VideoLogo: Synthetic Movies in a Learning Environment

by

Alexander Benenson

B.A., Cognitive Science
University of Rochester
Rochester, New York
1984

SUBMITTED TO THE MEDIA ARTS AND SCIENCES SECTION,
SCHOOL OF ARCHITECTURE AND PLANNING
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS OF THE DEGREE OF
MASTER OF SCIENCE
AT THE MASSACHUSETTS INSTITUTE OF TECHNOLOGY
June 1990

© Massachusetts Institute of Technology 1990
All Rights Reserved

Signature of the Author

....

Alexander Benenson
Media Arts and Sciences Section
May 11, 1990

Certified by

....

Andrew B. Lippman
Lecturer, Associate Director, Media Laboratory
Thesis Supervisor

Accepted by

....

Stephen A. Benton
Chairman
Departmental Committee on Graduate Students

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
JUL 06 1990
LIBRARIES
Rotch

VideoLogo:
Synthetic Movies in a Learning Environment

by
Alexander Benenson

Submitted to the Media Arts and Sciences Section,
School of Architecture and Planning
on May 11, 1990
in partial fulfillment of the requirements of the degree of Master of Science
at the Massachusetts Institute of Technology

Abstract

The cost of digital multimedia systems is decreasing, and the number of multimedia applications is increasing. Current multimedia application development environments tend to be restrictive or complex; there is a need for simple yet flexible development tools to make multimedia development more accessible to everyone.

This thesis focuses on multimedia application development in learning environments for elementary school children, specifically the Logo programming environment. VideoLogo, a Logo-based system using a set of video objects and manipulations based on current digital image coding and image processing approaches, is developed to allow the creation of flexible and dynamic multimedia applications.

Using the video primitives added to VideoLogo, children can easily build applications that create, access and manipulate graphics, still video, motion video, and animate video objects.

Thesis Supervisor: Andrew B. Lippman
Title: Lecturer, Associate Director, Media Laboratory

Work reported herein is supported by a contract from the Movies of the Future consortium, including AT&T, Columbia Pictures Entertainment Inc., Eastman Kodak Co., Paramount Pictures Inc., Viacom International Inc., and Warner Brothers Inc.

Acknowledgements

The author thanks the following people for their help and advice:

Andy Lippman, for helping me to focus on my area of interest by coming up with the original idea, and for letting me do it;

Alan Shaw, for the Logo interpreter code, and for supporting it above and beyond the call of duty;

Mario Bourgoin, for answering my questions, finding code for me to hack, and people for me to pester;

Idit Harel, for her advice, enthusiastic support, testing, and publicizing, and for helping to present VideoLogo to the American Educational Research Association;

Walter Bender and Glorianna Davenport, for helping me to refine the idea;

Alexandria, Roberta, Sequoia and Robert, for being delightfully enthusiastic guinea pigs,

and all inhabitants of the garden for keeping things interesting!

Contents

1 Introduction **7**
1.1 The Need for Accessible Video Manipulation Tools . . 8
1.2 Logo with Video Tools as a Learning Environment . . 8
1.3 Existing Logo Learning Environments 10
1.4 Instructionism vs. Constructionism 12
1.5 Synthetic Movies 13
1.5.1 Digital Image Representation and Compression . . . 14

2 Multimedia Applications **16**
2.1 Application Examples 16
2.1.1 Video Finger 16
2.1.2 Network Plus and Digital Interactive News 17
2.1.3 Palenque 18
2.1.4 Elastic Charles 19
2.2 Development Tools 20
2.2.1 Authoring Systems 20
2.2.2 High Level Languages 21
2.2.3 A Simple Multimedia Development Environment . . 22

3 The VideoLogo Environment **23**
3.1 Design of Primitives 23
3.1.1 The Turtle as Actor 24
3.1.2 Interaction with Video 25
3.1.3 Combining Graphics and Video 26
3.1.4 Shapes Design 27
3.1.5 Effects Design 28
3.2 System Environment 30
3.3 The Editor 31
3.4 Predefined Primitives 33
3.5 VideoLogo Workspace 34

3.6 VideoLogo Language 34
3.7 Turtle Graphics 35
3.8 VideoLogo Shapes 36
3.8.1 Creating Shapes 38
3.8.2 Shape Manipulation 39
3.8.3 Shape Housekeeping 40
3.8.4 Shape Animation 40
3.9 Video Stills 41
3.10 Motion Video 43
3.10.1 Video Playback Control 44
3.10.2 Masking & Keying 46
3.11 Video Effects 47
3.12 Video Capture/Recording 49

4 VideoLogo Software Description 51
4.1 Software Overview 51
4.2 Drawing Routines 52
4.3 Video Mode 53
4.4 Image Decompression 54
4.5 Multitasking 55
4.6 Video Playback Control 57
4.7 Waiting for Video Frames 58
4.8 Displaying Frames 59
4.9 Custom Microcode 61

5 Conclusions 62
5.1 Improved Functionality 62
5.2 Future Directions 64
5.3 Results 67

A VideoLogo Instruction Manual 68
B DVI System Overview 99
C Children's Experiences with VideoLogo 102

Bibliography 106

List of Figures

3.1 Primitive Organization 30
3.2 SHAPES Screen 37
3.3 Sample Video Effects Screen 48
3.4 Example Application Screen 50

4.1 Task Structure 55
4.2 Video Play Control Structure 58

A.1 Screen Coordinates 73
A.2 Neighboring Pixels 78

C.1 Screen from Children's Procedure 105

Chapter 1

Introduction

All-digital multimedia systems are becoming inexpensive enough to be placed in schools, businesses, and homes. Early versions of these systems are already powerful enough to implement still image and video coding for data rate and storage reduction, to add object-based information to images, and to generate real time video sequences formed from combining multiple video objects and sources.

With the increase in accessibility and decrease in cost, we can begin to think about how to integrate these tools in less constraining, more accessible and flexible ways in order to enable a wide spectrum of users, ranging from developers to teachers, from casual end users to elementary school children, to develop their own multimedia applications.

1.1 The Need for Accessible Video Manipulation Tools

This collection of new video capabilities provides a great potential for creating a new class of manipulable video data types. Each represents a unique set of features to be exploited, and in combination they promise an incredibly flexible and rich set of manipulations.

In the past, these capabilities have often been integrated into large, complex, closed applications. This constrains the user into a highly restricted set of interactions -- the operations that the author of the example intended. How much better if we could turn the things around so that instead of constraining, we encouraged creating instead? We have gone through all the trouble of inventing and developing all these capabilities, but we do them an injustice by treating them as *components* instead of *tools* [Harel90a].

What is needed is a environment where an extensive set of video capabilities and manipulations are made available, and access provided to them in a flexible and simple manner.

1.2 Logo with Video Tools as a Learning Environment

Until now, the closest things to the system being proposed have been authoring systems, which are often packaged with a large number of assumptions about what's to be authored [Harel90a]. The more atomic the set of manipulations and interactions provided, the easier it is to learn about the ways in which various basic capabilities can and should be applied.

A natural approach to solving this problem is to base the system upon a simple, extensible, and programmable environment, and this was why the Logo language was chosen. Logo already provides the ability to manipulate graphics, text, audio, and simple animations. Children are using it to create their own stories -- think about how much more compelling those stories could be, to both the authors and the readers, if they used completely interactive, manipulable video created by the authors themselves.

This thesis attempts to address this by creating a software environment, *VideoLogo*, which adds video manipulation capabilities to Logo, enabling users to merge conventional computer graphics objects with new types of video objects.

It is not simply a matter of providing better or more realistic image quality; an entire new class of data types become available. At the simplest level, children can provide short video clips of themselves demonstrating how to do something. The next level up encompasses manipulations, such as making the motion video window change size or shape, or copying an object from the background and using it as a paintbrush. At the highest level, multiple video objects and graphics can be animated and overlayed over motion video.

1.3 Existing Logo Learning Environments

Two existing Logo learning environments illustrate the potential and realizable benefits of the addition of video tools. The Instructional Software Design Project [Harel88, Harel90b], is a learning environment where elementary school children used Logo to develop instructional software to teach fractions to younger students. By working in a framework for dealing with general representations of knowledge, in the form of computer programs, procedures and data structures, and with more specific representations related to the problem at hand, such as algorithms and images, children were able to gain a greater understanding of fractions, one that took them beyond rigid scholastic models, to more personal and applicable ones.

The vast majority of applications created by the children used imagery as the primary method of communicating information. Line graphics drawings of objects such as houses, containers, money, clocks, and almost anything else capable of representing a fraction were shown, accompanied by text labels and prompts. Children spent a great deal of time fine-tuning the appearance of these images, indicating how absorbed they became in the process.

It is plausible to assume that the drawing capabilities of Logo helped the children form, and obvious that it helped them to communicate their representations. One cannot help but extrapolate that the availability of more sophisticated imaging tools, including motion and still video, would enable them to create even more compelling models, with an attendant increase in their grasp of the knowledge being worked with, and a greater potential to communicate to others [Farber90, Segall90].

The second example of a Logo learning environment, The Visual Telecommunication Workstation [Dickinson90], is one of the first Logo systems to integrate emerging digital video technologies. In this system, children were able to digitize video stills and combine them with Logo graphics. Children used the systems to compose video letters and transmit and receive them between the two schools involved in the project; schools in two different countries. By using video stills as elements in their projects, children could

> ...transport a specific image into the computer, write and "talk" about it, augment and transform it -- so that it represents an idea that they want to communicate to others. This ability to program, personalize, and appropriate the electronic media with which they are working -- is their first step towards building a visual language by which to communicate.

The Visual Telecommunication Workstation demonstrates the power of using video as a media for conveying ideas. Not only abstract ideas, such as mathematical concepts, but things such as emotional identification with the subject at hand -- for instance, students could not have used text and line graphics to show what they looked like!

1.4 Instructionism vs. Constructionism

An important thing to note about both the Instructional Software Design Project and the Visual Telecommunication Workstation learning environments was that the subject of the children's work was not imposed by the software. The software was not trying to "teach" anything; rather it was the children's use of the tools that enabled them to learn by building their own knowledge representations. this approach is called *constructionism*:

> We understand "constructionism" as including, but going beyond, what Piaget would call "constructivism." The word with the v expresses the theory that knowledge is built by the learner, not supplied by the teacher. The word with the n expresses the further idea that this happens especially felicitously when the learner is engaged in the *construction* of something external or at least shareable ... a sand castle, a machine, a computer program, a book [Papert90].

Conversely, as one might guess, the term *instructionism* refers to the more common, and more rigid, format of most software found in schools. Instructionist software presents prepackaged knowledge in a predefined format, thereby minimizing the student's opportunities for formulating their own personalized knowledge.

How many times have you heard the old adage "a picture is worth a thousand words" ? Taken in the constructionist context, this makes video seem to be an ideal tool for representing and conveying knowledge, and therefore presents a compelling reason for investigating what happens when video tools are made available in a constructionist environment.

1.5 Synthetic Movies

A synthetic movie is a motion video sequence that is built up from multiple image representations, on demand, in real time, as it is being displayed. [Watlington89] describes two types: intraframe and interframe synthetic movies.

The most familiar examples of int*ra*frame synthetic movies include some of the more realistic commercial video games, as this type of synthetic movie is the most desirable solution when the application requires a high degree of interactivity.

Intraframe movies aid interactivity because they separate the visual elements and the action. For example, in a video game, the elements consist of foreground objects, and the action consists of the user's and the game's control of the motion of the elements. It is helpful to think of this type of synthetic movie as consisting of a 'script' and 'actors', where the actors are the elements, and the script the instructions that make the actors appear, move, interact, change, and disappear.

Int*er*frame synthetic movies are more limited. The smallest manipulable object is an entire frame, so the only control permitted is over the speed and branching of the sequences. The most common example of interframe synthetic movies is interactive videodisc [Backer88].

1.5.1 Digital Image Representation and Compression

In digital multimedia systems, the digital video data can be compressed in order to reduce storage requirements and required bandwidth. While compression of still video images is desirable, compression of motion is absolutely necessary -- to record a single frame of motion video accurately requires approximately one megabyte of storage[1]. A transmission rate of 30 megabytes per second would be necessary to play video at this rate, requiring a transfer speed and storage capacity far beyond the capabilities of currently affordable and practical digital storage technologies [Luther89].

The data rates typically available to the digital multimedia system designer are far lower; a typical rate is 150 kilobytes per second, the rate of a CD-ROM. Rates as low as 64 kilobits per second are common in teleconferencing applications.

Compression of realistic images can take place in several areas. First, the temporal, spatial, or tone scale resolution of the image can be reduced, giving a significant savings [Schrieber86]. Which type of resolution to reduce, and the degree of reduction, is dependent upon the application, the system capabilities, and human observer; some amount of reduction will be undetectable. As is more often the case, noticeable resolution reduction is unavoidable based upon constraints of the system.

An additional degree of compression can be achieved by encoding the image in a manner which takes advantage of characteristics of the human visual system in order to determine how to reduce information (known as "lossy" encoding). For example, sensitivity to detail in chrominance is lower

---

[1] Assuming one byte per channel, 3 channels, 512 lines, 768 pixels per line.

than that of luminance [Netravali88] and therefore chrominance spatial resolution can be significantly lowered without a very noticeable impact in image quality.

Aside from compression, the manner in which digital data is used to represent an image can convey additional information and provide additional benefits. For instance, representing an image or sequence as a set of spatial and temporal frequency components, known as “pyramid” or “sub-band” encoding, allows for more efficient compression, and results in a representation naturally suited for applications that require partial decoding be sufficient to recognize an image, or to show a lower resolution version of an image [Adelson84, Butera88, Romano89]. Vector quantization is another example of a compression representation useful for embodying information about an image [King88, Romano89].

Chapter 2

Multimedia Applications

This first section of this chapter describes several multimedia applications as examples of the use of various synthetic movie approaches. The second section describes the current state of multimedia application development tools, using these applications as examples.

2.1 Application Examples

2.1.1 Video Finger

Video Finger is an application of an object-oriented digital video system developed on custom hardware for the Apple Macintosh II [Watlington89]. The intraframe synthetic movie system which Video Finger is based upon synthesizes video frames by combining multiple objects, in this case representing people, into a single scene. Each object has a specific depth in the frame, and multiple frames for each object are cycled through, animating the objects.

Video Finger uses these capabilities to provide a visual version of the UNIX™ "finger" utility. In its conventional form, finger outputs a text list of

who is currently logged in to the system, and what programs they are running. Video Finger uses this input, and combined with a video object database, presents a moving video image that is constantly updated to reflect the presence and activities of the system users.

For each potential user, a set of standard motions corresponding to computer use activities (e.g. standing up and walking away for logging out, sleeping for idle) was filmed on blue screen, digitized, and separated from the background.

2.1.2 Network Plus and Digital Interactive News

The Network Plus and Digital Interactive News applications are a logical extension to the concept of personalized electronic newspapers. The Network Plus system combines digitized television news video still images with wire service articles, decoding the closed caption[2] text broadcast with the video to provide keyword links between the video and the articles. A user can sit down at the Network Plus workstation, select a news story, and see presented both the article from the wire service and digitized stills from that evening's TV news broadcast(s), which the system automatically linked to the article using keywords found in the closed caption text [Bender88].

Digital Interactive News (DIN) is an enhancement of Network Plus that takes advantage of newer technology to record motion video and audio, instead of video stills. Because it has a video database which is constantly being updated and must be random-accessible, DIN cannot use analog videotape or videodisc. Instead, DIN uses Intel's DVI to digitize and compress

[2] Closed caption text is broadcast digitally in the vertical interval between video frames.

the video in real time, and store it on a hard disk. Using DVI video compression, video and audio for a half hour newscast require roughly 270 megabytes[3] of storage.

Using digital video brings another advantage to the system; the video may be transmitted from the recording station to the viewing station over a digital computer network.

The most challenging area of research facing DIN is ensuring accurate automated selection of video segments using only close caption text and characteristics of the video image and signal.

2.1.3 Palenque

Palenque is a multimedia/hypermedia interframe synthetic movie application developed at RCA Laboratories and Bank Street College of Education. Versions for both interactive videodisc and DVI were developed. The interactive video concepts embodied in Palenque are derived from earlier interactive movie research, such as the Aspen Movie Map [Lippman80, Mohl81].

Palenque is an open ended instructional application designed to allow grade school children to interactively learn about Mayan ruins in Mexico. Children use a joystick to travel along a series of pre-recorded video paths. The speed and direction are controllable, and path changes can be performed at predetermined branch points. At certain sections along the path, video icons representing human "guides" appear which, if selected, show a video or audio clip about a related subject. 360° panoramic scans are available at selected path intersections. The user may "take a picture" of the displayed

[3] DVI video data rate is approximately 150K/second. 150K x 60sec x 30min = 270,000K.

scene at any time and place it in their scrapbook. Aside from conventional text menus, Palenque uses still video icons, and sometimes uses objects in still video images as "buttons" [Benenson87].

2.1.4 Elastic Charles

The Elastic Charles is a hypermedia "movie magazine". Elastic Charles runs on an Apple Macintosh II equipped with a video overlay card and an analog videodisc player. The videodisc contains footage about many Charles river related subjects, and the Macintosh runs Hypercard, augmented by a set of hyper/multimedia tools implemented specifically for such projects.

An Elastic Charles "reader" may navigate through the stories and databases in a variety of ways: by selection through a table of contents, cross references from other stories, by topic, etc. Stories combine interactive text and graphics information on one monitor, and video on another.

The most significant capability of Elastic Charles lies in the power of its multimedia tools, which extend hypertext links beyond text into the realm of motion video. Users may create and modify links using a point-and-click interface. These links permit new sequences to be created by splicing together previously unconnected video sequences.

Video links are represented by "micons", meaning moving icon. A micon appears as a small independently moveable window containing a several-second long repeating video clip; selecting a micon will begin to play the actual video it represents. The micon frames are created by digitizing each video frame, subsampling it to a small size, and storing the frame sequence on the computer's hard disk, creating an interframe synthetic movie. Multiple

micons may be displayed simultaneously, and they operate transparently to the host application, in this case Hypercard [Brøndmo89].

2.2 Development Tools

Each of the applications described in the previous sections approach the issues of application design and development in a different manner. Most require some type of processing be performed on the motion video data. All have their own manipulation, presentation, and interaction methods.

Developing multimedia applications involves integrating disparate technologies such as digital computers, analog video, and text, in an interactive system. Because of the complexity of these applications, and the fact that they often fall into categories (such as training, database, etc.) which have similar needs independent of the subject matter, it becomes attractive to provide authoring tools to aid development of such applications.

2.2.1 Authoring Systems

A number of authoring systems exist for creating interactive video applications, such as IBM's Infowindows, Authology: Multimedia by CEIT, and MEDIAScript by Network Technology Corporation. A specific authoring system is designed to address a specific set of problems, in a specific class of applications. Because of the large number of assumptions about what's to be authored, developing with authoring systems can become a lot like choosing dishes from a menu, coloring in a coloring book, or filling out forms.

For example, an authoring system for developing training applications

may provide commands for presenting video, overlaying graphics, and branching on user inputs. To do something in an application outside of this restricted domain, the authoring system may provide "hooks" to enable addition of new software capabilities.

The natural tendency in such a situation is to address the shortcomings of the authoring system by adding features to it. As more features are added, complexity grows, and the original advantages of an authoring system may become endangered.

2.2.2 High Level Languages

Unfortunately, the alternative to a possibly constraining authoring system is almost always a dense and complex "high-level" language, such as C or Pascal. High-level in this case does not mean the same thing as providing useful abstractions for multimedia primitives, but rather much simpler and less useful (in the context of multimedia application development) abstractions of conventional computer programming elements, things that are actually quite low-level, such as adding or comparing two numbers. These languages are called "high level" because they do provide one layer of abstraction above the computer's native machine language.

The advantage of writing a multimedia application in this manner is that the designer can take advantage of every capability of the system, and combine them in novel ways that an authoring system designer may not have foreseen, and therefore not provided a method for.

The disadvantage is the amount of work involved. Absolutely everything, even the most atomic manipulations, such as presenting the contents of a text

file on the screen, require dozens of lines of code. Of course, subroutines and libraries ameliorate this problem to a large degree, but even if these are used as much as possible, other drawbacks of high level languages have their impact; among them the lack of interactive development.

Most high level languages impose a cyclical three-stage development process of edit, compile, and link, often using a separate program to perform each stage. Even in development systems where the three are integrated, building and using the application are still separated.

2.2.3 A Simple Multimedia Development Environment

An ideal system for investigating the applications of video tools in multimedia would avoid most of the drawbacks of both authoring systems and so-called high level languages. This requires finding a mid-point between the abstractions of authoring systems and the overwhelming detail and complexity of programming languages. It also means an environment that does not impose divisions between developing, debugging, and using a multimedia application.

Designing a system to satisfy the latter requirement is simple enough. By making the language interpreted instead of compiled, the environments for developing and using an application can be made the same.

Walking the thin line between too much and too little abstraction is a more difficult proposition. The approach taken in VideoLogo was to seek to implement a minimum useful level of abstraction with a carefully chosen collection of video primitives, thereby allowing users combine those primitives into procedures to formulate more abstract, higher level tools.

Chapter 3

The VideoLogo Environment

This chapter discusses the working environment of VideoLogo, VideoLogo objects and manipulations, and VideoLogo commands. Information in this chapter is supplemented by Appendix A.

3.1 Design of Primitives

The process of selecting primitives to add to VideoLogo was shaped by several requirements, including the desired level of multimedia capabilities (always to be weighed against ease of use), the capabilities of the DVI system that VideoLogo was implemented on, and the desire to expand upon existing Logo models whenever possible.

These requirements roughly correspond with three classes of VideoLogo primitives described below: interaction with video (the essential multimedia capabilities), combination of graphics and video (based on DVI capabilities), and video shapes and animation (an extension of a Logo graphics idea).

3.1.1 The Turtle as Actor

VideoLogo primitives adhere as closely as possible to the Logo convention of thinking of the turtle as an actor, and primitives as verbs or adjectives that make the turtle do something or change one of its attributes. Standard Logo primitives such as FORWARD, BACK, LEFT, and RIGHT, all are local (as opposed to global) in the sense that the state of the world before and after they are executed depends solely on the location and attributes of the turtle. This 'situatedness' is a central concept of Logo, enabling children to think about turtle geometry by 'playing turtle' [Papert80]. Whenever possible, new VideoLogo primitives were cast into this actor/action/attribute mold.

What can a turtle model bring to video? Several VideoLogo capabilities illustrate: for instance, the turtle's pen in conventional Logo controls whether the turtle will draw or not when the FORWARD or BACK commands are given. This model has usually been extended to other graphics primitives as well; the STAMP primitive in LogoWriter (by Logo Computer Systems Inc.), which leaves a turtle image on the screen, will have no effect when the pen is up. Extending the pen to motion video, where new frames are only displayed if the pen is down, makes it possible to do things like freeze-frame video, yet continue audio, simply by taking the pen up. This is achieved without having to use any new primitives, and uses a familiar Logo convention in the bargain. Other examples include using the turtle position to control the location of the video window, using it's background color (also known as erasing color) to control transparency, and using it's heading to rotate video shapes.

In several cases, primitives were designed as new attributes of the turtle.

For instance, much as SETC changes the turtle color in conventional Logos, VideoLogo's SETSCALE changes the size of the turtle. This follows the model of using the SET prefix to change an attribute; without the prefix the primitive reports the attribute. Whenever possible, the names selected for these primitives were chosen to be similar to those describing physical attributes, although the abstractions of video sometimes made this difficult; for instance the SYNC primitive deals with the abstract concept of synchronization, and because of system constraints, does not take effect until the next PLAY command.

3.1.2 Interaction with Video

The first, and easiest primitives to design were those that dealt with motion video, since the basic requirements were straightforward: the system should provide the ability to display a video window and give the user interactive control. A VCR-like set of primitives was selected (see section 3.10.1); this VCR model was chosen because it is familiar, simple, and closely approximates the way in which the DVI library routines actually control video.

The motion video primitives are designed hierarchically; the PLAY primitive is the only command required to start video playing; all other video primitives essentially act as modifiers on the executing PLAY. In this way, the capability that primarily separates VideoLogo from existing Logos could be learned quickly, thereby encouraging users to master increasingly capable commands.

Because DVI stores individual video segments in separate disk files, it

was tempting to provide a primitive that would 'open' a video file, and then have the other video primitives refer to that file when requesting actions such as play, pause, etc, allowing multiple files to be open at once. This was replaced by the simpler model which only allowed a single video file to be open at any time. By doing this, the atomic action of 'playing a video' can be accomplished with a single command, avoiding the abstraction of open vs. unopened files. The other reason was more pragmatic: memory restrictions in the DVI system precluded the existence of more than one set of motion video buffers at a time.

3.1.3 Combining Graphics and Video

More complex than determining what commands should control video was designing the interactions between video and graphics. Once again, an existing convention was borrowed, this time from interactive video rather than Logo. Interactive videodisc systems often include a graphics overlay board that can display graphics on top of the motion video [Lippman80, Mohl81, Brøndmo89]. In these systems, the overlaying is done in hardware; the only control the application has is over what pixels appear as transparent to video (i.e. which pixels are the 'key' color), and which do not.

In the DVI system, since both video and graphics exist as pixels in memory, the simulation of overlaying is actually performed by doing a copy-with-transparency on top of the video frame; therefore the overlay convention can be extended somewhat to gain extra flexibility.

Essentially, the concept of object, attributes, and actions was used once again, with primitives for defining the overlay region, called a "mask" (the

object and attributes), and controlling the manner in which it is overlayed, called "keying" (the action). See section 3.10.2 for a description of the primitives that control overlaying.

3.1.4 Shapes Design

Aside from the display of motion video and video stills, it was desirable that VideoLogo support the capability of manipulating parts of video images by moving them around the screen, rotating them, scaling, and re-coloring them. This capability enhances VideoLogo beyond the scope of 'graphics-overlayed-on-video', allowing animation and compositing of images from multiple elements.

Conveniently, once again Logo provides a model that can be expanded upon; that of the turtle shape. Shapes provide the convenience of fast and easy display; once the user has placed an image on screen by whatever method and copied it to a shape, there is no need to use a complex (and possibly slow) Logo procedure to redraw it later.

The shape mechanism is simple; the image of the turtle may be replaced by some other user-definable one. Whatever the turtle does, the shape does too; thus the shape may be moved, stamped onto the screen, and shown and hidden. By quickly cycling through several shapes, animation is possible as well. Shapes are initially created by copying a portion of the screen, and can be saved on disk.

VideoLogo makes few changes to these existing shape controlling primitives. Instead, the added functionality is attained by extending the definition of a shape from a small, fixed-size, single-colored, non-rotatable

bitmap to an arbitrarily sized and shaped bitmap that can be rotated and scaled.

The method of creating the shape is the main difference between VideoLogo's and conventional Logo's shape primitives. Since VideoLogo shapes can be any size or shape, the CUTOUT primitive (which replaces conventional Logo's SNAP primitive) must define a region of the screen to be the outline of the shape. This is done by defining a procedure that will draw the shape outline at the proper part of the screen, and passing it as input to CUTOUT.

This mechanism of primitives that take procedures as input is derived from the RUN[4] primitive in conventional Logo, and is used by other VideoLogo primitives that input procedures in order to define regions (FILLIN) or actions (ANIMATE).

See section 3.8 for a functional description of VideoLogo shape primitives and animation.

3.1.5 Effects Design

The DVI system is able to perform a number of real-time manipulations on still and motion video. These manipulations include those that occur at the pixel level (i.e. brightness and tint), frame level (i.e. mirror image and edge detect), and multiple frame level (average and difference).

These manipulations, or effects, are useful to different applications for different reasons. For instance, most of the pixel level effects correspond to the controls on a television set, and are therefore well-suited for adjusting the parameters of a video sequence.

[4] RUN takes a list as input and executes it.

Because it was impossible to predict which effects would be applied to which objects, and reasonable to expect that new effects would become available as the DVI system expanded, individual effects were not implemented as individual primitives. Instead, primitives for each type of object that could have an effect applied to it (motion video, shapes, screen) are defined, which take as input a list of effect names. Using a list of effects has another advantage over multiple primitives; the list inherently embodies sequence and parallelism -- which effects are in the list, and their order, can determine the final appearance of the video object.

See section 3.11 for a functional description of the video effects primitives.

	Action Primitives	Attribute Changing Primitives	Attribute Reporting Primitives
Conventional Logo	FORWARD BACK LEFT RIGHT HOME PENUP PENDOWN	SETH SETPOS SETPENST SETC SETBG SETSH	HEADING POS PENSTATE COLOR BG SHAPE
VideoLogo	*PLAY* *PAUSE* *STEP* *RESUME* *LOADPIC*	*SETSCALE* *SETCROP* *SETFONT* *SETSYNC* *SETVOLUME* *SETFX*	*SCALE* *CROP* *FONT* *SYNC* *VOLUME* *FX*

Figure 3.1 Organization of primitives as actions, attributes, and attribute reporters. Some primitives have been omitted for the sake of clarity; see Appendix A for a complete list.

3.2 System Environment

VideoLogo currently runs on a DVI-equipped 386 PC equipped with two monitors, one for display of text (primarily editing of Logo procedures and text input and output), and the other for the display of video and graphics, where the VideoLogo turtle appears.

This configuration is one of necessity rather than choice; at the time of development it was not possible to combine video, graphics, and the PC's text

output on a single monitor.

The VideoLogo environment, much like other Logo implementations, integrates a screen editor and a command-line interface. The user may enter direct commands to be immediately executed by the system, or by entering them in the editor, to have them available to VideoLogo when the editing is done.

3.3 The Editor

The VideoLogo system is integrated with a simple screen-oriented editor where users can enter and edit their procedures. The quickest solution within the scope of this thesis was to use a commercially available screen oriented editor application for DOS with configurable keyboard mapping, in order to make the keys for moving the cursor, inserting and deleting, etc. compatible with the ones used by LogoWriter, by Logo Computer Systems Inc.

Integration of the editor with VideoLogo was accomplished by having execution controlled by a DOS batch file[5] with the following logic, represented here as pseudocode:

```
ERASE EDITFILE
LOOP
RUN VIDEOLOGO WITH EDITFILE AS INPUT AND OUTPUT
IF EXITCODE = BYE THEN END
RUN EDITOR WITH EDITFILE AS INPUT AND OUTPUT
GOTO LOOP
```

[5] A batch file is a simple program that DOS can interpret and execute. Batch files can take inputs and have simple looping and testing constructs.

When a user starts VideoLogo, it is actually this batch file that is being run. In the above logic, EDITFILE represents a text file containing VideoLogo procedure definitions. When the batch file starts, any existing EDITFILE is erased, and VideoLogo is started. VideoLogo reads all the procedure definitions in EDITFILE (which is empty when the batch file starts). While VideoLogo is running, procedures may be changed, added or deleted at the command line. When VideoLogo encounters the EDIT command, all procedures in the workspace are written out to EDITFILE, and VideoLogo quits, setting an exit code to signal the batch file to run the editor. The editor reads the new EDITFILE, writes it out again with the user's changes, and the cycle repeats until the BYE command is encountered.

In addition to preserving procedure definitions across invocations of the editor, the integration is maintained by a set of status files that VideoLogo writes out each time it is exited, and reads in when it returns from the editor. These files contain variable definitions, and system video and graphics status information, such as pen state, shape definitions, etc. Some status information does not need to be preserved in files because the memory it resides in can safely be assumed to remain unchanged by the editor; this includes all memory on the DVI card which contains the graphics/video screen, and DOS extended memory which VideoLogo uses to store shape images.

This looping structure was chosen for several reasons. First, to run the editor as a sub-process of VideoLogo (or vice-versa) would have meant that system memory would have to have been shared between VideoLogo and the editor, and since the editor was a complete separate application the amount of overhead would have been undesirably high; using the batch file gives both

the editor and VideoLogo maximum system resources. Additionally, DVI software's multitasking kernel made the use of such sub-processes unreliable. Finally, using this structure makes it easy to switch to new editors as the situation warrants; all that is required is a change to a single line of the batch file; no programming is necessary.

3.4 Predefined Primitives

Several VideoLogo primitives are implemented as Logo procedures to save lower-level programming effort. Such procedures are collected together in a single file that VideoLogo reads every time it starts. Although these procedures can be displayed by the PO, POALL and TEXT[6] commands, they do not appear in the editor, nor can they be erased. No mechanism is supplied whereby typical VideoLogo users can modify this file, and its name begins with an underscore to make overwriting by the user's files unlikely (other files, including the sign-on, status and editor files, also start with underscores to avoid this problem).

Once the predefined procedures file is read, a second file is read to display a sign-on screen. The end of this file actually contains VideoLogo commands outside of the procedure definitions which run the sign-on procedures and then erase them.

[6] PRINTPROC and TEXT output the text of Logo procedures.

3.5 VideoLogo Workspace

In Logo parlance, "workspace" refers to the collection of procedures and variables currently defined. VideoLogo includes SAVE and LOAD primitives for writing and reading procedure definitions. Files that are saved and loaded may contain any number of procedures; when the SAVE command is given, all procedures in the workspace are written into the file, and when LOAD is encountered, the procedures defined in the file are added to the workspace. There is no provision for selecting particular procedures in the workspace or file for saving or loading.

Files that are loaded may contain any VideoLogo commands which will be executed as the file is being read, not just those commands used for defining procedures. This feature is useful for making self-running procedures. Note, however, that since VideoLogo outputs only procedure definitions for EDIT or SAVE, it currently is not possible to insert direct commands into files from within VideoLogo or the editor.

3.6 VideoLogo Language

Because the graphics primitive of the Logo language as implemented on current personal computers is fairly well-defined, this section will concentrate only on those new graphics and video primitives that VideoLogo adds to Logo.

The basic features of the Logo language itself, that is, the portions that handle data manipulation, parsing, text input and output, etc., have not changed. In addition, many of the graphics primitives that exist on commercial personal computer Logos, i.e. FILL, LOADPIC, etc. remain,

although in many cases improved VideoLogo variants exist as well.

Complete descriptions of all the VideoLogo primitives are included in Appendix A, the instruction manual. This chapter omits some primitives because they are not crucial to describing VideoLogo's functionality.

3.7 Turtle Graphics

One feature that most implementations of Logo share is the turtle. The first Logo turtles were real objects; a chassis with two motorized wheels on either side and a pen that could be lifted and lowered in the center, covered by a plastic dome which gave it a turtle-like appearance. The turtle was attached by a cable to a computer. Physical turtles have gradually been replaced by more precise and far more flexible screen turtles that exist as a picture of a turtle on a computer screen, but the idea of a turtle as an "object to think with" hasn't changed [Papert80].

A turtle has a location given by a set of cartesian coordinates, represented in Logo by a list of two numbers, e.g. [0 0] for the origin, and a heading in degrees represented as a single number. The origin is at the center of the screen, and the initial heading of zero means the turtle is pointing straight up along the Y axis.

The turtle's location can be changed in a relative manner by the movement commands FORWARD and BACK which take a distance as input; the smallest noticeable distance a turtle can move is a single "turtle step", which is nearly always equal to the width of a single pixel on the display. If the turtle moves off the edge of the screen it wraps around to the other side. The turtle can be moved to an absolute location using SETPOS, which takes a

coordinate as input.

The turtle's heading can be changed in either absolute or relative terms; the RIGHT and LEFT commands turn the turtle in place clockwise or counter-clockwise a given number of degrees, and the SETHEADING command allows the heading to be set absolutely.

Turtles carry a pen that leaves a trace along the turtle's path. Commands exist to lift (PENUP) and lower (PENDOWN) the pen, change its color (SETC) or background color (SETBG) and change the way it draws (PENERASE, PENEXCHANGE). The color of the turtle is the same as the pen's.

All of the commands discussed above have counterparts that report turtle and pen status: POS gives the turtle's location, HEADING gives its heading, COLOR the pen color, etc.

3.8 VideoLogo Shapes

Existing versions of Logo often provide a library of selectable turtle shapes consisting of small bitmaps (usually not much larger than the turtle image) that represent various objects, such as airplanes, faces, etc. The turtle shape can be stamped onto the screen using the STAMP primitive to become part of the background. Different shapes may be selected using the SETSH primitive, and new shapes can be created by copying part of the screen to the shape.

The number of shapes available in various Logo implementations depends on the shape size and available memory. VideoLogo contains 10 shape "slots"[7] which are initially empty, except for the first slot which contains

the default turtle shape. If an empty slot is selected, its shape continues to appear to be a turtle until it is changed. The first slot, number 0, may not be changed; if the user attempts to do so an error message will be displayed.

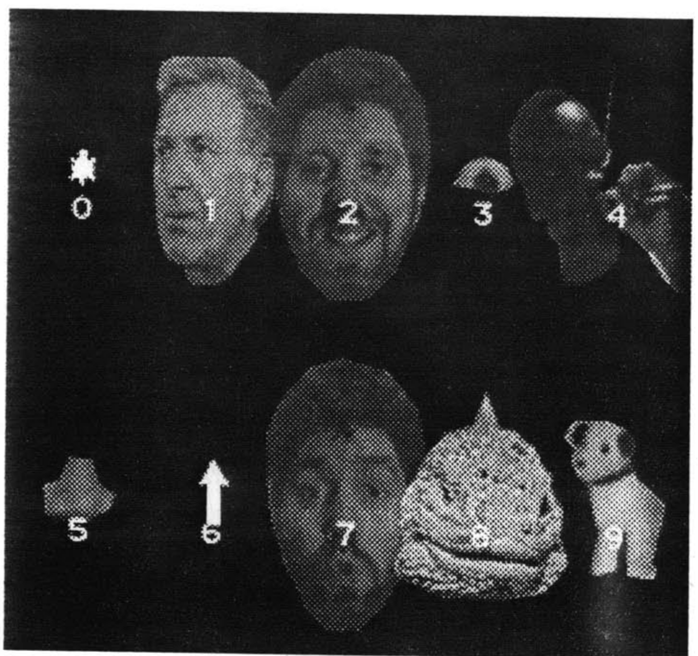


Figure 3.2 Screen displayed by SHAPES primitive, showing a collection of video and graphics shapes.

[7] Since shapes are kept in DOS extended memory, ten shapes at 64KB each fit into less than one megabyte, an acceptable minimum amount of extended memory.

3.8.1 Creating Shapes

The CUTOUT primitive is used to create a shape by using the turtle to outline an arbitrarily shaped[8] region of the screen. Once the area is defined, the portion of the screen inside the region becomes the new shape. The input to CUTOUT is a list which contains turtle motion commands. For example, CUTOUT [REPEAT 4 [FORWARD 100 RIGHT 90]] would cut out a square region of the screen and place it into the current shape slot.

Because VideoLogo allows the user to work with realistic video imagery, where the outline of an object in a scene is most likely irregular, an additional method of outlining the shape is provided: the use of a mouse. VideoLogo provides the MOVEMOUSE primitive which attaches the Logo turtle to the mouse, and draws connected line segments each time the mouse is clicked. When the mouse is double-clicked, the outline is closed and the MOVEMOUSE procedure stops. Therefore, the use of CUTOUT [MOVEMOUSE] affords the greatest amount of control when the user wants to create a shape from part of a video image. Aside from MOVEMOUSE, primitives for direct access of mouse position and button information are provided (see Appendix A).

The internal representation of a VideoLogo shape is a rectangular bitmap just large enough to hold the shape image at the orientation it was originally CUTOUT at, and a polygon that defines the part of the bitmap that actually contains the shape. Preserving the polygon that was used to define the shape, as opposed to just storing a bitmap containing the shape, allows for more

[8] Due to memory constraints, the current version of VideoLogo cannot handle shapes larger than one quarter of the screen area.

accurate rotation and scaling when using the DVI warp[9] algorithm.

3.8.2 Shape Manipulation

Once a particular shape slot is selected, the shape in that slot replaces the turtle shape. Like the turtle image, the shape will rotate to follow the turtle's heading when the LEFT, RIGHT or SETH commands are executed. This rotation is performed by rotating the shape's outline polygon and warping the original shape at zero heading into the rotated outline.

The shape may have transparent regions; although its outline is a polygon, any parts of the shape inside the bounding polygon that are the same as the current background color are treated as transparent. When the background color is changed using SETBG, the transparent areas of the shape, if any, change as well. The only exception is if the shape is number 0, the reserved turtle shape; in this case the turtle remains opaque even if the turtle and background color are the same[10].

The shape's size can be changed using the SETSCALE primitive. The vertices comprising the shape's polygon are multiplied by the input to SETSCALE, and the shape image is expanded or contracted to fit. Shape 0 cannot be scaled.

[9] Warp performs texture mapping from one arbitrarily-shaped polygon to another.

[10] Actually, the turtle and pen color are never exactly the same; to prevent the turtle from disappearing, its color is always slightly lighter or darker than the corresponding pen color.

3.8.3 Shape Housekeeping

Individual shapes may be stored in disk files. The current shape can be written to a file using the SAVESH primitive, and can be loaded back in with LOADSH. Both SAVESH and LOADSH take a filename as input; whatever shape slot is currently in use will be used as the source or destination. This approach differs from that of other Logo implementations[11].

If a shape slot is already occupied, it cannot be changed by CUTOUT or LOADSH until it is emptied using the ERASESH primitive.

3.8.4 Shape Animation

A simple technique for animation can be achieved by rapidly cycling the turtle through several shapes; for instance, a set of shapes extracted from a video of a man walking could be stepped through to create an animation of a man walking on top of any background.

Normally, the system does not attempt to synchronize the display of shapes with the video display scan, so shapes flicker when they are moved or rotated[12]. An additional distracting effect when moving and rotating turtle shapes involves the "rotate-in-place" phenomenon; the turtle cannot move and rotate at the same time, so when a shape's size reaches a certain threshold, it is easy to notice when it is turning to a new heading at the end

[10] For instance, LogoWriter loads and saves shapes as a single group; individual shapes within a group may be referred to only by number.

[12] The DVI drawing processor can draw a shape faster than the screen refresh rate, so synchronizing shape drawing to the screen's refresh rate would slow down all drawing operations.

of moving in a straight line, before it goes off in a new direction. Of course, the turtle shape could be hidden when rotated, but this just replaces the distraction of seeing the rotation with the one of seeing the turtle shape appear and disappear repeatedly.

The ANIMATE primitive provides a solution to these visual distractions. ANIMATE takes as input a list containing any turtle and/or shape controlling commands and executes them in a manner that eliminates the problems by synchronizing shape drawing to the display refresh rate, double buffering[13] the display, and not rotating shapes until the next FORWARD or BACK command is encountered. Because of the double buffering, ANIMATE blocks any attempts to draw on the screen -- if drawing were permitted, it would always have to be performed on both screens, with an attendant decrease in animation speed.

3.9 Video Stills

VideoLogo provides rapid display of digital compressed video stills loaded from hard disk or CD-ROM. These stills can be digitized by the users themselves, or be provided as part of a library of images on CD-ROM. VideoLogo screens can also be saved to disk and redisplayed later.

The LOADPIC primitives is used to display an image; this image can be any size, be in one of three DVI image formats. LOADPIC attempts to center the image at the turtle's position; if the image size is the same as the screen it will fill the entire screen regardless of the turtle's location.

LOADPIC can display files stored in DVI's compressed and uncompressed

[13] Double buffering refers to the technique of drawing on one screen while displaying another, and then switching screens when the drawing is done.

9-bit formats[14], and 16-bit compressed formats. The formats were selected based on various requirements; 9-bit formats have the best image quality for realistic digitized images, and compressing them does not noticeably affect the image quality. Therefore, compressed 9-bit is used for digitized images. Uncompressed 9-bit is used when VideoLogo saves a screen using the SAVEPIC primitive, because graphics drawn on top of the video image would be corrupted by the compression process[15]. Finally, 16-bit compressed mode is also supported because 3rd party CD-ROM image libraries and paint applications primarily use this format.

The current implementation of VideoLogo provides access to a CD-ROM containing 2000 digitized full-screen photographs covering a huge range of topics. This disk is a product called Photobase™, by Applied Optical Media Corp. Users may use the simple Photobase™ database software provided on the CD-ROM to find images they want using keyword and subject searching, and then display them from their VideoLogo applications.

---

[14] The 9-bit formats are not physically 9 bits per pixel; instead, 9-bit image files are comprised of three 8-bit planes, two of which are 1/16th the area of the first. If all the bits in the planes are added and then divided by the number of pixels in the image, the result is 9 bits per pixel.

[15] 9-bit compression is "lossy", that is, it does not attempt to preserve exact pixel values. This is acceptable when pixel values represent luminance or chrominance (i.e. video), but it fails when pixel values represent color lookup table indices (i.e. graphics).

3.10 Motion Video

VideoLogo is capable of displaying all of the DVI motion video algorithms, including the PLV (for "Presentation Level Video") and all RTV (for "Real Time Video") symmetric[16] algorithms, although it is optimized to use the 10 frame per second RTV 1.0 algorithm. This algorithm provides low but useable image quality; the primary sacrifice made to achieve a high compression ratio is a reduction in frame rate to about 10-11 frames per second (NTSC television, and non-symmetric DVI algorithms display at 30 fps). Another characteristic of the RTV 1.0 algorithm is that it actually takes substantially less than 1/10th of a second to decode a frame of video, so the frame can be processed in various ways by the DVI drawing processor, before it is displayed, without impacting the speed of the video playback (the frame processing includes masking, keying, and video effects, all described in later sections). In comparison, the PLV algorithm, at 30 frames per second, requires nearly all of that time to decode a frame. For this reason, RTV 1.0 is the preferred DVI video algorithm for use by VideoLogo.

The size of a motion video frame is 1/4 the area of the VideoLogo screen, so normally video appears as a window centered at the turtle position. However, a ZOOM primitive is included that causes the DVI display processor to put the screen into a lower-resolution mode, filling the entire screen with motion video. For PLV video sequences, the ZOOM primitive enables the full 30 frame per second playback rate because it causes decoded frames to be displayed directly, rather than having them copied to the screen, as is

[16] A symmetric video compression algorithm can be both encoded and decoded in real time on the same system.

normally the case with non-ZOOMed video.

3.10.1 Video Playback Control

Motion video playback control is provided by a set of commands that closely mimic those of a VCR. The video playback primitives PLAY, PAUSE, STEP, RESUME, and PLAYSTOP are self-explanatory[17]. Only PLAY takes an input; the name of the video file to play. All subsequent playback commands refer to this file.

Once video begins to play, control returns to VideoLogo and playback continues in the background. If no other playback commands are issued, the video will play all the way through and then stop, the video file will be closed, and the region of the screen under the video will be restored.

Drawing may only be done while video is paused or stopped; while video is playing, the turtle may be moved, but it will be hidden and will not draw with its pen. This is because the DVI system software which permits only one task at a time to perform video or graphics operations, preventing reliable use of graphics while video is playing. This problem could be solved by modifying VideoLogo internals to call graphics from within a hook routine during video playback, but with an unavoidable speed penalty (for a discussion of tasks and hook routines, see chapter 4). If VideoLogo were to be extended to support concurrency in the future, handling graphics in this manner would be mandatory.

In practice, this limitation is not very restrictive due to the speed of the pause/resume cycle; a typical operation such as drawing a text caption onto

[17] Rewind , fast forward, and random access to individual frames are not currently supported by the DVI motion video algorithms.

the motion video frame (e.g. PAUSE LABEL [THIS IS A CAPTION] RESUME) takes roughly two frame times (i.e. 2/10th sec.) and is almost unnoticeable.

When video is paused, the turtle will reappear and is able to draw; once video play resumes the video frame will be re-centered at the turtle's new position. The fate of any drawing the the turtle did in the region where the video will appear is determined by the keying and masking commands, detailed in the next section.

VideoLogo provides access to frame numbers with the SHOWFRAME, FRAME, LASTFRAME, and WAITFRAME and WAITDONE primitives. SHOWFRAME displays a frame number in the upper-left corner of the video frame; FRAME reports the currently displayed frame, and LASTFRAME reports the last frame that can be shown in the video sequence.

WAITFRAME causes VideoLogo to wait until a certain frame has been displayed before continuing. By using WAITFRAME, a procedure may begin playing video, wait for a certain frame, pause the video, draw on top of it, and then resume the video. The WAITDONE primitive simply waits until the last frame has been displayed.

3.10.2 Masking and Keying

Like many analog video based multimedia systems, VideoLogo provides the ability to make parts of the screen transparent to video ("keying") and other parts opaque to it ("masking")[18]. For example, video may play under graphics of a certain color, but over background-colored or video regions of the screen. Besides using graphics colors to determine masking, VideoLogo also allows a shape to be used as a mask; the effect of this is to show the video only inside the boundaries of the shape. A full description of the masking and keying commands may be found in Appendix A.

Because VideoLogo video is digital, the system does not distinguish between video and graphics; they are both simply collections of pixels. When a frame of video appears on the screen, it has been copied from an offscreen image which was reconstructed by the video decompression software. Therefore, the software must be capable of distinguishing between the two types of pixels to be able to overlay graphics on top of video.

Whenever video is about to begin playing, the system preserves the region of the screen it is about to appear on top of, and, depending on the type of masking and keying involved, pre-processes the region (for instance, by converting video pixels to the background color so that video regions in the mask appear transparent). During the video playing process, each time a new video frame becomes available for display, the frame and the processed mask are combined according to the current masking and keying settings, and then the combined frame is displayed on the screen.

[18] Unlike analog video systems, this capability is provided by software, not hardware.

3.11 Video Effects

Any effect can be applied to a shape, the screen, or motion video. For each type of object, a pair of VideoLogo primitives allows setting and reporting of the effects being applied to the object. For instance, SETSHFX and SHFX set and report the effects being applied to the current shape.

Effects are passed and returned from these primitives in the form of a list containing the names of effects followed by optional values for effects that have variable settings. For example, SETPLAYFX [BRIGHT 100] applies the BRIGHT effect to motion video and gives it a brightness level of 100. This syntax allows for any number of effects to be selected for a particular object[19]. See Appendix A for a full description of available effects and their settings.

Several effects deserve mention in this section. The AVERAGE and DIFFERENCE effects work only when applied to motion video; both perform an operation on pairs of video frames. AVERAGE displays video frames that are the arithmetic mean of the last two frames displayed, causing a type of blurring effect.. DIFFERENCE displays the difference between the last two frames; as a result, DIFFERENCE frames appear dark where there is little motion (i.e. little change from one frame to the next) and bright where there is a lot of motion. Because comparatively large regions of moving areas of the video frame (e.g. the center of a sail on a sailboat moving across the screen) do not change brightness although they are moving, only the edges of such regions will be picked out.

---

19 The current implementation of VideoLogo permits the application of only one effect at a time. All effects except the last in the list are ignored.

The EDGE effect is not temporal; it looks at horizontal pairs of pixels in the image, finds the difference between them, and thresholds the difference value so that differences below a certain value appear as black, and those above it appear white. The visual effect of EDGE depends very much upon the type of image it operates on and the threshold value. For instance, motion video images recorded on the VideoLogo system tend to have poor edge resolution because of artifacts in the DVI compression/decompression process, which would tend to diminish edge information in the scene, and at the same time, noise in the form of compression artifacts may be injected into the image which may appear as false edges.

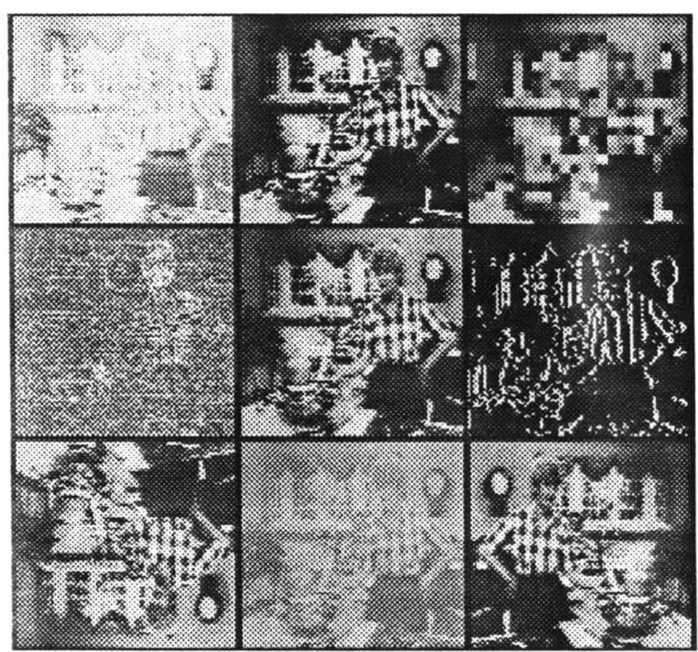


Figure 3.3: Sample video effects. Center: no effect; clockwise from upper left: high brightness, high contrast, blockiness, edges, mirror, low contrast, vertical flip, frame difference. This image was created by a VideoLogo application.

3.12 Video Capture/Recording

DVI provides a set of utility programs which can be used to record still and motion video and audio into disk files that can be immediately accessed by VideoLogo applications. Because of the complexity and scope of these operations, they cannot currently be performed from within the VideoLogo environment. Therefore, the VideoLogo system includes a set of simplified and packaged commands to run these utilities, which are described in Appendix A. It is expected that future versions of VideoLogo will provide primitives to perform these recording capabilities, using the same names as the utilities.

Utilities for recording still video, motion video, and audio are included. Additional utilities for converting images between various formats are provided as well; see Appendix A.

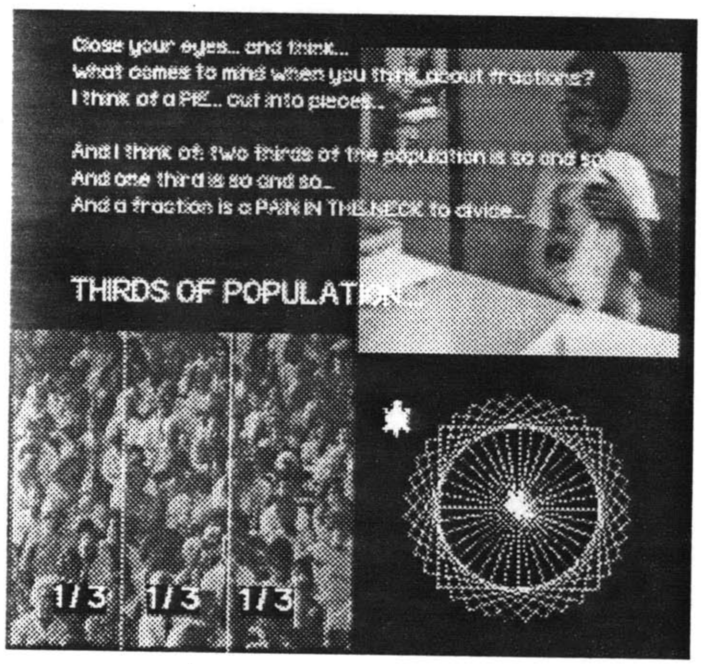


Figure 3.4 Screen from an application example showing text keyed over motion video (upper right), graphics drawn over decompressed video still (lower left), and conventional graphics (lower right).

Chapter 4

VideoLogo Software Description

This chapter details the internal software architecture of the VideoLogo system.

4.1 Software Overview

VideoLogo software was written entirely in Microsoft C. DVI microcode developed especially for VideoLogo was written using the DVI A82750PA assembler. The bulk of the VideoLogo software can be divided into two halves; the Logo interpreter code, and the video/graphics code.

The Logo interpreter code was originally written for the Macintosh in Symantec Corp.'s Think C by Alan Shaw. By using Shaw's code and modifying and building upon its user interface and graphics routines, a great deal of time was saved, and compatibility with existing versions of Logo was assured.

The graphics and video portion of the code was written to use existing DVI library routines whenever possible, and it was designed to be as independent of the rest of Logo as possible, allowing the video manipulations of VideoLogo to be easily separated from Logo should they be needed in a different

environment.

4.2 Drawing Routines

Practically all the primitives that draw graphics or video onto the screen make calls to DVI library routines which in turn use the Intel A82750PA drawing processor. The A82750PA is a display list driven device with programmable microcode; therefore a call to a DVI graphics routine, such as GrLine, performs the following steps invisibly to the caller:

- Load the line-drawing microcode from disk into the VRAM microcode buffer if it is not present.
- Parse the high level parameters to the line call into a low level form compatible with the microcode. This usually involves calculating VRAM addresses and pitches[20].
- Add a new display list entry containing the parameters for the line call and a pointer to the line microcode.
- Return to the caller.
- When the display list interpreter (which itself is a microcode routine) encounters the display list entry for the line call, the parameters and microcode are loaded onto the A82750PA chip and executed by the interpreter as a subroutine.

If it were never necessary to have the host CPU access VRAM directly, the existence of a display list could be ignored by the application. However, certain operations, such as loading and saving shapes and images, require that the host access VRAM. In these cases, it is crucial that the the host wait

---

[20] The pitch of a bitmap is the number of pixels that must be added to get to the corresponding pixel on the next line. It is not necessarily the same as the bitmap width.

until all pending display list operations have completed, or a situation could arise in which the host assumes a region of VRAM has been written to by the A82750PA before it actually has.

An additional microcode synchronization issue arises during double buffering; it is necessary to wait for the display list to complete before the screen being drawn into can be shown, otherwise the drawing operations could be visible.

4.3 Video Mode

DVI offers a large variety of pixel depths, modes, and resolutions. The 9Y pixel format was selected for VideoLogo because it supports intermixed color-mapped and subsampled chrominance pixels. In 9Y mode, the display consists of three 8-bit deep bit-planes. The first plane, the Y plane, is the same resolution as the screen. The least significant bit of the Y pixel determines if it is a color-mapped graphics pixel or a video pixel. If it is a graphics pixel, the pixel value is used as an index into a color table entry which determines the pixel's brightness and hue. If it is a video pixel, it is converted directly into luminance information, and the hue of the pixel is found in the remaining U and V chrominance[21] bitmaps. The U and V bitmaps are each one sixteenth the area of the Y bitmap; the Intel A82750DA display controller chip interpolates them up to full resolution of the screen when displaying them.

VideoLogo line drawing, fill and text operations are performed with color

[21] Y,U and V are coordinates in color space, much like RGB, HLS, etc. Y corresponds to brightness, and U and V together determine hue and saturation.

map pixels; these operations affect only the Y bitmap. A fixed palette of eight colors is provided[22]. Operations such as drawing shapes, or displaying motion or still video, draw video pixels into the Y plane, and also write into the U and V planes.

The resolution of the VideoLogo screen is 512 pixels across by 480 lines. There are two screen buffers which are used primarily for double buffering by the ANIMATE primitive; while not animating, the system makes use of the undisplayed buffer for various purposes, including undo and image decompression.

4.4 Image Decompression

Encoded 9-bit image data is loaded from disk into a VRAM buffer and decoded directly onto the VideoLogo screen. Each plane is individually loaded and decoded. When decoding 16-bit images, the image is first decoded as a single 16-bit plane into an offscreen buffer, then converted into a 9-bit image by bit shifting and masking, and then subsampling and filtering the chrominance.

4.5 Multitasking

DVI is a multitasking environment. This means that many system processes can run simultaneously. DVI processes are called “tasks”.

[22] Up to 128 graphics colors could be made available by choosing from combinations of 8 brightness levels and 16 hues.

```
REPEAT FOREVER
    WAIT FOR EVENTS
    ACT ON EVENTS
    CAUSE EVENTS
END REPEAT
```

Figure 4.1 Typical task structure

Figure 4.1 is a simplified representation; a task does not have to perform all of the operations listed, most do some subset.

Of course, a multitasking system is not really running tasks simultaneously; rather it is "time-slicing" between them, giving each task a small amount of time to run. Tasks are switched on and off when they wait for events; while one task waits, others may run. An internal multitasking scheduler keeps track of which tasks are waiting for which events, and decides which task to activate next based on the type of event and the task's priority. Tasks must be well-behaved; since the scheduler only runs when a task waits for an event, the multitasking nature of the system can be destroyed by a task that never waits.

Tasks are useful in the DVI environment in general, and VideoLogo in particular, because of the high degree of integration required between multiple system resources. While VideoLogo plays video, several tasks are active: The VideoLogo task itself, which interprets VideoLogo commands; the player task which VideoLogo communicates with asynchronously, using events, to control video playback, and several low level system tasks which DVI software uses to control the various processes needed to play video, such as responding to disk and display interrupts, reading in and buffering video

data, and scheduling decoding and display of video frames.

The simplest type of event a task can wait for is a timeout. This means that a task simply "puts itself to sleep" for a period of time and is "awakened" by the multitasking scheduler when the time interval expires. This is how the VideoLogo interpreter task can run while DVI tasks play video.

Normally, when video is not playing, the VideoLogo command interpreter constantly checks to see if the user has struck the stop key[23] by simply inspecting the keyboard input buffer. However, when video is playing, the check requests a timeout of a brief interval (approximately 1/15th of a second) in order that DVI video tasks may execute. The net effect is that both video playback and Logo interpretation appear to occur simultaneously. Not only the command interpreter, but all VideoLogo operations that wait for a specific time interval, such as the WAIT and TONE primitives, or those that wait for user input, such as READLIST and READCHAR and command input, are designed to use task events instead of simple loop constructs.

WAIT and TONE use the aforementioned timeout event; routines that take keyboard input use a system-supplied keyboard event; instead of simply calling a C library routine to read a character or a line, these routines wait for the keyboard event, and then read the input once the event has occurred. In the meantime, video is able to play.

[23] The stop key stops the execution of all executing Logo procedures and returns to the command prompt.

4.6 Video Playback Control

Because normal graphics or video operations, such as drawing a line or rotating a shape take only fractions of a second, it is perfectly reasonable to have the VideoLogo system wait for these operations to complete before returning control. However, playing a video file requires that the system be able to provide asynchronous control of the video.

Achieving this control is accomplished by having a play task act as intermediary between VideoLogo playback controls and the actual calls to the DVI AVSS[24] software. This play task is created when the PLAY command is first given, and remains in existence until the video file has finished. The play task can be envisioned as a VCR, and the logo playback primitives as pressing buttons on it. A primitive such as PAUSE issues an event that "presses the pause button". The play task pauses the video by making a direct call to AVSS, and no further attention is required by the button-presser to ensure that the video has paused. Certain primitives that call more complex and error-prone sequences of operations, such as PLAY, wait for a confirming event to return from the play task before completion.

[24] Audio Video Subsystem

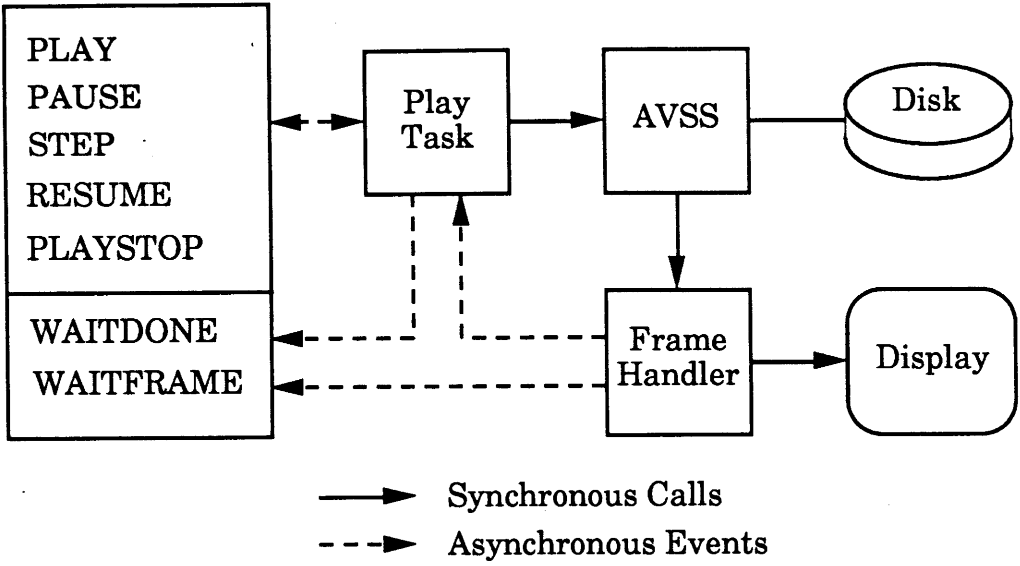


Figure 4.2 Block diagram of the play task communication structure.

4.7 Waiting for Video Frames

VideoLogo's WAITFRAME and WAITDONE primitives require access to the current frame number. AVSS provides a direct call for obtaining this information, but it cannot be used because it would be necessary to enter a loop and constantly check the returned frame number. As we have seen, such looping in DVI's multitasking environment is counter-productive. Instead, DVI provides a set of "hooks" whereby a user-supplied routine may be called from various steps along the image decoding pipeline.

VideoLogo uses one of these hooks to have a frame handling routine be called after each frame has been decompressed. AVSS passes various types of data to this routine, among them the current frame number. The WAITDONE or WAITFRAME primitive sets the requested frame number as

a global variable and then waits for a frame event. Each time a frame is decompressed, the frame handling routine compares the current frame number with the desired one; once the desired frame is reached, the frame handler posts a frame event which is detected by the waiting primitive. In this manner synchronization is achieved with minimum impact on system performance.

4.8 Displaying Frames

Because of the large number of post-processing options VideoLogo provides for video frames (i.e. keying, masking, effects) it is necessary that the decoded frame be intercepted by VideoLogo software before it is displayed. The same frame handling routine used for counting frames is also used to process and display them. Besides passing a frame number each time the hook is called, AVSS also provides the bitmap containing the newly decoded frame. The hook routine modifies this bitmap as necessary by copying various masks onto it, and by applying various effects to it, and then copies the frame bitmap to the correct location on the screen. If no effects or masks are required, the hook routine simply copies the frame onto the screen.

The post processing of a frame occurs in three stages, each stage basically being a copy-with-modify operation, where the output of one stage is the input to the next. First, if any effect is required, it is applied by copying the frame to a temporary bitmap while performing the effect. Subsequently, if some type of keying or masking is in effect, the mask bitmap is copied with transparency on top of the temporary bitmap. Which regions of the mask are transparent depends on the masking and keying settings; see Appendix A.

Finally, the frame is copied without any modification onto the screen.

AVSS is able to maintain a constant playback frame rate as long as the total number of post-processing operations (including the final copy onto the screen) do not take longer than one frame time. The DVI video algorithm preferred for VideoLogo provides approximately 10 frames per second, but the amount of time needed to actually decode a frame is substantially less than that. Only when the operations become complex, such as temporal frame averaging or difference, does the playback rate drop below normal.

Because this hook mechanism increases effective frame decode time by adding at least one, and often several bitmap copy operations, it is undesirable for video algorithms that devote practically the entire frame time to decoding. For example, DVI's PLV[25] algorithm, which runs at 30 frames per second, uses practically all of the 30th of a second for decoding. The amount of time left between the completion of decoding and when the frame must be displayed is insufficient to perform even a single frame bitmap copy. However, because VideoLogo users may conceivably want to play such sequences, the ZOOM primitive is provided whereby the frames are displayed directly by AVSS instead of being copied by the hook routine. This necessarily causes two restrictions: the video must play full screen, and it cannot be post-processed in any manner.

[25] Presentation Level Video

4.9 Custom Microcode

Although a large number of microcode routines are supplied by Intel with the DVI system, microcode had to be written in order to make certain VideoLogo primitives operate more quickly or more memory-efficiently. These include routines which operate on a single bitmap, to convert all pixels into video or graphics or to check a bitmap to find out if it contains pixels of either type; routines that operate on all three bitmap planes simultaneously to copy transparent colors from the Y plane to the U and V planes, and copy-and-modify routines, for performing various effects such as edge detection, frame averaging, and differencing.

Chapter 5

Conclusions

This chapter discusses possible improvements, future directions, and applicability of the ideas explored by VideoLogo.

5.1 Improved Functionality

The types of improvements that can be made to VideoLogo fall into roughly four classes:

DVI system software improvements. VideoLogo does not offer primitives for video fast-forward, rewind, or random access, because DVI does not support these functions. Since the DVI symmetric video algorithms used by VideoLogo are intraframe coded, there is no inherent reason why this restriction could not be lifted in the future. Because of the highly integrated and packaged nature of the current DVI calls for playing video, it is not possible to add rewind, fast forward, etc. to the current system without modifying internal DVI code. Another area where DVI limitations impact VideoLogo's flexibility is in video, audio, and still recording. Because the DVI software routines for performing these functions were not documented, it was

necessary to package them as external utilities.

Speed Improvements. Although the Intel A82750PA drawing processor chip runs at 12 million instructions per second, that speed is not sufficient to enable certain graphics and video operations to take place concurrently while motion video is playing. For example, distorting the motion video frame into a shape outline (as opposed to clipping it, as is done now), chroma- or luma-keying the incoming video frame, and performing most video effects on 30 frame per second playback, all would slow the playback speed far below the rate which it was recorded at. If the next generation DVI chips can perform at roughly double the speed of the current ones, these features will become feasible.

Microcode functionality. Although roughly a dozen custom microcode routines were written for the VideoLogo system, more could be done to expand image-processing functionality, and make VideoLogo easier to use. One such possibility is the area of edge and feature detection and chroma-keying, for instance to make a 'smart' version of CUTOUT that would perform an automatic extraction of the object from the background. Although these operations would probably take too much time to be used in motion video, they would be suitable for primitives, such as CUTOUT, that do not have to run at the fastest possible speed. Other potential applications of new microcode include translucency effects, multiple effects being performed in a single pass, and color mapping of video images.

Logo environment. Several Logo concepts explored in other systems would

lend themselves well to VideoLogo. In particular, MultiLogo [Resnick88], by giving explicit control over multiple tasks with message passing, is a convenient model for dealing with the many things that must be simultaneously handled by an interactive multimedia application, such as animation, control of video playback, and responding to user inputs. A less ambitious, but still useful extension would be support for multiple turtles, such as those in LogoWriter.

5.2 Future Directions

*Range Camera.*Through the use of a range camera, it is possible to capture depth information across a scene. There are two varieties of range cameras, each suited to a particular task. One type uses a laser to scan a static scene; another uses two lenses with different focal depths. These devices provide a unique depth value for each pixel in a scene (after moderate to heavy computation). Once the depth is derived, the image can be observed from different points of view, keyed based on depth, and combined with 3D graphics objects [Bove89, Linhardt89].

Depth chromakey. Much coarser depth information (giving only one of two possible depth values for each pixel) is possible when using a Polaroid-type range finding device combined with chroma keying, using the following method: the subject steps in front of a monochromatic background, and the sensor returns a distance reading which is closer than before. From this one assigns the initial sensor value to all background-colored pixels, and give other pixels (those consisting of the subject) the later, closer value. Although

this technique is very restricted, it has the great advantage of being able to be run in real time; little or no computation is required.

Three dimensional rendering. This capability would allow combination of object models and range-camera objects (perhaps the users themselves), creating a world that could be traveled through by using Logo commands, giving a "turtle's-eye-view". Animation and object modelling could potentially allow the creation of virtual LEGO/Logo machines to exist alongside the objects in the microworld. Conceivably, a miniature camera could be attached to a LEGO/Logo robot, placing video input under Logo control.

Combine separate objects. Using pre-calculated, chroma or depth keying, multiple motion video objects can be combined in a single scene. Objects are stamped onto the screen; the order of stamping determines the appearance of depth. The objects can be clipped out from their backgrounds using chroma keying or the range estimation techniques discussed earlier. A depth value for each object, or for every pixel in each object can be preserved (or assigned), so that 2-D and 3-D clipping and overlap can be performed. Note that these can be combined with generated 2-D and 3-D graphics objects as well.

Add depth information to motion capture. Range cameras and Polaroid sensors will enable creation of 2-1/2 D image sequences[26]. Playback software will be able to display the sequence from a range of possible view angles (by doing simple transforms before displaying pixels), support depth-keying of other objects, and mixing with 3-D graphics objects (see above). Note that 2-1/2 D images are not true 3-D representations; one cannot see 'behind' 2-1/2 D

[26] 2-1/2 D means that a 2-D image has depth information associated with each pixel.

objects, so the range of rotations permitted before noticing holes in the image may be fairly small [Bove89, Linhardt89].

Texture mapping. Computer graphics textures or digitized images may be applied to surfaces of objects in 2-1/2 D or 3-D sequences. This manipulation provides the ability to change the appearance of objects, by, say, making a house look like it's made of wood, brick, or LEGO blocks, simply by assigning a different texture to it. Textures could be tiled to fill larger regions; for instance, a small tile of grass texture could be repeated to cover the 'floor' of a 3-D scene.

3D input manipulation. Depending on the 3D display and capture capabilities offered, it would be a good idea to provide some type of 3D input device, such as a dataglove or joyball. This would enable Logo users to travel through a 3-D video microworld without programming the travel path, much as a joystick or mouse is currently used [Sturman89].

Video input manipulation. A LEGO/Logo robot [Resnick90] could be equipped with a miniature video camera. The intention is not to build a Logo-controllable vision system, but rather to enable Logo users to have a spatially controllable video source. For instance, a child could construct a robot that could turn the camera upside down, or bring it into an area where a normal video camera would not fit, i.e., under things, or into narrow spaces.

5.3 Results

As expected, the most challenging area of building the VideoLogo system was not the technical programming, but rather the design of primitives. Choosing a given primitive involved deciding on its name, how it should interact with other primitives, both at the Logo language level and through underlying effects on the system, and what types of inputs it would take and outputs it would give. All of these considerations had to made in light of other primitives in order to maintain a logical grouping and fairly uniform spread of functionality.

This difficulty is not a surprising one, given the wide range of possible video manipulations available with a system such as DVI. It was tempting to simply assign primitives to practically every possible manipulation, but this would have defeated the purpose of the system -- that it supply the most efficient set of building blocks for users to combine in order to create their own new primitives.

Testing VideoLogo with children (see Appendix C) has proven that video is an extremely compelling data type to add to Logo. Because video is so easy to identify with and understand, it has an immediacy and depth of communication that cannot be matched by text or graphics.

Appendix A

VideoLogo User's Manual

Screens

VideoLogo uses two screens. The command screen is for entering and editing VideoLogo commands. The graphics screen is where the turtle is, and where graphics and video appear.

Starting VideoLogo

Type **LOGO** and press Enter. After a few moments, both screens will clear, and then you'll see

Welcome to VideoLogo!
?

on the text screen, and a welcome display on the graphics screen. The system is now ready to be used.

Entering Commands

VideoLogo shows a **?** (question mark) on the text screen when it's ready to accept the next command. If you are defining a procedure using **TO**, a **>** character will be displayed instead, prompting you to type the next line of the procedure. The prompt will change back to **?** when you finish defining the procedure with **END**.

Special Keys

VideoLogo interprets several keys in a special way:

F3	Repeats the last line whenever the **?** or **>** prompts are showing.
Esc	Stops the currently-running procedure or video.
Ctrl-F	Go to the editor (the **EDIT** command will also do this).

The Editor

You can use the editor to enter and edit procedure definitions. You may move between the editor and VideoLogo as often as you like. All the currently defined procedures will appear in the editor.

The VideoLogo editor uses the following keys:

↑ ↓ ← →	Move the cursor one line or one character.
Ctrl-↑	Move the cursor to the beginning of the edit text.
Ctrl-↓	Move the cursor to the end of the edit text.
Ctrl-←	Move the cursor to the beginning of the current line.
Ctrl-→	Move the cursor to the end of the current line.
PgUp	Move to the previous page of edit text.
PgDn	Move to the next page of edit text.
F1	Select. Sets the cursor position as the selection point.
F2	Cut. Saves and removes text between selection point and cursor.
F3	Copy. Saves text between the selection point and the cursor.
F4	Paste. Inserts saved text at the cursor.
Ctrl-F	Return to VideoLogo.

If there is an incorrect procedure definition (e.g. missing END, empty procedure, TO in a procedure, or a bad command) in the editor when you try to return to VideoLogo, you will be told which procedure had the error, and VideoLogo will return to the editor to let you fix the problem.

Saving and Loading

To save all procedures into a disk file, use VideoLogo's **SAVE** command (exit the editor first). **SAVE** requires a file name; for instance **SAVE "MYFILE** .

To load procedures into VideoLogo, use **LOAD** and give it a file name; for example, **LOAD "MYFILE** . If the file that you **LOAD** contains a procedure with the same name as one that is already defined, the new definition will replace the old one.

The file that you **SAVE** or **LOAD** can have more than one procedure in it. If you are working on a large project that requires procedures from several files, it's a good idea to load them all, and then save them once into a single file. Then whenever you want to work on the project, just load that file. To add some more procedure files to your project file, load the project, then the procedures, and then save the project again.

Logo Language

Many VideoLogo commands are the same as the ones in other versions of Logo. The following Logo commands will work in VideoLogo:

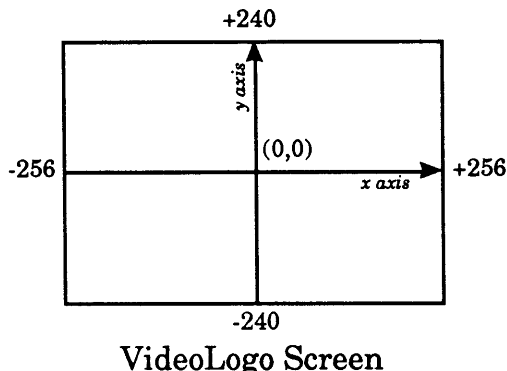
back/bk	bg	break	butfirst
butlast	bye	cc	cg
clean	clearname(s)	color	cs
end	false	fill	first
forward/fd	fput	heading	home
ht	if	ifelse	label
last	left/lt	list	lput
make	member?	name	output/op
pc	pd	pe	penstate
po	poall	pons	pops
pos	pr	print	printnames
pu	px	random	rerandom
readchar	readlist	recycle	repeat
rg	right/rt	run	setbg
setc	seth	setpenstate	setpos
setsh	setx	sety	shapes
show	shownames	st	stamp
stop	stopall	text	thing
to	tone	true	type
wait	xcor	ycor	

Commands specific to VideoLogo, and those that work differently, are described in the next section, **VideoLogo Commands**.

VideoLogo Commands

Turtle

The screen coordinate range is 512 turtlesteps wide and 480 turtlesteps high. The coordinate (0,0) is in the center of the screen.



VideoLogo Screen

Figure A.1

The initial turtle position is (0,0), and it's initial heading is zero degrees, which means it is pointing straight up along the Y axis.

The turtle has a pen, which has a certain color and can be up or down. The pen can also erase with a background color.

FORWARD *steps*
FD *steps*
BACK *steps*
BK *steps*

Move the turtle ahead or back a given number of turtlesteps. The size of a turtlestep is the width of the smallest dot that can be shown on the screen. If the pen is down, it will leave a line between the turtle's old and new position. If the turtle moves off the edge of the screen, it will wrap around to the other side.

LEFT *degrees*
LT *degrees*
RIGHT *degrees*
RT *degrees*

Turn the turtle left or right a given number of degrees. Right is clockwise, left counter-clockwise.

HEADING
SETH *degrees*

HEADING reports the turtle's heading in degrees, and SETH sets it to a particular value.

XCOR
YCOR
SETX *coordinate*
SETY *coordinate*

XCOR and YCOR report the turtle's x and y coordinates. SETX and SETY set the x and y coordinates.

HOME

Move the turtle to the center of the screen, coordinate (0,0).

POS

SETPOS *coordinates*

Report and set the turtle position. The position is a list of two numbers; the first is the turtle's x coordinate (XCOR) and the second is the turtle's y coordinate (YCOR).

PD

Put the pen down. The turtle will draw using the pen color the next time it moves. When VideoLogo starts, the pen is down.

PU

Take the pen up. Stops the turtle from drawing.

PE

Pen erase. Makes the turtle erase what was drawn. The background color is used for erasing.

PX

Pen exchange. Draws black where there was white, and vice-versa. When drawing on colors besides black and white, and on video, the drawing color will be unpredictable. PX doesn't work with any of the FILL commands (see below).

ST

Shows the turtle. When VideoLogo starts, the turtle is showing.

COLOR

SETC *color*

Report and set the pen color. When VideoLogo starts, the pen is white. There are a total of 8 available colors:

0	Black	2	Red	4	Blue	6	Blue-Green
1	White	3	Green	5	Yellow	7	Purple

BG

SETBG *color*

Report and set the background color. Available background colors are the same as pen colors. The starting background color is black (0). The background color is used for erasing by PE (pen erase) and CS (clear screen), and it controls which parts of a shape (see **Shapes**) and masks (see **Motion Video**) are transparent.

PENSTATE

SETPENSTATE *list*

Report and set the pen state. PENSTATE outputs a list consisting of the commands that set the pen to its current settings. SETPENST will input this list and change the pen's settings appropriately.

For example, suppose a procedure called MONSTER changed the pen state. To preserve the pen state and restore it afterwards, use

```
MAKE "PS PENSTATE
MONSTER
SETPENST :PS
```

Screen

These commands affect the entire screen.

CG

Clear graphics. Erases the screen to background color.

CC

Clear command. Erases the text screen.

CS

CLEAN

Clear screen. Erases the screen to background color and moves the turtle to home.

RG

Reset graphics. Stops video and resets everything to starting values.

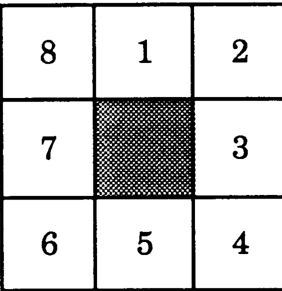
SCRUNCH

SETSCRUNCH *scrunch*

Report and set the aspect ratio of the screen. Use 1 if the pixels on your display are square. Use a value greater than 1 to make objects taller, less than 1 to make them shorter. Setting SCRUNCH won't change what's already on the screen. SCRUNCH starts at 1.

FILL

Fills all the neighboring pixels of the same color as under the turtle to the pen color. A neighboring pixel is one of the eight pixels surrounding a particular pixel:



Neighboring Pixels

Figure A.2

FILLTO *color*

Fills all neighboring pixels with the pen color stopping at a boundary of a given color. You should use FILLTO instead of FILL when drawing on top of video.

FILLIN *list of commands*

Draws a filled-in shape. The commands executed in the list will be used to create the outline of the shape, and everything inside the outline will be filled with the current pen color. For example, FILLIN [REPEAT 4 [FD 20 RT 90]] will draw a filled-in square. If the outline has more than 64 line segments, or it wraps around the edge of the screen, it won't be filled in.

LOADPIC *file*

Loads an image file onto the screen. If the image is smaller than the whole screen, it will be centered at the turtle position. If the pen is up, nothing will be displayed. For example, LOADPIC "BIRDS would show the image file BIRDS.

SAVEPIC *file*

Saves the whole screen to an image file. You can display the file later with LOADPIC.

Text and Fonts

VideoLogo lets you display text on the graphics screen with your choice of font, size, and style. Each font comes in a variety of sizes; not all fonts are available in the same sizes.

FONT
SETFONT *font*

Report and set the current font. The starting font is SANS; other available fonts are ULTRA, ULTRAB, SOFTV, SLIMV, SOFTF and SLIMF. For example, SETFONT "ULTRA changes the font to ULTRA.

FONTSIZE
SETFONTSIZE *size*

Report and set the current font size; 18 is the starting size. The SANS and ULTRA fonts have sizes 12, 14, 18 and 24; the SLIM and SOFT fonts only come in size 8. If the size you want isn't available, VideoLogo will find the closest size and use that instead. For example, if the font is ULTRA and you SETFONTSIZE 30, VideoLogo will use size 24.

FONTSTYLE
SETFONTSTYLE *style*

Report and set the current font style. There are 3 styles:

0 Transparent background (VideoLogo starts with this style)

1 Solid background

2 Shadow

LABEL *input*

Print to the graphics screen starting at the turtle position, using the current font, size, and style. LABEL takes the same types of inputs as PRINT. If text goes off the right edge of the screen, it will wrap around to the left side and move one line down. LABEL leaves the turtle at the end of the last character it draws.

Shapes

VideoLogo has turtle shapes that can be rotated, scaled, and moved. Shapes may be loaded and saved on disk. A shape can be much larger than the turtle, up to a quarter of the screen size.

Background-colored areas of the shape are treated as transparent when the shape is drawn; if you change the background color, and parts of the shape are background colored, they will become transparent. The appearance of the shape also changes if you apply a video effect to it.

There are 10 shapes which always start out looking like a turtle. You can select any shape and change it (except the first one which is always the turtle shape) by erasing it, loading a shape file into it, or by creating a new shape. Unlike shapes in other Logos, VideoLogo shapes are not kept together in one group. Shapes must be loaded and saved individually, and any shape file can be loaded into any shape.

SHAPE
SETSH *shape*

Report and set the current shape. All shapes start out looking like a turtle. VideoLogo has 10 shapes numbered 0-9; shape 0 is reserved for the turtle shape and can't be changed.

SHAPEUSED?

Outputs true if the current shape is used, false if it isn't. Shape 0, the turtle, is always in use.

CUTOUT *list of commands*

Create a shape. The commands in the list will be used to create the outline of the shape, and everything on the screen inside the outline will become part of the current shape; for example, CUTOUT [REPEAT 4 [FD 20 RT 90]] will create a square shape. The shape outline can't have more than 64 line segments.

LOADSH *file*

Load a shape from disk into the current shape.

SAVESH *file*

Save the current shape to disk. For example, SAVESH "FACE will save the current shape into a file called FACE.

ERASESH

Erase the current shape, changing it back into a turtle.

SHAPES

Shows all the shapes that are currently in use. Shapes that are not in use appear as turtles.

SCALE

SETSCALE *scale*

Report and set the scaling applied to the current shape. If scale is greater than one, the shape will get larger; if less than one, it will get smaller. If the shape gets too big to fit into 1/4 of the screen area, you'll get an error. Scaling doesn't affect shape 0, the turtle.

For example, SETSCALE 1.5 will make the shape one and a half times larger.

STAMP

Puts an image of the current shape (at its current size and orientation) on the screen. STAMP doesn't do anything if the pen is up.

ANIMATE *list of commands*

Animate mode is provided to give smoother motion when moving or changing shapes. Only commands that move the turtle (FD, BK, RT, LT, HOME, SETX, SETY, SETPOS) and STAMP will work while animating. For example, ANIMATE [REPEAT 10 [FD 40 RT 10]] will move the current shape along a curve without flickering.

Motion Video

Video play occurs in the background; once video has started playing other VideoLogo commands can be executed. While video is playing, the turtle can be moved, but it will be hidden and it won't draw. Commands that change the screen, such as STAMP or LOADPIC, must be done when video is paused. A set of VCR-like commands can be used to control playback.

When video is being played, an image called a mask can be overlayed on the video, and the mask can be "keyed" to control which parts of the mask the video shows through. There are several ways of making masks, and several types of keying.

PLAY *file*

Plays a video file. The video will be centered at the turtle position. The maximum size of the video is 1/4 of screen. If a mask or key have been defined before running PLAY, they will be used; if keying has been set, but there's no mask, a temporary mask taken from the area where the video will appear is used, just as if you had used the SNAPMASK command (see below). If the pen is up, video won't be shown.

PLAY returns once the first frame of video has appeared on the screen (this may take several seconds). If the stop key is pressed while video is playing, it will stop.

DONTKEEP

KEEP

KEEP?

These commands control whether the last video frame stays on the screen when video stops or pauses.

DONTKEEP tells VideoLogo not to keep the last frame of the video on the screen; this is the setting used when VideoLogo starts.

KEEP keeps the last frame of the video on the screen, KEEP? reports true if the video will stay on screen, false if not.

PLAYSTOP

Stops playing video, and restores the area of the screen under the video if DONTKEEP was set.

PAUSE

Pauses the currently playing video. Unless KEEP is set, the video will disappear until it's resumed.

STEP

Advances to the next video frame, then still-frames. Will pause video if it was playing.

RESUME

Resumes full-speed playback after PAUSE or STEP. If the turtle moved while video was paused, video will resume centered at the turtle's new position.

PLAYSTATE

Reports RESUME if video is playing, PAUSE if it is paused, or PLAYSTOP if video is not playing.

FRAME
LASTFRAME

FRAME reports the currently-displayed frame number, or 0 if no video is playing. LASTFRAME reports the value of the last frame in the currently playing video, or 0 if no video is playing.

WAITDONE
WAITFRAME *frame*

WAITDONE waits until all the frames in the currently playing video have been displayed.

WAITFRAME waits until a specified frame in the currently playing video has been displayed.If the video stops before the last frame is reached, WAITDONE and WAITFRAME will stop waiting.

SHOWFRAME
HIDEFRAME

Turns on and off the display of the current frame number in the upper left corner of the video frame. This is useful for finding points in a sequence suitable for stopping, pausing, etc.

VOLUME
SETVOLUME *volume*

Get and set the audio volume. The starting volume is 100. SETVOLUME 0 will mute the audio. This command will take effect immediately if video is playing.

FILTER
SETFILTER *frequency*

Get and set the audio filter cutoff frequency; frequencies above *frequency* Hz are filtered out. The starting value is 16666. This command will take effect immediately if video is playing.

SYNC
SETSYNC *frames*

Get and set the number of frames of video to wait before starting audio; the starting value is 0. If *frames* is a negative number, audio will start before video. This command will take effect at the next PLAY.

CROP
SETCROP *[top bottom left right]*

Report and set the number of pixels to discard on each edge of the video. These commands take and output lists. Video that you've recorded yourself usually has 'dirty' edges, so VideoLogo starts with CROP set to [4 8 8 8]. Use SETCROP [0 0 0 0] to show the entire video frame, dirty edges and all.

DONTMASK
SNAPMASK
SHAPEMASK
MASK?

These commands determine what type of mask to use for playing video, if any. The contents of the mask determine how the video will be keyed. DONTMASK tells VideoLogo not to use a mask with PLAY; this is the setting used when VideoLogo starts.

SNAPMASK uses the area of the screen centered around the turtle as a video mask; the mask size will be the same size as a video frame.

SHAPEMASK tells VideoLogo to use the current shape as a mask. When the mask is a shape, video will only show inside the boundaries of the shape, and the shape's outline, size and orientation at the time of the PLAY command will be used to determine the mask's shape.

MASK? outputs true if a mask has been defined, false if not.

DONTKEY
KEYBACK
KEYOVER
KEYSTATE

These commands determine how the current mask will be keyed over video.

DONTKEY makes video show through all areas of the mask; this is the key state that VideoLogo starts with.

KEYBACK makes video show through only background-colored areas of the mask.

KEYOVER makes video show through background colored *and* video areas of the mask.

KEYSTATE reports DONTKEY, KEYBACK, or KEYOVER depending on the current key state.

ZOOM
DONTZOOM

Use ZOOM to make VideoLogo play video full-screen instead of 1/4 screen. Use DONTZOOM to change back to the usual 1/4 screen size. While video is zoomed, the rest of the screen will be covered up by the video. The screen is restored when video stops playing. ZOOM and DONTZOOM take effect the next time a PLAY command is used.

Video Effects

Video effects can be applied to playing video, shapes, or the screen. Some video effects change the image much like picture controls on a TV set; others pick out particular details, such as edges, and still others move parts of the image around, like mosaic.

Each kind of object that can have a video effect (playing video, shapes, or the screen) has a set of commands to report and set its effect. These commands need the name of the effect and (in most cases) a number that is the setting for the effect. For example, if the effect is brightness, the setting controls how much the brightness should change.

Some objects cannot use certain effects because it wouldn't make any sense for them to do so; for instance, the DIFF effect (see below) only works with playing video, because it shows the differences between successive video frames .

Inputs to Effect Setting Commands

For any kind of object, the commands for setting the effect input the same thing: the name of the effect and an optional effect setting. If the effect doesn't need a setting, and you give one, it will be ignored. If the effect does need a setting, and you don't give one, it will choose a setting for you.

If you give both a name and a setting, they must be in a list with the name first, e.g. [BRIGHTNESS 50]. If you just give a name, it must be in a list by itself, e.g. [FLIP]. In the descriptions below, the inputs to these commands are shown as [*effect (setting)*].You don't need to type the parentheses around the setting, they are shown just to remind you that the setting is optional.

Effect Setting and Reporting Commands

Here are the commands that set and report the effect for each type of object (shapes, playing video, and the screen):

SHFX
SETSHFX *[effect (setting)]*

Report and set the current shape effect. If the shape effect is changed while a shape other than the turtle is being displayed, the effect will be applied immediately.

PLAYFX
SETPLAYFX *[effect (setting)]*

Report and set the current video playing effect. If the video playing effect is changed while video is playing, the effect will be applied immediately. Some effects can cause playback to slow down. This may cause the sound to break up or get out of step with the video.

SCRFX
SETSCRFX *[effect (setting)]*

Report and set the current screen effect. This will change the appearance of the screen immediately.

Effect Names

Here are the names of the 13 available effects and what they do:

...these effects don't need settings...

NONE	Don't do anything.
FLIP	Flips the picture vertically.
FLOP	Flops the picture left-to-right.
NOCOLOR	Remove the color from the picture.
GRAPHICS	Change video pixels to random colors.

...these effects need settings...

BRIGHTNESS	Adjust the brightness of the picture.
COLOR	Adjust the amount of color in the picture.
CONTRAST	Adjust the contrast of the picture.
EDGES	Show edges in the picture.
BLOCKY	Make the picture look blocky.
TINT	Adjust the tint of the picture.

...these effects are for playing video, and don't need settings...

AVERAGE	Averages the last two video frames together.
DIFFERENCE	Difference between last two video frames.

Effect Settings

The BRIGHTNESS, COLOR, CONTRAST, and TINT settings can range from –255 to 255. If the setting is greater than 0, it means 'more' (e.g. brighter, more colorful, etc.). If it is less than 0 it means 'less' (e.g. darker, less colorful, etc.), and if it is exactly 0 the effect won't change anything.

The BLOCKY setting controls the size of the blocks in turtlesteps. It can range from 1 to 255; when it is 1 there will be no change.

The EDGES setting controls how sharp an edge must be for it to appear. It can range from 0 to 255; settings of 1 to 10 are best.

Effect Examples

SETSHFX [CONTRAST 30]	Increase shape contrast.
SETPLAYFX [AVERAGE]	Average playing video frames.
SETSCRFX [BLOCKY 50]	Make screen very blocky.
SETSHFX [NONE]	Change shape back to normal.
SHFX	Report shape effect name & setting.

Miscellaneous Commands

CLEARALL

Erases all variables and procedures.

CLEARNAMES

CLEARNAME *name*

CLEARNAME *[name name...]*

Erases all, one, or some variables. For example, CLEARNAME "SIZE erases the variable called SIZE; CLEARNAME [SIZE SPEED] erases the variables SIZE and SPEED.

CLEARPROCS

CLEARPROC *name*

CLEARPROC *[name name...]*

Erases all, one, or some procedures. For example, CLEARPROC "HOUSE erases the procedure called HOUSE; CLEARPROC [HOUSE CAR] erases the procedures HOUSE and CAR.

EDIT

Runs the editor. Ctrl-F will also run the editor -- even when the command prompt isn't showing, or when video is playing. See **Using VideoLogo** for more about editing.

LOAD *name*

Reads a file. See **Using VideoLogo** for more about LOAD.

MOUSEPOS

MOUSEPOS reports the mouse coordinate as a list of two numbers.

MOUSEX

MOUSEY

MOUSEX and MOUSEY return the mouse x or y coordinate. The coordinate range of the mouse is the same as the turtle's.

MOUSEBUTTON?

MOUSEBUTTON? reports true if either mouse button is pressed down, false if not.

For example, a procedure to move the turtle with the mouse until a mouse button is clicked could be defined as:

```
TO MOVE
IF MOUSEBUTTON? [STOP]
SETPOS MOUSEPOS
MOVE
END
```

MOVEMOUSE

MM

Attach the turtle to the mouse; the turtle will follow the mouse until you press both mouse buttons at once. If you press one mouse button down and the shape is a turtle, the pen will draw. If the shape isn't a turtle, the shape will be used like a paintbrush.

MOVEMOUSE behaves differently when you are using it to create an outline with FILLIN or CUTOUT (i.e. CUTOUT [MM]). Each time you click the mouse button, a line is drawn from the last place you clicked.When you click twice in the same place, the outline will be closed and MOVEMOUSE will stop.

The pen state will be restored when MOVEMOUSE stops.

PRINTNAMES

Prints out the names and values of all variables. For example, if the variable :NAME contains the value "TOM and the variable :ANIMALS contains the value [HORSE ZEBRA RABBIT], SHOWNAMES will print out

:ANIMALS is [HORSE ZEBRA RABBIT]

:NAME is "TOM

PRINTPROC *procedure*

Prints out the procedure definition for a given procedure.

PRINTPROCS

Prints out the names of all procedures.

SAVE *name*

Saves the edit buffer to a file. See **Using VideoLogo** for more about SAVE.

SHOWNAMES

See PRINTNAMES.

SHOWPROC *procedure*

See PRINTPROC.

SHOWPROCS

See PRINTPROCS.

RANDOM *range*

Outputs a random integer in the range 0 to n-1.

RERANDOM

Restart the random number generator. Each time VideoLogo starts, if you give the same sequence of inputs to RANDOM, it will give the same outputs. RERANDOM makes RANDOM behave as if VideoLogo was started over again.

TONE *frequency duration*

Play a tone of a certain frequency for a specific amount of time. The *duration* units are about 20 per second, e.g. to play a tone for two seconds, use a duration of 40. VideoLogo waits until the tone is finished before going on to the next command. The tone can be stopped with the stop key. **Note:** you cannot change the volume of TONE with SETVOLUME, SETVOLUME only works with PLAY.

WAIT *time*

Wait for a certain amount of time without doing anything. The *time* units are about 20 per second, e.g. to wait half a second use WAIT 10.

Image and Video Utilities

The current version of VideoLogo does not include commands for capturing still or motion video or video file housekeeping, but these things can be done outside of VideoLogo using special utility programs.

If you are in VideoLogo, use the BYE command to exit before using these programs. When you are done using them, type LOGO to go back to VideoLogo.

Note: Unlike VideoLogo commands, these commands should not have a double-quote mark placed before a file name; e.g. use SNAP FLOWER instead of SNAP "FLOWER.

Video and Audio Input

A video and/or an audio source, such as a camera, VCR, or microphone must be attached to the system's inputs for these commands to work. The display will show live video before capturing. The descriptions below mention which keys to press for certain functions, but this information also appears on the text screen.

TAKEPIC *file*

Take a still video picture. Watch live video, and press any key to take a picture of it. Once the picture is taken, it is processed into VideoLogo's image format, and then displayed. The picture can be displayed in VideoLogo using LOADPIC, and you can shrink it using SHRINKPIC (see below).

RECORD *file seconds*

Record video with optional sound for a given number of seconds. The display will show live video and play live sound. To begin recording video and sound, press C. To record video without sound, press V. When the recording is finished, press the space bar, then press P to play it, Q to exit, or C or V to re-record. **NOTE:** be careful not to press C or V while recording; it will cause the recording to start over again when it finishes.

The recorded file can be played back in VideoLogo using PLAY.

LISTEN *file*

Record sound only. Press the space bar to start recording; press it again to quit.

The recorded file can be played back in VideoLogo using PLAY.

Image Files

PICTURES

Select pictures from the CD-ROM image database. When you start the PICTURES utility, there will be a delay while the system is restarted to run the CD-ROM software. When the Photobase screen appears, press Esc to begin. Refer to the screen for information on how to access images. VideoLogo's LOADPIC command and the COPYPIC utility (see below) can directly access the images using their photo ID number. Whenever you see a picture you like, write down the ID number so you can use it later.

COPYPIC *photo-id file*

Use COPYPIC to copy images from the CD-ROM to the hard disk so that they can be accessed more quickly (and to give them names that are easier to remember!). *photo-id* is the photo ID number of the image on the CD-ROM, and *file* is the name to give to the copy.

For example, COPYPIC U230342 HOUSE will make a copy of the CD-ROM image U320342 and call it HOUSE. The image HOUSE can be shown using VideoLogo's LOADPIC command.

SHRINKPIC *oldfile newfile*

Use SHRINKPIC to make a smaller copy of an image file. The size of *newfile* will be half the width and half the height of *oldfile.*

CONVPIC *file*

Use CONVPIC to convert images from other image formats to VideoLogo format.

CONVPIC is useful to change screen images created using VideoLogo's SAVEPIC command so that they load much faster when loaded with the LOADPIC command. **CAUTION:** if there are any turtle graphics colors in the image, CONVPIC will change them to black and white!

Appendix B

Digital Video Interactive System Overview

DVI Technology by Intel Corp. [Intel90], is one of the first commercially available all-digital multimedia system which can deliver motion video, audio, and graphics from a single, integrated PC-based unit, and thus is a good indicator of the types of capabilities that should be seen on more and more PCs and workstations in the future.

In addition to being an ideal platform upon which to implement multimedia applications, DVI also has the virtue of running on the IBM PC/AT and IBM PS/2, systems that can already run various implementations of Logo.

DVI's power lies in it's high-performance (12 MIPS) programmable pixel engine, unique frame buffer design, and a large body of software.

Several versions of DVI motion video are available, each involving a different compression time/frame rate/image quality tradeoff. All the motion video algorithms are designed to work at a data rate of 1.5 megabits per second, the data rate of a CD-ROM. The highest quality approach, called Presentation Level Video (PLV) achieves a 30 frame per second image with quality comparable to a VCR, but requires compression be performed in

nowhere near real time, on special hardware. An intermediate version, called Real Time Video (RTV) gives nearly as good quality, but with half the frame rate, and it is symmetric -- it can be compressed in real time on the DVI system itself. Intel lately has developed yet another symmetric RTV algorithm that gives lower quality, but still useable 30 fps video.

The subjective image quality of any of these algorithms depends a great deal upon the resolution of the image. The more pixels on the entire screen (and consequently, the smaller the video window on the screen), the better the image looks. Hardly any of them look very good when they are shown full screen, but practically all are acceptable when played in a 1/4 screen window.

DVI hardware is well suited for implementing video/graphics applications. The Intel A82750PA pixel engine chip has a dual-bus pipelined architecture, containing specialized hardware for transforming and filtering bitmaps and for statistical decoding of image data. The frame buffer display is controlled by the Intel A82750DA display chip, and is YVU based, instead of RGB, simplifying the implementation of video and image processing algorithms. It can handle a variety of resolutions and pixel modes, including VGA, 16- and 32- bit per pixel, and also can directly display bitmaps where chroma information is present at 1/4 the resolution of luma (i.e., for every 4x4 block of Y pixels there's one U and one V pixel), taking advantage of the human visual system's low spatial chrominance resolution. The frame buffer is also capable of digitizing incoming NTSC video in a single frame time. The audio processor is based on a TMS32010 DSP chip and does ADPCM digitization and playback.

DVI software can be divided into two parts: code that runs on the host machine and code that runs on the A82750PA. Host software controls I/O,

multitasking, and device synchronization, ensuring that data arrives from devices such as hard disks or CD-ROMs, and gets to the pixel engine and audio processor at the right time, while simultaneously handling requests from the user. This code also controls the pixel engine by loading microcode into video memory and maintaining a display list of pending operations and their parameters.

A82750PA code is very different. It consists of a library of small microcode routines, each designed to perform a specific video, graphics, or image processing function. The A82750PA chip contains a memory large enough to hold a single routine (or a major part of one), and in the course of normal operations, different microcode routines are constantly being loaded into chip memory -- one routine can be loaded in the time it takes a single scan line to be traced out on the CRT. There are microcode routines to compress and decompress single frames of video, to draw and copy shapes, and to scale, rotate, and transform images. Microcode routines can be written using an assembler-type language, and can be accessed through a host-level interface.

Intel supplies a set of libraries that package the above capabilities into high level calls for handling multitasking, playing audio and video, and performing graphics calls. The typical DVI programmer need never directly deal with microcode or the mechanics involved in reading data from disk, putting it into video memory, and having it reconstructed into a frame from a video sequence. Intel also supplies simple tools for performing data gathering and editing tasks, primarily for digitizing and compressing audio and video.

Appendix C

Children's Experiences with VideoLogo

Two test sessions have been conducted with grade school children. Because there have not been enough sessions to formulate any reliable conclusions, this section will be presented anecdotally; when more children have used the system it may become practical to conduct a real study of their responses.

The first session was three hours long with two fifth grade girls who had previous experience with LogoWriter; the second session was several days later, lasted about four hours and was conducted with the same two girls, and two younger children, a boy and a girl, who had very little Logo experience.

In the first session, the girls, Alexandria and Roberta, initially were timid about trying out VideoLogo video manipulations, and were embarrassed by the idea of recording themselves! They perused the CD-ROM, but did not find any images they liked from among the subjects they had selected -- abstract art, babies, and couples (while looking at the CD-ROM images, Alexandria mentioned how "real" the pictures looked).

After a suggestion from an experimenter that they could try cutting out parts of one picture and stamp them onto another, they digitized still images

of all three experimenters, and used the CUTOUT, MOVEMOUSE and associated shape primitives to create several video shapes (noses, eyes, beards) and proceeded to rearrange the faces they had digitized. They really enjoyed this activity.

It was evident that they girls were easily picking up on the primitives; by the end of the three hours they had complete grasp of all the shape primitives, including scaling, selection, loading and saving of shapes and were not asking the experimenter for any hints.

Near the end of the first session an experimenter asked the girls to create a procedure using the VideoLogo editor, so they wrote a procedural version of what they had been doing before by typing commands directly and by using the mouse. They had no difficulty adapting to VideoLogo's editing environment, and their procedure, to cycle through two still images, select, scale, rotate, and label a shape, was completed successfully.

The second session was conducted with the same two girls, and two younger children, Robert and Sequoia. During this session, the children spent most of their time planning and recording rap songs and talk show skits, or just playing with the camera and watching the monitor.

The next most popular activity was again cutting and pasting parts of faces. The most interesting child during this session was Robert. Although it was evident that he had not had much experience with Logo, he was very enthusiastic about the system (he hardly ever left it) and spent most of his time using the MOVEMOUSE primitive to draw lines and stamp video shapes onto the screen. He also used the SETSCALE primitive with many different fractions to see how different fractions would affect the size of the shape.

The only procedure work done during this session was an improvement to Alexandria's and Roberta's first session procedure; they added voiceover to the text that appeared on the screen.

Several interesting findings came to light from observing these two sessions. As a vehicle for introducing manipulable video, VideoLogo's programmability was not an important issue; the children became absorbed in using the video camera to record and play movies, and using the mouse to paint and make shapes. Programmability did not become important until the children had become familiar enough with primitives they had used in direct command mode. Another interesting finding was their apparent lack of interest in using the images on the CD-ROM; this was probably due to the slow access time and unpredictable image quality -- they had little difficulty in learning to use the database search software supplied with the disk. In general, one would also expect that creating one's own images, even of inferior quality, would be far more engaging than choosing them from a disk. If the access time were increased, the children probably would use the disk as a source of not only screens, but as a source of shapes as well.

Figure C.1 Still from Alexandria and Roberta's procedure, showing a video shape being rotated, scaled, and overlayed with text.

Bibliography

[Adelson84] F. H. Adelson, C. H. Anderson, J. R. Bergen, P. J. Burt and J. M. Ogden, "Pyramid Methods in Image Processing", *RCA Engineer.* 29(6): 33-41, 1984

[Backer88] D. S. Backer, "Structures and Interactivity of Media: A Prototype for the Electronic Book", PhD. dissertation, Massachusetts Institute of Technology, 1988

[Bender88] W. Bender and P. Chesnais, "Network Plus", paper presented at SPSE Electronic Imaging Devices and Systems Symposium, 1988

[Benenson87] A. Benenson and S. Morris, "Digital Video Interactive Technology in Education and Training", Society for Applied Learning Technology, Ninth Annual Interactive Videodisc in Education and Training Conference Proceedings, 1987

[Bove89] V. M. Bove Jr., "Synthetic Movies Derived from Multi-Dimensional Image Sensors", PhD. dissertation, Massachusetts Institute of Technology, 1989

[Brøndmo89] H. P. Brøndmo and G. Davenport. "Creating and Viewing the Elastic Charles - a Hypermedia Journal", Hypertext II Conference Proceedings, 1989

[Butera88] W. J. Butera, "Multiscale Coding of Images", S. M. Thesis, Massachusetts Institute of Technology, 1988

[Dickinson90] S. Dickinson and M. Schaffer. "The Visual Communication Workstation: Preliminary Experiments with Children's Integration of Logo, Video Images, and Telecommunications", paper presented by Massachusetts Institute of Technology Epistomology and Learning Group to American Educational Research Association, 1990

[Farber90] N. G. Farber, "Through the Camera's Lens: Video as a Research Tool", paper presented by Massachusetts Institute of Technology Epistomology and Learning Group to American Educational Research Association, 1990

[Harel88] I. R. Harel, "Software Design for Learning: Children's Construction of Meaning for Fractions and Logo Programming", PhD. dissertation, Massachusetts Institute of Technology, 1988

[Harel90a] I. R. Harel and S. Papert. "Instructionist Products or Constructionist Tools? On the Role of Technology-Based Multimedia in Children's Learning", Unpublished paper, Massachusetts Institute of Technology, 1990

[Harel90b] I. R. Harel and S. Papert. "Software Design as a Learning Environment", *Interactive Learning Environments.* 1(1). Ablex, 1990

[Intel90] Intel Corporation, DVI Technology: The Multimedia Solution, Publicity / Sales brochure, Princeton New Jersey, 1990.

[King88] King and Nasrabadi. "Image Coding Using Vector Quantization: A Review", *IEEE Transactions on Communications.* 6(36), 1988

[Linhardt89] P. M. Linhardt, "Integration of Range Images from Multiple Viewpoints into a Particle Database", S. M. Thesis, Massachusetts Institute of Technology, 1989

[Lippman80] A. Lippman, "Movie Maps: An Application of the Optical Videodisc to Computer Graphics Laboratory", ACM SIGGRAPH, 1980

[Luther89] A. C. Luther, Digital Video in the PC Environment, McGraw Hill, New York, 1989

[Mohl81] R. Mohl, "Cognitive Space in the Interactive Movie Map: An Investigation of Spatial Learning in Virtual Environments", PhD. dissertation, Massachusetts Institute of Technology, 1981

[Netravali89] A. Netravali and B. Haskell, Digital Pictures: Representation and Compression, Plenum Press, New York, 1989

[Papert90] S. Papert, Introduction from Constructionist Learning, Collection of papers presented by Massachusetts Institute of Technology Epistomology and Learning Group to American Educational Research Association, 1990

[Papert80] S. Papert, Mindstorms: Children, Computers, and Powerful Ideas, Basic Books, New York, 1980

[Resnick88] M. Resnick, "MultiLogo: A Study of Children and Concurrent Programming", S. M. Thesis, Massachusetts Institute of Technology, 1988.

[Resnick90] M. Resnick and S. Ocko "LEGO/Logo: Learning Through and About Design", paper presented by Massachusetts Institute of Technology Epistomology and Learning Group to American Educational Research Association, 1990

[Romano89] P. J. Romano, "Vector Quantization for Spatiotemporal Sub-band Coding", S. M. Thesis, Massachusetts Institute of Technology, 1989.

[Schreiber86] W. F. Schreiber, Fundamentals of Electronic Imaging Systems, Springer-Verlag, New York, 1986

[Segall90] R. G. Segall, "Learning Constellations: A Multimedia Research Environment for Exploring Children's Theory-Making", paper presented by Massachusetts Institute of Technology Epistomology and Learning Group to American Educational Research Association, 1990

[Sturman89] D. Sturman, D. Zeltzer, S. Pieper, "Hands-on Interaction with Virtual Environments", UIST '89: ACM SIGGRAPH Symposium on User Interface Software and Technology, 1989

[Watlington89] J. A. Watlington, “Synthetic Movies”, S. M. Thesis, Massachusetts Institute of Technology, 1989