To the Graduate Council:

I am submitting herewith a thesis written by Chu Wang entitled "Algorithms for Region Filling with Mixed Colors." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

J. Michael Moshell, Major Professor

We have read this thesis
and recommend its acceptance:

David Matuszek

Accepted for the Council:

L Evans Roth
Vice Chancellor
Graduate Studies and Research

Thesis
13
Wa2
cop 2

# ALGORITHMS
# FOR
# REGION FILLING WITH MIXED COLORS

A Thesis

Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Chu Wang

March 1983

3066085

To my parents

Mr. and Mrs. Daw-Jen Wang

and my sisters.

# ACKNOWLEDGEMENTS

The author wishes to thank to Dr. J. Michael Moshell for his guidance, assistance and patience in the preparation of this thesis. Appreciation is also expressed to Dr. David Matuszek and Dr. David Straight for their support as committee member. The author also wishes to thank to Dr. Martha R. O'Kennon for her time and suggestion. Finally, appreciation is expressed to Mr. Seck Wong and Mr. Carl Gregory for their editing and suggesting of this thesis.

## ABSTRACT

Computer graphics applications are becoming widespread. Line drawing alone no longer satisfies the average user's needs. Region-filling is one advanced feature to make a computer graphics system more sophisticated. Roughly speaking, there are two categories of filling algorithms. One is to fill a pixel on the screen and recursively fill its four neighboring pixels. The other is to fill a whole line within the boundary, then fill the neighboring lines if they are not filled yet.

This study uses the latter algorithm to develop a package of software that fills bounded regions of a graphics screen with varied solid or halftone colors.

The package is implemented on the Apple II computer. It does line drawing, color filling and record keeping. It has a user friendly interface. A major effort has been dedicated to speed the process by converting critical parts of the code to machine language. Drawing a typical full-screen scene required 130 seconds in Pascal, and less than three seconds in assembly language.

# TABLE OF CONTENTS


| CHAPTER | | PAGE |
|---|---|---|
| I. | INTRODUCTION | 1 |
| | 1.1 Introduction | 1 |
| | 1.2 Some Definitions | 2 |
| | 1.3 Survey of Some Algorithms | 6 |
| II. | PROTOTYPE PROGRAM DEVELOPMENT | 11 |
| | 2.1 Low-resolution | 11 |
| | 2.2 High-resolution | 14 |
| | 2.3 Prototype Program | 18 |
| | 2.4 Data Structures | 21 |
| | 2.5 Problems | 25 |
| III. | IMPLEMENTATION | 26 |
| | 3.1 Assembler/Pascal Relationship | 26 |
| | 3.2 Experimental Implementation | 29 |
| | 3.3 Full Implementation | 32 |
| | 3.4 Storing a Keystroke Record | 36 |
| IV. | USER'S MANUAL | 40 |
| | 4.1 Preparation | 40 |
| | 4.2 Drawing Outline | 40 |
| | 4.3 Fill Region | 45 |
| V. | CONCLUSIONS | 47 |
| | 5.1 Accomplishments | 47 |
| | 5.2 Future Development | 48 |
| BIBLIOGRAPHY | | 49 |
| VITA | | 51 |

# LIST OF FIGURES


| FIGURE | | PAGE |
|---|---|---|
| 1.1 | Interior-defined Regions ..... | 3 |
| 1.2 | Boundary-defined Regions ..... | 4 |
| 2.1 | Single Color Byte Table ..... | 15 |
| 2.2 | Symbol of Representation ..... | 16 |
| 2.3 | Two Colors Not Mixed ..... | 16 |
| 2.4 | Two Colors Mixed ..... | 17 |
| 2.5 | Three Colors Matching ..... | 17 |
| 2.6 | Record of SCENE ..... | 21 |
| 2.7 | The Screen-Memory Relationship ..... | 23 |
| 2.8 | Low Order Bits ..... | 24 |
| 2.9 | High Order Bits ..... | 25 |
| 3.1 | Macros used in Debugging ..... | 28 |
| 3.2 | Macro MPUSH and MPOP ..... | 33 |
| 3.3 | System Memory Map ..... | 35 |
| 3.4 | New Record of SCENE ..... | 36 |
| 3.5 | Record of INLIST ..... | 37 |
| 3.6 | Procedure GETIN ..... | 38 |
| 4.1 | State Diagram of The Program ..... | 41 |

# CHAPTER I

# INTRODUCTION

## 1.1 Introduction

Computer graphics has been widely used recently in many fields such as medical computing, geological computing, education, animation, and so on. There are also many microcomputers capable of simple graphical work such as the Apple II, Atari 800, IBM PC, TI 99/4A. Before using these machines, programmers often have to figure out what background scene should appear on the monitor screen and how to make this scene.

Scene drawing is getting more and more important in computer graphics, and many graphics editor programs have been developed for microcomputers. For instance, "Delta Drawing" (11) can be used as an educational tool and a recreational game. It uses simple keystrokes to control a cursor, and the cursor control the action as it moves, adds, draws, or erases parts of a scene. When a picture is drawn, the program keeps track of every keystroke and records the sequence of commands as a disk file. One useful feature is that the program has a command that can switch back and forth between graphics and text mode, which allows the user to write a program and see the picture right away.

Line and point drawing techniques have been used for a long time. It is not hard for a programmer to draw the

outlines of a picture. This project involves the experimental development of a region-filling algorithm that produces a large number of different halftone colors, and of an editor that demonstrates the region-filling system. This project begins with a survey of region-filling algorithms. Experiments were conducted to determine how best to produce halftone colors on the Apple II computer. Test programs were developed using UCSD Pascal. To make filling faster, all the critical procedures were converted to assembler language.

The demonstration program uses RASCAL (a cartoon animation library from Gentleware Corp. Knoxville, TN. details described later) to draw the outline of a region. Some RASCAL functions and procedures were modified due to the limited memory space of Apple II computer which has only 64K bytes of memory.

## 1.2 Some Definitions

A region is a group of adjacent, connected pixels. Regions can be created or defined either by assigning a specific color to the pixels contained in the region or by assigned a specific color to the pixels bordering on the region.

Regions defined by interior colors are called interior-defined: all pixels in the region have the "old-color", and all pixels on the boundary have another

color. Algorithms which set a region's old-color to a new-color are called flood-fill algorithms. Figure 1.1 shows two interior-defined regions. Character x and character o represent the old-color. Flood-fill is to change x and o to another character.

```
   xxxxxxx                 oooooooo
  xxxxxxxxxxx              oooooooo
    xxxxxxxx               o     ooo
xxxxxxxxxxxxxx             o     ooo
   xxxxxxxxxxx           ooo     ooo
   xxxxxxxxxxx             o     ooo
   xxxxxxxxxxx           oooooooo
```

Figure 1.1 Interior-defined regions

Regions defined by a boundary are called boundary-defined: pixels on the boundary have the "boundary-color". Algorithms which set a region's color to a new-color are called boundary-fill algorithms. Figure 1.2 shows two boundary defined regions. Character A represents the boundary-color. Boundary-fill is to change the blanks within the boundary into a new color.

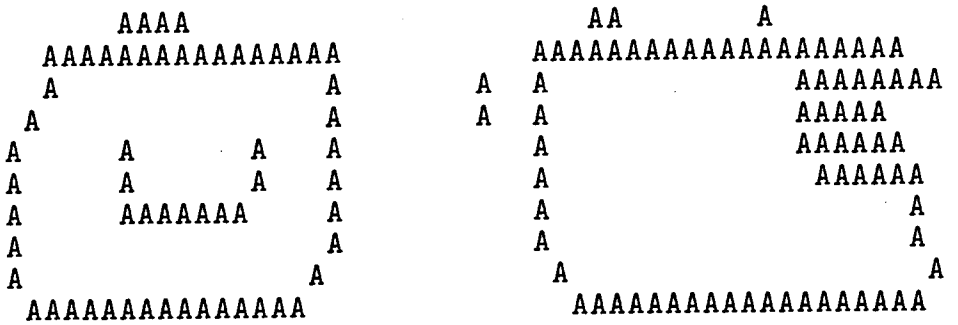


Figure 1.2 Boundary-defined regions

In most microcomputer graphics systems, there is a fixed set of memory locations which stores the screen information, called the FRAME-BUFFER. Each pixel has its own location in the FRAME-BUFFER. The color of a pixel on the screen is determined by the value of that pixel in memory. To fill a pixel with a color, technically speaking, is to put the value of that color into the pixel's memory location. The color representation and FRAME-BUFFER's location are system dependent.

RASCAL is a cartoon animation library. It contains procedures for Apple UCSD Pascal which allow for the manipulation of movable "figures" in front of a fixed background scene. A movable figure consists of a description of a series of lines: lengths, colors and directions. One of the figure which is used as a "PAINT BRUSH" in this project is called STICK. Some useful procedures of RASCAL are as follows:

TURNON(FIGURE): causes the figure to be shown on the screen.

TURNOFF(FIGURE): removes the figure from the screen.

POSITION(FIGURE,X,Y): locates the figure at position (X,Y) on the screen.

SETANGLE(FIGURE,ANGLE); sets the figure's angle with respect to its original position of definition.

SETSCALE(FIGURE,SCALE): causes a figure to grow or shrink. Scale range is from 0 to 200, with normal size = 100.

SETCOLOR(FIGURE,COL): sets the figure's color to the color named by the value COL. There are six colors of the figure: black, white, blue, green, violet and orange. Color is a scalar Pascal data type, defined by the RASCAL unit.

DELAY(TIME): causes a pause of 'TIME' tenths of a second.

PAINT(FIGURE): causes a figure, in its current orientation, color, and size, to be added to the scene.

FAREND(X,Y): returns the location of the far end of the figure, by assigning values to reference parameters X and Y.

CLEARTEXT: clears the text screen and

positions the cursor at (0,20) for writing messages in the the four-line text window.

## 1.3 Survey of Some Algorithms

The region-filling algorithm that Newman & Sproull (9) provide is a recursive procedure:

```
PROCEDURE FILL(X,Y,OLD-COLOR,NEW-COLOR: INTEGER);
 BEGIN
  IF GETPIXEL(FRAMEBUFFER,X,Y)=OLD-COLOR THEN BEGIN
     SETPIXEL(FRAMEBUFFER,X,Y,NEW-COLOR);
     FILL(X+1,Y,OLD-COLOR,NEW-COLOR);
     FILL(X-1,Y,OLD-COLOR,NEW-COLOR);
     FILL(X,Y+1,OLD-COLOR,NEW-COLOR);
     FILL(X,Y-1,OLD-COLOR,NEW-COLOR);
   END
 END
```

GETPIXEL is a function that returns the value of the pixel. SETPIXEL is to set a new color value to a pixel. A user invokes this procedure by pointing to a pixel within the region to be filled, and indicating a new-color. The procedure uses GETPIXEL to check the pixel. If it contains the old-value then it is changed to new-color, otherwise just skip it. From the same point, it checks the 4 neighboring pixels and fills them, and does the same thing until all pixels have been checked and changed.

The algorithm that Foley & Van Dam (10) provided is similar to the above. These procedures are simple but highly recursive, and the many levels of recursion take time and may cause stack overflow when memory space is limited.

The algorithms of Pavlidis (2), Lieberman (1) and Smith (3) are much more logical than the above algorithms. The filling procedures fill one line at a time instead of a pixel, then check the lines above and below the current filled line. These algorithms reduce the recursive depth.

Take Smith's algorithm, for example. It is scanline oriented; that is, all procedures try to fill along a scanline before it changes Y, the vertical coordinate. The basic notion is that the FILL procedure fills all pixels connected to the starting point (seed-point) on the first scanline. Then it looks at the scanline above and below for points connected to the scanline segment just filled. A number of these points sufficient to guarantee connectivity go on a stack. When the scans are finished, a point is popped from the stack and this becomes a new seedpoint.

```
PROCEDURE FILL (SEEDX,SEEDY,NEWPV);
  INTEGER SEEDX,SEEDY;
  PXVAL   NEWPV;
  BEGIN
    X:=SEEDX; Y:=SEEDY;
    NEW:=NEWPV; OLD:=GET;
    IF OLD=NEW THEN RETURN;
    YREF:=Y; PUSH;
    WHILE STACKNOTEMPTY DO BEGIN
      POP;
      IF GET=NEW THEN CONTINUE;
      FILLLINE;
      IF HINEIGHBOR THEN SCANHI
        ELSE IF LONEIGHBOR THEN SCANLO
        ELSE BEGIN
          SCANHI; SCANLO;
        END;
      YREF:=Y; LXREF:=LX; RXREF:=RX;
    END
  END
```

Function GET returns the value stored at location (X,Y). Its complement SET sets location (X,Y) to the new value.

Function PUSH pushes X and Y onto the stack. POP pops the top two locations from the stack into X and Y. STACKNOTEMPTY returns true if the stack is not empty or false if it is.

Function SAVEX saves the current value of X in temporary storage. It is retrieved by RESTOREX.

The other routines are defined by these procedures.

```
PROCEDURE FILLINE;
BEGIN
  FILLRIGHT; FILLEFT;
END;


PROCEDURE FILLRIGHT;
BEGIN
  SAVEX;
  WHILE GET=OLD AND X<RX DO BEGIN
    SET; X:=X+1;
  END;
  RX:=X-1; RESTOREX;
END;


PROCEDURE FILLEFT;
BEGIN
  SAVEX; X:=X-1;
  WHILE GET=OLD AND X>LX DO BEGIN
    SET; X:=X-1;
  END;
  LX:=X+1; RESTOREX;
END;
```

The speed of the fill algorithm can be improved by noticing that when neighboring scanline segments are being filled, one of the procedure calls to SCANHI or SCANLO may be redundant. Thus the functions HINEIGHBOR and LONEIGHBOR are designed to check for the cases when it can skip one of the scans above or below.

```
FUNCTION HINEIGHBOR;
BEGIN
  IF Y=YREF+1 AND LX>LXREF-1 AND RX<RXREF+1 THEN
  HINEIGHBOR:=TRUE ELSE HINEIGHBOR:=FALSE;
END;

FUNCTION LONEIGHBOR;
BEGIN
  IF Y=YREF-1 AND LX>LXREF-1 AND RX<RXREF+1 THEN
  LONEIGHBOR:=TRUE ELSE LONEIGHBOR:=FALSE;
END;
```

My program is based on Smith's algorithm, using both boundary-fill and flood-fill techniques. The screen is set to black at the beginning of the program. A boundary can be drawn using six colors: black, white, green, orange, violet and blue, but a black boundary cannot be correctly filled by the program. When an outline is drawn, the filling algorithm changes the black pixels within the boundary to the specific color.

A demonstration program was designed to illustrate a typical application of a fast regions-filling procedure. The functions of the demonstration program are designed to be:

1. Using PAINTER (a line-drawing editor from the RASCAL system) to draw the outline of a scene.

2. Choosing a desired color or color pattern with which to fill.

3. Calling an efficient procedure to fill the region within the outline.

4. Storing the whole scene in a small space.

The first step to accomplish the above goals is to write a test program to see if Smith's algorithm can work, then write a prototype Pascal program using the algorithm. Chapter II describes these stages and also introduces some library units used in the program. Chapter III describes the modifications of the prototype program and changes in the library. Chapter IV describes the implementation of the program, and is a kind of user's manual. Chapter V describes the problems encountered and the future development of the program.

# CHAPTER II

# PROTOTYPE PROGRAM DEVELOPMENT

## 2.1 Low-resolution

To test Smith's (3) algorithm, a simple filling program using low-resolution graphics was developed first. Hi-resolution on the Apple computer is too complicated for a prototype program to use, and what we want is to know if the algorithm can work.

The low-resolution screen can show an array of blocks 40 wide and 48 high. Eight blocks at the bottom are for mixed-text use, so actually it is a 40 x 40 screen. Each byte in memory represents two blocks on the screen, stacked vertically. Each nybble (one half byte) can hold a value from zero to 15, which determines the color of that nybble.

Two RASCAL units, LORES and FREEJOY are used in this program. LORES enables the program to do graphics in the low-resolution mode. FREEJOY allows the keyboard to simulate a "JOYSTICK" while updating numerical, character and color input functions according to the most recent keystroke. Some useful functions and procedures are described as follows:

LOGRAFMODE: puts the system into a 40 x 40 color graphics mode, with four lines of text displayed at the bottom.

DOT(color,x,y): produces a dot at the location (X,Y) of the color specified by the integer "color".

LOFILL(color): fills the screen with the color named.

WHATDOT(x,y): returns the integer between 0 and 15 which represents the color at position (X,Y).

JOYCTR: resets the pseudo-joystick to its current center position.

JOYCHAR: returns the most recently struck character.

The main routine initiates the low-resolution mode and fills the screen with black. A cursor is shown on the center of the screen. The user can use paddle(0) to choose the color of the cursor. The keypad (a set of keys around K on keyboard, functions provided by FREEJOY unit) can control the moving direction of the cursor. Type "I" to move cursor up one line, "<" to move down one line, "J", "L" to left and right respectively. While moving the cursor, the color dot will remain on the screen and form a boundary. When a closed boundary is formed, then the filling routine can be called.

In the filling routine, the user can use paddle(0) to choose an internal color, and use KEYJOY to move the cursor to the starting point. When the user is ready to fill the

inside region, he types "S" on keyboard. The program pushes starting point's location (X,Y) onto a stack, then enters a loop. In the loop, (X,Y) are popped from stack and the algorithm fills to the right and left until it hits the boundary. When one line has been filled, the procedures SCANHI and SCANLO are called to find the next starting point. This is the heart of the filling program. When an unfilled dot is found, the location of that dot is pushed onto stack. The procedure then pops one dot's location (X,Y) from the stack and start to fill that line. The filling program will stop when the stack is empty.

An integer RX is set equal to the position value of the rightmost pixel of the inner region when the routine hits the boundary dot. The scanning procedure will start from the same X position but in a lower or higher line and expand to the right until it finds an unfilled dot or if the X value is not less than RX. If it is less than RX, that means a new starting point has just been found, and (X,Y) is pushed onto the stack; otherwise exit from the routine. The LOWNEIGHBOR and HIGHNEIGHBOR functions are routines that can check the lower or higher line to see if they are filled already. These two routines help reduce the scanning steps.

The test program draws an outline of a picture and then fills the inner region with a selected color. At that point, we can say the algorithm is essentially working and can be changed to high-resolution.

## 2.2 High-resolution

Apple high-resolution graphics mode can display 52760 dots in a matrix 280 wide and 192 high. Thirty two lines at the bottom of the screen are reserved for mixed-text mode. So the graphics screen is actually 280 x 160 dots. The screen can display black, white, violet, green, red and blue dots. The colors and their representative byte patterns are shown in Figure 2.1.

The high-resolution graphics mode stores its data in 8192 bytes of memory, called a "picture buffer". There are two picture buffers: one for the primary page called "FOREGROUND" and one for the secondary page called "BACKGROUND". Animation thus can be made by altering the contents of the two pages.

Each dot on the screen represents one bit from the picture buffer. Only seven of the eight bits of each byte are displayed on the screen, with the eighth bit (called the phase bit) used to select the color of the dots in that byte. There are some limitations on the color display:

1) Dots in even columns must be black, violet, or blue.

2) Dots in odd columns must be black, green, or red.

3) Each byte must be either a violet/green or a blue/red byte. It is not possible to mix green and

blue, green and red, violet and blue, or violet and red in the same byte.

4) Two colored dots side by side always appear white.

| COLOR | ODD BYTE | HEX | EVEN BYTE | HEX |
|---|---|---|---|---|
| BLACK | 00000000 | 00 | 00000000 | 00 |
| WHITE | 01111111 | 7F | 01111111 | 7F |
| GREEN | 00101010 | 2A | 01010101 | 55 |
| ORANGE | 10101010 | AA | 11010101 | D5 |
| VIOLET | 01010101 | 55 | 00101010 | 2A |
| BLUE | 11010101 | D5 | 10101010 | AA |

Figure 2.1 Single color byte table

A screen filled with colors actually is an arrangement of color bytes in the foreground and background. Each color has its unique representation, or rather its "value." The color will appear on the screen simply by putting the selected value into the desired memory location. Multi-color filling is the most desirable way of filling since single-color filling seems too dull. Because the color bytes are different, not only in their representation but also in their location, color matching in multi-color filling is somewhat difficult. This program provides four options of color filling described as follows:

Single color filling: Colors (See Figure 2.1) are represented according to their odd or even location on the horizontal axis. The symbol representation is shown on Figure 2.2.

A: one byte of color A.
B: one byte of color B.
C: one byte of color C.
a: half byte of color A.
b: half byte of color B.

Figure 2.2 Symbol of representation

Two colors not mixed: Colors are represented according to their odd or even location on both the horizontal and vertical axes. The bytes for each color are alternated by row and column. (See Figure 2.3)

| EVEN | ODD | EVEN | ODD |
|---|---|---|---|
| A | B | A | B |
| B | A | B | A |
| A | B | A | B |
| B | A | B | A |

Figure 2.3 Two colors not mixed

Two colors mixed: One half byte of each color is used. The two half bytes are combined to form a "halftone" color. Then they are treated like "two colors not mixed". (See Figure 2.4)

| EVEN | ODD | EVEN | ODD |
|---|---|---|---|
| ab | ab | ab | ab |
| ba | ba | ba | ba |
| ab | ab | ab | ab |
| ba | ba | ba | ba |

Figure 2.4 Two colors mixed

Three color filling: Three colors are represented in sequence in both vertical and horizontal locations. Each color uses the odd or even representation it needs. (See Figure 2.5)

| EVEN | ODD | EVEN | ODD | EVEN | ODD |
|---|---|---|---|---|---|
| A | B | C | A | B | C |
| B | C | A | B | C | A |
| C | A | B | C | A | B |
| A | B | C | A | B | C |
| B | C | A | B | C | A |
| C | A | B | C | A | B |

Figure 2.5 Three colors matching

An integer variable, MODE, is set when the filling method has been decided. This integer will control the color byte when filling starts. MODE = 1 means single color, MODE = 2 is two colors not mixed, MODE = 3 is three colors, and MODE = 4 is two colors mixed. Each mode has its own color selection method.

## 2.3 Prototype Program

Several external units and procedures are used in the program:

RASCAL: Contains some procedures which allow for using "PAINT BRUSHES" to draw a figure.

DIRLST: Provides a procedure to show what files are on the disk.

KEYJOY: Simulates a joystick when updating integer, character and color input functions according to the most recent keystroke.

Memory addresses in the screen buffer of the Apple II do not correspond in a simple way with vertical pixel position. An external function YMEMLOC returns the actual memory address value corresponding to a given position on the screen. This function is stored in UTILITY.

UTILITY is a unit which contains some assembler code for rotate, setscale, setangle, spin ...

operations graphics in RASCAL. When using this unit, the INTERFACE unit must also be on disk. INTERFACE connects a Pascal host program to the assembler code in RASCAL.OBJ.

The test program contains several user options; their functions are described as follows:

NEWSCENE

Allow the user to create a new scene. If a working scene is on the screen, the program will ask the user to make sure he wants to discard the current scene. If the user types "Y", the process proceeds, otherwise this procedure ends. The user is asked to give a name to the new scene. When a scene is created, the boolean variable SCENE.EXIST will be set to true. The program can easily check if a scene is on the screen by this boolean value.

PAINTER

This mode is used to create boundaries to be filled. The whole procedure is in an endless loop until "Q" is typed. STICK is turned on to work as a paint brush. Paddle(0) and paddle(1) control the angle and scale of the STICK. The user can turn the knobs to get the

exact direction and length. The color of the STICK is not limited to white; the user can type "6" through ":" on the upper row of the keyboard to choose the desired boundary color. The STICK will be shown on the screen but will not be painted into the background. When a line is desired, the user can type "A" to add a line and move stick to the far end of the original line, or type "S" to add a line without moving the stick. The user can also use the keypad (the keys around "K") to move the stick one pixel at a time in any direction.

SAVEONE

This routine uses SAVESCENE from RASCAL to store the whole scene on disk. Each scene will take up 16 blocks of disk space.

GETONE

This routine uses GETSCENE to get a scene from disk. The user provides the name of the scene he wants, and the routine will show that scene on the screen. The scene to be gotten must be on the disk. An error message will show on screen when trying to get a non-existent scene.

KILLSCENE

This routine can clear the screen by filling with black. The program will recheck with the user before killing the whole scene.

DIRLISTING

This is a procedure in the DIRLST unit which can list the file names on your disk. The user has a choice of disk #4 and #5.

## 2.4 Data Structures

The program first creates a record "SCENE" which has the scene name and a boolean value to verify its existence. (See Figure 2.6) Then it calls PAINTER to draw the outline of a scene. When the outline of a scene has been drawn, the user can call a filling program to fill it. The filling program contains two parts: choosing the colors and filling the region.

```
SCENE=RECORD
   EXIST:BOOLEAN;
   NAME:STRING
   END;
```

Figure 2.6 Record of SCENE

The color choosing part will ask the user how many colors and what colors he or she wants. If the user types

"1": The program will ask which color is desired, then set MODE = 1 and put the color byte in COLOR1.

"2": The program will ask which two colors the user wants and if a halftone byte is desired. If the user answers "N", the MODE is set to 2 and the two colors are put in COLOR1 and COLOR2. If the answer is "Y", the MODE is set to 4.

"3": Three colors are chosen to put in COLOR1, COLOR2 and COLOR3. The MODE is set to 3.

The colors arrangement is described in Chapter 2.2.

The filling algorithm used in high-resolution is the same as in low-resolution, only the pixel unit is different. The filling program uses a whole byte as a unit to do the manipulating work. When filling starts, the process will keep filling to the right and left direction of the starting point until it hits a boundary byte. Every time a byte was filled, all the references that were used to decide the color byte are updated either incremented or decremented according to the filling direction, right or left.

There are often several bits remaining between the boundary bit and the byte already filled. To fill this gap, it must be understood that dots shown on the screen are placed in reversed order in memory. That is, the rightmost "low order" bit in a byte represents the leftmost dot on the screen of the same byte (See Figure 2.7). There are two different boundary filling methods depending on whether the high order bits or low order bits of the boundary byte are to be filled.

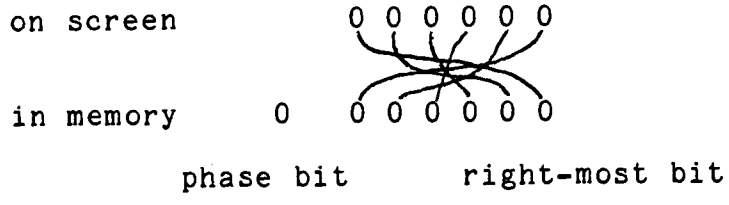



Figure 2.7 The screen-memory relationship

Low order bits (dots on the left side of the screen byte): The module locates the bit in the boundary byte representing the actual boundary by scanning from the right until a set bit is found. A fill byte is adjusted by truncating the bits higher than this bit in the desired color byte. This byte is then joined with the boundary byte by concatenating the old "outside" color and the new "inside" fill color. This is the byte that can fill the gap. See Figure 2.8 for detail.

```
screen          0 0 1 1 0 0 0
memory      0   0 0 0 1 1 0 0
color mask  0   0 1 0 1 0 1 0
                        <   Truncate the left side
remainder   0   0 0 0 0 0 1 0
                        +   Concatenate
boundary    0   0 0 0 1 1 0 0
                        =
new byte    0   0 0 0 1 1 1 0
screen          0 1 1 1 0 0 0
```

Figure 2.8 Low order bits

High order bits (dots on the right side of the screen byte): First the phase bit must be checked. If the boundary byte is blue or orange, then the phase bit is set. The unfilled gap will be located between the phase bit and the boundary bit. In this case, the boundary bit cannot be found by simply scanning from the left until a set bit is found. So when the phase bit is set, it must be reset to zero. The module locates the rightmost bit which is set, then truncates the lower bits in the color byte, then concatenates the boundary byte to form a new byte as in the low order case. This byte will fill the gap between the boundary and the filled byte. Figure 2.9 shows the high order bit filling.

```
screen          0 0 1 1 0 0 0
memory      0   0 0 0 1 1 0 0
color mask  0   1 1 1 1 1 1 1
                        >   Truncate the right side
remainder   0   1 1 1 0 0 0 0
                        +   Concatenate
boundary    0   0 0 0 1 1 0 0
                        =
  new byte  0   1 1 1 1 1 0 0
screen        0 0 1 1 1 1 1
```

Figure 2.9 High order bits

## 2.5 Problems

The prototype program uses procedures SAVESCENE and GETSCENE to store and fetch scenes to and from disk. The speed is very fast, but storing 4480 pixels of information takes 16 blocks of disk space. This causes problems on a small microcomputer. A time and space saving method for storing scenes, then, is desired.

Also, region-filling uses a lot of comparisons and loops to search for the next available line and unfilled byte. It is extremely slow when implemented in the Pascal language. A full screen region takes approximally 130 seconds to fill. A fast filling method, then, is desired.

# CHAPTER III

# IMPLEMENTATION

## 3.1 The Assembler/Pascal Relationship

To optimize code to meet time and memory requirements, assembler language was used to replace the principal procedures of the region-filling algorithm. The Apple computer uses the 6502 microprocessor which runs the Apple Pascal System 6502 Assembler.

A Pascal program involving an assembler routine has to declare it EXTERNAL as a FORWARD routine. Calls to the external routine use standard Pascal syntax and the compiler checks that each call to the routine agrees in type and number of parameters as declared before.

When a Pascal host program calls an external assembler language routine, parameters are pushed on to the stack as they are encountered in the calling statement. The first parameter is pushed on to the stack, higher byte first, then the second parameter, and so on. Parameters in functions are usually passed by values while they are passed by address in procedures. Two words (4 bytes) of meaningless value are also pushed on to the stack in functions. When all the parameters have been passed, the host program's return address is pushed on top of the stack. Records in Pascal programs are also passed by address. The elements of a record are stored in order as declared in the TYPE

statement. When referring to an element, you just add the offset to the address. The return address is pushed onto the stack when an assembler routine is finished. The host program will pop that address and execute the instruction at that location.

All registers are available in the Apple computer when running 6502 assembler language. Zero page memory locations 0 through 35 (Hex) are used as temporaries. They are often used by the P-machine as well. The user can use these locations as temporary storage but should not expect data to remain there for a long period of time. For instance, locations 20, 21, 22, 23 can not be used as long term storage, since they are often used by the system.

One of the most difficult things about programming in any computer assembly language is debugging. The Apple II+ computer does not provide any tools to trace a program, step by step, and to see where it goes wrong. Dr. Michael Moshell (12) supplies some macros, MONWHERE, MONCALL, and RETPAS to break out of an assembly program and later to return to it.

```
.MACRO MONWHERE
JSR     $1
PLA
STA     16
PLA
STA     15
.ENDM
```

```
.MACRO MONCALL
STA     12
STX     13
STY     14
STA     0C082
JMP     0FF59
.ENDM


.MACRO RETPAS
STA     0C088
RTS
.ENDM
```

Figure 3.1 Macros used in debugging

MONWHERE is used to determine where in memory the code file is loaded when executing MONWHERE . It stores the current address in location 15 and 16.

MONCALL stores the accumulator, registers X and Y into locations 12, 13 and 14. Then it enables the monitor ROM and jumps to the monitor.

RETPAS returns to the host program.

Since locations 12, 13, 14, 15 and 16 are used in these macros to store registers and the stopping address, they can not be used as variable storage.

When assembler programs execute these macros, the system goes to the monitor. Several monitor commands of the Apple II computer can be used to trace the program. Some useful commands are:

G: "go" transfers control to the machine language and executes it.

L: disassembles and displays 20 instructions.

adr1 . adr2: dumps the contents of memory between adr1 and adr2 on the screen.

adrs: val val ...: stores the values in consecutive memory locations starting at adrs.

The assembler routine is ready to be used after it has been assembled and linked to the host program. It is hard to write and to understand the assembler code, but the time it saves in execution of a program is worth it.

## 3.2 Experimental Implementation

The first assembler routine developed was to replace the FILLEFT and FILLRIGHT procedures in the original Pascal program. When a starting point is given, the filling routine fills the empty byte with the selected color, then expands that horizontal line to the right or left until it hits the boundary.

The host program passes nine parameters to the assembler routine. They are:

COLORs that are chosen in the host program.

Position of the starting point (X,Y).

Memory location of the starting point (X,Y), both on FOREGROUND and BACKGROUND.

An integer, IDE, that tells which direction to fill, right or left.

An integer, MODE, that tells how the region should be filled: single color, two colors mixed, three colors, or two colors not mixed.

When the host program calls the assembler routine, it passes the nine parameters through the stack. The assembler routine gets the parameters by popping them from the stack, and stores them in page zero. Since the parameters are passed by their location, not by value, the assembler routine can access the parameters only by indirect addressing. The steps that the assembler routine does are described as follows:

1. Check the byte is in between 0 and 39 of X axis, 0 and 159 of Y axis.

2. Calculate the references, XPOS, YPOS, PAR and YPA which decide the color combination.

3. Check the byte to see if it is a boundary. If the byte is not a boundary, then put the desired color at that byte location. If it is a boundary, then modify the color byte and fill the gap.

4. Update the references and the X value according

to IDE. If IDE is 0, it is a right fill, and every reference should be incremented. If IDE is 1, every reference should be decremented.

5. After filling the boundary, the routine terminates and returns to the host program.

Two methods are used to fill the gap between the filled byte and the boundary according to the filling direction (right or left). The method of filling the gap was described in Chapter II. Since the assembler language does not have a command to measure how many bits in a byte are filled, we have to use the shift command.

The shift command works by transferring the left or right-most bit into the carry flag in the processor status register. Using the BCS command, we can check when a set bit is shifted into the carry bit. If there are n bits left in the gap, the color byte will be shifted (8 - n) bits, then joined with the boundary bits to form a new boundary byte. This byte will fill the hole.

The experimental program worked using a Pascal host program to search for unfilled bytes and an assembler routine to fill the whole line within a boundary region. The running time is substantially reduced; a full-screen scene can be filled in 36 seconds by the assembler routine. The Pascal prototype took 130 seconds for the same scene.

## 3.3 Full Implementation

The final goal was to change the entire filling Pascal program into assembler language, step by step. Each subroutine in assembler has exactly the same function it had in Pascal. Even the names are the same. In a Pascal program, it is easy to set up a stack by keeping a linked list or a pointer. But there is no such advantage in assembler programs.

To solve this problem, a "private stack" was used. A block of memory starting from STPTR, at location 1D00 (Hex), is set up to store the parameters. Two macros, MPUSH and MPOP, are also formed to push or pop the parameters. (See figure 3.2 for macros)

An integer YSTASH is initialized to zero at the beginning of the program. When using MPUSH, YSTASH will be used as the offset to the beginning of STPTR. The parameters (X,Y) will be stored at a location a distance of YSTASH away from STPTR. Whenever a parameter is stored, the YSTASH will be updated. MPOP does the opposite thing; it retrieves parameters by the offset to the STPTR. Whenever a parameter is retrieved, the YSTASH will also be updated. The stack will be empty when YSTASH is zero.

```
.MACRO MPUSH
LDY    YSTASH     ;GET YSTASH IN Y REGISTER
INY               ;UPDATE THE YSTASH
LDA    %1+1       ;GET HIGH BYTE OF FIRST PARAMETER
STA    STPTR,Y    ;STORE AT YSTASH OFFSET TO STPTR
INY               ;UPDATE YSTASH
LDA    %1
STA    STPTR,Y
INY
LDA    %2+1
STA    STPTR,Y
INY
LDA    %2
STA    STPTR,Y
STY    YSTASH     ;STORE YSTASH
.ENDM


.MACRO MPOP
LDY    YSTASH     ;PUT YSTASH IN Y REGISTER
LDA    STPTR,Y    ;GET Y LOW BYTE
STA    %2
DEY               ;UPDATE YSTASH
LDA    STPTR,Y    ;GET Y HIGH BYTE
STA    %2+1
DEY
LDA    STPTR,Y
STA    %1
DEY
LDA    STPTR,Y
STA    %1+1
DEY
STY    YSTASH      ;STORE YSTASH
.ENDM
```

Figure 3.2 Macros MPUSH and MPOP

Six parameters are passed to the assembler routine from the host program. They are MODE, three COLORs and the starting point (X,Y). The memory location of (X,Y) is calculated inside the assembler routine. In the experimental program, the memory location is calculated by an external function YMEMLOC. It is easy to use .DEF and

.REF commands in the main assembler program to refer to YMEMLOC. But rather than link the UTILITY library to the program, which is already overcrowded, the function YMEMLOC was modified and written in a new assembler routine called YMEML. Function YMEML can calculate the memory location of the Y line, then add the X position and return the (X,Y) memory location.

Ordinarily, people would think that assembler code occupies less memory space than Pascal code. It is not true, at least not in the Apple computer. The Apple Pascal Compiler converts the text of a Pascal program into the compressed P-code version of the program. This P-code is the "machine language" of the Pascal pseudo machine, usually only 1/3 size of the corresponding assembler code.

When the filling program was changed to assembler language, the program occupied more memory space than the all-Pascal version. Stack overflow happened when the heap expanded and hit the program code in memory. (See figure 3.3 for the system memory map) Since every procedure and function in the program had been declared as a segment to reduce the size of memory, the only way to solve this problem was to eliminate unnecessary functions and procedures from the RASCAL unit. One third of the procedures of the RASCAL unit, not used in this particular program, were deleted for this reason. This solved the problem.

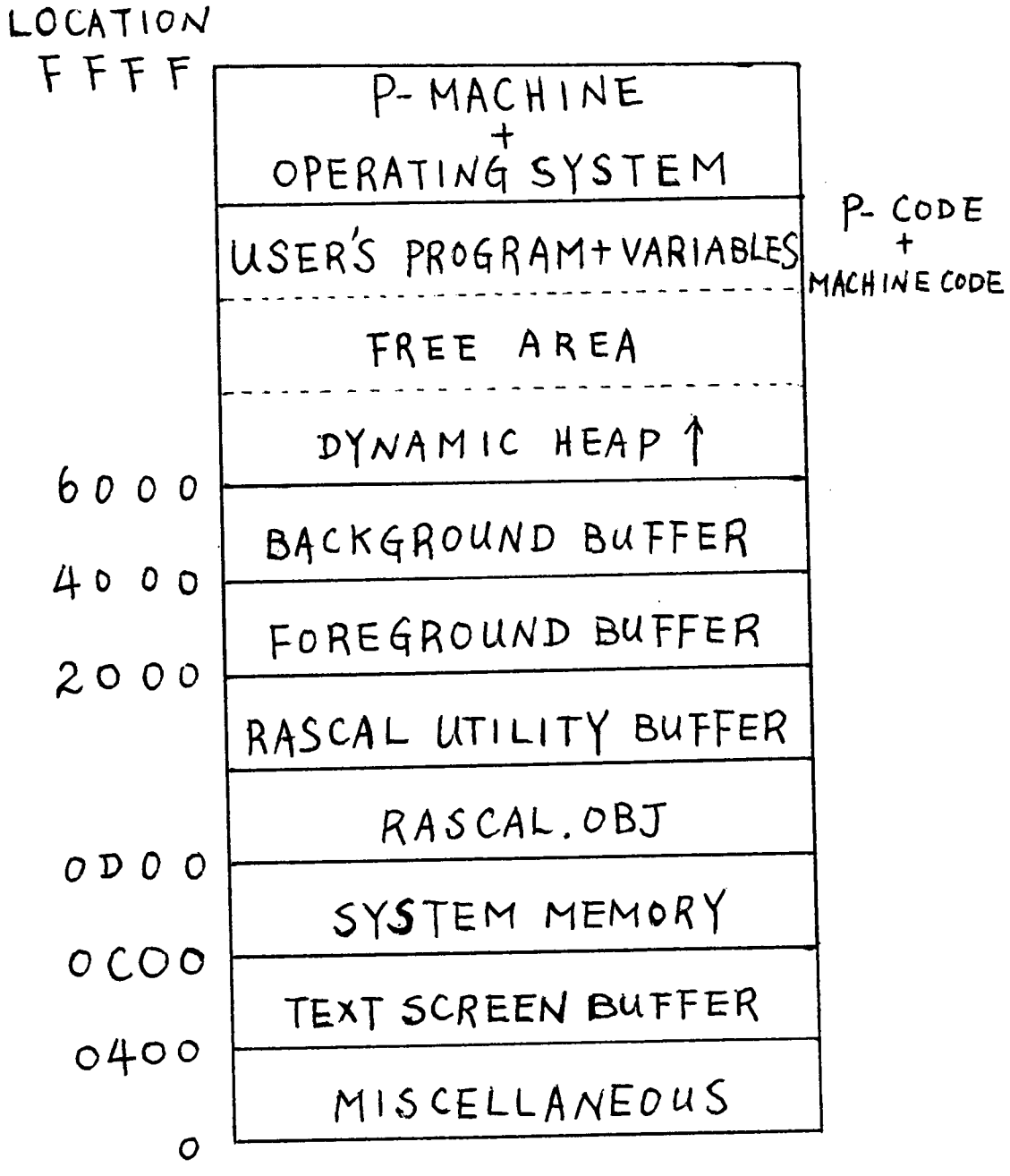



Figure 3.3 System memory map

## 3.4 Storing a Keystroke Record

To keep track of all inputs, the record for SCENEs has been modified to store more information. (See figure 3.4)

```
SCENE = PACKED RECORD
        EXIST : BOOLEAN;
        INOUT : BOOLEAN;
        NAME  : STRING ;
        LENGTH: INTEGER;
        LIST  : INNAME
      END;
```

Figure 3.4 New record of SCENE

INOUT is an indentifier to distinguish whether the input is from an existing record or is currently being built. LENGTH is the length of the input number of keystrokes. LIST is an array of 200 INLIST records. (See Figure 3.5) When a new SCENE is defined, EXIST and INOUT are set to true. When an old SCENE is fetched, INOUT is set to false. The LENGTH is 0 for a new SCENE; it is the number of INLIST records for an old SCENE. INLIST is a record of the inputs as follow:

```
INLIST = PACKED RECORD
     PADO : INTEGER;
     PAD1 : INTEGER;
     CHIN : CHAR
   END
```

Figure 3.5 Record of INLIST

PADO, PAD1 are the inputs from game paddles. CHIN is the input character from the keyboard.

There are two ways that the program gets input. One is from KEYJOY's READ command, the other is regular READ commands in the program. To unify the input command, a procedure GETIN (See Figure 3.6) was created to replace all the READ commands. GETIN checks whether this is a new scene being created by the user or an old scene description retrieved from the disk. If the scene is a new one, GETIN gets an input character from the keyboard and stores the character and the paddles values in an INLIST record. If it is a replay of an old scene, the INLIST pulls one character and paddles' values from the INLIST record. The program will use these data to perform the drawing.

```
PROCEDURE GETIN;
BEGIN
  IF SCENE.INOUT THEN
  BEGIN
    IF INPTR<=200 THEN
    BEGIN
      READ(KEYBOARD,CH);
      KJOYCHAR:=CH;
      SCENE.LIST[INPTR].PADD:=PAT1;
      SCENE.LIST[INPTR].PASS:=PAT2;
      SCENE.LIST[INPTR].CHIN:=CH;
      INPTR:=INPTR+1
    END ELSE BEGIN
      KJOYCHAR:='Q';
      WRITELN('WARNING: INPUTS EXCEEDS 200')
    END
  END ELSE IF INPTR<SCENE.LENGTH THEN
  BEGIN
    CH:=SCENE.LIST[INPTR].CHIN;
    KJOYCHAR:=CH;
    PAT1:=SCENE.LIST[INPTR].PADD;
    PAT2:=SCENE.LIST[INPTR].PASS;
    INPTR:=INPTR+1
  END ELSE SCENE.INOUT:=TRUE
END;
```

Figure 3.6 Procedure GETIN

An integer, INPTR, is used as the index of the array. When the GETIN is called, The INPTR will be updated. The final INPTR value of a scene is the LENGTH of that scene.

To save a scene is to store the array of the scene's records. Apple Pascal provides a command, BLOCKWRITE, to write an array onto a disk. The I/O-checking compiler option is turned off when writing to or reading from a disk. If any I/O error occurs, the error will be handled by our code and shown on the screen instead of crashing.

With the keystroke record demonstration program and the fast filling algorithm, the full implementation of the filling program has been finished. The program fulfills all the goals that had been set at the beginning of the project:

1. Draw an outline of a scene.
2. Choose colors as the user desires.
3. Quickly fill a region.
4. Store and retrieve a scene compactly.

The filling speed is very fast; a full-screen scene can be filled in less than 3 seconds. All the user inputs are stored in an array which can be written into a disk, occupying only one block of space.

# CHAPTER IV

# USER'S MANUAL

## 4.1 Preparation

When using this filling program, the user must have a RASCAL 1081 keychip (a hardware chip that comes with RASCAL) on the Apple II computer. This security device protects the RASCAL software from illegal duplication. There is a special SYSTEM.LIBRARY which must be present when the program is used.

Finally, the machine language file RASCAL.OBJ must also be present.

## 4.2 Drawing the Outline

All commands are invoked by single keystrokes. Most of the procedures are user friendly; they will double check with users to avoid accidental mistakes.

Figure 4.1 shows the state diagram of the whole program. When the program is executed, it enters a main state from which the user can select a command (sub-state) according to the main menu shown on the bottom of the screen. The main state is controlled from this menu:

K)ILL, N)EW, M)AKE, D)IR
F)ILL, G)ET, S)AVE, Q)UIT

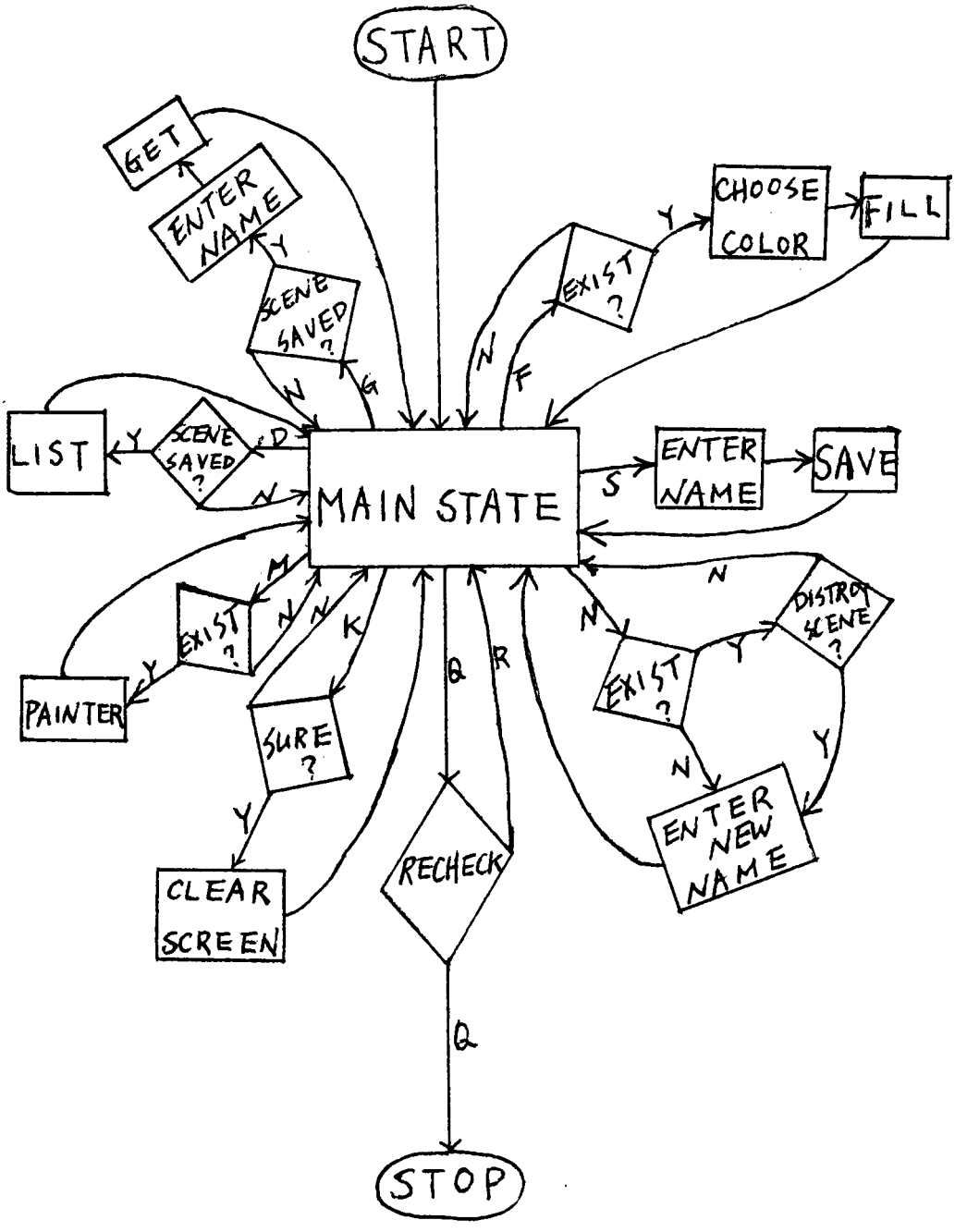



Figure 4.1 State diagram of the program

K)ILL. The user can discard the current working scene by typing "K". The screen will be filled with black color. INPTR (index of INLIST) of the current scene will be initialized to 0.

N)EW: Starting a new scene, the user should enter this sub-state. The program will ask the user to provide a name for the new scene, then return to the main state. If a scene is currently on the screen, the program will ask the user if he wants to discard the old scene. The user replies "Y" to start a new scene and "N" to return to the old scene.

M)AKE: Users enter this sub-state to draw the outline of a scene. The sub-state is controlled by this menu:

```
6 TO :  CHOOSE COLOR,  K)ILL SCENE
PADDLE(0) SET ANGLE,  (1) SET SCALE
A)DD LINE,  S)TART FROM ONE POINT, Q)UIT
```

Color control: The color of the stick can be changed by typing one of the color key set, "6", "7", "8", "9", "0" and ":" representing black, white, blue, green, violet and orange respectively.

Paddle(0): The orientation of the stick is set by the values of paddle(0). The angle of the

stick can be controlled in the range of -150 to 360 degrees.

Paddle(1): The scale of the stick is set by the value of paddle(1).

Keypad: The position of the stick on the screen can be moved by a "keypad" around the key "K". Typing the "I" key causes the stick to move upward one pixel, the "M" key causes the stick to move downward and to the right one pixel. The other keys ("U", "J", "L", "O", ",", ".") cause movement similar to their direction from "K".

Drawing: The user can type "A" to draw a line on the screen and move the stick to the end of it. Or, he can type "S" to draw a line but keep the stick as it was.

Redrawing: The user can type "K" to redraw the outline. The screen will be filled by black color, and the INPTR of the scene is reset to 0.

Quitting: To quit drawing the outline, a user can type "Q" to exit from the drawing procedure.

D)IR: The user can get a list of file names on disk by typing "D". When entering this sub-state, the program will ask whether the current scene has been saved. If the user types "Y", then he will have a

choice between #4 and #5 disk. When returning to the main state, the screen will be filled by black color. The INPTR of the scene is reset to 0.

S)AVE: To save the current scene on disk #4, the user can type "S" from the main state. The program will ask the name of the scene to be saved. This name need not be the same as the working scene.The reason for this is that a working scene can be developed into several different scenes and stored with different names. When the saving routine is finished, the program goes to the main state, and the working scene is still on the screen.

G)ET: The user can get a scene from disk by typing "G". The program will ask the name of the scene. When the user types in the name, the program will search the disk #4 for the scene. If the scene exists, the whole scene will be displayed on the screen by executing the commands stored in the array. If the scene does not exist, an error message will tell the user that the file was not found.

Q)UIT: When finishing the whole program, the user can type "Q" to exit. The program will recheck with the user to confirm it. If the user types "R", the

program returns to the main state, otherwise, the program terminates.

F)ILL: When the user wants to fill a region, he can type "F" to enter the filling program.

## 4.3 Fill Region

When entering the filling program, the first thing is to choose the number of colors to fill. The screen will show:

```
HOW MANY COLORS DO YOU WANT TO FILL?
1)ONE, 2)TWO, 3)THREE OR Q)UIT
```

If the user types "Q", the program will exit from color choosing. If "1" is typed, the program will ask what is the color to fill in. If "2" is typed, two colors will be asked by the program. The program will also ask if the user want to have halftone-fill or not. If "3" is typed, three colors will be asked by the program. When the program gets all the colors, the process goes to sub-state "FILL" and the screen shows:

```
PADDLE(0) SET ANGLE, (1) SET SCALE
USE KEYJOY TO MOVE STICK
TYPE "S" TO START FILLING
     "E" TO EXIT
```

A stick will appear on the left-bottom of the screen. The user can use paddle(0) and paddle(1) to adjust the direction and the length of the stick. When the stick points to the interior of the region to be filled, the user can type "s" to start filling. If adjusting the paddles is not enough to reach the region, the user can use KEYJOY (as described before) to move the stick. When the region has been filled, the user can type "E" to exit the filling program, or use paddles to point to another region to fill again.

# CHAPTER V

# CONCLUSIONS

## 5.1 Accomplishments

Region filling is a major step forward from simple line drawing. It provides a more vivid picture and enriches the flavors of a dull geometric shape. This paper investigates various algorithms, and chooses the most efficient one for implementation on the Apple II computer. A scene editor has been put together as a demonstration of the region filling implementation.

Speed is one of the key factors in evaluating a filling program. To make the process faster, all the filling part of the program has been implemented in assembly language.

To make a beautiful scene, color in a region is not limited to simple solid colors. The user has choices of colors and several multicolor techniques.

All commands in the demonstration program are invoked by single key strokes. Every input is kept in an array of records. A scene is saved by saving this record on disk. This record occupies only one block of space on disk. That is, a disk can store many scenes.

## 5.2 Future Development

Some further studies which will enhance the power of the project have been considered as follows:

1) Use a smaller unit in color representation.

   This project uses "BYTE" as the basic unit. The program will not fill areas of width less than a byte.

2) Speed up the drawing of a scene on the screen.

   The demonstration program is currently very slow in manipulating the stick. Most of the time is spent in waiting for the stick's showing.

3) Allow choice of color from a pre-defined menu.

   This will save user's time in choosing colors.

4) Adjust the range of KEYJOY.

   This will reduce the number of commands required for long-distance movement.

# BIBLIOGRAPHY

# BIBLIOGRAPHY


1. Lieberman, H. R., "How to Color in a Coloring Book", SIGGRAPH '78 Proceedings, published as Computer Graphics, 12(3), August 1978, pp. 111-116.

2. Pavlidis, T., "Contour Filling in Raster Graphics", SIGGRAPH '81 Proceedings, published as Computer Graphics, 15(3), August 1981, pp. 29-36.

3. Smith, A. R., "Tint Fill", SIGGRAPH '79 Proceedings, published as Computer Graphics, 13(2), August 1979, pp. 276-283.

4. Moshell, J. M., "Rascal & Interpas Reference Manual", Gentleware Corp., 1982.

5. Apple Pascal Language Reference Manual, Apple Computer Inc., 1980.

6. Pascal Operating System Reference Manual, Apple Computer Inc., 1980.

7. Apple II Reference Manual, Apple Computer Inc., 1981.

8. Wang, C. C., "RASDAZL: An Extended Graphics Tool for Use with the Rascal Animation System", M. S. Thesis, University of Tennessee, Knoxville, Tn., Dec. 1981.

9. Newman, W. M., Sproull, R. F. "Principles of Interactive Computer Graphics", Second Edition, McGraw-Hill Book Co., 1982.

10. Foley, J. D., Van Dam, A. "Fundamentals of Interactive Computer Graphics", Addison-Wesley Co., 1982.

11. Creative Computing, "Delta Drawing" advertisement, Ahi Computing Inc., February, 1983, pp. 16-17.

12. Moshell, J. M., "Assembly Language Programming with UCSD PASCAL", Compute Magazine, February, 1980.

# VITA

Chu Wang was born on January 5, 1951, in Taichung, Taiwan, Republic of China. He received his B. E. degree in Fishery Technology from The College of Marine and Oceanic Technology. After two years mendatory service in the Chinese Army, he worked with Cathay Pacific Airways in Taipei, as a ground staffer for one year.

In September 1978, he came to The University of Tennessee, Knoxville, as a graduate student pursuing his Master's degree in Food Technology. In the Spring of 1979, he transferred to Computer Science. He will receive his Master's degree in the Spring of 1983.