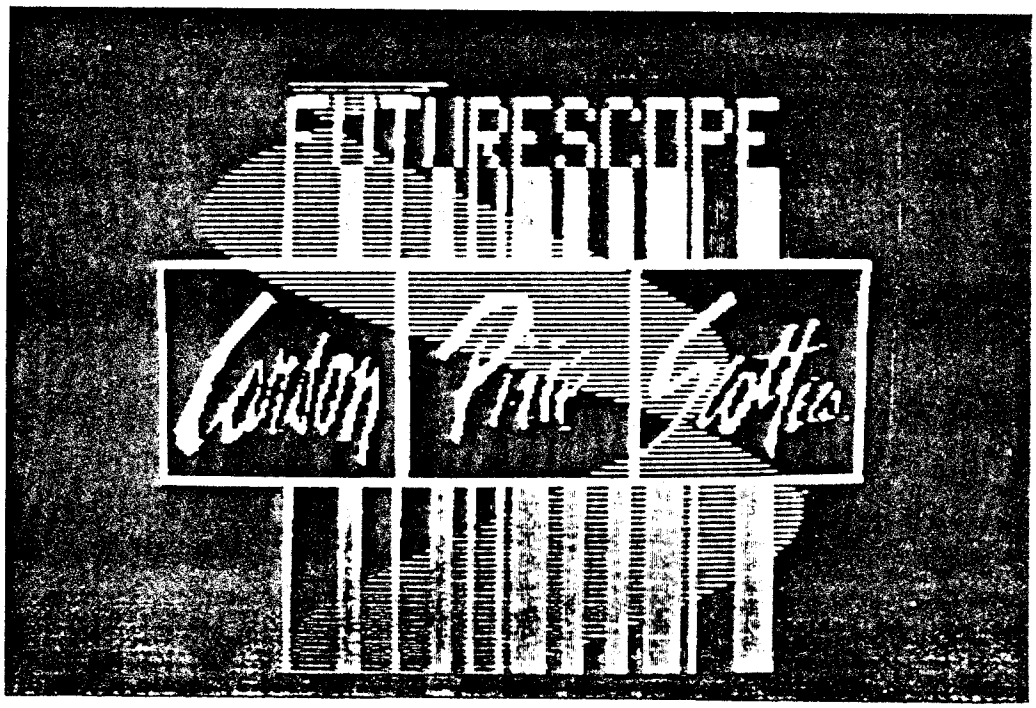


Photo 8: Image from a promotional campaign for Carson Pirie Scott, & Co., a Chicago retailer. Image and programming by Copper Giloth.

Zgrass Control Structures in Detail

Programs in Zgrass are called "macros." Macros are stored as ASCII strings and are in no way different from strings of characters except that they 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.

The built-in commands are given at the end of this article.

Zgrass does not have declare statements for variable types (with the exception of array dimensioning). The software automatically does all conversions that make sense based on the context. Any argument to anything 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. Add the fact that there are no restrictions on name length and you get rather readable code in general.

User-level Extensibility

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

Macros, again, are simply strings of ASCII characters. When a macro is called, a Macro Invocation Block (MIB) is automatically built. It gives information on the caller, the passed argument list, pointers to local variables, and provides room for the returned value. Macro Invocation Blocks form a stack which implements the subroutining and block structuring of the language. When the macro returns, the Macro Invocation Block 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 starts asking 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 2 shows two types of sorts being compared for execution speed in real-time, a tricky task in most languages, easy in Zgrass.

The background parallelism is achieved by interleaving the macro statements. The Macro Invocation Block has all relevant context for execution including a pointer to the next command to execute, so switching MIB's after each line is complete is simple and gives the funtional 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, and anytime the user types a command line, it is slipped in and executed at a higher priority than the background macros. If the user ini-

tiates a macro at keyboard level, it will finish before the background macros continue. In any event, the keyboard overides the background, again in an obvious, predictable way.

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 at one to 64K sixtieths of a second. These foreground macros execute on a higher priority level than the keyboard and background macros so they will startup pretty much on time (again, delayed only by a time-consuming graphics command). Foreground macros allow a keyboard command to be slipped in during context switching between themselves.

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 names 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 non-compiled 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 EPROM's. A transfer vector in low memory and a series of Z-80 RST instructions allow communication with about one hundred system routines which do parsing, type conversion, primitive graphics, 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 whatnot to be written and distributed to others on audio tape, disk, or over phone lines. Terry Disz wrote a debugger (itself a disk-resident command) for setting breakpoints, 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 non-resident commands is 4K bytes. Since the typical Zgrass machine has 30K of user RAM, the amount of potential custom code is immense. All housekeeping for storage allocation and deletion, maintenance of temporary scratch areas and general cleanup is done for you 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 at. 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 19.2K bits per second!

Error Handling, Debugging and Automated Instruction

Zgrass was designed from the beginning to be a language for writing computer-assisted instruction (CAI) programs. In particular, it was designed to teach itself to a fairly high degree.

In Zgrass as CAI machine, the student always has the power of the whole language to explore yet the author of the CAI programs is also in control, the result of providing parallelism, string manipulation and good error handling.

Since macros are 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 an article in itself, so let's just mention the system features which make CAI possible.

Errors 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 problems.

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

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

Infinite loops are caught with the "loopmax" command which sets a limit to the number of control transfers (skips and gotos). Once the limit is exceeded, an error is generated and trapped as above. So, you can catch non-terminating programs or be real picky and require efficiency from advanced students by lowering the loopmax appropriately.

Logic errors are trickier and, in the general case, impossible to decipher. 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 using a GRASS-coded prototype (called GAIN, by Tom Towle), with great success.

Conclusions

Zgrass is a language/system designed to provide easy access to computer graphics, and computing in general. It has sophisticated real-time structures and control capability, is friendly, extensible and fun. It's better than BASIC, more user-oriented than FORTRAN or PASCAL and it's one good way to control what appears on your television set.

Glossary of Zgrass Commands

ANYARGS

SYNTAX: ANYARGS()

(esoteric)
Function which returns 0 if no arguments are left in argument list and returns 1 if there are arguments in the argument list.

EXAMPLE:

```
ADDEMUP:[SUM=0
IF ANYARGS()==1,INPUT A;SUM=SUM+A;SK 0
PRINT SUM]
```

ARCCOS

SYNTAX: ARCCOS(NUMBER)

Function which returns the inverse cosine of the number.

ARCSIN

SYNTAX: ARCSIN(NUMBER)

Function which returns the inverse sine of the number.

ARCTAN

SYNTAX: ARCTAN(NUMBER)

Function which returns the inverse tangent of the number.

ARRAY.INT

SYNTAX: ARRAY.INT NAME,NUMBER

Creates an integer array with elements called NAME(0),NAME(1), ..., NAME (NUMBER-1). If two NUMBERs are specified, a two-dimensional array is created. Similarly, three NUMBERs will create a three-dimensional array.

EXAMPLE:
ARRAY.INT NEWARAY 8,8 will create a 81 element array of 9 rows and 9 columns.

ARRAY NAME

SYNTAX: ARRAY NAME,NUMBER

Creates a floating point array of elements called NAME(0),NAME(1),..., NAME(NUMBER-1). If two NUMBERs are specified, a two-dimensional array is created. Similarly, three NUMBERs will create a three-dimensional array.

EXAMPLE:
ARRAY CHECKERBOARD 7,7 will create a 64-element array with 8 rows and 8 columns.

ARRAY.STRING

SYNTAX: ARRAY.STRING NAME,NUMBER

(You cannot PUTUNIX string arrays as yet)
Creates a string array with string elements NAME(0),NAME(1),...., NAME(NUMBER-1). Multidimensional arrays are also possible with ARRAY.STRING.

EXAMPLE: ARRAY.STRING SEMANTICS,10

BOX

SYNTAX: BOX XCENTER,YCENTER,XSIZE,YSIZE,COLOR OPTION

This draws a filled-in rectangle of XSIZE by YSIZE centered at XCENTER,YCENTER with the drawing mode specified by COLOR OPTION (values 0-15). See COLOR OPTION in the Words Glossary for the meaning of the 16 color option values.

EXAMPLE:
BOX 0,0,20,30,5
This will create a rectangle 20 pixels wide and 30 pixels high in the center of the TV screen with color option 5.

CIRCLE

SYNTAX: CIRCLE XCENTER,YCENTER,XSIZE,YSIZE,COLOR OPTION

Draws a filled-in circle of XSIZE by YSIZE centered at XCENTER, YCENTER with the drawing mode specified by COLOR OPTION (values 0-15). See COLOR OPTION in the Words Glossary for the meaning of these 16 color option values.

EXAMPLE:
CIRCLE 0,0,20,30,5
This will create a circle (actually an ellipse) 20 by 30 pixels in the center of the TV screen with color option 5.

CLEAR

SYNTAX: CLEAR

Clears the TV/Monitor screen but not the computer's memory. RESTART clears the computer's memory.

CLEAR.CRT

SYNTAX: CLEAR.CRT

Clears the CRT terminal screen but not the computer's memory. RESTART clears the computer's memory.

CMPARA

SYNTAX: CMPARA(A1,A2)

(esoteric)
Function which returns values depending on the comparison of two arrays (This is usually used to compare SNAPs).
The values returned are:

```
0 if all elements of A1<=A2
1 if all elements of A1==A2
-1 if all elements of A1>=A2
-2 otherwise
```

COMPILE

SYNTAX: COMPILE NAME

(esoteric)
Replaces the macro named NAME with the compiled macro of the same name. Compiled macros are larger but run faster. However, compiled macros cannot be stored on disk or tape. Several commands will not work in the compiler;these are: EDIT,CORE, and HELP.

NOTE: You should always COMPILE a copy of the macro you want sped up because COMPILE deletes the macro it compiles.

EXAMPLE:

If your macro is called FOO, do the following:

```
CFOO=FOO
COMPILE CFOO
CFOO
```

CONTROL

SYNTAX: CONTROL STRING

(esoteric)
Takes the first character of the STRING and does the same function as the corresponding control character on the keyboard. This allows you to put the same functions in a program that you can

obtain by typing control characters.

CHARACTER	CONTROL CHARACTERS MEANING
A	(unused at this time)
B	return to default colors
C	stop currently running macro(s)
D	enter/exit debug mode (single step)
E	suppresses echo on terminal
F	turn foreground jobs off/on
G	reset all control characters
H	
I	
J	
K	
L	
M	(unused at this time) (actually NEXTLINE)
N	turn on beep for GETUNIX or PUTUNIX
O	suppress output to screen
P	halt output to screen
Q	resume output to screen
R	used by editor
S	user accessible via $CS
T	user accessible via $CT
U	delete whole line being typed
V	user accessible via $CV
W	set printout to twenty-line window
X	list/unlist macro as it is executing
Y	turn on '!' to mark nextline
Z	interrupt executing macro to enter commands, etc. (resume execution by typing NEXTLINE)

CORE

SYNTAX: CORE

(esoteric)
Tells you how much memory you have in BYTES in how many fragments. The first number is the hex address which you should ignore. A byte will hold one character so if you have a macro that's 500 characters long (USEMAP will give it's length once its in memory), CORE has to show a fragment with at least 500 BYTES for you to GETUNIX it.

COSINE

SYNTAX: COSINE(NUMBER)

Function which returns the cosine of the number.

DELETE

SYNTAX: DELETE NAME

Deletes the NAME (variable, array, string) from memory and reclaims the memory for further use. Certain things cannot be deleted ($variables, variables A-Z, commands) so an appropriate error message accompanies illegal deletion requests.
NOTE: Never delete anything that is referenced in a compiled macro unless you have already deleted that compiled macro.

EXAMPLE:

```
DELETE RICK
```

The item named RICK is deleted from memory.

DISPLAY

SYNTAX: DISPLAY NAME,XCENTER,YCENTER,MODE

Takes SNAPped NAME and writes it at center indicated.
The MODE is one of four writing options:
0 means do nothing
1 means XOR the SNAPped NAME
2 means OR the SNAPped NAME
3 means plop (that is, write) the SNAPped NAME

A SNAPped NAME is actually an array specially created by the SNAP command and is essentially an exact copy of an area of screen memory. You can then use DISPLAY to do animations.

EXAMPLE:

There is an apple drawn at the center of the screen and it fits in a rectangle 10 x 12 pixels. The following code will SNAP it and move it on a joystick+:

```
MOVEIT=[SNAP MAPPLE,0,0,16,18
 .LEAVE EXTRA WHITE AROUND FOR ERASING
$X1=0
$Y1=0
 .SET JOYSTICK1 X AND Y TO ZERO
DISPLAY MAPPLE,$X1,$Y1,3;SKIP 0]
```

EDIT

SYNTAX: EDIT NAME

Edits the macro specified.

EDIT CONTROLS

DEL	move cursor left a character
TAB	move cursor down a line
LINE FEED	move cursor up a line
BACK SPACE	move cursor right a character
	insert before character cursor is under
HOME	delete character cursor is under
CLEAR	delete line cursor is on
SHIFT+LINE FEED	insert line after current line
CONTROL+C	get out of EDIT
ESC	delete last inserted character (insert mode only)

GETERROR

SYNTAX: GETERROR()

(esoteric) (assembled out)
Function which returns the error number that last occurred. Usually used in conjunction with ONERROR to figure out programmatically what error condition arose. Cannot be used outside of the macro in which the error occurred.

GETERROR.ARG

SYNTAX: GETERROR.ARG()

(esoteric) (assembled out)
Function which returns a number corresponding to the place where the error occurred on the line. 1 refers to the command name, for example and 4 would tell you that argument number 3 (the fourth word) was where the error was found. Used with ONERROR.

GETERROR.STR

SYNTAX: GETERROR.STR()

(esoteric) (assembled out)
Function which returns the command line in error as a string. It can be used in conjunction with GETERROR.ARG to pinpoint the part of the command in error and point it out friendly-like to the used of your macro.

GETUNIX

SYNTAX: GETUNIX NAME

Gets the named file from disk.
Control+N will cause terminal to beep every 256 characters read.

EXAMPLE:
GETUNIX MOO
Loads a program named "MOO" into core memory.

GOTO

SYNTAX: GOTO LABEL

Causes the line containing the LABEL to be executed next instead of the following line. LABELs start with numbers.

EXAMPLE:

```
ARCHERY=[A=40
1AGAIN CIRCLE 0,0,A,A,A/10+3
IF A>10,A=A-1;GOTO 1AGAIN]
```

HELP

SYNTAX: HELP

HELP lists system commands available.

LINE LN(NUMBER) LOG(NUMBER) LOOPMAX

IF

SYNTAX: IF CONDITIONAL,COMMAND

If the CONDITIONAL is satisfied the command following is executed. Otherwise, control is skipped to the next line.

A CONDITIONAL is an expresssion which evaluates to 0 (false) or 1 (true). Using relational operators ('==', '>', '<', etc.) expressions are true or false after being evaluated and the rest of the line (including ';'s) is executed if the condition is true. Anything that evaluates to 0 or 1 can be used as the first part of an IF statement.

EXAMPLE:

```
IF A==10,STOP SAM ;.stop SAM only if A is eqal to 10
IF 1,FIXUP ;.this will always happen
IF FLAG,B=C+D ;.this will happen if FLAG==1
IF SIN(BRADIANS)*1.25<=.7,DRAW
     this last example shows complex expressions are
     perfectly OK in an IF statement.
```

INPUT

SYNTAX: INPUT NAME

Gets a value from the user or the argument list passed to the macro and stores it as a number in the NAME.

EXAMPLE:

```
PROMPT "Gimme a number"
INPUT A
 Computer will put the inputted value into the variable "A".
```

INPUT.NAME

SYNTAX: INPUT.NAME NAME

Gets a string of characters from the user or the argument list passed to the macro and checks it for valid name syntax and then puts it in NAME as a string.

EXAMPLE: INPUT.NAME CHUCKIT

INPUT.STRING

SYNTAX: INPUT.STRING NAME

(esoteric)
Gets a string of characters and then puts it into NAME. This option is good for reading an entire line from the terminal, including commas. It must also be used to pass a string with commas or spaces as an argument, in which case it should be enclosed in quotes or other string delimiters.

EXAMPLE: INPUT.STRING READLINE

INT(NUMBER)

SYNTAX: INT(NUMBER)

Function which returns the integer part of a floating number.

EXAMPLE:
INT(5%8)
This will give 5,6, or 7 without the fractional part.

LINE

SYNTAX: LINE XCOORDINATE,YCOORDINATE,COLOR OPTION

Draws a line from the previous line endpoint to the endpoint specified by XCOORDINATE and YCOORDINATE in the COLOR OPTION indicated. The '4' in LINE X,Y,4 will move the endpoint without drawing anything and should be used to set the first line endpoint and should be used to set the first line endpoint.

EXAMPLE:
LINE 0,0,4
This command places the line at the center of the screen but does not draw (option 4).
LINE 100,100,5
Draws a line from the center at 45 degrees in color option 5.

LN

SYNTAX: LN(NUMBER)

Function which returns the natural log of the number.

LOG

SYNTAX: LOG(NUMBER)

Function which returns the logarithm base 10 of the number.

LOOPMAX

SYNTAX: LOOPMAX NUMBER

(esoteric)
This command allows you to catch infinite loops by setting a maximum for the number of skips and gotos that can occur before an error is caused.

EXAMPLE:
LOOPMAX 500
This allows the program to run thru its cycle 500 times before an error is caused.

ONERROR

SYNTAX: ONERROR LABEL

(esoteric)
Sets up a transfer to label when an error occurs.
You can turn off ONERROR by specifying no label (ONERROR by itself turns the normal error reporting back on). You normally put an ONERROR LABEL before a statement that is likely to cause an error. You can only have one ONERROR setup per macro at a time but you can change it in the macro at any time.

EXAMPLE:
ONERROR GOOF
The control will transfer to the area of the program labelled GOOF and will squelch normal error messages.

PIXEL

SYNTAX: PIXEL(XCOORDINATE,YCOORDINATE)

Function which returns the value of a pixel addressed by the two coordinates given. 0 means that color 0 is at that address on the screen, 1 means that color 1 is there, etc.

EXAMPLE:
PIXEL(20,30)
Say the computer returns a 5, that means color option 5 is at location x=20,y=30. Not too tough,huh?

POINT

SYNTAX: POINT XCOORDINATE,YCOORDINATE,COLOR OPTION

Draws a point at XCOORDINATE,YCOORDINATE in the color specified. A point is one pixel in size and is the same as a box with size 1x1 and a circle with size 1x1.

See COLOR OPTION in the Words Glossary.

EXAMPLE:

```
POINT 0,0,5
```

This draws a single pixel in the screen center in color option 5.

POWER

SYNTAX: POWER(NUMBER,NUMBER)

Function which returns the first number raised to the power of the second number.

PRINT

SYNTAX: PRINT STRING

Prints the value of the STRING on the CRT followed by a NEXTLINE. Several STRINGS can be used. If you separate them by commas, a space is printed between them. If you do not want the space, separate them with '&'s. Stuff in quotes can also be used (like PRINT "THE ANSWER IS:",A) PRINTS (and PROMPTS) are suppressed if there are arguments passed to the macro.

EXAMPLE:

```
PRINT 2+5
 Prints the sum, or 7.
A=100
PRINT "The sum of 90 + 10 is:",A
 The computer responds with:
  The sum of 90 + 10 is 100
```

PRINT.FORCE

SYNTAX: PRINT.FORCE STRING

Like PRINT but forces printing whether or not an argument list is passed to the macro.

PROMPT

SYNTAX: PROMPT STRING

Just like print but does not print the NEXTLINE at the end.
PROMPT has a .FORCE option.

EXAMPLE:

```
PROMPT "Gimme a number"
INPUT A
PRINT A + 3
```

This asks for a number from the user then adds three to it and then will print the sum.

PUTUNIX

SYNTAX: PUTUNIX NAME

Puts the named file onto disk.
Control+N causes a beep every 256 characters transferred.

RENAME

SYNTAX: RENAME NAME1,NAME2

Renames NAME1 to NAME2.

EXAMPLE:

```
RENAME FOO,BOO
```

This renames program named FOO, BOO. This command is helpful for getting a program into core from UNIX, renaming it, changing it and then saving the changed copy under the new name.....the original program is still on UNIX under the old name.

RESTART

SYNTAX: RESTART

Clears memory and restarts ZGRASS. The "BREAK" key is a software reset that does not clear memory.

RETURN

SYNTAX: RETURN VALUE

(esoteric)
Returns the value indicated and control to the calling macro. Useful for creating user defined function calls which return values.

EXAMPLE:

```
MAX=[INPUT a,b,c ;.NOTE LOCAL VARIABLES
IF a<b,IF b<c,RETURN c
IF a>b,IF a>c,RETURN a
RETURN b]
```

This will return the maximum of the three parameters passed and could be used in:
BIGGEST=MAX(OF,THESE,THREE)
HONEY=MAX(CRUNCH1,CRUNCH2,KISS)

SCROLL

SYNTAX: SCROLL XCENTER,YCENTER,XSIZE,YSIZE,NUMBER

Scrolls stuff in area indicated NUMBER of pixels up or down. Scrolling means moving the area up or down. A positive number means scroll up; a negative number means scroll down. This is the only way to scroll text on the TV monitor.

EXAMPLE:
SCROLL 0,0,40,40,10
This scrolls the area 40 pixels wide by 40 pixels high located at the center of the screen up 10 pixels.

SINE

SYNTAX: SINE(NUMBER)

Function which returns the sine of the number.

SKIP

SYNTAX: SKIP NUMBER

Skips the given number of lines (including the one you're on). It transfers control by counting the number of NEXTLINE's indicated.

NOTE: SKIP does not allow labels. Use GOTO if you want labels.

EXAMPLE:
SKIP 0 - hangs in place
SKIP 2 - skips the next two lines
SKIP -3 - goes back three lines
SKIP 1 - does nothing
SKIP 999 - is the same as RETURN
SKIP -999 - will get you back to the beginning of the macro.

SNAP

SYNTAX: SNAP NAME,XCENTER,YCENTER,XSIZE,YSIZE

Takes the pixels in the area indicated and saves them in an array called NAME. The DISPLAY command can then redraw them somewhere else.

EXAMPLE:

SNAP APPLE,0,0,40,40
This takes the information about the area 40 by 40 pixels at the center of the screen and saves it in an array called APPLE.

SQRT

SYNTAX: SQRT(NUMBER)

Function which returns the square roote of the number.

TEXT

SYNTAX: TEXT XCOORDINATE,YCOORDINATE,COLOR,STRING

This is like PRINT but puts it on the TV monitor. The first character printed is centered at XCOORDINATE,YCOORDINATE.

TEXT.SCALE

SYNTAX: TEXT.SCALE XCOORDINATE,YCOORDINATE,XSIZE,YSIZE,COLOR,STRING

This is like TEXT but the size is determined by XSIZE,YSIZE. XSIZE and YSIZE can be controlled indépendently.

EXAMPLE:
TEXT.SCALE 0,0,20,30,5,"HOWDY FOLKS!"
This prints characters 20 pixels wide by 30 pixels high beginning at the center of the screen in color option 5.

USEMAP

SYNTAX: USEMAP

Gives a list of names currently in use and the number of BYTES they take up.

WHATSIS

SYNTAX: WHATSIS NAME

(esoteric)
Returns a value for the type represented by the name.
The above command descriptions compiled by Joanne Culver.

Glossary of Terms Used

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 0-3 to the 256 available COLORS. The hardware looks at the values of $L0-$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 $L's and the $R's. You get to the $R's by setting $HB.

COLOR OPTION

the possible values for COLOR OPTION are 0-15. You may need to study your truth tables for XOR and OR 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, blue)
11	or with 10 (if green or white, stay green, if red or blue, 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

is 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 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

is a collection of characters (numbers, letters, punctuation) delimited (enclosed) by single or double quotes or balanced square or curly brackets. If you have to use a string delimiter in a STRING, make sure 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]
[]
```

SWITCH

is an option for a COMMAND or MACRO. The only SWITCHES defined for MACROs are .B and .F which cause the MACRO to be executed in the background and foreground respectively. Many COMMANDS (INPUT, ARRAY, etc.) have SWITCHES which are given in the Command Glossary as separate entries. SWITCHES are always preceeded by the NAME they are modifying and a '.' Examples:

```
INPUT.STR SAM
ARRAY.INT FOO,123
DEATHWEAPON.B
```

XOR

is a logical operation (also called 'exclusive-or') used to draw PIXELS on the screen. What gets drawn is a value from 0-3 and is computed by the XOR function of what was there and what you give it to write there. The reason for this complexity is that a couple of neat tricks are made possible by XOR. First, if you draw anything on the screen with XOR (COLOR modes 4-7) or DISPLAY a SNAPped picture element with mode 2, you can erase it by simply drawing or DISPLAYing it again the same way. In other words, two XOR's is the same as nothing. Second, by setting $L3=$L2 (and $R3=$R2 if you mess with $HB), you can make anything XOR written with COLOR 1 pass 'behind'

anything written with COLOR 2 (you have to try it to believe it). At any rate, the XOR table is as follows, assuming 0=white, 1=red, 2=green and 3=blue:

COLOR GIVEN	COLOR THERE			
XOR	white	red	green	blue
white	white	red	green	blue
red	red	white	blue	green
green	green	blue	white	red
blue	blue	green	red	white

besides being tricky, XOR is good for relieving boredom in Zgrass wallpaper art.

.B

means run the MACRO in the background over and over again until CONTROL+C or STOP is seen. Any MACRO or COMMAND issued from the keyboard or .F mode will take precedence. Example:

```
ANIMATE=[DISPLAY APPLE,$X1,$Y1,0]
ANIMATE.B
```

will move the APPLE (a SNAPped picture element) under control of the first joystick until further notice). (esoteric).

.F

is a way of telling a MACRO to execute every 1/60 second. Such macros should be short since they take precedence over regular and .B MACROS. Example:

```
TIMESUP=[timer=timer+1
IF timer==180,PRINT 'THREE SECS ARE UP';STOP TIMESUP]
```