The TV Turtle
A Logo Graphics System for Raster Displays

Henry Lieberman
Logo Group
MIT Artificial Intelligence Lab

Until recently, most computer graphics systems have been oriented toward the display of line drawings, continually refreshing the screen from a *display list* of vectors. Developments such as plasma panel displays and rapidly declining memory prices have now made feasible *raster* graphics systems, which instead associate some memory with each point on the screen, and display points according to the contents of the memory. This paper discusses the advantages and limitations of such systems. Raster systems permit operations which are not feasible on vector displays, such as reading directly from the screen as well as writing it, and manipulating two dimensional areas as well as vectors. Conceptual differences between programming for raster and vector systems are illustrated with a description of the author's *TV Turtle,* a graphics system for raster scan video display terminals. This system is imbedded in Logo, a Lisp-like interactive programming language designed for use by kids, and is based on Logo's *turtle geometry* approach to graphics. Logo provides powerful ideas for using graphics which are easy for kids to learn, yet generalize naturally when advanced capabilities such as primitives for animation and color are added to the system.

1. Drawing with turtles

The fundamental concept of the Logo approach to computer graphics is the *turtle.* A turtle is an imaginary creature which lives on the display screen. It has a *position* on the screen and a *heading,* the direction in which the turtle is facing. The turtle also has a *pen* which can be either *up* or *down*. The commands FORWARD and BACK change the turtle's position, moving it along the direction of the heading. If the pen is down when the turtle moves, a line is drawn between the old and new positions. LEFT and RIGHT change the heading. (See Figure 1).

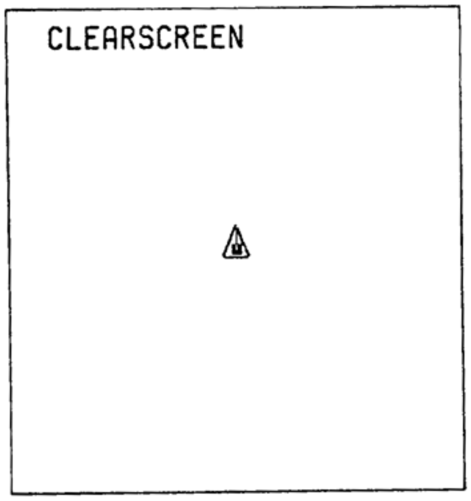


Figure 1a. This is the initial state. The turtle, represented by a triangular cursor, starts out at its home at the center of the screen. Its heading points upward. The square at the center of the turtle indicates its pen is down.

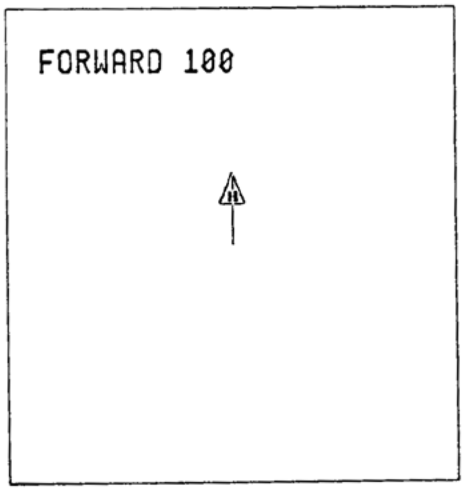


Figure 1b. The turtle moves 100 steps in the direction of the heading. When the turtle moves with the pen down, it draws a line.

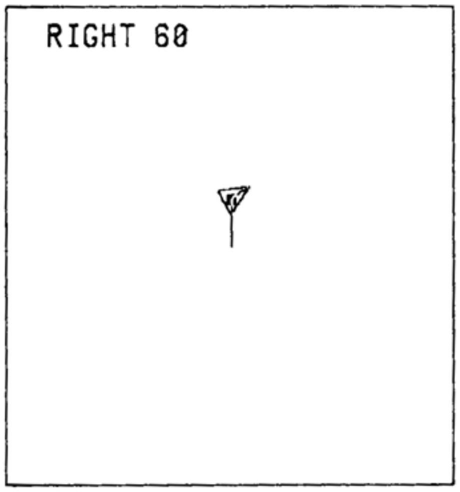


Figure 1c. The heading is turned 60 degrees.

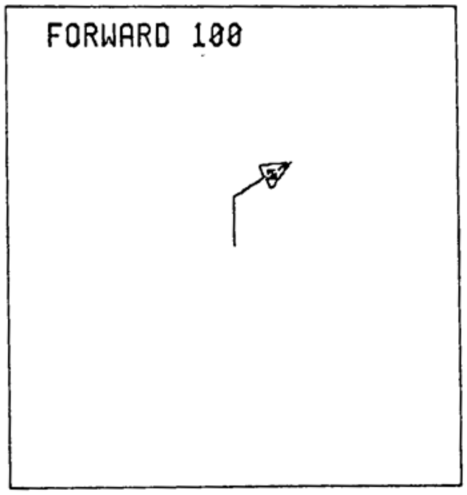


Figure 1d. The turtle now moves in the direction of the new heading.

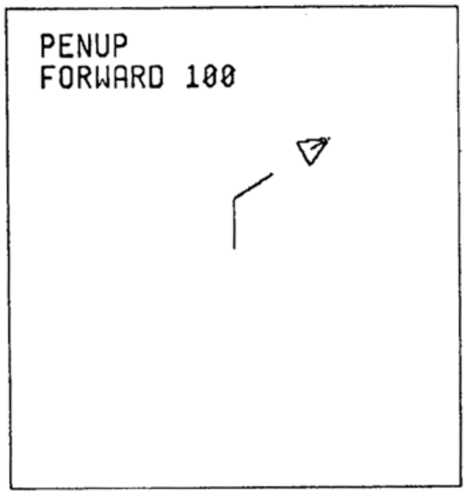
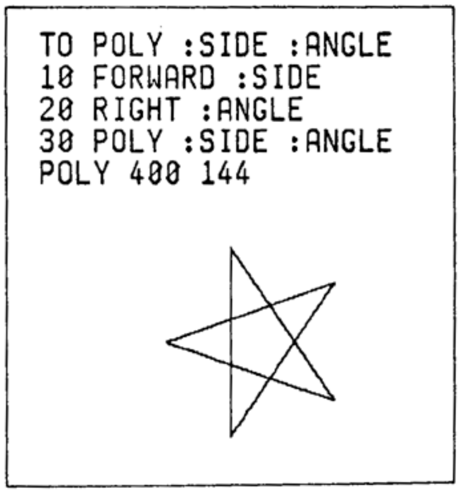


Figure 1e. When the pen is up, the turtle does not draw a line when it is moved.



```
TO POLY :SIDE :ANGLE
10 FORWARD :SIDE
20 RIGHT :ANGLE
30 POLY :SIDE :ANGLE
POLY 400 144
```

Figure 1f. A typical procedure for drawing polygons. Note that the recursion does not terminate, and the procedure must be stopped by hand.

This scheme is generally much more convenient for interactive graphics than the conventional approach using absolute cartesian coordinates, and its mathematical implications lead naturally toward a new view of geometry called *turtle geometry.* The Logo graphics philosophy and turtle geometry are discussed in detail in [1], [2], and [3]. The turtle is normally represented visually on the screen by a triangle shaped cursor pointing in the direction of the heading, although any picture can be substituted for this purpose. Any number of turtles can be created, each with its own local state.

These basic turtle commands are common to a variety of Logo systems for creating pictures using several different types of devices. In addition to the raster system, there are graphics systems which operate using vector display list displays, such as the DEC 340 and GT40 [5]. Another version uses character-only display terminals, where the turtle is constrained to move about a rectangular grid of characters [6]. The same turtle commands can also be used to control a robot turtle, which draws pictures on the floor with a felt tip pen [4].

2. Vectors versus video

Raster graphics systems such as the TV Turtle represent a significant departure from the vector display list approach which has characterized most previous graphics systems. Since the primitive capabilities of raster systems differ from those offered by vector display list systems, raster systems encourage a new outlook on graphics, and strongly influence the style of graphics programming.

One aspect of this is illustrated by an analogy with the contrast between compilers and interpreters for general purpose programming languages. The primary component of most vector display list systems is a *display list compiler* which compiles graphics commands into a *display list,* a set of instructions for a display processor. Raster systems eliminate the display list, making the display list compiler unnecessary. As a result, raster systems take on more of the flavor of an interpreter. The advantages in generality, simplicity, and ease of interaction offered by interpreters are especially important for systems like Logo, which are designed for use by beginners and are highly interactive. Although the extra conceptual complexity imposed by a compilation process can be minimized by a clever compiler when programs are simple, it becomes troublesome when an attempt is made to include more sophisticated capabilities.

The display list compiler found in most present graphics systems is necessary because the processor controlling the displays in such systems has primitive instructions which display visible objects such as vectors *momentarily.* In order for a picture to remain on the screen, the processor must repeatedly execute a loop containing the instructions to draw the picture, refreshing the screen when the image fades. The user would normally like to have commands which cause objects drawn to stay on the screen rather than flash and disappear. This is accomplished by having graphics commands operate by instructing a display list compiler to compile the user's requests into the instructions understood by the display processor. The global display list representing the picture is then edited to insert the compiled instructions into the refresh loop. This refresh loop is a separate process running in parallel with the user's program. The display list compiler usually must have the ability to compile incrementally, so that when the picture is changed it is not necessary to recompile all of it. The repertoire of visible objects available in the instruction set of most display processors typically includes vectors, points, and text. In addition, many display processors have the capability to call subroutines, so that a sequence of display instructions may be used from many places in the display list. Vector display list systems and display list compilers are discussed in detail in [7].

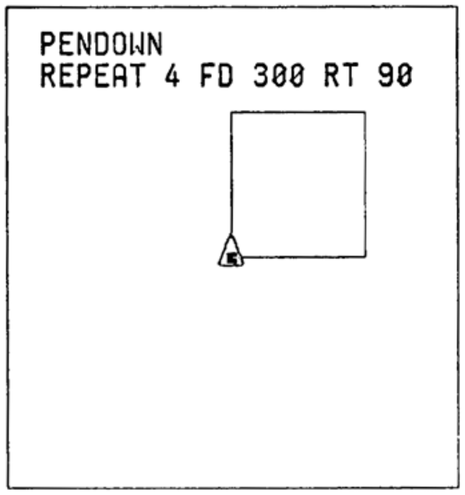
In raster systems like the TV Turtle, all graphics commands cause changes to the display screen directly by modifying the memory associated with each point on the screen. Hardware independently refreshes the screen from the contents of the memory, so the software need not concern itself with maintaining a separate process to refresh the screen. The primitive operation in a raster graphics system is to display a visible object which *persists* on the screen until it is explicitly erased or modified. Since changes are made directly to the displayed visual images, there is no need for a hidden data structure like a display list, and no need for a display list compiler. The visible effect of a command occurs immediately as the command is interpreted.

Interpretive systems only require a naive user to understand the semantics of the source language. Compilers tend to force the user to explicitly consider the semantics of the target language, and the process of translating programs from one language into the other. Thus, graphics systems based on display list compilers may require the user to know about the display processor's instructions and think of commands in terms of how they compile into display lists. With compilers, the user must also understand the effects of declarations as well as imperative commands; for example, declarations to a graphics compiler to "open" or "close" a display list [7]. Compiler systems are much less flexible than interpreters in interactive debugging. Examining compiled representations such as display lists is usually little help in debugging, and problems may arise because of discrepancies between source and compiled versions of a program. The compilation process frequently loses important information about the original program The user must tolerate the delay of compile time, during which untended changes to the program become effective. Local changes to a program often necessitate extensive recompilation. As raster systems interpret rather than compile graphics commands, they are able to avoid many of these problems associated with compilers.

Raster systems also permit a greater degree of flexibility in the choice of data structures used to represent pictures. Graphics systems of the vector display list type provide a very strong inducement for the user to conceptualize pictures as consisting solely of sets of lines. In many such systems, if the description of the picture is very complex in terms of the vector representation, the display will start to flicker objectionably, as the display processor will not be able to interpret all the instructions rapidly enough to refresh the screen. Curved lines are approximated by many short vectors and tend to require large display lists. Very few vector type systems can display a sufficient number of vectors to make shading areas feasible. The raster system, however, is not limited by the complexity of the picture. It will not flicker, even if the picture consists of a large number of vectors, text, points, curves, or filled in areas. Raster graphics encourages experimentation with a greater variety of representations for visual information. Pictures can be represented as sets of points, vectors, run length codes, procedures, or other alternatives.

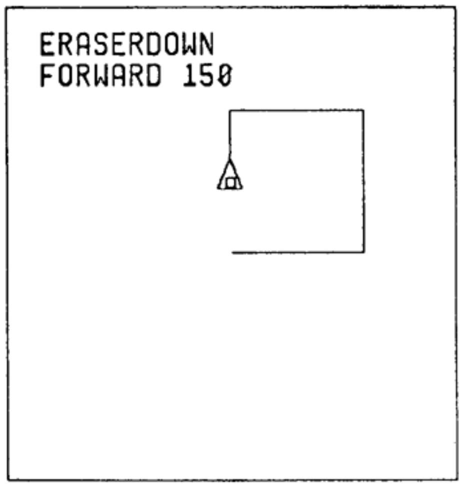
3. Erasing pictures

In raster graphics systems, points, vectors and other visible objects can be erased just as easily as they are drawn, by turning off bits in the screen memory instead of turning them on. We can supply the turtle with an *eraser* in addition to a pen. When the eraser is down, points along the turtle's path are erased, just as they are drawn when the turtle passes over them with the pen down. Alternatively, this can be thought of as drawing in the same color "ink" as the backgrounds causing previously drawn pictures to disappear. (See Figure 2.)



```
PENDOWN
REPEAT 4 FD 300 RT 90
```

Figure 2a. The turtle draws a square with the pen down.



```
ERASERDOWN
FORWARD 150
```

Figure 2b. The turtle erases part of one side of the square by going over it with the eraser down.

In vector display list systems, erasing a picture is accomplished by deleting the instructions which draw it from the refresh loop of the display processor. This is in many respects less flexible, as the structure of an object to be deleted must correspond exactly to an object drawn. The decision to enable erasing an object must be made when the display list is compiled, whereas in raster systems this decision can be made at any time during the execution of a program. It is generally not possible in display list systems to erase only *part* of a vector or vector subroutine previously displayed.

One problem with using the turtle's eraser to remove pictures from the screen is that it often does not behave as expected for pictures that are considered to *overlap* on the screen. When considering two dimensional pictures as representations of three dimensional objects, an opaque object prevents objects behind it from being seen by an observer. If the opaque object is removed, obscured objects which were behind it reappear. Thus, it is expected that drawing an object and erasing it will leave a picture undisturbed. The semantics of the erase command desired, then, is to erase an object, but restore whatever was visible behind it before the object was drawn. In vector display list systems, instructions for displaying two intersecting vectors will result in displaying the intersection point *twice* during the refresh cycle, so that removing one of the vectors will not cause the intersection point to go away, as it will be displayed by the instruction for the remaining vector. In the TV Turtle, if intersecting lines are drawn with the pen down, and one line is erased by backing over it with the eraser down, the point of intersection will be turned off as well. When a picture is drawn over another, the obscured picture is not saved, and so is not restored when another picture passing through the same points is erased.

One mechanism for solving this problem is to explicitly save a portion of the old picture before drawing a new picture over it, so that the old picture can be restored when the new picture is erased. This complicates programs which do frequent display and erasing of overlapping pictures. More elaborate hardware could provide a "two-and-a-half dimensional" feature, in a system with several bits per point, where each bit is assigned a plane and hardware determines the visibility of each point according to its plane [8].

In addition to the pen and the eraser, the turtle is also equipped with an *exclusive OR* (XOR) mode. In this mode, the turtle complements the state of points as it passes over them, turning on points which were previously off and turning off points which were on. This mode is often useful with simple overlapping pictures, where display of the intersecting area is not critical, but it is important to avoid disturbing a background picture while some other picture is drawn and then erased. Using XOR mode, a picture can be drawn and subsequently erased using the same procedure, transparent to any previously existing picture. It is also very useful when a particular picture should remain distinctly visible, even against backgrounds of heavily shaded pictures. For these reasons, XOR mode is used to display the triangle cursor which marks the state of the turtle on the screen.

4. Reading the screen

Another important conceptual extension permitted by raster graphics is that the screen *remembers* what is drawn upon it; it is possible to read from the screen as well as write on it. Individual points on the screen can be tested to see if they are on or off. This allows programs to *perceive* what is on the screen as well as modify it. A version of the FORWARD command can easily be defined which will stop if it the turtle hits any displayed "obstacles" in its path. A "seeing eye turtle" can report evidence of visible objects in the direction of the turtle's heading. The turtle can be programmed to find its way out of mazes or navigate around obstacles. The shading facility also depends in an essential way upon being able to read the screen to search for the boundaries of a region.

In almost every vector display list system, it is impossible to examine a display list after it is created to determine what is on the screen. Display lists are usually made accessible only from machine language, and require knowing the format of the display processor instruction set. Even if this information were available, figuring out whether a particular point on the screen was on would be a lengthy computation, as it would involve examining all the display lists existing in the system. Programs which need to know what has been drawn must instead maintain some auxiliary data structure to record changes to the screen, and modify that structure whenever a display command is issued. Instead of using display lists as the primary data structure for pictures, in raster systems the user deals directly with the screen memory, and can think of sensing and displaying points as simply reading or writing the screen.

5. Words and pictures

All interaction with the system involving text, such as command input, program editing, and error messages occurs on the *same* display screen used for graphics. This has the advantage of concentrating the attention of the user on a single screen, rather than dividing it between a graphic display and text terminal. An auxiliary processor is employed to speed up the drawing of text on the screen during normal character input and output. One can, however, use the graphics commands to display text as well, perhaps to print using a different font than the system's default. For convenience, a split screen mode is normally used in which the top area of the screen is reserved for graphic output, and the bottom for typein and typeout. The sizes of the text and graphics sections are easily adjustable so that, for example, the text area can be expanded while editing the text of a program, or the graphics area enlarged while watching pictures being drawn. Text and graphics can interact flexibly, and the user can print labels or captions on pictures, or draw pictures to accompany text. Printed copy of both text and graphics is available.

6. Shading areas

Raster style graphics systems enable the user to think about *areas* as picture components as well as just lines and points. This allows the user to fully exploit the two dimensional nature of the display screen. In the TV Turtle, primitives are provided for constructing, naming and manipulating areas.

The concept of the turtle's pen which draws lines is readily generalized to that of a brush which sweeps out areas. An arbitrary picture may be designated as the turtle's brush, and that picture is swept across the path along which the turtle moves. This permits the turtle to draw or erase lines of any thickness, as well as lines normally one point wide. In conjunction with input from a device such as a mouse or tablet, the brush can be used for painting regions freehand.

For programs, however, the brush mode is not completely satisfactory. Programs for constructing filled in areas with even simple geometric shapes quickly become complicated. To allow easy creation of shaded regions, the system provides a SHADE command. A closed curve is drawn with the pen down, outlining the region to be filled in, and the turtle is placed at any interior point of the region before issuing the command. The system then scans the screen to find the boundaries of the region and fills it. The area to be shaded may be of any shape or size. An argument can be given to SHADE to specify a *shading pattern*, which tells the turtle how to fill the area. The default shading pattern, SOLID, turns on every point in the region. The system also provides a set of predefined shading patterns, which are sufficient for typical uses of this feature, like distinguishing between a few neighboring regions. These perform functions such as filling areas with horizontal or vertical lines. The user also has the option of using any saved picture as a pattern, in which case the region is filled with that picture, repeating it "wallpaper" style if necessary. Alternatively, the user can supply a function to decide how to shade the area. (See Figure 3.)

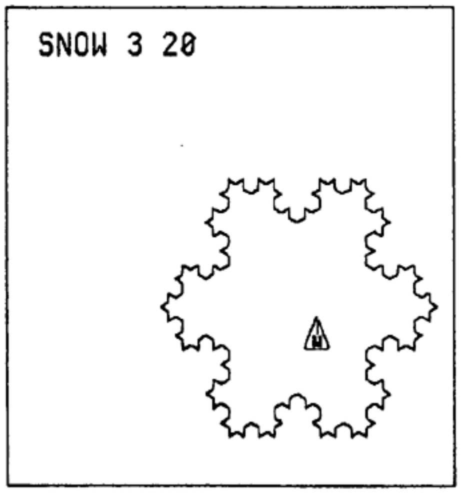


Figure 3a. The turtle is inside a "snowflake curve".

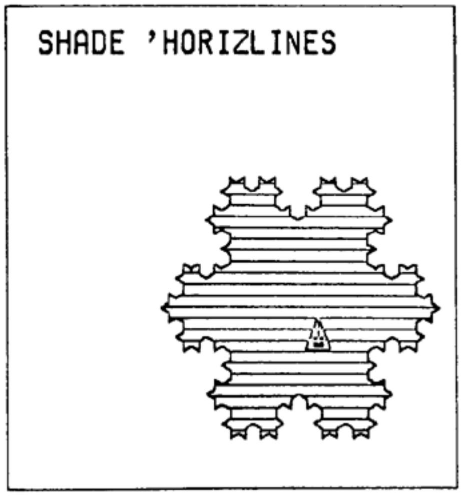


Figure 3b. The curve is filled in with horizontal lines

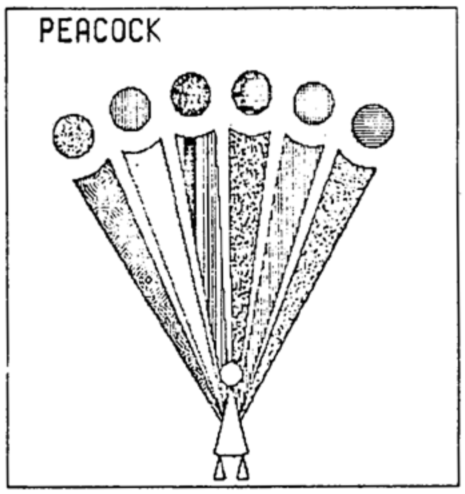


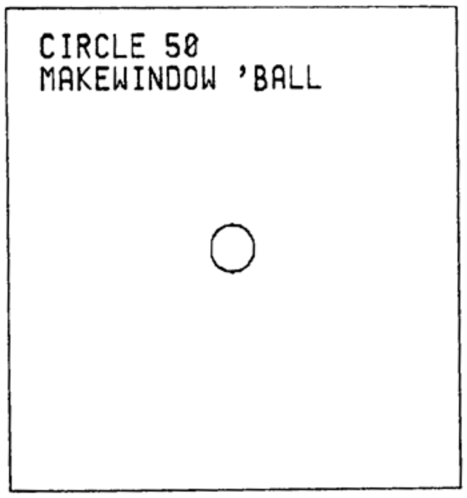
Figure 3c. A picture created using several different shading patterns.

In the color version of the system described below the shading facility is especially useful in filling areas with different colors.

7. Animation

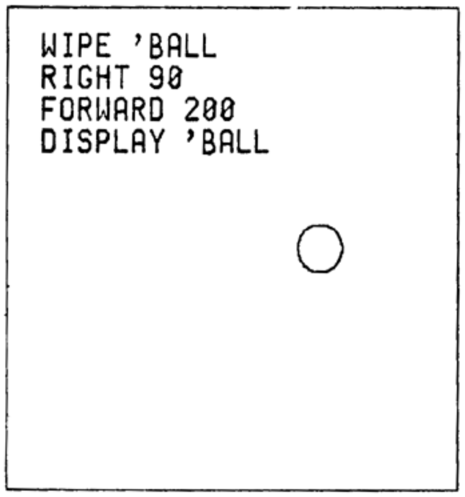
For real time animation, it is important that the user be able to create, display, erase and transform pictures rapidly enough to give the appearance of motion. Neither the vector display list nor raster graphics systems are usually fast enough to make it feasible to draw each frame from scratch, dynamically computing the transformations of the picture from one frame to the next. In vector display list systems, the limitation is usually not the speed of the display processor, but the speed of the display list compiler. Instead, movies are usually made by drawing frames or key components of frames, and saving them in some lower level form which is faster to redisplay, erase, and move than re-executing the original procedure that drew the picture. In the case of vector displays, this low level representation for pictures is a display list.

In the TV Turtle system, the principal data structure used for this purpose is called a *window*. A window is an array of points representing the picture in a rectangular area on the screen. It can be thought of as a "photograph" of that part of the screen within a given rectangular "viewfinder". Once created, a window can be displayed or erased at any location on the screen. (See Figure 4.)



```
CIRCLE 50
MAKEWINDOW 'BALL
```

Figure 4a. A picture of a ball is saved in a window.



```
WIPE 'BALL
RIGHT 90
FORWARD 200
DISPLAY 'BALL
```

Figure 4b. The ball is moved 200 steps to the right of its original position.

Redisplay of a window is implemented simply by transferring the contents of the window array into the screen memory, which is a very fast operation. Transformations such as rotation, reflection, and scaling of windows are currently not provided as primitives, but may be programmed by the user or a systems programmer. Showing a movie by displaying pictures saved in windows is usually much faster than executing procedures to draw each picture. Windows also provide a means of clipping pictures, as any portion of the picture outside the rectangular area will be excluded from the window.

The speed of displaying windows depends only upon how much area the picture occupies. It is totally insensitive to the complexity of a picture within a region, unlike display lists. The window representation is therefore ideal for animation in which most changes happen within small areas of the screen, but must happen very fast. An example is a game like space war, where objects that move, appear and disappear (ships and torpedoes) might be complex pictures, but are small relative to the size of the screen, and need to be changed rapidly. Windows may tend to be somewhat space consuming for some applications. This problem could be reduced by converting to a run length encoding scheme, at the cost of increasing the amount of time necessary to redisplay a window. It is also possible to construct other types of low level representation for pictures, optimized for pictures with particular known characteristics.

A disadvantage of the window commands as they currently exist is that the area saved in a window must be specified as a rectangular area. This may make it inappropriate for saving pictures with irregularly shaped boundaries. Saving the enclosing rectangular area may save in the window portions of some other unwanted picture adjacent to it, or at least waste space saving empty area. Overlapping pictures are also a source of difficulty, making it difficult to assign separate names to the output of two procedures if the pictures they draw can overlap on the screen. Erasing a window also erases the picture "behind" it, a problem also encountered when using the eraser mode of the turtle, as discussed above. As with the eraser mode, this problem can be alleviated to some extent by using XOR mode, or explicitly saving and restoring pictures. Rotation and scaling transformations are more difficult to apply to pictures using the window representation than is the case with vector display lists.

In Logo implementations for vector displays such as those described in [4] and [5] the basic picture saving capability is provided by the SNAP command. This command saves a picture (usually the entire screen) as a display list subroutine. A snap can subsequently be redisplayed anywhere on the screen by inserting a call to that subroutine into the current display list. Removing the call erases the picture saved in the snap from the screen. The implementation described in [5] provides such capabilities as copying snaps, editing them by appending instructions to the subroutine, and restricting the scope of the picture saved in a snap to the display list output by a particular piece of source code.

A problem with saving pictures as display list subroutines using the SNAP command is that the relationship between pictures visible on the screen and pictures saved in snaps is not always clear. This is especially true when multiple copies of snaps may exist, when snaps can be edited, and when there is extensive overlapping of pictures. There is no way to restrict the extent of a snap to a particular area on the screen. Windows, however, have the property *that what you see is what you get*. Everything visible within the specified area will be included in the window, regardless of what caused it to appear, including vectors, text, points, shaded areas, and other displayed windows. Windows allow selecting a particular region of the screen to be saved, whereas snaps allow selectivity with respect to the time at which the picture was drawn. Another problem with snaps is that they cannot be examined in any reasonable way, and must remain somewhat mysterious to a naive user. The internal structure of a window, however, is simply an array of points, and can easily be examined and modified by the user or by programs.

8. Color

In addition to the black and white system, another version of the TV Turtle displays full color pictures on an Advent projection television screen. The color version understands most of the same commands as the black and white version, with additional extensions for creating and manipulating colors. The system supplies a set of colors initially, but the user may define new colors using the MAKECOLOR command, which creates a new color from a specified mix of the primary colors red, green, and blue. The turtle's pen has a *pen color*, and all drawings done by the turtle appear on the screen in the currently selected pen color. The system also keeps track of the color of the background against which pictures are drawn. Pictures are erased by redrawing them with the eraser down, using this background color, called the *eraser color*. CLEARSCREEN fills the screen with the eraser color, thereby erasing everything on the screen.

The screen memory for the color system uses several bits to describe the state of each point. These bits address an array called the *palette*, which holds a set of currently available colors. A point may be displayed in any color which is a member of this set. The colors chosen for the palette may be changed at any time. Thus, any number of colors may be defined, but the number of colors actually visible on the screen at any particular time is limited by the capacity of the palette. Whenever a PENCOLOR command is issued, the palette is searched to find the new pen color. If it is present, that color is selected. If not, the new color is inserted into the next available space in the palette. CLEARSCREEN deletes all the colors from the palette except for the current pen and eraser colors. Thus, the user need think only in terms of symbolically named colors rather than numerical indices into the palette, and will only get an error if the limit on the number of different colors in use simultaneously is exceeded. The palette can, however, be explicitly accessed if desired.

It is very easy to change the color to which a particular number representing the state of a point corresponds, simply by modifying the palette. This has the visual effect of simultaneously changing all points on the screen in that color. A REPLACECOLOR primitive is supplied, which given two colors, changes all pictures on the screen in one color to another. This operation is much faster than achieving the same effect by redrawing the picture in another color. Using this, it is easy to program psychedelic effects by rapidly changing colors randomly in a multicolored picture. It is also possible to exploit this feature for animation, so that a picture could be made to appear or disappear very quickly by using REPLACECOLOR to change it from the background color to some contrasting color and back again.

9. Implementation

The TV Turtle is a package of functions written in the MacLisp dialect of Lisp for the PDP10 [9]. It can be used either directly from Lisp or from our Lisp implementation of Logo [5]. A similar graphics system is currently being implemented for the PDPII/45 in assembly language. Both the black and white and the color displays are compatible with standard broadcast television, allowing the use of standard TV monitors or videotape equipment. Resolution is approximately 575 points horizontally by 454 points vertically. The black and white system has one bit per point (no gray levels). The color system has four bits per point, allowing up to sixteen different colors to appear on the screen at once. Hardware for the black and white system was designed by Tom Knight, for the color system by Ron Lebel.

Acknowledgments

The author would like to thank Hal Abelson, Ken Kahn, and Ira Goldstein for conversations which helped to crystallite the ideas presented here, and for suggestions which were very helpful in debugging this paper.

Bibliography

[1] Seymour Papert
A Computer Laboratory for Elementary Schools
Logo memo 1, MIT Artificial Intelligence Lab, October 1971

[2] Seymour Papert
Teaching Children to be Mathematicians Vs. Teaching About Mathematics
Logo memo 4, MIT Artificial Intelligence Lab, July 1971

[3] Howard Austin
The Logo Primer
Logo working paper 19, MIT Artificial Intelligence Lab, January 1973

[4] Hal Abelson, Nat Goodman, Lee Rudolph
PDPII Logo Manual
Logo memo 7, MIT Artificial Intelligence Lab, August 1974

[5] Ira Goldstein, Henry Lieberman, Harry Bochner, Mark Miller
LLogo: An Implementation of Logo in Lisp
Logo memo 11, MIT Artificial Intelligence Lab, March 1975

[6] Ira Goldstein
Germland
Logo working paper 7, MIT Artificial Intelligence Lab, February 1973

[7] William Newman, Robert Sproull
Principles of Interactive Computer Graphics
McGraw Hill, 1974

[8] Nicholas Negroponte
Raster Scan Approaches To Computer Graphics
MIT Architecture Machine Group memo

[9] David A. Moon
MacLisp Reference Manual
MIT Project MAC memo, December 1975