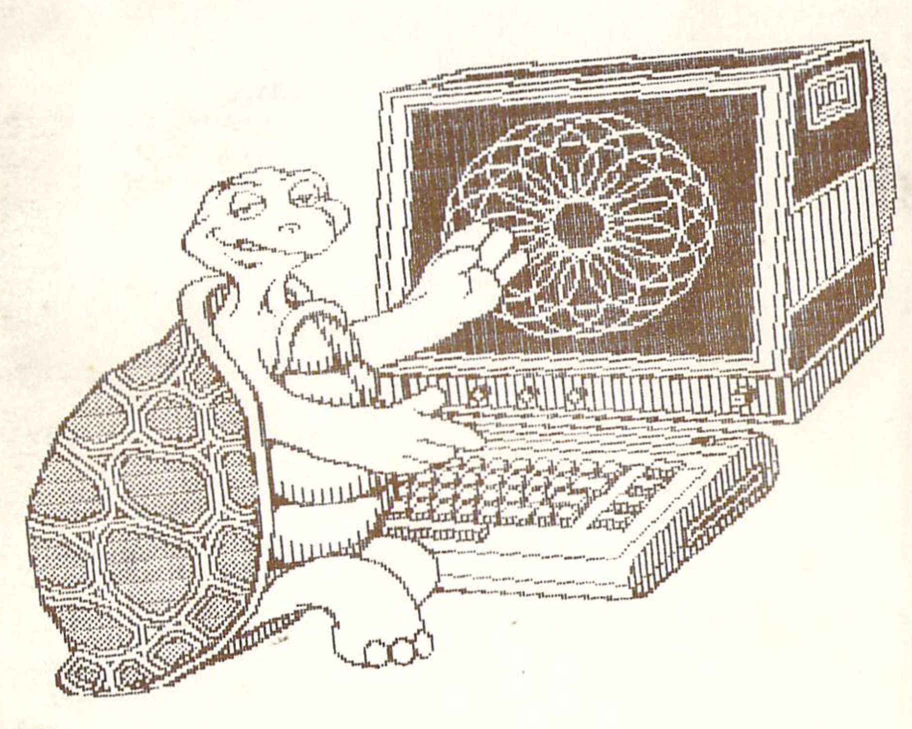
# THE AMAZING ADVENTURES OF CAPTAIN COMAL™
## BOOK 5

## CAPTAIN COMAL'S GRAPHIC PRIMER



MINDY SKELTON STEPS YOU THROUGH BOTH
TURTLE GRAPHICS AND SPRITE CONTROL

# Captain COMAL

Presents

A Graphic Primer

by

Mindy Skelton

ISBN 0-928411-04-4

The original manuscript of this book is copyright (C)1984 by Melinda Skelton.

This edition is copyright (C)1984 by Comal Users Group, U.S.A., Limited and published by permision of the author.

All rights are reserved. No part of this book may be reproduced in any way or by any means without permission of the publisher.

**COMAL USERS GROUP, U.S.A., LIMITED**
5501 Groveland Ter., Madison, WI 53716

**Trademarks:**

CAPTAIN COMAL of COMAL Users Group, U.S.A., Ltd
COMMODORE 64 of Commodore Electronics Ltd
Easy Script of Commodore Electronics Ltd

TABLE OF CONTENTS

Introduction .......................................... 5

Chapter 1: Some COMAL Basics ........................ 7
Getting Started
Loading from disk
Your first programs
1-Line numbering and renumbering
2-A short word about Procedures and Functions
3-Storing and Recalling Programs
Conventions of this Book

Chapter 2: Turtle Graphics .......................... 14
Why a Turtle?
Screen Coordinate System
Setting up the Screen
Meet your Turtle
Maneuvering the Turtle
Text on a Graphic Screen

Chapter 3: Sprite Graphics .......................... 29
What is a Sprite?
Making a Sprite
Moving a Sprite

Chapter 4: Program Listings ......................... 42

Chapter 5: Useful Procedures/Functions ............. 59

Glossary ............................................. 69

Appendix A: Defined Function Keys .................. 74

Appendix B: Sprite Chart ............................ 75

Appendix C: MULTI COLOR Sprites ..................... 76

Appendix D: Two More Ways to Design Sprites ........ 77

Index ................................................ 82

I would like to thank Andy Skelton, Len Lindsay, Colin Thompson, Kevin Quiggle, John McCoy, and David Stidolph for their help and encouragement in this project. I would also like to thank Donald Pipkin, Perry Brickley, and Jesse Knight for use of their procedures and functions in Chapter 5.

Special thanks go to Wayne Schmidt for his cover Doodle.

This primer was written in Easy Script, on a Commodore 64, and was printed on a Gemini-10 printer. Thank you, Precision Software, Commodore Business Machines, and Star Micronics.

# INTRODUCTION

Perhaps you are like me, and one of the reasons you bought your Commodore 64 was because you wanted to use the graphics capabilities that your fast-talking computer salesman showed you. Then you got your machine home and discovered the joys of attempting hi-res or sprite graphics on your Commodore. After wandering in the wasteland of PEEKS and POKES, VIC chips and sprite registers, not to mention the intricacies of bit-mapped graphics, you decided you'd shelve graphics for a little while, and somehow never got back to it. Maybe you mastered sprites and bit-mapped graphics, but would like a simpler way of doing all that plotting. Whichever camp you belong to, welcome to COMAL graphics.

COMAL (COMmon Alogrithmic Language) was designed by Borge Christensen and Benedict Loefstedt to be used by people who wanted to get the most from their computers, but who did not necessarily wish to become "hackers". With this goal in mind, the authors of COMAL made graphics easily accessible to even the casual user.

Included as part of version 0.14 COMAL, and as a quickly accessible "package" in version 2.0 COMAL, are easily used and understood commands for *turtle graphics* (which may already be familiar to you from LOGO or PILOT) and *sprite graphics*. If you really want to appreciate the ease of COMAL sprites, read through the sprite section in your *Commodore 64 Programmer's Reference Guide* (pp. 131-149).

This booklet will guide your through the basic use of the commands, give you some procedures and functions to use, and show you some demonstrations and programs to play with and change to suit your needs. There is also a listing of graphic commands. For further detail of these commands, see *COMAL FROM A TO Z*, by Borge Christensen, available from COMAL User's Group, USA.

If you are unfamiliar with COMAL, please read the first chapter for some hints on getting started. If you are an old hand at COMAL, just glance at the "Conventions of This Book" section to make sure we are on the same wavelength.

# Chapter 1
# Some COMAL Basics

## Getting Started

### LOADing 0.14 from disk

Put the COMAL disk in your drive. Type LOAD "BOOT*",8. Press RETURN. When your machine returns the READY prompt, type the word RUN, then sit back and wait for a few minutes, as COMAL loads.

Once the program is loaded, you have a choice of seeing several demos, reading some general information, or beginning to program. If you haven't seen the demos or read the information, I would urge you to do so. If you have, indicate that you want to begin programming by either entering

p (for (p)rogram)

or

c (for (c)omal)

depending on your version of the COMAL menu. Press RETURN. When you get your next prompt, enter NEW to clear out the demo program and the "Hi" program, and you are ready to start programming.

## Your first programs

### 1 - Line Numbering and Renumbering:

One of the niceties of COMAL is its automatic line numbering. Before you start writing a program, enter the command AUTO, then press RETURN. COMAL will supply you with line numbers in increments of ten (the default value), and all you have to do is type in the lines of code. To cancel the auto numbering hit RETURN twice. If you wish to continue the same program in the auto numbering mode, enter AUTO and the line number where you wish to begin. For example, if your last program line was 120, you would enter AUTO 130, and press RETURN.

You can also number in the regular way, by not using auto, and entering the line numbers yourself. For example, if you wish to insert some code betweem lines 10 and 20, just start your line number with 11 and number as high as 19, with no ill effects.

Let's say that you want to include more code between two lines than can be included in nine lines (or however many lines you have available). What can you do? COMAL has an automatic line renumbering command. If you type RENUM and press return, your entire program will be renumbered in increments of 10. If you wish to have different increments, you can specify both the increment and the beginning line number. For example RENUM 100 would change your first line number to 100, and increment by 10. RENUM 100,100 would change your first line number to 100 and increment by 100. RENUM 100 would leave your first line number as whatever it was, but would increment by 100. [Note: To get some experience with numbering and renumbering, look at program 1.1 in the sample program listings.]

2 - A short word on Procedures/Functions:

As you probably all know, COMAL is a structured language, and one of the things that makes a structured language, is the use of *procedures* rather than the GOTOs and GOSUBS you may recall from BASIC. The procedure is really the basis of all COMAL programs. With procedures you can break up a long, complicated program into several smaller pieces, each of which performs a particular task, or part of a task. The pieces are then used as needed by being accessed (or *called*) in the MAIN section of the program, or from another procedure. Procedures could often stand on their own, outside the program and so might be thought of as a mini-program.

Procedures have their own particular structure. They all begin with the word PROC, followed by the procedure name. If *parameters* (values for variables to be used in the procedure) are to be passed to the procedures, the name of the procedure is followed by information in parentheses (these are the *arguments* of

the procedure or function). When the procedure is called, if there are values to be sent to the procedure or function, a list of values is included in parentheses. These values are assigned, *in the order they are given*, to the arguments of the procedure or function. [Note: If you are unfamiliar with parameters, and passing values, please spend some time with one of the many books on COMAL programming currently available. We will be using parameters in many of the programs in this book. You are welcome to plunge ahead, and they may make sense as you see them used.] Parameters make procedures and functions extremely flexible. As an example, let's look at a procedure which draws a box.

```
proc box
 for i:=1 to 4 do
  forward 10
  right 90
 endfor
endproc box
```

Now let's add arguments to this procedure:

```
proc box (h,v,heading,length)
  moveto h,v
  setheading headading
  for i:=1 to 4 do
   forward length
   right 90
  endfor
 endproc box
```

By sending different values for h,v, heading and length we can draw boxes of different sizes, in different positions, and in different locations on the screen. For example, if, later in your program, you *called* the procedure by including the line:

```
box(0,0,0,20)
```

you would draw a box 20 pixels to a side, starting in the lower left corner of the screen. Calling the procedure causes the program to do whatever action is

specefied in the procedure. Calling the procedure by the inclusion of this line:

```
box(10,100,45,50)
```

would draw a diamond shape, 50 pixels to a side, 10 pixels up from the bottom of the screen, and 100 pixels over from the left edge. Please notice in the example that the first value in the call is assigned to the first variable, the second to the second, the third value to the third variable, and the fourth to the fourth. Be sure to give your values in the correct order.

Any commands or structures (such as if-then-else statements, case structures, for-endfor loops, etc.) can be used in a procedure or function. Just be sure to end your procedure with the word ENDPROC, so COMAL will know when your procedure is finished.

FUNCTIONS are a specialized kind of procedure. All functions are started with the word FUNC (rather than PROC). You can pass parameters to functions. In version 0.14 you cannot define string functions. Somewhere in the function, you must use the word RETURN to return a value for your function.

```
e.g. func getpencolor
       return peek(646)
     endfunc getpencolor
```

This function could be called with a line like

```
print getpencolor
```

The system would then execute the function getpencolor, and print the the value that is returned. Again, we will be using functions, and they will make more sense as you see them used.

### 3 - Storing and Recalling Programs:

You can list your program to the screen, either in whole or in part, by entering the command LIST.

LIST by itself will list the entire program. To see portions of your program, follow the LIST command by the range of line numbers you wish to see. For example, LIST -100, would list everything up to line 100. LIST 20-70, would list lines 20-70, inclusive. LIST 100- would list everything from line 100 on. The LIST command shows you the program complete with line numbers and indentation. The command EDIT will show you the program without indentation. Both can be slowed by holding down the control key, or stopped and restarted by pressing the space bar.

You are no doubt, familiar with saving programs in BASIC with the SAVE command. This command also works in COMAL. If you enter the command

```
SAVE "0:Myprogram"
```

you would store a PRG (program) file, called Myprogram, on your disk. In order to call this program back into memory at a later time, you would type

```
LOAD "0:Myprogram"
```

and press RETURN. If a file has been SAVEd, in can also be called up by using the command CHAIN, followed by the program name. The CHAIN command loads the file and runs it.

COMAL has another way of saving a file. You can LIST a file to disk. Entering the command

```
LIST "0:Myprogram.l"
```

stores Myprogram.l on disk as a SEQ (sequential) file. You might notice that this filename ends with '.l'. This is done, for your convenience, to help you remember which files are SAVEd versions, and which ones are LISTed. To recall a LISTed file, the command ENTER is used, followed by the program name.

```
ENTER "0:Myprogram.l"
```

LISTed files cannot be chained, but they can be manipulated like any other sequential file. For

example, if your word processor creates sequential files, you can read in a LISTed file for inclusion in a paper created on your word processor. (This really cuts down on typos!)

There's lots more that's involved in COMAL programming, but this is a book on graphics. For additional info on programming, refer to one of the many Captain Comal books.

## Conventions of This Book

Definitions:

(a) IMMEDIATE MODE: There will be times in this booklet when I will refer to immediate mode or programming mode. *Immediate mode* (at least while you are reading this booklet) means any commands you type in which are executed immediately. This can happen either in text or graphic mode. If your commands are contained in a program and are executed as the program runs, we'll consider that *programming mode*.

(b) DEFAULT: The term *default* value or status means the value of a variable or condition, as it is when the the system begins operation. For example, if a variable has a default value of 10, or the screen has a default color of blue, it means that when COMAL loads in and becomes operational, that variable has a value of 10, and the screen is blue, without your doing anything. If you want the value to be anything other that the default value, you must issue commands to change it.

(c) Q.V.: The abbreviation *q.v.* stands for the Latin phrase *quid vide*. It means literally, "which see", and in common usage, "see also". In context, it refers you to relevant sections, definitions and related commands.

Helpful things to know:

1- A reference to the CCGP disk is a reference to the *Captain COMAL's Graphic Primer* disk which

accompanies this book.

2- Keywords will be capitalized in text in order to make them visable. Don't attempt to enter the programs with the keywords capitalized, or COMAL will spit up.

3- Information to be supplied by the user will be enclosed in single < >. If, for example, you saw PENCOLOR<color> in a program, you would enter the actual color number you wish to use, *without* the enclosing < >s. For example, if you wanted a black pen, you would enter 'PENCOLOR 0'.

4- Information to be supplied by the user which requires the inclusion of parentheses will be enclosed in (). If you find any parameters enclosed in (), please be sure to include the ()s when you type in the code. For example, the information following the command DATACOLLISION must be in parentheses: DATACOLLISION (<sprite#>,<reset>). When you type this command, you would include the (), but not the <>.

5- Actions to be performed by the user will be enclosed in double << >>. For example, if you needed to press both the Commodore key and another key (in this example, A) simultaneously, you would see <<Com A>>. If you were to press the RETURN key, you would see <<RETURN>>.

6- In some of the program listings you will see the symbol '^'. In order to get this symbol, press the *up arrow* key, immediately to the left of the RESTORE key. The ^, in COMAL (as in Commodore Basic), is the exponentiation sign. For example, 2 is the exponent in the statement 8^2. 8^2 means 8 raised to the power of 2, or 8 squared, or 8*8. 2^8 would be 2*2*2*2*2*2*2*2.

# CHAPTER 2:
# TURTLE GRAPHICS

This chapter is divided into six sections. The first is just background on turtle graphics, and can be skipped if you're in a big hurry to get going. After each other section heading, there will be a list of commands which will be explained in that section.

## Why a turtle?

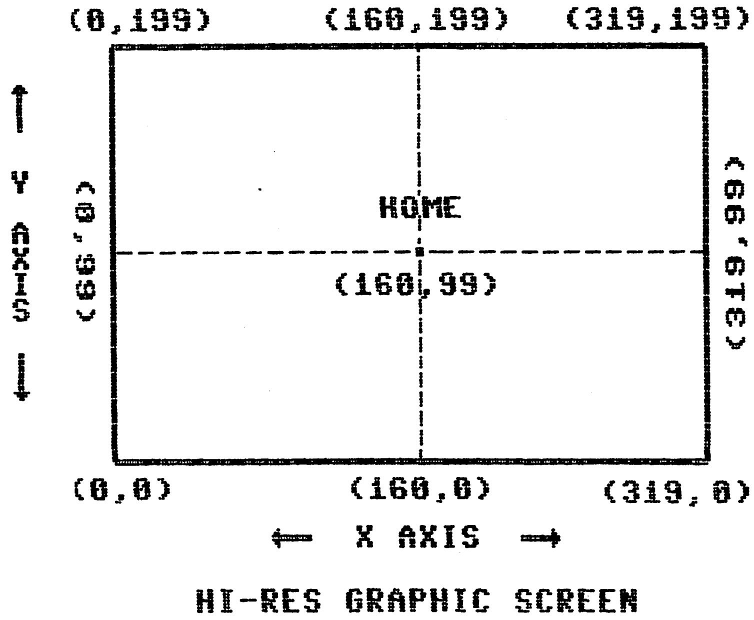
Turtle Graphics is a generic term for a system of computer graphics found in COMAL, as well as LOGO and PILOT. In this system, the user controls a triangular or turtle-shaped sprite (known as a "turtle") in order to "draw" in HI-RES or MULTI-COLOR graphics. Let me try to make that definition a little clearer.

About 17 years ago at Massachusett Institute of Technology (MIT), Seymour Papert and a group of people who were interested in artificial intellegence developed a language called LOGO. LOGO grew out of another language called LISP, which was and is used extensively in programs which simulate intelligence. Originally the language LOGO was used to control a slow-moving robot shaped somewhat like a turtle. The robot was attached to a mainframe computer and could be controlled by means of a "button box". The "turtle" could be made to move forward and back as well as rotate a specified number of degrees. Children (for whom the language was aimed) could work out problems with the real, tangible turtle, and later, as the language developed, could duplicate their solutions with the "turtle-shaped" cursor on the computer graphic screen. It took a while before home computers were powerful enough to hold the language, and by that time the robot was no longer a feature of the language. The name turtle has hung on to the turtle-shaped graphic cursor, long after most people stopped using the robot.

Screen coordinate system.

A number of the commands for moving your turtle depend on your understanding of a graphic coordinate system. If you remember Geometry and the Cartesian co-ordinate system, think of you screen as the upper right-hand section of a regular 4 section graph. The horizontal axis is labeled x, and the vertical axis is labeled y. Every spot on your graphic screen can be defined by an x and y coordinate pair. You use this system to tell the computer where you want something to be placed.

Until you get used to thinking of your screen as a grid, it may be helpful for you to have the following chart to refer to.




The left border of your screen is the 0 x axis, and the bottom of your screen is the 0 y axis. Where these axes intersect in the lower left corner of your screen is location 0,0. The place where the x and y axes intersect in the center of the screen (160,99) is the

turtle's HOME(q.v.).

Setting up the screen.
Commands: SETGRAPHIC, SETTEXT, BACKGROUND, BORDER, CLEAR, FULLSCREEN, SPLITSCREEN, FRAME

The first thing you have to do to use COMAL graphics, is *initialize*, or turn on, the graphic screen. To do this, either in a program or in immediate mode, issue the command

SETGRAPHIC <type>

followed by the *type* of graphic screen you want. SETGRAPHIC 0 gives you a HI-RES screen. SETGRAPHIC 1 gives you a MULTI-COLOR screen. It is only necessary to enter the type designation the first time you use the SETGRAPHIC command. After that, unless you are changing the type, the command SETGRAPHIC alone is sufficient.

You may be wondering what the difference is between HI-RES and MULTI COLOR screens. In HI-RES graphics, your screen is 200 (0 to 199) pixels high and 320 (0 to 319) wide. The screen is further divided into 1000 *blocks* (40 across, 25 down), each made up of 64 pixels (8 across, 8 down). That's a lot of pixels, and reasonably, this mode give you the greatest resolution. Each of those pixels can be any of the Commodore colors, but unfortunately, each block can only hold two colors, the background color and the color of the last pixel you turn on in that block. If, at a later time, you attempt to turn on another pixel in that same block, in a third color, each turned-on pixel in the entire block changes to the third color. This may not give you *exactly* the effect you had planned.

A partial solution to the problem is offered by MULTI COLOR graphics. In MULTI COLOR mode your screen is still divided into 1000 blocks, but each block is now made up of 32 pixels (8 across, 4 down). This gives you a screen 160 pixels by 200, and consequently lower resolution. Each line you draw is now two pixels wide. As a compensation for the loss of resolution, each block will now hold four colors, rather than two. Each

pixel in the block can now be one of the following

(a)screen color
(b)background color#1
(c)background color#2
(d)character color.

Each turned-on pixel in the entire block will still change color if you attempt to introduce a fifth color.
In order to get a feeling for these modes, type in the sample program dealing with HI-RES vs MULTI COLOR.
When the SETGRAPHIC command is issued, the graphic screen is turned on, the turtle is shown at coordinates 160,99 (see HOME), in size 10 (see TURTLESIZE), heading vertically upward (setting 0 - see SETHEADING). If HIDETURTLE (q.v.) is in effect, the turtle will not be visible. To return to the text screen from within your program, enter the command

SETTEXT

In immediate mode you have the choice of using SETTEXT or pressing the F1 function key. [Note: Appendix A gives a listing of the assigned actions of the function keys.] The two screens operate independently, so it is possible to switch from one to the other without losing either, or even to have two differnt things going on at the same time. For example, you could have direction on your text screen while a picture is being drawn on the graphic screen, then switch to graphics and a completed picture. Conversely you can have informatioin printed to the text screen while a picture is being drawn, which you can access later.
Once you have your screen initialized, you can change the color of the screen or border to any of the 16 Commodore colors. The background is set by using the command

BACKGROUND <color#>

where <color#> is any number between 0 and 15. If you are in HI-RES, the color won't change on the entire screen until you issue the command

CLEAR

CLEAR clears the graphic screen, but does not affect any sprites present. Until you issue the CLEAR, the background will only change in 8x8 pixel blocks around any lines or points you put on the screen (leading to some really strange images). The color of the screen border can be set using the command

```
BORDER <color#>
```

where <color#> is an integer from 0 to 15.

You can further choose to have the full graphic screen displayed by using the command

```
FULLSCREEN
```

or you can have a text window on the top two lines of the graphic screen by using the command

```
SPLITSCREEN
```

In immediate mode, using SPLITSCREEN, you can see your commands as you type them, which is sometimes helpful [Note: SPLITSCREEN doesn't work in programming mode]. Your commands are recorded on the text screen even in FULLSCREEN, so by toggling back and forth, you can keep track of your commands, even without the "window".

The FRAME command allows you to choose the area of your screen in which the pen will be active. A frame is designated by four coordinates;

```
FRAME <x1>, <x2>, <y1>, <y2>
```

<x1> and <y1> are the lower left corner of the frame, while <x2> and <y2> are the upper right corner. The default value of FRAME is 0, 319, 0, 199, giving a frame which covers the entire graphic screen. The turtle's pen will not leave any marks outside your frame, although, if the frame is smaller than the graphic screen, the turtle may move outside the frame and still be visible.

Let's look at short program (demo 2.1 on the CCGP disk) which uses some of the commands discussed above. Don't worry about MOVETO and DRAWTO; we'll get to them later.

```
 10 setgraphic 0            //this will initialized the
                              graphic screen to HI-RES
 20 fullscreen              //this selects a full
                              graphic screen without
                              text window
 30 for loop1 := 0 to 15 do
 40  background loop1       //this chooses a background
                              of color# loop1 (all
                              possible colors)
 50  clear                  //this implements the color
                              change selected in line
                              40
 60  for loop2 := 0 to 15 do
 70   border loop2          //this selects a border
                              color of color# loop2
                              (all possible colors)
 80   for pause := 1 to 90 do null
 90  endfor loop2
100 endfor loop1
110 frame 50,100,50,100     //this selects a frame
                              with lower left corner
                              at 50,50 and upper
                              right corner at 100,100
120 for scoot :=50 to 100 step 5 do
130  moveto scoot,40        //lines 120-150 draw lines
                              across the framed area.
                              The turtle is always
                              visible, but the lines
                              show only in the framed
                              area
140  drawto scoot,120
150 endfor scoot
160 frame 0,319,0,199       //returns the frame to
                              default setting
                              (entire screen)
170 for scat :=150 to 200 step 5 do
180  moveto scat,40         //lines 170-200 draw lines
                              across
```

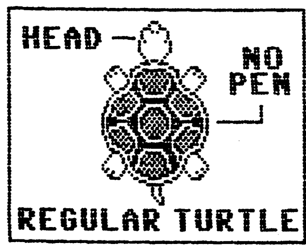
```
                                   the same area as above.
                                   The lines are now visible
                                   for their entire length,
                                   as the frame now covers
                                   the whole screen
190  drawto scat,120
200 endfor scat
210 while key$=chr$(0) do null  // line 210 causes
                                   the program to
                                   pause until a
                                   key is pressed
220 settext                   //you then return to text
                                   screen
```

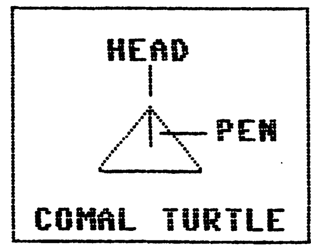
If you enter this program (remember you can use auto line numbering) and run it you should get some idea of the things you can do to set up your screen.

Now you have the basics. You can set up your screen just the way you want it. Now you can rest for a minute, as you get ready to...

Meet the Turtle.

Commands: TURTLESIZE, SHOWTURTLE, HIDETURTLE, SETHEADING, PENCOLOR, PENDOWN, PENUP, GETCOLOR




If you issued the SETGRAPHIC command, you should have a rather attractive triangle, similar to the one shown above on the right, sitting in the center of your screen. This is your turtle. (If your turtle looks like the one on the left, remove your pet from the screen.) Actually, the COMAL turtle is a sprite, sprite 7 to be precise. This sprite comes with lots of

predefined images, so as your turtle moves, its shape will change to reflect any rotation. Its size can also change. Until we consider sprites, we will not be able to make the turtle larger, but we can make it smaller right now. A command

TURTLESIZE <size>

where <size> is an integer from 0 to 10, will alter your turtle. The default size is 10 (the largest), but you can make it anything down to 0 (almost invisible). [Note: There is a function GETTURTLESIZE included in Chapter 5 which will tell you the current size of your turtle.]

Speaking of invisible, there is a command to make the turtle invisible

HIDETURTLE

When this command is in effect, the turtle can still draw things, but you can't see it. Drawings go faster when the turtle is hidden, and personally, for most uses I think the screen looks better without our little friend. To bring the turtle back from invisiblity, use the command

SHOWTURTLE

and BINGO there it is.

When your turtle first appears, the head will be pointing toward the top of your screen. This is heading 0, the default setting. The command

SETHEADING <degrees>

will change the turtle's heading to anything between 0 and 359. The turtle is moved in a clockwise direction, so heading 90 points to the right, heading 180 points down, etc. The turtle does not move to the position specified by <degrees> by turning through the intermediate settings. It goes directly to the specefied setting. If you ask for a heading of over 360 degrees, COMAL will subtract 360 from your request

until the result is less than 360, and set the turtle to the resultant heading. For example SETHEADING 400, would have the same effect as SETHEADING 40. The SETHEADING command always operates from heading 0. The current heading of the turtle when the SETHEADING command is issued has no effect on the outcome. That is, if your turtle is at heading 40, and you issue the command SETHEADING 50, the turtle will be set directly to heading 50 not heading 90.

Your turtle may be set to any of the 16 usual colors available to you, by using the command

```
PENCOLOR <color#>
```

Color# will be an integer from 0 to 15. The color of the turtle (even if HIDETURTLE is in effect) determines the color of the line drawn by the turtle's pen, and also the color of text plotted on the graphic screen, or on the text screen. A handy trick I use sometimes (since I find it very hard to remember what all 16 color number stand for) is to write a procedure which assigns the color number to the approprite color, for example WHITE := 1. Then in my programming, if I want to switch colors, I can issue commands using the name of the color I want, for example

```
PENCOLOR WHITE
```

This isn't required. It's just helpful. This procedure is called Color'select on your CCGP disk.

There is a function in chapter 5 called GETPENCOLOR which returns the current pencolor. You can use this function to store the current pencolor before you change colors, and then can call up the original color when you are through. There is also a built-in function

```
GETCOLOR <x>,<y>
```

which returns the color of location x,y. If there is no graphic data on the x,y point, GETCOLOR returns the color of the background. GETCOLOR cannot recognize sprites.

The line you see issuing from the head of your turtle is the "pen". The pen can be either up or down. The command

PENDOWN

"puts the pen on the paper", or causes the pen to become "active". If the pen is down, the turtle will usually leave a line when it moves (for an exception see MOVETO). The command

PENUP

"lifts the pen from the paper", or causes the pen to become "inactive". When the pen is up, the turtle will usually *not* leave a line (see PLOT for an exception).

Let's see what some of these command do (this program is demo 2.2 on your CCGP disk):

```
 10 setgraphic 0
 20 background 0
 30 clear
 40 for t'size := 0 to 10 do // show all possible
                                turtle sizes
 50  turtlesize t'size
 60  for wait := 1 to 20 do null  //pause
 70 endfor t'size
 80 for blink := 1 to 10 do  // "blink" turtle
 90  hideturtle
100 for wait := 1 to 75 do null
110  showturtle
120 endfor blink
130 for t'color := 0 to 15 do // show all possible
                                turtle colors
140  pencolor t'color
150  for wait := 1 to 90 do null
160 endfor t'color
170 for t'turn := 0 to 360 step 5 do // rotate turtle
180  setheading t'turn
190 endfor t'turn
```

Maneuvering the Turtle.

Commands: HOME, FORWARD, BACK, RIGHT, LEFT, MOVETO, DRAWTO, SETXY, PLOT, FILL

The turtle starts out in the "home" position. Issuing the command HOME will send your turtle from wherever it is to the center of the screen (coordinates 160,99). The turtle will be pointed toward the top of your screen (heading 0).

The command

FORWARD <distance>

will cause your turtle to move forward (toward his head) <distance> number of units (in this case pixels). For example, issuing the command FORWARD 20 will move your turtle 20 units forward. The command

BACK <distance>

will cause the turtle to move back (away from its head), <distance> number of units. For example, the command BACK 30 will move your turtle back 30 units.

The command

RIGHT <degrees>

will move the turtle <degrees> degrees to *its* right. For example if your turtle is pointing to the bottom of the screen, and you issue the command RIGHT 90, the turtle will turn 90 degrees to the right (clockwise) of its present position. Since the turtle is upsidedown, it will be turning to your left. This can get confusing, so be alert. By the same token, the command

LEFT <degrees>

will turn the turtle <degrees> degrees to the turtle's left (counter-clockwise). You may have noticed that all the movement commands we've seen so far, with the exception of SETHEADING, act in relation to the turtle's current heading.

The next three commands we are going to talk about now aren't concerned with current heading. They act on

the coordinate system discussed earlier, and move you fromyour current position, to a specified location, regardless of the turtle's heading.

```
MOVETO <x>,<y>
```

will move the turtle from it's current position to the coordinates specified by <x>,<y>, regardless of the current heading or position. This command does not leave a line regardless of whether the pen is up or down.

```
DRAWTO <x>,<y>
```

will draw a line from the current position to the specified coordinates. Due to a quirk in COMAL, the pen must be down for this command to leave a mark on the screen. The heading of the pen has no effect on the line.

```
SETXY <x>,<y>
```

does much the same thing as MOVETO, that is, it moves the turtle from the curent position to the specified position. If the pen is down, SETXY draws a line as it moves to the specified position.

[HINT: Due to the aforementioned quirk, Len Lindsay recommends leaving the pen down, and using moveto to move to particular points without leaving a line. It's so easy to forget to put the pen back down otherwise.]

The last two commands we will consider are PLOT and FILL.

```
PLOT <x>,<y>
```

marks the specified location by turning on the pixel at the x,y coordinates, in the currrent pencolor. PLOT works whether the pen is up or down, and leaves no line in moving to the specified location.

```
FILL <x>,<y>
```

fills the area containing the x,y coordinates. FILL

continues filling until it reaches the border of the screen, or the border of the frame or an area of another color. Be carefull. FILL can spill out of a space even one pixel wide and cover your whole screen.

One nice thing about turtle graphics is its variability. Once you have a procedure which draws a particular design, you can, by varying such things as the turtle heading and the x and y coordinates, draw your design anywhere in the screen. You can repeat it over and over to form new patterns. You can vary its size and color. You are limited only by your imagination. [Note: For some excellent examples of this, look at the programs called Arabesque1 thru Arabesque7 on the User Group disk #4.]

Now that we've read about the commands, let's try moving our turtle. This program will make a mess, but you can see how the different commands work.

```
 10 setgraphic 0
 20 background  0
 30 clear
 40 fullscreen
 50 for box := 1 to 4 do //draws box
 60  forward 20
 70  right 90
 80 endfor box
 90 fill 161,100          //fills box
100 moveto 50,50          //lines 80-90 move turtle to
                            new location and draw line
110 drawto 50,70
120 for dot := 1 to 20 do  plot  rnd(0,319),rnd(0,199)
    //line 100 plots 20 points at random locations
130 home
140 penup                //lines 130-160 show no marks
                           are left when in the penup
                           condition
150 left 90
160 forward 50
170 pendown
180 right 80
190 setxy 10,10
200 home
```

Text on a graphic screen.
Commands: PLOTTEXT

Text can only be plotted on a HI-RES screen, not on a MULTI COLOR. The command

PLOTTEXT <x>,<y>,<text string>,

plots the text contained in <text string> on the graphic screen, with the lower left corner of the first letter in position x,y. This is one time when COMAL outsmarts itself, and you. Text on your computer screen is 8 pixels high (remember those 8x8 blocks?). This gives you 25 lines of text on the screen, and, if you will, 25 "bottom lines" for text, something like invisible lined paper. The 40 characters you can put on a line are each an 8x8 block, so you also have 40 invisible vertical lines. If your coordinates would place your text "between" what the computer considers the lines for text, COMAL will "correct" you and put your text on the line equal to or less than your coordinates (either horizontally or vertically). For an example, change the value of *x* in line 40 of the program below to some value other than 8 (or some multiple thereof). For example, replace the 8 in line 40 with 13, or 5, or 20 and see what happens.

Although PLOTTEXT allows for only horizontal plotting, there are applications such as *PlotChar* by John McCoy, and *Gutenberg* by Kevin Quiggle, which allow you to print sideways, upsidedown and at angles. Copies of both are available from Comal Users Group,U.S.A.,Limited.

Here's a look at PLOTTEXT.

```
10 setgraphic 0
20 background 0
30 clear
40 x:=8
50 dim text$ of 14
60 counter := 10
70 text$ := "i speak comal!"
80 for show := 100 to 50 step - x
90  plottext counter,show,text$
```

```
100  counter := counter + x
110 endfor show
```

Well, that's all the commands for turtle graphics. There are some sample programs in chapter 5. Look at them, type them in and play around with them. You should be crankin' out graphics in no time.

# CHAPTER 3
# SPRITE GRAPHICS

This chapter is divided into three sections. The first section will explain a little about what a sprite is. The next two sections will show you how to design both HI-RES and MULTI COLOR sprites, and how to move your sprites. A list of any commands discussed in a section will be included after the section heading.

## What is a Sprite?

We all know from experiences with "arcade" style games that sprites can add real zest and sparkle (sometimes literally) to a program. They're the sort of thing to impress your friends with your virtuosity at the (Commodore) keyboard. However, if you've read the section on programming sprites in your *Commodore 64 Programmer's Reference Guide*, you may have come away thinking that sprites are beyond your grasp. All that stuff about VIC chips, POKES, sprite registers, color locations, etc. can look and sound like Greek to the beginning (and even the advanced) computer user. Forget all that junk! Sprites are easy in COMAL! Let me repeat that, "Sprites are easy in COMAL!" Now, keep that in mind, take a deep breath, and let's talk sprites.

Sprites are:

(a)characters in "A Midsummer Night's Dream"

(b)a special kind of user definable character which you can move around the screen to any location(s) you choose

(c)too much work to bother with.

If you answered (c), go back to paragraph 1. If you answered (a), you're right of course, but you may be disappointed in the rest of this chapter. If your answer was (b), give yourself a gold star, and keep reading.

Consider this paragraph the Cliff-Notes on sprites. Sprites are blocks of information which give a graphic representation of an image you create and store in th computer memory. Sprites are 24 pixels wide and 21

pixels long in hi-res, or 12 pixel pairs wide by 21 pixels long in multi-color mode. You can expand a sprite by a factor of 2, horizontally, vertically, or in both directions. You have up to 8 sprites at a time (numbered 0-7) to work with on your screen. You can assign a *priority* to your sprite to determine whether it will pass over or under other sprites or data on the screen. You can check to see if your sprite has collided with another sprite, or piece of data. You can turn off or "hide" srites when they are no longer needed.

So how do you do all this? In COMAL, you have a number of commands and procedures which will make the job easy for you. Let's take it one step at a time.

Making a Sprite:
DEFINE, IDENTIFY, SPRITECOLOR, SPRITESIZE, SPRITEBACK, HIDESPRITE, SPRITEPOS

Now that I've told you how easy sprites are, let me confuse you. You determine the design of your sprite by telling the computer whether you want a particular pixel turned off or on, and if on, what color you want it. There are a number of methods to do your designing, ranging from complicated to easy, but what all of them do is define a sprite *image*, rather than the actual sprite. HUH?

[Note: The following two paragraphs are really important, but they may not make much sense the first time you read them. If that is the case, take a deep breath and try again. Once this makes sense, you're more than halfway to using sprites.]

You need to make a distinction now between *sprites* and *sprite images*. A sprite *image* is a 64 character string expression which consists of 63 bytes of image information, and a 64th byte which determines whether the sprite is HI-RES or MULTI COLOR. The definition string tells the computer which blocks of the sprite you want on or off. You can have up to 56 (0 to 55) of these images stored in memory (54 if you are using the turtle). A *sprite* is a movable block, 21 pixels high by 24 pixels wide (unexpanded, HI-RES sprite). You can have up to 8 sprites (0 to 7) on the screen at one time

(7 if you are using the turtle). A sprite is given its shape by being assigned one of the stored sprite images. More than one sprite can have the same image, and during the course of a program the same sprite can have more than one image assigned to it (just not more than one at the same time).

You build a sprite *IMAGE* with the command

```
DEFINE <image#>, <definition string>
```

where <image#> is an integer from 0 to 55, and <definition string> is a 64 character string. You assign each sprite one of your stored images with the command

```
IDENTIFY <sprite#>, <image#>
```

where <sprite#> is an integer from 0 to 7, and <image#> is an integer from 0 to 55, representing one of your pre-defined images.

It's all very well to say "you build a 64 character string", but what are the mechanics of the construction? I'm going to talk about several methods (and there are probably other ways, so don't think you *have* to do it the way I say). The first thing you should do, no matter what system you use, is get some idea of what you want your sprite to look like. You have 21 x 24 pixels at your disposal, and each one can be either on or off. All you have to do is tell the computer what's what.

If the on-off business sounds like binary math, it is. One method of designing sprite images is to build up strings of 1s and 0s (binary notation). A 1 indicates the block is on, a 0, it's off. You then use a machine language subroutine to build the binary code into a definition string. This is quick and easy, but you have to have the machine language subroutine (unless you can write one for yourself). [Note: See Appendix D for more information. See also the program READ'SPRITE on your CCGP disk.]

Another *very* easy way to design your image is to use a sprite designer. A sprite designer is a program which allows you to move your cursor around a

representation of a sprite, turning the block on and off with the flick of a switch. You then save the result and load it into your program later. This is the easiest way of doing it, but again you need the sprite designer. [Note:The program FLURRY on your CCGP disk uses sprite data generated by a sprite designer. Look at appendix D for more information on sprite designers.]

Let's imagine for a minute that you have only yourself, your machine and COMAL. You can still design sprite images. You've probably seen sprite design grids (If you haven't, look at Appendix B). If you have, you've noticed that each of the 21 rows is made up of 24 columns. The 24 columns are divided into three sections of eight rows each. Each block in each row has a particular value associated with it. To generate the data to define your sprite image, you fill in the block you want turned on and add the values for the darkened blocks. You get a total for each of the three sections of each line, which give you the three data statements to define that line. Three totals per line, for 21 lines give you 63 of the 64 characters you need for your sprite definition string. The 64th character determines whether the sprite is HI-RES or MULTI COLOR. A 0 indicates HI-RES, anything but a 0 indicates MULTI COLOR. For now I will be dealing only with HI-RES sprites. MULTI COLOR will come a little later.

Let's design a sprite image. In this example, our sprite is going to be a solid block. That means every pixel in our image is on. I chose a block to simplify things. Since all the lines in our sprite are the same, all our data statements will be the same. Since all our pixels are on, all the numbers in our data statements will be 255. If you are unsure as to how we arrived at that figure, look at Appendix B. The values of the columns in each section are 1, 2, 4, 8, 16, 32, 64, 128 (a binary progression). To get a numeric data item, you add together the numbers associated with the colums containing pixels which are to be turned on. When you add all the values in a section together (as in our example where all the pixels are on), you get 255. The following code will design the image of a

block:

```
  10 setgraphic 0
  20 hideturtle
  30 dim image$ of 64
  40 for i := 1 to 63
  50  image$ := image$ + chr$(255)
  60 endfor i
  70 image$ := image$ + chr$(0)
```

Line 10 turns on the graphic screen. Line 20 dimensions the image string. Lines 30-50 build all of the definition string, except the character which designates HI-RES or MULTI COLOR mode. Line 60 adds this final character. We now have a completed image definition. All that remains is to assign our definition to one of our sprite images. This is done with the

```
DEFINE <sprite image#>,<definition$>
```

command. For example, by adding the line

```
  80 DEFINE 0,image$
```

to our program. we assign our defined image to sprite image 0.

This does not give us a sprite yet. In order to assign the image to a sprite, we need to use the

```
IDENTIFY <sprite#>,<sprite image#>
```

command. The IDENTIFY command assigns the sprite image specified to the desired sprite. By adding the line

```
  90 IDENTIFY 0,0
```

to our program, we have now IDENTIFYed sprite # 0 as having the shape of image #0.

We're not through yet. Now we need to assign a color to our sprite, decide on its shape, and put it on the screen. A HI-RES sprite can have two colors in it. One of the colors is the background color of the

screen. This is called the *transparent* color, since it really doesn't show. Your primary color is determined through the

```
SPRITECOLOR <sprite#>,<color#>
```

command. To make our sprite (sprite #0) red (color 2), we enter the line

```
100 SPRITECOLOR 0,2
```

Color becomes a bit more complicated for MULTI COLOR. If our sprite were MULTI COLOR, we would have a choice of four colors, rather than the two allowed in HI-RES. We would have the transparent color, the primary color and two background colors. Due to the way color is handled on the Commodore, all the MULTI COLOR sprites on your screen will have *common* background colors. These colors are set with the

```
SPRITEBACK <color#1>,<color#2>
```

command. Primary color is set with SPRITECOLOR. [Note: MULTI COLOR sprites are discussed in more detail in Appendix C.]

Now, do we want the sprite its normal size, or do we want it expanded horizontally or vertically? If we want it normal size, we either do nothing, or put FALSE or 0 for <x expand?> and <y expand?>. If we want to change the sprite size, we use the

```
SPRITESIZE<sprite#>,<x expand?>,<y expand?>
```

command. If you want the sprite to be double height, put TRUE or 1 in <y expand?>; for double width, answer TRUE or 1 to <x expand?>, and obviously, two TRUEs for expansion in both directions. Let's make our sprite double size by adding the following line to our program.

```
110 SPRITESIZE 0,1,1
```

The only thing left to do now is put the sprite on the

screen. Surprisingly, there is a command that does that very thing:

```
SPRITEPOS <sprite#>,<x>,<y>
```

For the time being, let's put the sprite halfway up the screen, on the left border.

```
120 SPRITEPOS 0,0,99
```

If all has gone well, you should have a double-size red block sitting halfway upthe left side of your screen (demo 3.1 on CCGP disk). Once your sprite is on the screen, you may notice that the CLEAR command has no effect on it. How do you get a sprite off your screen? The command

```
HIDESPRITE <sprite#>
```

will turn off your sprite. (Note: There is a procedure in chapter 5 called SHOWSPRITE which you can use to turn your sprite back on, or you can re-IDENTIFY your sprite.]

Perhaps you'd like to design a slightly more exciting sprite. Fine! I'll show you the way I do it ,with the clear understanding that there are many other ways of doing it. [Note: See Appendix C for directions on two other ways.]

The graph on the next page shows the sprite designer graph I normally use. You will notice that some of the squares are darkened. The values associated with these blanks have been added to give the three values listed for each line. These values will be made into data statements.

| SECTION A<br>1 2631<br>28426842 1 | SECTION B<br>1 2631<br>28426842 1 | SECTION C<br>1 2631<br>28426842 1 | SUM OF A | SUM OF B | SUM OF C |
|---|---|---|---|---|---|
| | | | 4 | 0 | 32 |
| | | | 10 | 0 | 80 |
| | | | 17 | 255 | 136 |
| | | | 33 | 0 | 132 |
| | | | 6 | 0 | 96 |
| | | | 8 | 0 | 16 |
| | | | 18 | 0 | 72 |
| | | | 33 | 129 | 132 |
| | | | 32 | 0 | 4 |
| | | | 32 | 24 | 4 |
| | | | 32 | 24 | 4 |
| | | | 16 | 68 | 8 |
| | | | 8 | 60 | 16 |
| | | | 6 | 0 | 6 |
| | | | 1 | 0 | 128 |
| | | | 0 | 255 | 0 |
| | | | 0 | 102 | 0 |
| | | | 0 | 102 | 0 |
| | | | 4 | 102 | 64 |
| | | | 5 | 102 | 160 |
| | | | 8 | 231 | 16 |

If you make the following changes and additions to the code you've already typed in, we'll have two sprites on the screen in no time. First, amend line 20 of our program to read,

```
  20 dim image$ of 64, image1$ of 64
```

Now add these lines to the program we are creating,

```
  55 read info
  56 image1$ := image1$ + chr$(info)
  75 image1$ := image1$ + chr$(0)
  85 DEFINE 1,image1$    //assign image1$ to image 1
  95 IDENTIFY 1,1        //assign image 1 to sprite 1
 105 SPRITECOLOR 1,7     //make sprite 1 yellow
 115 SPRITESIZE 1,0,0    //sprite 1 is unexpanded
 125 SPRITEPOS 1,200,199//sprite 1 is at top of screen
                           most of the way across
 400 data 4,0,,32,10,0,80,17,25,136 //data to define
                                       sprite#1
 410 data 33,0,132,6,0,96,8,0,16
```

```
420 data 18,0,72,33,129,132,32,0,4
430 data 32,0,4,32,24,4,16,66,8
440 data 8,60,16,4,0,32,3,0,192
450 data 0,255,0,0,102,0,0,102,0
460 data 2,102,64,5,102,160,8,231,16
```

If after renumbering (use RENUM), saving your code, and running it you have two sprites on your screen, well done (demo 3.2 on CCGP disk)! What's that you say? You want your sprites to move? Very well..

Moving a Sprite:
PRIORITY, SPRITECOLLISION, DATACOLLISION

Once your sprite is DESIGNed, IDENTIFYed, and put into position with SPRITEPOS, you're probably going to want to move it around. You may even want it do different things if it encounters other sprites or data. No problem.

Let's make the box move across the screen while the creature goes up and down. Go back to our program, and add the following lines, between lines between 180 and 190,

```
185 x := 0;y := 199
187 repeat
```

Now change lines 190 and 200 to read

```
190 SPRITEPOS 0,X,199
200 SPRITEPOS 1,200,Y
```

Now add the following lines between 200 an 210. [HINT: Type RENUM 100,100, then press <<return>>. Then type AUTO 2010, press <<return>>, and you can enter the lines.]

```
if x>319 then
 x := 0
else
 x := x+1
endif
if y<0 then
```

```
    y := 199
   else
    y := y-1
   endif
  until key$ <> chr$(0)
```

Now type RENUM again, and press <<return>>. If you've entered everything, your program listing should look like this:

```
  0010 setgraphic 0
  0020 hideturtle
  0030 dim image$ of 64, image1$ of 64
  0040 for i :=1 to 63 do
  0050  image$ := image$ + chr$(255)
  0060  read info
  0070  image1$ := image1$ + chr$(info)
  0080 endfor i
  0090 image$ := image$ + chr$(0)
  0100 image1$ := image1$ + chr$(0)
  0110 define 0,image$
  0120 define 1,image1$
  0130 identify 0,0
  0140 identify 1,1
  0150 spritecolor 0,2
  0160 spritecolor 0,2
  0170 spritesize 0,1,1
  0180 spritesize 1,0,0
  0190 x := 0; y := 199
  0200 repeat
  0210  spritepos 0,x,99
  0220  spritepos 1,200,y
  0230  if x>319 then
  0240   x := 0
  0250  else
  0260   x := x+1
  0270  endif
  0280  if y<0 then
  0290   y := 199
  0300  else
  0310   y := y-1
  0320  endif
  0330 until key$<>chr$(0)
  0340 data 4,0,32,10,0,80,17,25,136
```

```
0350 data 33,0,132,6,0,96,8,0,16
0360 data 18,0,72,33,129,132,32,0,4
0370 data 32,0,4,32,24,4,16,6,8
0380 data 8,60,16,4,0,32,3,0,192
0390 data 0,25,0,0,102,0,0,102,0
0400 data 2,102,64,5,102,160,8,231,16
```

Save this, run it, and you should have two sprites slipping across the screen. How 'bout that? (on your CCGP disk, this is demo 3.3)

That's almost it for sprites. There are a couple more commands to look at, and you're home free. The

```
PRIORITY<sprite#>,<p>
```

command determines whether sprite number <sprite#> passes over or under graphic data. If <p> is TRUE, the sprite will have lower priority, and will pass under the graphics. If <p> is FALSE, the sprite passes over. The priority among sprites themselves is pre-determined. The lower the number, the higher the priority. Sprite 0 has the highest priority (passes over all other sprites), sprite 7 has the lowest priority (passes under all other sprites).

In addition to passing over and under graphics and other sprites, sprites can recognize *collisions*. That is, you can detect if a particular sprite has come in contact with another sprite, or some graphic information, and, if you wish, perform some other action at that point. The commands for detection are

```
DATACOLLISION(<sprite#>,<reset>)
```

and

```
SPRITECOLLISION(<sprite#>,<reset>)
```

DATACOLLISION checks automatically each time a sprite is drawn, to see if the specified sprite has collided with a non-background sprite pixel or a non-background graphics pixel. When this happens DATACOLLISION returns a value of TRUE. If <reset> is set to TRUE, the collision detection flag is reset. If the flag is

reset, the statement will return TRUE the next time it is encountered; otherwise, it will return FALSE. SPRITECOLLISION works the same way, but it returns TRUE *only* if the specified sprite has collided with another sprite. The one exception to all this is in MULTI COLOR sprites. The first background color (01) will not cause a collision. For this reason, this color is often used to fill in areas of a sprite which you do not want to be detected as a collision.

To add a collision detection routine to our example, add the following lines between lines 220 and 230. Obviously, you will need to renumber again. Once again, type RENUM 100,100, press <<return>>, and type AUTO 2210, and press <<return>> one more time.

```
  if SPRITECOLLISION (1,1) then
   if x>319 then
    x:=0
   else
    x:=x+1
   endif
   if y<0 then
    y:=199
   else
    y:=y-1
   endif
```

Now, assuming that you renumbered by 100, add this line between line 3200 and 3300:

```
 endif
```

Type RENUM, press <<return>> once more, and, if all your code is correct, your large red block should wait and let your creature pass. (demo 3.4 on CCGP disk)

Congratulations! You can now use sprites with the best of them (well, maybe not the BEST...). Moving right along.. did I hear someone say MULTI COLOR? MULTI COLOR sprites are dealt with in Appendix C. [Note: You can simulate MC sprites with HI-RES sprites. If, for example, you superimposed a sprite shaped like a box with holes in it over our box sprite, the colors of the solid sprite would show through. Look at the

sprite example in Chapter 4 - demo 4.4]

After you feel comfortable with the material covered in the last chapters, take apart the programs in the next chapter. See how they work. Change them around. Have fun.

# SAMPLE PROGRAMS

This section consists of several programs for you to enter. Each of them gives you practice in a particular area, but any or all of them can be torn apart to see how a COMAL graphics program is put together. I hope you will find them to be useful. All these programs are included on your *Captain COMAL'S Graphic Primer* disk.

Numbering and Renumbering:
Demo 4.1 on the CCGP disk

Enter this program using automatic line numbering:

```
0010 proc init
0020  setgraphic 0
0030  hideturtle
0040  fullscreen
0050  background 0
0060  border 0
0070 endproc init
0080 //
0090 proc square(h,v,l) closed
0100  moveto h,v
0110  for loop :=1 to 2 do
0120   forward l*.75
0130   right 90
0131   forward l
0132   right 90
0140  endfor loop
0150 endproc square
0160 //
0170 proc poly(h,v,l,s) closed
0180  moveto h,v
0190  for loop :=1 to s do
0200   forward l
0210   right 360/S
0220  endfor loop
0230 endproc poly
0250 //
0260 init
```

```
0270 counter := 0
0280 repeat
0290  counter := counter + 1
0300  scale := rnd(5,40)
0310  l := rnd(5,40)
0320  s := rnd(3,8)
0330  h := rnd(0,320)
0344  v := rnd(0,200)
0350  pencolor rnd(1,15)
0360  ct := rnd(1,3)
0370  case ct of
0380   when 1
0390    poly(h,v,l,s)
0400    if counter>20 then
0410     counter := 0
0420     clear
0430    else
0440     null
0450    endif
0460   when 2
0470    square(h,v,l)
0480   otherwise
0490    null
0500  endcase
0510 until key$<>chr$(0)
0520 end
```

Now add the following lines to the program right after the procedure 'poly'. Try using RENUM 100,100 and inserting the new lines between lines 2500 and 2600.

```
//
proc circle (h,v,scale) closed
 moveto h,v
 aspect := 1.3
 y := 0
 first := true
 d'theta := .1
 c := cos(d'theta)
 s := sin(d'theta)
 n := 64
 for i := 1 to n do
```

```
   temp := scale * c - y * s
   y := y * c + scale * s
   scale := temp
   sx := aspect * scale + h
   sy := v - y
   if first then
    moveto sx,sy
    first := false
   else
    drawto sx,sy
   endif
  endfor i
endproc circle
```

Now add the following between lines 4700 and 4800:

```
  when 3
   circle(h,v,scale)
```

Now if you want, you can renumber the entire program.

HI-RES and MULTI COLOR:
Demo 4.2 on CCGP disk

Enter and save (or list) this program (remember auto line numbering can make life much easier). Then run it. Look closely for areas where the colors "bleed" into each other. Then change line 80 to read SETGRAPHIC 1. Run the program again and notice the differences.

```
0010 //  delete "0:house.1.1"
0020 //   m.skelton
0030 //  save "0:house.1.3"
0040 //
0050 //  procedures
0060 //
0070 proc init
0080  setgraphic 0
0090  hideturtle
0100  border 0
0110  background 0
0120  clear
0130 endproc init
```

```
0140 //
0310 proc ground closed
0320  pencolor 5
0330  x:=130
0340  for i:=0 to 48 do
0350   moveto 0,I
0360   if i mod 2=0 then
0370    pencolor 5
0380   else
0390    pencolor 13
0400   endif
0410   drawto x,i
0420   x:=x+.5
0430  endfor i
0440  x:=190
0450  for i:=0 to 48 do
0460   moveto 319,i
0470   if i mod 2=0 then
0480    pencolor 5
0490   else
0500    pencolor 13
0510   endif
0520   drawto X,I
0521   x:=x-.5
0522  endfor i
0530 endproc ground
0540 //
0550 proc house
0560  moveto 120,49
0570  setheading 0
0580  for i:=1 to 2 do
0590   forward 51
0600   right 90
0610   forward 80
0620   right 90
0630  endfor i
0640 endproc house
0650 //
0660 proc roof
0670  home
0680  moveto 120,100
0690  right 55
0700  back 5
```

```
0710  forward 54
0720  right 70
0730  forward 54
0740 endproc roof
0750 //
0760 proc door
0770  moveto 155,50
0780  setheading 0
0790  forward 20
0800  right 90
0810  forward 10
0820  right 90
0830  forward 20
0840  plot 162,60
0850 endproc door
0860 //
0870 proc window(x,y,h,w)
0880  setheading 0
0890  moveto X,Y
0900  for i:=1 to 2 do
0910   forward h
0920   right 90
0930   forward w
0940   right 90
0950  endfor i
0960  setheading 0
0970  moveto x+4,y
0980  forward h
0990  moveto x,y+4
1000  right 90
1010  forward w
1020 endproc window
1030 //
1040 proc trunk(x,y,l)
1050  moveto x,y
1060  setheading 0
1070  pencolor 10
1080  forward l
1090 endproc trunk
1100 //
1110 proc tree(l)
1120  pencolor 5
1130  if l>4 then
```

```
1140   right 45
1150   forward l
1160   tree(l*.75)
1170   if l<5 then blossom
1180   back l
1190   left 90
1200   forward l
1210   tree(l*.75)
1220   if l<5 then blossom
1230   back l
1240   right 45
1250  endif
1260 endproc tree
1270 //
1280 proc blossom
1290  pencolor 7
1300  x:=xcor
1310  y:=ycor
1320  plot x,y
1330  plot x,y+1
1340  plot x+1,y
1350  plot x+1,y+1
1360  pencolor 5
1370 endproc blossom
1380 //
1390 func xcor closed
1400  high'bit#:=peek(27255)
1410  return high'bit#*256+peek(27256)
1420 endfunc xcor
1430 //
1440 func ycor closed
1450  return 199-peek(27260)
1460 endfunc ycor
1470 //
1480 proc bird1(x,y)
1490  moveto x,y
1500  left 45
1510  forward 5
1520  back 5
1530  right 90
1540  forward 5
1550 endproc bird1
1560 //
```

```
1570 // main
1580 //
1590 init
1610 ground
1620 pencolor 1
1630 house
1640 roof
1650 door
1660 window(130,60,9,8)
1670 window(180,60,9,8)
1680 window(130,80,9,8)
1690 window(180,80,9,8)
1700 window(155,80,9,8)
1710 pencolor 2
1720 fill 125,51
1730 pencolor 1
1740 fill 125,101
1750 trunk(60,49,31)
1760 tree(20)
1770 trunk(60,49,26)
1780 tree(15)
1790 trunk(265,49,31)
1800 tree(20)
1810 trunk(265,49,26)
1820 tree(15)
1830 pencolor 1
1840 bird1(100,180)
1850 repeat
1860  null
1870 until key$<>chr$(0)
1880 settext
1890 print chr$(147) // shifted clear/home
1900 end
```

Turtle Graphics:
Demo 4.3 on CCGP disk

This program shows that fairly complicated drawings can be created using simple graphic commands.

```
0010 // delete "0:city'scape.1"
0020 //   m.skelton
0030 // save "0:city'scape.3"
```

```
0040 //
0050 proc init
0060  setgraphic 1
0070  fullscreen
0080  hideturtle
0090  border 0
0100  background 11
0110  clear
0120  pencolor 0
0130 endproc init
0140 //
0150 proc ground'floor(x,y) closed
0160  moveto x,y
0170  setheading 0
0180 endproc ground'floor
0190 //
0200 proc build(x,y,z,q)
0210  ground'floor(x,y)
0220  forward z
0230  right 90
0240  forward Q
0250  right 90
0260  forward z
0270 endproc build
0280 //
0290 proc build2(x,y,z,q)
0300  ground'floor(x,y)
0310  forward z
0320  right 90
0330  forward q/2
0340  left 90
0350  forward q/4
0360  right 90
0370  forward q/2
0380  right 90
0390  forward z+(q/4)
0400 endproc build2
0410 //
0420 proc build3(x,y,z,q)
0430  ground'floor(x,y)
0440  forward z
0450  for i:=1 to 2 do
0460   right 90
```

```
0470   forward q/5
0480   left 90
0490   forward q/5
0500  endfor i
0510  right 90
0520  for i:=1 to 2 do
0530   forward q/5
0540   right 90
0550   forward q/5
0560   left 90
0570  endfor i
0580  forward q/5
0590  right 90
0600  forward z
0610 endproc build3
0620 //
0630 proc build4(x,y,z,q)
0640  ground'floor(x,y)
0650  forward z
0660  right 90
0670  forward q/2
0680  right 90
0690  forward q/4
0700  left 90
0710  forward q/2
0720  right 90
0730  forward z
0740 endproc build4
0750 //
0760 proc build5(x,y,z,q)
0770  ground'floor(x,y)
0780  forward z
0790  right 90
0800  for i:=1 to 2 do
0810   forward q/4
0820   left 90
0830   forward q/4
0840   right 90
0850  endfor i
0860  forward q/4
0870  right 90
0880  forward z+q
0890 endproc build5
```

```
0900 //
0910 proc window(h,v)
0920  moveto h,v
0930  pencolor 7
0940  plot h,v
0950 endproc window
0960 //
0970 // main
0980 //
0990 init
1000 build2(0,0,150,40)
1010 build(41,0,110,30)
1020 build5(71,0,163,60)
1030 build(116,0,130,17)
1040 build(133,0,134,10)
1050 build3(143,0,100,25)
1060 build4(169,0,85,20)
1070 build2(189,0,170,35)
1080 build3(225,0,100,45)
1090 build5(271,0,80,40)
1100 build4(301,0,125,20)
1110 pencolor 0
1120 moveto 0,180
1130 fill 0,199
1140 pencolor 1
1150 plot 125,155
1160 plot 40,180
1170 plot 300,150
1180 plot 150,150
1190 plot 10,190
1200 plot 200,195
1210 plot 160,195
1220 plot 260,180
1230 pencolor 0
1240 for i:=120 to 123 do
1250 window(i,20)
1260 endfor i
1270 window(125,100)
1280 window(245,50)
1290 window(260,10)
1300 window(200,120)
1310 window(205,90)
1320 window(230,80)
```

```
1330 window(245,50)
1340 window(260,10)
1350 window(210,10)
1360 window(10,100)
1370 window(30,120)
1380 window(35,5)
1390 window(45,85)
1400 window(46,85)
1410 window(56,26)
1420 window(60,26)
1430 window(80,150)
1440 window(107,175)
1450 window(82,50)
1460 window(102,34)
1470 window(122,124)
1480 window(125,12)
1490 window(137,42)
1500 window(139,120)
1510 window(150,10)
1520 window(160,70)
1530 window(280,60)
1540 window(290,55)
1550 window(310,100)
1560 window(312,9)
```

Sprites:
Demo 4.4 on CCGP disk

I hope this program is helpful to you in setting up your sprite programs. Lines 1100 - 1720 do the sprite work here. Take a second to notice that the three different sprite shapes all have labels in the data statements. This is not required, but it helps you keep things straight.

```
0010 // save "0:butterfly.3"
0020 //   mindy skelton
0030 // delete "0:butterfly.1"
0040 //
0050 // procedures
0060 //
0070 proc init
0080  dim cloud$ of 64, shape$ of 64, shape1$ of 64
```

```
0090  setgraphic 0
0100  background 6
0110  hideturtle
0120  co:=0
0125  ko:=0
0130 endproc init
0140 //
0150 proc ground
0160  right 90
0170  moveto 0,0
0180  pencolor 13
0190  forward 320
0200  for i:=1 to 40 do
0201   moveto 0,i
0202   if i=40 then
0203    pencolor 5
0204   else
0205    pencolor 13
0206   endif
0210   forward 320
0220   moveto 0,i
0230  endfor i
0240 endproc ground
0250 //
0260 proc grass
0265  pencolor 5
0270  x:=rnd(5,315)
0280  moveto x,40
0290  setheading 0
0300  forward 10
0310  for i:=2 to 8 step 2 do
0320   x1:=x-i
0330   moveto x1,40
0340   forward (10-i)+rnd(0,3)
0350   x2:=x+i
0360   moveto x2,40
0370   forward (10-i)+rnd(0,3)
0380  endfor i
0390 endproc grass
0400 //
0410 proc flower
0420  x:=rnd(5,315)
0430  leaf(x)
```

```
0440  moveto x,40
0450  setheading 0
0460  forward 15
0470  moveto x,58
0480  petal
0490 endproc flower
0500 //
0510 proc petal
0520  old'color:=peek(646)
0530  pencolor 4
0540  for i:=1 to 16 do
0550   forward 8
0560   right 170
0570  endfor i
0580  pencolor old'color
0590 endproc petal
0600 //
0610 proc leaf(x) closed
0611  pencolor 5
0620  moveto x,40
0630  setheading 0
0640  for i:=1 to 2 do
0650   for j:=1 to 13 do
0660    forward 1
0670    right 7
0680   endfor j
0690   right 90
0700  endfor i
0701  fill x+2,42
0710  moveto x,40
0720  setheading 0
0730  for i:=1 to 2 do
0740   for j:=1 to 13 do
0750    forward 1
0760    left 7
0770   endfor j
0780   left 90
0790  endfor i
0791  fill x-2,42
0800 endproc leaf
0810 //
0820 proc sun
0830  pencolor 7
```

```
0840  aspect:=1.3
0850  h:=260
0860  v:=180
0870  r:=10
0880  y:=0
0890  first:=true
0900  d'theta:=.1
0910  c:=cos(d'theta)
0920  s:=sin(d'theta)
0930  n:=64
0940  for i:=1 to n do
0950   temp:=r*c-y*s
0960   y:=y*c+r*s
0970   r:=temp
0980   sx:=aspect*r+h
0990   sy:=v-y
1000   if first then
1010    moveto sx,sy
1020    first:=false
1030   else
1040    drawto sx,sy
1050   endif
1060  endfor i
1070  fill h,v
1080 endproc sun
1090 //
1100 proc butterfly
1110  for i:=1 to 63 do
1120   read xx
1130   shape$:=shape$+chr$(xx)
1140  endfor i
1150  for i:=1 to 63 do
1160   read xx
1170   shape1$:=shape1$+chr$(xx)
1180  endfor i
1190  shape$:=shape$+chr$(0)
1200  shape1$:=shape1$+chr$(0)
1210  define 0,shape$
1220  define 2,shape1$
1230  identify 0,0
1240  identify 1,2
1250  spritecolor 0,7
1260  spritecolor 1,2
```

```
1270  for i:=0 to 1 do spritesize i,0,0
1280 endproc butterfly
1290 //
1300 proc cloud'make
1310  for i:=1 to 63 do
1320   read code
1330   cloud$:=cloud$+chr$(code)
1340  endfor i
1350  cloud$:=cloud$+chr$(0)
1360  define 1,cloud$
1370  identify 4,1
1380  identify 3,1
1390  identify 5,1
1400  identify 6,1
1410  identify 7,1
1420  spritecolor 4,1
1430  spritecolor 3,1
1440  spritecolor 5,1
1450  spritecolor 6,1
1460  spritecolor 7,1
1470  spritesize 4,1,1
1480  spritesize 3,1,1
1490  spritesize 5,1,1
1500  spritesize 6,1,1
1510  spritesize 7,1,1
1520  spritepos 4,40,200
1530  spritepos 3,260,220
1540  spritepos 5,180,150
1550  spritepos 6,60,205
1560  spritepos 7,65,195
1570 endproc cloud'make
1580 //
1590 proc move'sprite
1600  for x:=1 to 300 do
1610   if (x mod 2)=0 then
1620    spritepos 0,x,100
1630    spritepos 1,x,100
1640   else
1650    spritepos 0,x,101
1660    spritepos 1,x,101
1670   endif
1680   for j:=1 to 3 do null
1690  endfor x
```

```
1700 endproc move'sprite
1710 //
1720 // main
1730 //
1740 init
1750 ground
1760 repeat
1770  flower
1780  co:=co+1
1790 until co>6
1800 repeat
1810  grass
1820  ko:=ko+1
1830 until ko>6
1840 sun
1850 cloud'make
1860 butterfly
1870 repeat
1880  move'sprite
1890 until key$<>chr$(0)
1900 //
1910 // data
1920 //
1930 cloud
1940 data 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
1950 data 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
1960 data 0,255,0,3,255,192,7,255,224,7,255,224,15,
         255,240
1970 data 127,255,254,255,255,255,127,255,254
1980 butterfly1
1990 data 192,0,3,248,0,31,184,0,29
2000 data 158,36,121,143,153,241,255,219,255
2010 data 255,255,255,225,231,135,255,231,255,248,
         231,31
2020 data 255,231,255,126,102,126,7,255,224,31,231,248
2030 data 63,126,252,127,231,254,103,165,230
2040 data 103,231,230,63,195,252,15,129,240,7,0,224
2050 butterfly2
2060 data 192,0,3,248,0,31,248,0,31
2070 data 254,0,127,255,153,255,255,219,255
2080 data 255,255,255,255,255,255,255,255,255,255,
         255,255
2090 data 255,255,255,127,255,254,7,255,224,31,255,248
```

```
3000 data 63,255,252,127,231,254,127,231,254
3010 data 127,231,254,63,195,252,15,129,240,7,0,224
```

# USEFUL PROCEDURES/FUNCTIONS

Grateful acknowledgement is made to David Stidolph, Donald Pipkin, John McCoy, Perry Brickley, Jesse Knight, Kevin Quiggle, and Len Lindsay for spending enough time dissecting COMAL to come up with these enhancements. A listing of most of these enhancements can be found in *COMAL Today Issue3*, published by COMAL Users Group, U.S.A., Limited. Listing is alphabetic.

## CIRCLE

There are several ways to correct COMAL's tendency to draw ovals instead of circles. Here are three.

Circle #1:

```
setgraphic 0
hideturtle
fullscreen
background 0
border 15
for radius:=10 to 100 step 10 do
 at'x0:=119
 at'y0:=100
 circle(radius,at'x0,at'y0)
endfor radius
end
proc circle(radius,x0,y0) closed
 // draw circle of given radius
 //   centered at x0, y0
 if radius<50 then
  n'steps:=36
 else
  n'steps:=72
 endif
 dtheta:=6.283192/n'steps
 cos'dtheta:=cos(dtheta)
 sin'dtheta:=sin(dtheta)
 first'x:=radius; first'y:=0
 move'to(x0+first'x,y0+first'y)
 prev'x:=first'x; prev'y:=first'y
 for point:=1 to n'steps do
```

```
   next'x:=prev'x*cos'dtheta-prev'y*sin'dtheta
   next'y:=prev'x*sin'dtheta+prev'y*cos'dtheta
   draw'to(x0+next'x,y0+next'y)
   prev'x:=next'x; prev'y:=next'y
  endfor point
 endproc circle
 proc move'to(x,y) closed
  adjust(x,y)
  moveto x,y
 endproc move'to
 proc draw'to(x,y) closed
  adjust(x,y)
  drawto x,y
 endproc draw'to
 proc adjust(ref x,ref y) closed
  scrunch:=1.34
  // note: using a scrunch factor
  //   corrects the difference in
  //   vertical and horizontal units/
  //   max x is reduced to  238
  //   horizontal center is 119
  x:=scrunch*x
 endproc adjust
```

Circle #2:
[Note: This procedure gives a particularly nice visual effect.]

```
// improved circle
setgraphic 0
hideturtle
fullscreen
pencolor 1
background 0
border 15
for radius:=10 to 100 step 10 do
 at'x0:=119
 at'y0:=100
 circle(radius,at'x0,at'y0)
endfor radius
end
proc circle(radius,x0,y0) closed
 // draw circle of given radius
```

```
 //   centered at x0, y0
 // j. michener algorithm
 x:=0; y:=radius
 d:=3-2*radius
 while x<y do
  plot'sym'points(x,y,x0,y0)
  if d<0 then
   d:=d+4*x+6
  else
   d:=d+4*(x-y)+10
   y:=y-1
  endif
  x:=x+1
 endwhile
 if x=y then plot'sym'points(x,y,x0,y0)
endproc circle
proc plot'sym'points(x,y,x0,y0) closed
 xx:=x; yy:=y
 xy:=y; yx:=x
 adjust(x0,y0)
 adjust(xx,yy)
 adjust(xy,yx)
 plot x0+xx,y0+yy
 plot x0+xy,y0+yx
 plot x0+xx,y0-yy
 plot x0+xy,y0-yx
 plot x0-xx,y0-yy
 plot x0-xy,y0-yx
 plot x0-xx,y0+yy
 plot x0-xy,y0+yx
endproc plot'sym'points
proc adjust(ref x,ref y) closed
 scrunch:=1.34
 // note: using a scrunch factor
 //   corrects the difference in
 //   vertical and horizontal units
 //   max x is reduced to  238
 //   horizontal center is 119
 x:=scrunch*x
endproc adjust
```

Circle #3:

```
proc circle(h,v,scale) closed
 moveto h,v
 aspect:=1.3
 y:=0
 first:=true
 d'theta:=.1
 c:=cos(d'theta)
 s:=sin(d'theta)
 num:=64
 for loop:=1 to num do
  temp:=scale*c-y*s
  y:=y*c+scale*s
  scale:=temp
  sx:=aspect*scale+h
  sy:=v-y
  if first then
   moveto sx,sy
   first:=false
  else
   drawto sx,sy
  endif
 endfor loop
endproc circle
```

## GETBACKGROUND

This function will return the number of the current background color.

```
func getbackground closed
 return peek(53281) mod 16
endfunc getbackground
```

## GETBORDER

This function will return the number of the current border color.

```
func getborder closed
 vic:=53248
 return peek(vic+32) mod 16
endfunc getborder
```

## GETPEN

This function will tell you if your pen is up or down by returning TRUE if it is down, and FALSE if it is up.

```
func getpen closed
 return peek(27333)
endfunc getpen
```

## GETPENCOLOR

This function will return the number of the current pen color.

```
func getpencolor closed
 return peek(646)
endfunc getpencolor
```

## GETSPRITECOLOR

This function will return the number of the current sprite color.

```
func getspritecolor(num) closed
 vic:=53248
 return peek(vic+num+39) mod 16
endfunc getspritecolor
```

## GETTURTLESIZE

This function returns the size of the turtle (0-10).

```
func getturtlesize closed //does not work if hideturtle
 t'size:=peek(27258)-10
 if t'size=130 then t'size:=10 //prior to turtlesize
                                           command
 return t'size
endfunc getturtlesize
```

## GRAPHICSTATE

This function will return TRUE if the graphic screen is MULTI-COLOR, and FALSE if the screen is HI-RES.

```
func graphicstate closed
 //return 0=hires, 1=multicolor
 return peek(27261)
```

```
endfunc graphicstate
```

## HEADING

This function returns the turtle's current heading as an integer from 0 to 359.

```
func heading closed
 a:=peek(27277); b:=peek(27278)
 case a of
  when 0
   return 90
  when 129
   return 89
  when 130
   return 88-(b div 64)
  when 131
   return 86-(b div 32)
  when 132
   return 82-(b div 16)
  when 133
   return 74-(b div 8)
  when 134
   return 58-(b div 4)
  when 135
   if b<53 then
    return 26-(b div 2)
   else
    return 386-(b div 2)
   endif
  when 136
   return 322-b
  when 137
   return (194-(b*2))-(peek(27279) div 128)
 endcase
endfunc heading
```

## HIDESCREEN

(see also SHOWSCREEN)
This procedure lets you hide the entire screen, much as you can SHOW or HIDE the turtle.

```
proc hidescreen closed
 vic:=53248
```

```
 x:=(peek(vic+17) mod 32) div 16
 if x then
  poke vic+17,peek(vic+17)-16
 else
  return
 endif
endproc hidescreen
```

POLYGON

This procedure allows you to draw a polygon of any radius.

```
proc polygon(sides,radius) closed
 //this routine draws a polygon
 // centered at the current
 // position with sides and radius
 // given
 pi:=3.14159265
 length:=2*radius*sin(pi/sides)
 angle:=180*(1-(sides-2)/sides)
 penup
 forward radius
 right 90+angle/2
 pendown
 for side:=1 to sides do
  forward length
  right angle
 endfor side
 right 90-angle/2
 penup
 forward radius
 right 180
 endproc polygon
```

SHOWSCREEN

(see also HIDESCREEN)
This procedure lets you show the entire screen, much as you can SHOW or HIDE the turtle.

```
proc showscreen closed
 vic:=53248
 x:=(peek(vic+17) mod 32) div 16
 if x then
```

```
  return
 else
  poke vic+17,peek(vic+17)+16
 endif
endproc showscreen
```

SHOWSPRITE

This procedure allows you to turn on a sprite previously hidden with HIDESPRITE, without knowing the identity of the sprite.

```
proc showsprite(num) closed
 vic:=53248
 x:=2^num
 y:=(peek(vic+21) mod (2*x)) div x
 if not y then poke vic+21,peek(vic+21)+x
 y:=(peek(27276) mod (2*x)) div x
 if not y then poke 27276,peek(27276)+x
endproc showsprite
```

SPRITESTATE

This function will return TRUE if the specified sprite is visible, and FALSE if the sprite is hidden.

```
func spritestate(num) closed
 x:=(peek(27276) mod 2^(num+1)) div 2^num
 return x
endfunc spritestate
```

SPRITEXCOR

This function returns the x coordinate of the specified sprite.

```
func spritexcor(num) closed
 x:=(peek(53264) mod 2^(num+1)) div 2^num
 return x*256+peek(53248+2*num)
endfunc spritexcor
```

SPRITEXSIZE

This function tells you if the specified sprite is double height by returning TRUE if the sprite is expanded on the x coordinate (width).

```
func spritexsize(num) closed
 x:=(peek(53277) mod 2^(num+1)) div 2^num
 return x
endfunc spritexsize
```

SPRITEYCOR

This function returns the y coordinate of the specified sprite.

```
func spriteycor(num) closed
 return peek(53249+2*num)
endfunc spriteycor
```

SPRITEYSIZE

This function tells you if the specified sprite is double height by returning TRUE if the sprite is expanded on the y coordinate (height).

```
func spriteysize(num) closed
 y:=(peek(53271) mod 2^(num+1)) div 2^num
 return y
endfunc spriteysize
```

TURTLESTATE

This function tells you if the turtle is visble or not by returning TRUE if SHOWTURTLE is in effect, and FALSE if HIDETURTLE is in effect.

```
func turtlestate closed
 //return 1=turtle on, 0=turtle off
 return peek(27295)
endfunc turtlestate
```

XCOR

This function returns the x coordinate of the turtle.

```
func xcor closed //return x
 high'bit#:=peek(27255)
 return high'bit#*256+peek(27256)
endfunc xcor
```

## YCOR

This function returns the y coordinate of the turtle.

```
func ycor closed //return y
 return 199-peek(27260)
endfunc ycor
```

## GLOSSARY

Here is an alphabetic listing of the COMAL graphic commands, along with the syntax and a *brief* explanation of each. More complete explanations of each command will be found in Chapters 2 (turtle) and 3 (sprites). Turtle commands and sprite commands are listed separately.

Since it may appear that some of the commands are listed twice on a given line, it may be of help for you to know all commands are listed in the following format:

COMMAND: SYNTAX: DEFINITION

### TURTLE COMMANDS

BACK: BACK <distance>: Moves turtle backward <distance> units from the direction of the point of the turtle. If pen is down (see PENDOWN), the turtle leaves a line in the current pencolor (see PENCOLOR).

BACKGROUND: BACKGROUND <color>: Sets the background to the specified color. In hi-res, the change does not take effect until a CLEAR command has been executed (see CLEAR).

BORDER: BORDER <color>: Sets the border to the specified color.

CLEAR: CLEAR: Clears the graphic screen, but does not remove sprites (see HIDESPRITE).

DRAWTO: DRAWTO <x>,<y>: Draws a line, in the current pencolor, from the present location to the position (<x>,<y>). Due to a bug in COMAL, this command requires that the pen be in the 'down' position in order to draw a line. If the pen is 'up', your turtle will move to the specified position, but no line will be drawn.

FILL: FILL <x>,<y>: Fills, with the current pencolor, an area containing the specified location. Fills until

encountering either a boundary formed by points of another color or points on the edge of the present frame (see FRAME).

FORWARD: FORWARD <distance>: Moves the turtle forward <distance> units. Forward is the directioin pointed to by the head (point) of the turtle. If the pen is down, a line will be left in the current pencolor.

FRAME: FRAME<x1>,<x2>,<y1>,<y2>: Defines the area (frame) within which graphic activity will occur. No drawing will take place outside the framed area, even though the turtle will be displayed. <x1>,<y1> determine the lower left corner of the frame. <x2>,<y2> determine the upper right corner. The default is a frame covering thre entire graphic screen with values of FRAME 0,319,0,199.

FULLSCREEN: FULLSCREEN: Shows the entire graphic screen without the window of text on the upper two lines (see SPLITSCREEN).

GETCOLOR: GETCOLOR <x>,<y>: Returns the color of the specified point. Returns background color if no graphic pixel on. Not affected by sprites.

HIDETURTLE: HIDETURTLE: Makes the turtle invisible and speeds up some graphics. (See SHOWTURTLE)

HOME: HOME: Moves turtle to center of screen (160,99), with the head vertically upward, at heading 0. (See SETHEADING)

LEFT: LEFT <degree>: Moves the turtle's head <degree> degrees to the turtle's left (counterclockwise).

MOVETO: MOVETO<x>,<y>: Moves the turtle to the specified coordinates. Does not leave a line on the screen.(See SETXY)

PENCOLOR: PENCOLOR <color>: Sets the pen to color <color>. <color> is an integer from 0 to 15. The cursor will appear in the specified color, as will text.

PENDOWN: PENDOWN: "Lowers" the turtle pen. Causes the pen to leave a trace on the screen as long as the pencolor is different from the background color and as long as the turtle is within the current frame. (See PENUP)

PENUP: PENUP: "Raises" the turtle pen. No mark is left by the turtle. PLOT and DRAWTO (q.v.) still function. (See PENDOWN)

PLOT: PLOT <x>,<y>: Marks position <x>,<y> in the current pencolor.

PLOTTEXT: PLOTTEXT <x>,<y>,<text$>: Plots text string <text$> on the HI-RES graphic screen, in the current pencolor. The lower left corner of the text expression is placed at <x>,<y>. <x> and <y> will be adjusted to the greatest multiple of 8 less than or equal to the specified values.

RIGHT: RIGHT <degrees>: Turns the turtle's head <degree> degrees to the turtle's right (clockwise).

SETGRAPHIC: SETGRAPHIC <type>: Initializes the graphic screen. Used intially with <type> declaration. <Type> of 0 sets screen to HI-RES, <type> of 1 sets the screen to MULTICOLOR. Does not affect the text screen.(See SETTEXT)

SETHEADING: SETHEADING <degree>: Sets the turtle to the heading specified by <degree>. The heading is calculated clockwise from the 0 degree heading (vertical) rather than from the current heading of the turtle.

SETTEXT: SETTEXT: Displays the text screen and hides the graphic screen. Does not clear the graphic screen or prevent graphic instructions from being carried out on the graphic screen.(See SETGRAPHIC)

SETXY: SETXY <x>,<y>: Moves turtle to the designated coordinates, drawing a line if the pen is down. (See MOVETO)

SHOWTURTLE: SHOWTURTLE: Causes the turtle to become visible on the graphic screen. This is the default setting. (See HIDETURTLE).

SPLITSCREEN: SPLITSCREEN: Causes a text "window" to be displayed on the top two lines of the graphic screen.(See FULLSCREEN)

TURTLESIZE: TURTLESIZE <size>: Sets the size of the turtle to <size>. <size> is an integer from 0 to 10. The default size is 10 (largest).

## SPRITE COMMANDS

DATACOLLISION: DATACOLLISION (<sprite>,<reset>): A function which returns a value of TRUE if the sprite number <sprite> collides with graphic information (e.g., text, screen graphics or sprites). The collision detection is done automatically each time a sprite is drawn. If <reset> is given a value of TRUE(1), the collision flag will be reset. If <reset> is given a value FALSE(0), the collision flag is stored for use with the next DATACOLLISION statement.(See)

DEFINE: DEFINE<image#>,<definition$>: Assigns the string <definition$>, which contains the 64 characters of the sprite image, to image<image#>. <image#> is an integer from 0 to 55.

HIDESPRITE: HIDESPRITE <sprite#>: Causes sprite <sprite#> to become invisible. (See procedure SHOWSPRITE)

IDENTIFY: IDENTIFY<sprite#>,<image#>: Gives sprite<sprite#> the image contained in image number <image#>. More than one sprite may share the same

image. One sprite may, at different times, have different images.

PRIORITY: PRIORITY<sprite#>,<p>: If <p> is FALSE (0), sprite<sprite#> has a higher priority than screen graphics. This means the sprite will pass over the graphics. If <p> is TRUE (1), the sprite will pass under the graphics. The priority of sprite against sprite is determined by the number of the sprite, with sprite 0 having the highest priority.

SPRITEBACK: SPRITEBACK <color1>,<color2>: Sets the common colors for multicolor sprites. <color1> and <color2> are integers from 0 to 15.

SPRITECOLLISION: SPRITECOLLISION (<sprite#>,<reset>): This function returns TRUE if (and only if) a sprite <sprite#> has collided with another sprite. If <reset> has a value of TRUE or 1, the collision flag is reset. (See DATACOLLISION)

SPRITECOLOR: SPRITECOLOR <sprite#>,<color>: Makes sprite number <sprite#> color <color>. <color> is an integer from 0 to 15.

SPRITEPOS: SPRITEPOS <sprite#>,<x>,<y>: Sets the position of sprite number <sprite#> to be such that the upper left corner of the sprite is at position <x>,<y>.

SPRITESIZE: SPRITESIZE <sprite#>,<xsize>,<ysize>: If <xsize> is TRUE (1), sprit number <sprite#> is expanded 2 times in width. If <ysize> is TRUE (1), sprite number <sprite#> is expanded 2 times in height.

# APPENDIX A

Defined Function Keys:

F1 .......................... TEXT SCREEN

F3 .......................... SPLITSCREEN

F3 .......................... FULLSCREEN

These three predefined function keys are available for use after the graphic screen has been initialized. They can be accessed in immediate mode from either the text or graphic screen.

F1 - has the same effect as issuing the command SETTEXT.

F3 - has the same effect as issuing the commands SETGRAPHIC and SPLITSCREEN.

F5 - has the same effect as issuing the commands SETGRAPHIC and FULLSCREEN.

## APPENDIX B:
## Sprite Chart

Here is a chart you can use to sketch out your hi-res sprites, if you want to try one without using a "sprite designer". I would suggest either making several copies, or overlaying this chart with a sheet of acetate. Fill in the blocks to make the design you want, then add up the values of the blacked-in spaces, for each of the three sections. The three numbers you get will be the data statements which define that line of your sprite.

| SECTION A | SECTION B | SECTION C | SUM OF A | SUM OF B | SUM OF C |
|---|---|---|---|---|---|
| 1<br>2631<br>84268421 | 1<br>2631<br>84268421 | 1<br>2631<br>84268421 | | | |

## APPENDIX C:
## MULTI COLOR Sprites

MULTI COLOR Sprites differ in several ways from HI-RES. They can hold four colors (screen color, primary color, background color#1, background color#2) instead of the two allowed by HI-RES sprites.

The block which defines a MULTI COLOR sprite is only half as wide (12 rather than 24) since in MULTI COLOR mode your pixels are double width.

For my money, MULTI COLOR sprites are most easily defined using the Sprite Designer (q.v.), but they can also be defined using the Read'sprite procedure (q.v.). If you are using the Read'sprite procedure, the screen (transparent) color has a value of 00, background color#1 has a value of 01, the primary color has a value of 10, and the background color#2 has a value of 11.

All multi color sprites on the screen must share the same two common background colors. The two colors are set using the SPRITEBACK <color1>,<color2> command. The primary color of a MULTI COLOR sprite is set using the SPRITECOLOR <sprite#>,<color> command.

## APPENDIX D:
## Two more ways of making sprites

### Sprite Designer:

The Sprite Designer allows you to design your sprite without thinking about any values for any pixel. When you load up the program, you will be asked a series of questions. There were no instructions, but Captain COMAL and his friends have figured out what to do. Use 1 for yes answers, and 0 for no answers. The first question asks if the sprite is to be MULTI COLOR or not. Reply with a 1 for MULTI COLOR, and a 0 for HI-RES. You will then be asked to pick your primary color. Use the CONTROL key or the COMMODORE key and the numbers 1 through 8. For example, CONTROL 1 is black, COMMODORE 1 is orange, etc.. The next two questions ask if you want your sprite expanded on the x and y axes. Use 1 or 0 to respond. You will then be asked to choose your background color, and if your sprite is MULTI COLOR, your background colors #1 and #2. Again, use your CONTROL, COMMODORE and number keys to select your colors.

You now can design your sprite by moving the cursor around the large field. Use the cursor control keys to position your cursor. Turn on a pixel by pressing 1, turn it off by pressing 0. In multicolor mode press

0 (transparent)
1 (background #1)
2 (primary color)
3 (background #2)

to access your possible colors.

You can save your design by pressing S. To load in a previously saved design, press L. Press A to append a sprite to another program. Press Q to quit.

The sprite designer program is very long, and not many people would actively enjoy typing it in (not to mention de-bugging it after you type it in), so a working version has been included on your CCGP disk. Look for SPRITE DESIGNER.

To use one of the sprites you create with the

designer, read it into your program with a piece of code something like this,

```
open file 2,"<your sprite image name>",unit 8 read
read file 2: image$  //this assigns the information
                       file 2 to the variable
                       image$
close file 2
DEFINE 0,image$
```

You can now use the defined image just as you would use an image created any other way.

Read'sprite:

The Read'sprite procedure allows you to enter your sprites as strings of 1s and 0s. These strings are then read by a short machine language program which uses your binary notation to define the sprite. To define a HI-RES sprite by this method, enter

1 for blocks you want in the primary color
0 for blocks you want in the background color

For a MULTI COLOR sprite, remember that it takes two numbers to define a block. Twenty four numbers define a sprite 12 two-pixel-wide columns wide. The numbers to use are

00 for transparent
10 for primary color
01 for background color#1
11 for background color#2

For example the line

<data "100111001001110010011100">

would create one line of the 21 needed to define a MULTI COLOR sprite. This particular line repeats the sequence (primary color, 1st background, 2nd background, transparent) three times. This would

eventually give you a striped sprite.
To use the Read'sprite program, either enter the code below, or look on your CCGP disk and load Read'sprite.
To use this program for your own sprite, all you have to do is insert your own numbers in lines 780-860 (which design a multi-color sprite) or lines 880-1090 which design a hi-res sprite). The following is an example by Captain Comal and his Friends.

```
0010 // delete "readsprite/demo2"
0020 //  by captain comal and friends
0030 // save   "readsprite/demo4"
0040 //
0050 func find'string closed
0060  pointer1:=peek(51)
0070  pointer2:=peek(52)
0080  address:=pointer2*256+pointer1
0090  return address+4
0100 endfunc find'string
0110 //
0120 proc read'sprite'ml'data closed
0130 //
0140  data 0,0,0,169,0,168,170,141
0150  data 168,2,141,169,2,141,170
0160  data 2,169,128,141,220,2,189
0170  data 0,4,41,1,240,10,185,168
0180  data 2,24,109,220,2,153,168
0190  data 2,232,224,24,240,8,78,220
0200  data 2,144,229,200,208,221,96
0210  total:=0
0220  for x:=680 to 731 do
0230   read a
0240   poke x,a
0250   total:+a
0260  endfor x
0270  if total<>5747 then
0280   print "error in data statements"
0290   stop
0300  endif
0310 endproc read'sprite'ml'data
0320 //
0330 proc read'sprite(blk) closed
0340  if peek(683)<>169 then
```

```
0350   read'sprite'ml'data
0360  endif
0370  dim line$ of 24
0380  dim sprite$ of 64
0390  sprite$:=""; count:=1
0410  read line$
0420  while (line$(1)="0" or line$(1)="1") and count<64
      and (not eod) do
0430   while len(line$)<24 do line$:=line$+"0"
0440   if line$="" then null
0450   addr:=find'string
0460   poke 702,addr mod 256
0470   poke 703,addr div 256
0480   sys 683
0490   for x:=0 to 2 do
0500    sprite$(count):=chr$(peek(680+x))
0510    count:+1
0520   endfor x
0530   read line$
0540  endwhile
0550  while count<64 do
0560   sprite$(count):=chr$(0)
0570   count:+1
0580  endwhile
0590  case line$(1) of
0600  when "h","H"
0610   sprite$(64):=chr$(0)
0620  when "m","M"
0630   sprite$(64):=chr$(1)
0640  otherwise
0650   print "error in sprite data statements"
0660   stop
0670  endcase
0680  DEFINE blk,sprite$
0690 endproc read'sprite
0700 //
0710 read'sprite'ml'data
0720  SETGRAPHIC (0)
0730  read'sprite(0)
0740  IDENTIFY 1,0
0750  SPRITEPOS 1,75,75
0760  SPRITECOLOR 1,1
0765  read'sprite(0)
```

```
0766  IDENTIFY 1,0
0770 //
0780 data "0000110000" // any length
0790 data "0000110000" // up to 24
0800 data "0000110000"
0810 data "1111111111"
0820 data "1111111111"
0830 data "0000110000"
0840 data "0000110000"
0850 data "0000110000"
0860 data "h" // "h" means HI-RES
0870 //
0880 data "111111111111111111111111"
0890 data "111000000001100000000111"
0900 data "111100000001100000001111"
0910 data "110110000001100000011011"
0920 data "110011000001100000110011"
0930 data "110001100001100001100011"
0940 data "110000110001100011000011"
0950 data "110000011001100110000011"
0960 data "110000001101101100000011"
0970 data "110000000111111000000011"
0980 data "110000000011110000000011"
0990 data "111111111111111111111111"
1000 data "110000000011110000000011"
1010 data "110000000111111000000011"
1020 data "110000001101101100000011"
1030 data "110000011001100110000011"
1040 data "110000110001100011000011"
1050 data "110001100001100001100011"
1060 data "110011000001100000110011"
1070 data "110110000001100000011011"
1080 data "111111111111111111111111"
1090 data "m" // "m" means MULTI-COLOR
```

You now have several ways to design sprites. I hope you find one of them comfortable to use.

INDEX

An index to the procedures included in Chapter 5 follows the regular index.

AUTO (line numbering) ........................ 7

BACK ..................................... 24
BACKGROUND ........................... 16-17
BORDER ................................ 16,18

CHAIN .................................... 11
CLEAR ................................. 16-17
COORDINATES, (see SCREEN COORDINATES)

DATACOLLISION ..................... 13,37,39
DEFAULT (defined) ....................... 12
DEFINE ............................ 30,31,33
DRAWTO ................................ 24-25

ENTER .................................... 11

FILL .................................. 24,26
FORWARD ................................. 24
FRAME ................................. 16,18
FULLSCREEN ............................ 16,18
FUNCTION KEY (predefined) .............. 16,74

GETCOLOR .............................. 21,23

HIDESPRITE ............................ 30,35
HIDETURTLE ......................... 17,21-22
HOME .................................. 17,24

IDENTIFY ........................... 30-31,33
IMAGE (sprite) ......................... 30-31
IMMEDIATE MODE .......................... 12

LEFT .................................. 24-25
LIST (a program to screen) ............. 10-11
(a program to disk) ...................... 11
LOAD ..................................... 11

MOVETO ........ 24-25
PENCOLOR ........ 13,21-22
PENDOWN ........ 21,33
PENUP ........ 21,33
PLOT ........ 24,26
PLOTTEXT ........ 27-28
PRIORITY ........ 39
PROCEDURE (explained) ........ 8-10
PROGRAMMING MODE ........ 12
READ'SPRITE ........ 78-81
RENUM (renumber) ........ 8
RIGHT ........ 24-25
SAVE ........ 11
SCREEN
  Coordinate system ........ 15
  HI-RES ........ 16
  MULTI COLOR ........ 16-17
SETGRAPHIC ........ 16-17,21
SETHEADING ........ 17,21-22
SETTEXT ........ 16-17
SETXY ........ 24,26
SHOWTURTLE ........ 21-22
SPLITSCREEN ........ 16,18
SPRITE
  Defined ........ 27-28
  HI-RES ........ 28-30
  MULTI COLOR ........ 28-30,76
SPRITEBACK ........ 30,34
SPRITECOLLISION ........ 37,39
SPRITECOLOR ........ 30,34
SPRITEPOS ........ 30,35
SPRITESIZE ........ 30,34

TURTLESIZE ........ 17,21

PROCEDURES

CIRCLE ........ 59-62

GETBACKGROUND ........ 62

GETBORDER .................................. 62
GETPEN ..................................... 63
GETPENCOLOR ................................ 63
GETSPRITECOLOR ............................. 63
GETTURTLESIZE .............................. 63
GRAPHICSTATE ............................... 63

HEADING .................................... 64
HIDESCREEN ................................. 64

POLYGON .................................... 65

SHOWSCREEN ............................. 65-66
SHOWSPRITE ................................. 66
SPRITESTATE ................................ 66
SPRITEXCOR ................................. 66
SPRITEXSIZE ................................ 66
SPRITEYCOR ................................. 67
SPRITEYSIZE ................................ 67

TURTLESTATE ................................ 67

XCOR ....................................... 67

YCOR ....................................... 68

THE PROGRAMS FOR THIS BOOK ARE ON THE MATCHING DISK WITH THE LABEL SHOWN AT THE BOTTOM OF THIS PAGE

## THE AMAZING ADVENTURES OF CAPTAIN COMAL

BOOK 1
**CAPTAIN COMAL GETS ORGANIZED**
BY LEN LINDSAY
BOOK AND DISK - $19.95

BOOK 2
**COMAL FROM A TO Z**
BY BORGE CHRISTENSEN
BOOK - $6.95

BOOK 3
**COMAL LIBRARY OF FUNCTIONS & PROCEDURES**
BY KEVIN QUIGGLE
BOOK AND DISK - $19.95

BOOK 4
**CARTRIDGE GRAPHICS AND SOUND**
BY CAPTAIN COMAL'S FRIENDS
BOOK - $9.95

BOOK 5
**CAPTAIN COMAL'S GRAPHIC PRIMER**
BY MINDY SKELTON
BOOK AND DISK - $19.95

BOOK 6
**COMAL WORKBOOK**
BY GORDON SHIGLEY
BOOK - $6.95

ISBN 0-928411-04-4

**GRAPHICS PRIMER DISK**
C64 COMAL 0.14 © 1983
COMAL Users Group (USA)
5501 Groveland Ter. Madison, WI 53716-3251
**LOAD "BOOT*", 8 then RUN**