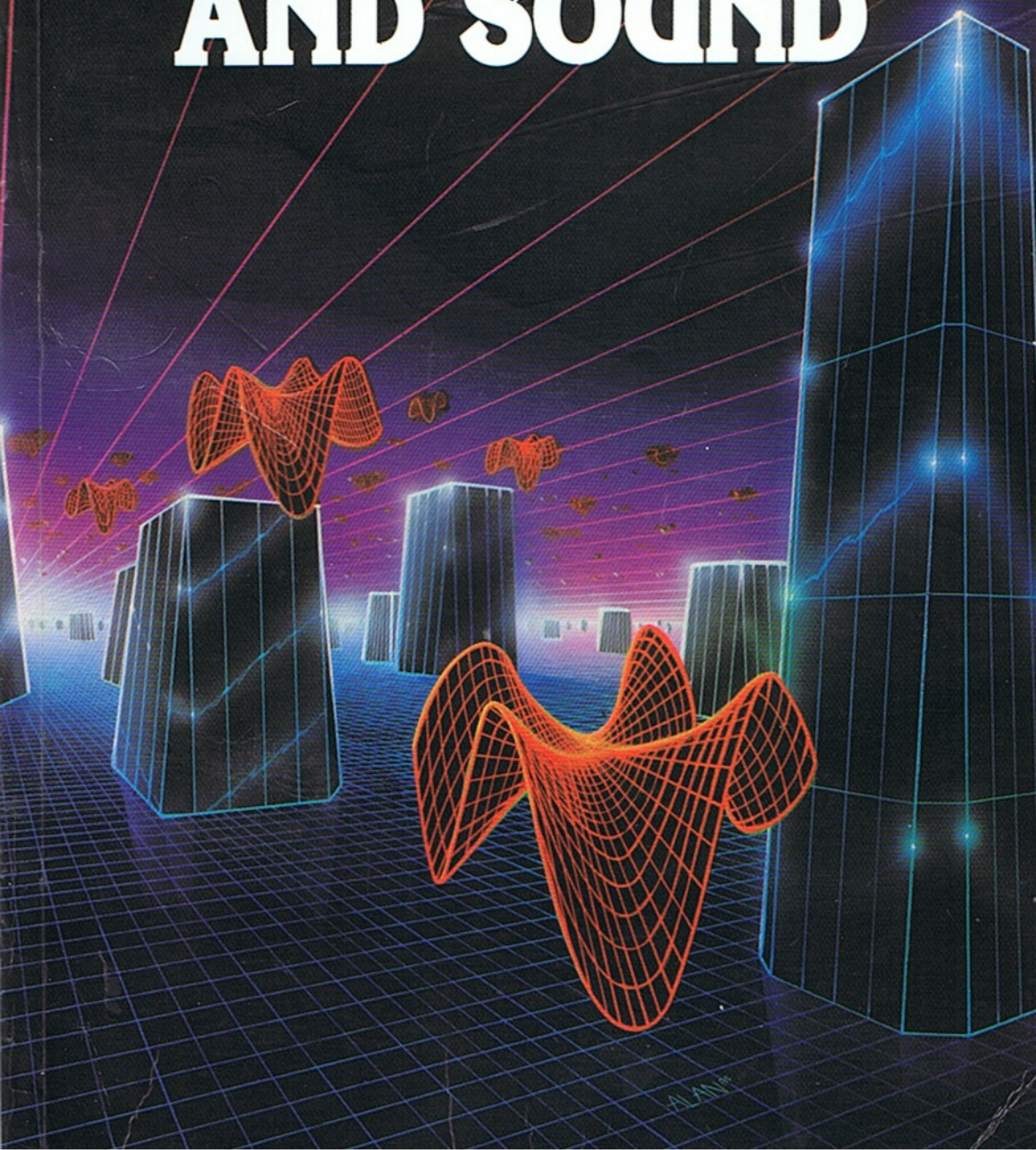


**Steve Money**

# COMMODORE 64 GRAPHICS AND SOUND



# **Commodore 64 Graphics and Sound**

**Steve Money**

**GRANADA**

London Toronto Sydney New York

Granada Technical Books  
Granada Publishing Ltd  
8 Grafton Street, London W1X 3LA

First published in Great Britain by  
Granada Publishing 1984

Copyright © S. A. Money 1984

*British Library Cataloguing in Publication Data*

Money, Steve A.

Commodore 64 graphics and sound.

1. Commodore 64 (Computer)—Programming
2. Computer sound processing
3. Computer graphics

I. Title

001.64'43      QA76.8.C64

ISBN 0-246-12342-7

Typeset by V & M Graphics Ltd, Aylesbury, Bucks  
Printed and bound in Great Britain by  
Mackays of Chatham, Kent

All rights reserved. No part of this publication may  
be reproduced, stored in a retrieval system or  
transmitted, in any form, or by any means, electronic,  
mechanical, photocopying, recording or otherwise,  
without the prior permission of the publishers.

# Contents

<i>Important Note</i>	vi
<i>Preface</i>	vii
1 Introduction	1
2 Character Graphics	14
3 High Resolution Graphics	31
4 Adding Colour	61
5 Setting Objects in Motion	82
6 Sprite Graphics	98
7 Graphs and Charts	117
8 Depth and Perspective	140
9 The Sound Generator	157
10 Making Music	178
<i>Index</i>	193

# Important Note

In many program listings for the Commodore 64 computer the cursor, reverse video and colour control codes are included in text strings of PRINT statements. They appear on the listings as reverse video symbols but must be entered on the keyboard by using cursor keys or combinations of certain keys with the CTRL or LOGO keys. The method of keying in these codes is described in Chapters 2 and 4. Figs. 2.2, 2.7, 4.1 and 4.2 show the listing symbols for these codes.

Control codes can also be entered using the CHR\$(N) form where N is the ASCII code number. Since the listing symbols for these codes can sometimes be difficult to decipher in printed listings the CHR\$ form has been used for most listings in this book, but some programs in Chapter 7 do use embedded control codes in text strings.

In Fig. 7.4 lines 41Ø and 46Ø, Fig. 7.6 line 44Ø, and Fig. 7.8 lines 44Ø and 51Ø, the text string is produced by using the keys

[SPACE] [CRSR UP] [CRSR LEFT]

# Preface

Modern home and personal computers usually have very good colour graphics and sound facilities, and the Commodore 64 is no exception in this respect. The graphics display capability of this machine is extremely good, and the sound generator is perhaps the most versatile available.

The primary graphics mode on the Commodore 64 makes use of character graphics, and is basically a development of the graphics provided on the earlier Commodore PET computer. The main difference is that the Commodore 64 has a *colour* display. In Chapter 2 we take a look at some of the techniques involved in using character graphics. To a large extent, using character graphics is like assembling a jigsaw puzzle. Once you have sketched out a rough outline of the picture to be produced, you simply choose suitable symbols from the available set, and then place them on the screen to build up the desired picture.

When the existing character set doesn't provide the symbol you want it is possible to create your own symbols. This is a fairly complicated process on the Commodore 64, but you will find a full description of how to do it in Chapter 2.

High resolution graphics (where individual dots on the screen can be controlled) were not available on the earlier PET and VIC 20 computers, but have been provided on the Commodore 64. This makes use of what is called the 'bit mapped graphics' mode. The BASIC language of the Commodore 64 does not provide any drawing commands for the high resolution graphics, so points and lines have to be set up on the screen by POKEing data directly into the graphics display memory. Some of the techniques involved in drawing on the high resolution screen are described in Chapter 3. However, drawing using BASIC is rather a slow process, and for serious work in this mode programs written in machine code will usually be required.

Chapter 4 looks at the colour capabilities of the Commodore 64, both in the text and character graphics mode and in the high

resolution mode. Sixteen colours are available, and by using the multicolour mode it is possible to produce multicoloured symbols.

For most games programs, animation of objects on the screen is an important factor. The basic principles involved in animating simple objects and detecting collisions are explained in Chapter 5. An important variation of animation makes use of *scrolling*, where the entire picture is moved up or down or from side to side. Typical applications of this technique are in 'road race' games and those of the Defender type, where a landscape moves across the bottom of the screen. Scrolling requires the use of machine code routines since BASIC is far too slow for this type of operation. Three machine code routines are included which can be loaded and called by BASIC and provide downward and side-to-side scrolling actions.

One very important feature of the Commodore 64 is its *sprite graphics*. In Chapter 6 we look at how sprites are created, controlled and positioned on the screen. The chapter goes on to look at animation using sprites. For applications involving animation the sprite graphics are far superior, and more flexible in use than conventional character graphics.

An important application of the graphics capabilities of any computer is graphs, charts and other similar displays. In Chapter 7 the principles of displaying gauges and meters are discussed, and the chapter goes on to show how bar charts and scientific graphs can be produced.

Chapter 8 introduces 'three-dimensional' displays with three-axis bar charts and circular graph plots which give an illusion of depth. The chapter goes on to look at the principles involved in producing perspective views.

In Chapter 9 the versatile sound generator of the Commodore 64 is explored. There are many features in the special sound chip used in this machine, and many useful hours can be spent trying out the different possibilities. The principles of setting up sound frequencies and waveforms are explained. The chip has a comprehensive ability to tailor the 'envelope' of sounds, and the techniques involved in producing some common sound effects are explained.

The sound chip can, of course, be persuaded to produce music, and in Chapter 10 the basic principles of music and the techniques for translating written music into data for playing a tune are explained. Later in the chapter a program is developed which allows the Commodore 64 to be used as a musical instrument that can be played directly on the keyboard.

Steve Money

# Chapter One

## **Introduction**

One of the attractions of the modern personal or home computer is its ability to provide highly detailed graphics displays, usually in colour, and to produce a wide range of sounds. These facilities are, of course, very important for one of the main uses of home computers: playing various types of video game. The graphics capabilities, however, are not limited to use for games. They can also provide diagrams and displays for business and educational use. It is also possible to turn the computer into a form of automated drawing board, and (if a suitable printer or plotter is available) to produce quite acceptable drawings or diagrams on paper. Finally, of course, computer graphics displays may be used purely as an art form.

The Commodore 64 computer provides an excellent graphics display which can produce both low and high resolution pictures in colour. One disadvantage, however, is that the BASIC language provided as standard on this machine has virtually no commands to handle the high resolution graphics and sound facilities. This can mean that programming graphics may at first seem to be a complicated process. The techniques are not, however, all that difficult to master.

The actual graphics displays are controlled by a special display chip known as the VIC11 Video Interface Chip (which may also be referred to by the chip type number 6566 or 6567), and its functions are controlled by inserting appropriate data into its internal registers using the BASIC POKE and PEEK instructions. This is a very complex device, and has a total of 47 internal registers to control its various operating modes.

### **Video displays**

Before going on to explore the display capabilities of the



## 2 Commodore 64 Graphics & Sound

Commodore 64, it might be as well to look at the basic techniques involved in producing computer displays.

All current home or personal computers make use of a television-type display to present text and graphics outputs. In most cases a domestic television receiver is used, and the signal from the computer is fed into the aerial input of the TV receiver. As far as the television set is concerned the signals from the computer appear to be just another television channel. Usually the output of the computer is set to an unused channel, generally channel 35, and the TV is tuned to this using one of the spare channel selector buttons.

Passing the computer video signal through the complete television receiver can cause some loss of sharpness on high resolution graphics displays, especially when they are in colour. An alternative arrangement, which can produce greatly improved results, makes use of a television monitor unit. Here the video signals from the computer are not converted into a broadcast-type television signal but are fed directly to the television display circuits. The special television monitor units usually employ much higher-grade display tubes and circuits, which themselves allow a much sharper and more steady picture to be produced.

In a television receiver the picture on the screen is built up by sweeping a single dot of light across and down the screen. The dot moves rapidly from left to right, tracing out a series of horizontal lines across the screen. At the end of each sweep the spot moves almost instantaneously back to the left-hand side ready to trace out the next line. At the same time as it is moving across the screen the dot also moves slowly *down* the screen, so that each successive line is drawn just below the previous one. Every  $1/25$ th second ( $1/30$ th second in America and Japan) the spot traces over the complete area of the screen.

The complete picture is traced out 25 (or 30) times per second. This could cause a flickering effect, since the eye can just about detect changes in a scene occurring at this rate. To avoid such problems a system known as *interlaced scanning* is generally used. In this scheme the dot is scanned down the screen in  $1/50$ th ( $1/60$ th) second but only traces out alternate scan lines. So on the first scan the dot might trace out all of the odd numbered lines of the complete picture, and on the next all the even lines, filling in the gaps between the lines of the first scan. The effect of this, as far as the viewer is concerned, is to increase the flicker rate to 50 (or 60) per second, making it undetectable; but the rate at which complete pictures are presented is still 25 (or 30) per second.

If the brightness of the spot is varied as it traces over the screen then a picture made up of light and shaded areas is built up on the screen each time the spot completes a scan. In a colour television display three separate spots of light are scanned simultaneously over the screen. One spot is red, and the others are green and blue respectively. On the screen the three dots are kept very close together so that, to the viewer, they appear to be a single point of light. The red, green and blue light from the three dots is effectively combined, so that the viewer sees a white dot if the three separate dots are lit simultaneously. By varying the relative brightness of the red, green and blue dots any desired colour of light may be produced at each point as the group of dots scans over the screen.

## Text displays

Much of the output from a computer will consist of printed text displayed on the TV screen. For our computer display we can conveniently divide the screen up into a series of small rectangular areas, called *symbol spaces*, and in each of these rectangles we can arrange to display a single text symbol. The Commodore 64 screen display allows 25 rows of text to be displayed with 40 symbol spaces in each row. This allows a total of 1000 text characters, or roughly 150 words of text, to be displayed on the screen at a time.

If you examine text symbols displayed on a broadcast television picture, for instance during programme credit titles, you will see that each symbol is actually built up from a pattern of short horizontal lines and dots. For a computer display the symbol space on the screen is divided up into a matrix of individual dots. If you look very closely at the text displayed by your Commodore 64 you will see that each letter is built up from a pattern of dots. In fact for each symbol space the Commodore 64 uses 8 rows of dots with 8 dots in each row. By selectively lighting some of the dots and leaving the others dark the outline of the required text symbol can be picked out as shown in Fig. 1.1.

In order to display text the computer needs to know the patterns of dots for each of the symbols it will be required to display. Typically there will be capital and lower case letters, numbers, and a selection of punctuation and other signs. A typical *character set* might contain about 96 different symbols.

The usual method of storing the dot patterns is to use a memory chip similar to those used for the computer's main memory. A

#### 4 Commodore 64 Graphics & Sound

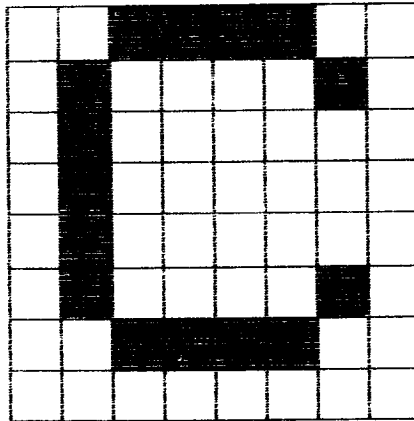


Fig. 1.1. Display of a text symbol using a dot matrix.

memory chip consists of thousands of tiny electronic cells, each of which can be turned on or off. Each cell stores one item of data which is called a 'bit'. Each of these bits has two possible states which are called '1', when the cell is turned 'on', and '0' when the cell is 'off'. Each cell may equally be used to indicate if a dot on the screen has to be lit (1) or dark (0). With 64 dots in each character pattern, and assuming a set of 128 different characters, then a total of 8192 bits of memory are required to store the complete set of dot patterns. Typically the memory is arranged as groups of 8 bits, called *bytes* or *words*, so the dot-pattern memory would have 1024 bytes which can easily be fitted into a single memory chip.

We could, of course, use a piece of the main computer memory to store the character dot patterns, but the problem here is that when the computer is turned off the contents of the memory are lost. To overcome this problem a special type of memory device is used. This chip has the data pattern written permanently into it when it is made, and the pattern is retained even when the power is turned off. We can read data from the memory but cannot alter what is stored in it, so the device is called a Read Only Memory or ROM. When the ROM is used to hold the dot patterns for producing character displays it is generally called a 'character generator' ROM. The character generator fitted to the Commodore 64 is quite complex, and it contains the dot patterns for two complete character sets with 256 symbols in each set.

## The display memory

The television picture is scanned 25 (or 30) times a second to produce a continuous display on the screen. This means that we have to trace out the dot patterns repeatedly, so the display circuits will need rapid access to data telling them what symbols are to be displayed. This involves the use of some sort of memory to store the information to be displayed. We could of course store all of the dot patterns for the screenful of text; but this would use up a lot of memory, so it is normal to allocate a number code, or *character code*, to each character in the available set and then store just the character codes for the text to be displayed. Since a 'memory word' of 8 bits can have 256 different combinations, a single memory word is sufficient to define each text symbol to be displayed.

The Commodore 64, with its  $40 \times 25$  screen format, can display 1000 symbols on the screen and will need 1000 words of memory to store the character codes of the text to be displayed. This memory is, in fact, just a section of the computer's main memory, and is referred to as the *screen memory*. At switch-on the screen memory is automatically located at addresses 1024 to 2023, but it is possible to move the area reserved as screen memory to other addresses if desired.

To produce the display, the VIC11 chip reads the sequence of character codes for a row of text as the dot scans across the screen. Each character code is in turn used to call up a row of dots from the corresponding character dot pattern, and the dot on the screen is switched on and off in sympathy with the dot patterns as it sweeps across the screen. Each row of text is scanned eight times, with a different row of dots from the character pattern being called up for each scan line.

The general arrangement of the text display system is shown in Fig. 1.2.

### *Character graphics*

In the early days of computers the only form of output available, either on a display screen or printed on paper, was in the form of text. However, enterprising programmers soon discovered they could produce crude pictures, graphs and charts by using the text symbols. Some symbols, such as the M and W, will appear darker than others, such as I, so by careful choice of the pattern of symbols printed on a page a picture can be built up. If viewed from a distance such a picture can look quite good!

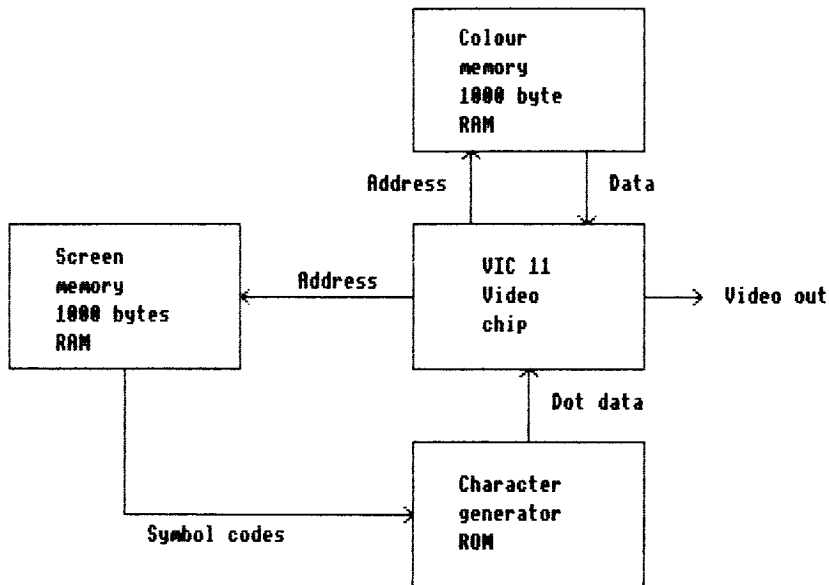


Fig. 1.2. Display system for text and character graphics displays.

The next step in the development of graphics displays was the addition of special graphics symbols to the displayable character set. These symbols would consist of a selection of horizontal and vertical lines, shaded blocks and some specialised symbols such as the suit markers for playing cards. One of the early personal computers to use such a character set was the Commodore PET, and the Commodore 64 provides a similar set of graphics symbols. Some examples of these symbols are shown in Fig. 1.3.



Fig. 1.3. Typical character graphics symbols on the Commodore 64.

The graphics symbols provide segments of lines which can be assembled together to produce remarkably effective displays, and they can be very useful in producing such things as bar charts. There are some inherent limitations to this method of producing graphics displays, since the special symbols can only be positioned in the standard matrix of character spaces on the screen, and some care is needed in designing the display layout.

## Mosaic graphics

With the normal text mode we can, by having the whole character space lit or dark, produce a graphics dot pattern which is 40 dots wide and 25 dots high. By selectively lighting the character spaces we could produce simple patterns. This is generally referred to as *low resolution graphics*. We talk of 'resolution' as a measure of the fineness of detail that can be produced in the picture, and it is quoted as the number of dots or picture elements that can be individually controlled across and down the screen. For the Commodore 64 text screen, using whole character spaces as picture elements, we would have a resolution of 40 dots across and 25 down which is usually abbreviated to  $40 \times 25$ .

One of the variations of character graphics, which is often used on home computers to provide low to medium resolution displays, is called *mosaic graphics*. Here the symbol space is divided up into a matrix of four or six blocks arranged in two columns. If the space is divided into four quarters, and each can be set on or off, then the original  $40 \times 25$  text screen now becomes an  $80 \times 50$  graphics matrix. Some typical mosaic symbols are shown in Fig. 1.4. The picture is built up in much the same way as when character graphics symbols are used – it's rather like assembling a jigsaw puzzle.



Fig. 1.4. Typical mosaic graphics symbols on the Commodore 64.

Using mosaic graphics symbols we can now selectively light 80 dots across the screen and 50 down the screen, which allows us to produce low to medium resolution graphics. Thus for mosaic graphics on the Commodore 64 the resolution would be  $80 \times 50$ .

Mosaic graphics provide greater flexibility in building up a picture when compared with the line and shape elements of the graphics character set. Within the limits of the  $80 \times 50$  screen resolution any desired pattern can be produced, although the results will usually look rather crude compared with those that can be achieved by careful use of character graphics. The mosaic graphics symbols also provide a convenient method of filling in areas of colour on the screen.

### Programmable characters

Normally the dot patterns for all the text and graphics symbols that can be displayed on the screen are held in a special character generator ROM. This device, however, is just a piece of memory as far as the computer is concerned. Therefore it should be possible for the VIC11 chip to display the data patterns stored in any part of the computer's working memory. Since we can write whatever we like into the working memory, we can produce customised characters by setting up the appropriate patterns of '1's and '0's in the memory and telling the VIC11 chip to use them as character dot patterns.

The Commodore 64 *does* in fact allow us to use the computer memory and VIC11 chip in this way, so that any desired symbol pattern within an  $8 \times 8$  character matrix can be set up in memory and displayed on the screen. When these user-defined symbols are being used by the VIC11 chip, however, the normal character generator ROM is disabled. This means that if we want to use some of the standard symbols as well as the custom-designed ones, the required dot patterns have to be copied from the ROM into the main memory, or RAM as it is generally called. Now the standard character patterns will form part of the user-defined symbol set, and can be displayed on the screen. During this copying process we can, if we like, rearrange the order of the standard symbol set; or we might select characters from both standard and alternate character sets for use simultaneously.

The arrangement of the display system when user-defined graphics symbols are being used is as shown in Fig. 1.5.

### Bit mapped graphics

The mosaic graphics scheme effectively provides us with a matrix of  $80 \times 50$  dots over the screen on which we can build up a pattern. The text symbols themselves are built up from a matrix of  $8 \times 8$  dots, and if we could control each individual dot in every character space then much higher resolution displays become possible. This mode of graphics operation is available on the Commodore 64, and it is called *bit mapped* or *high resolution* graphics.

In the case of the Commodore 64 the number of dots across the screen will be  $8 \times 40$  or 320 dots, since there are 40 symbol spaces each 8 dots wide. In the vertical direction we have 25 rows of symbols with 8 dots for each symbol, so the number of dots down the

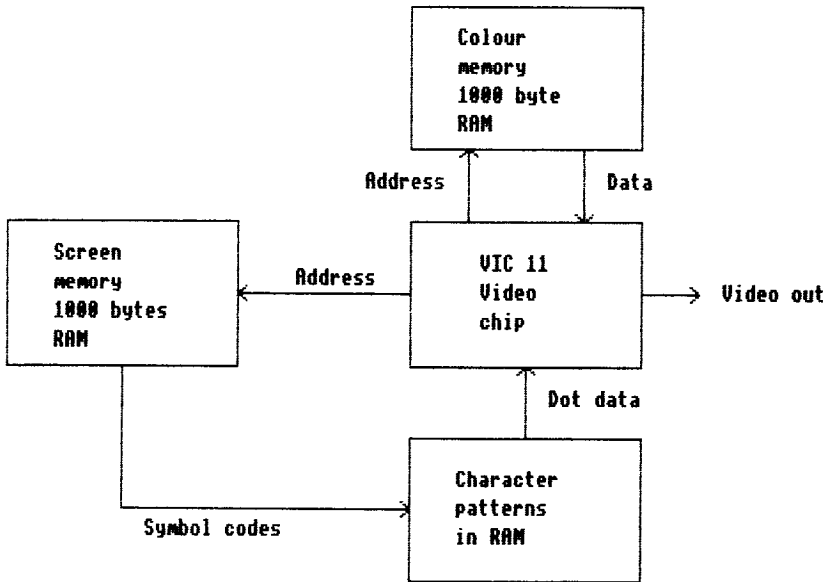


Fig. 1.5. Display arrangement for user-defined text and graphics.

screen is  $25 \times 8$  or 200. Thus for the high resolution mode the Commodore provides a graphics resolution of  $320 \times 200$ , which will allow quite detailed graphics displays to be produced.

In the character graphics mode an 8-bit code defines the symbol to be displayed, and 1000 bytes of memory are used to hold these codes. For high resolution graphics a different arrangement is required. Let us suppose that each dot on the  $320 \times 200$  screen can be set either 'on' or 'off'. We can store the state of the dot as a single data bit which may be either a '1' or '0'. The screen itself contains  $320 \times 200$  or 64000 dots altogether, so we are going to need rather a lot of memory to store the required information. Since there are 8 bits per word we can use each memory word to store the data for 8 adjacent dots on the screen. As a result, a total of 8000 words of memory are required to deal with the high resolution screen display.

To handle high resolution graphics the Commodore 64 switches from its normal text display to a different display mode called the Bit Mapped mode. In this mode the VIC11 video chip calls up its display data from an 8000-word section of the RAM instead of from the normal 1000-word screen memory area. In fact, this bit map area of memory is simply an extension of the memory area that would normally be used to store the dot patterns for user-defined symbols. The change in the display mode is controlled by changing the data in



two of the control registers of the video display chip, as we shall see in a later chapter.

One slight problem is that when the display mode is switched to high resolution graphics the computer will no longer display text symbols – in fact the normal text screen memory area is used for another purpose. We can, however, mix text with graphics by copying the required dot patterns from the character ROM into the bit map memory at the desired position.

The arrangement of the display system when the bit map mode is selected is shown in Fig. 1.6.

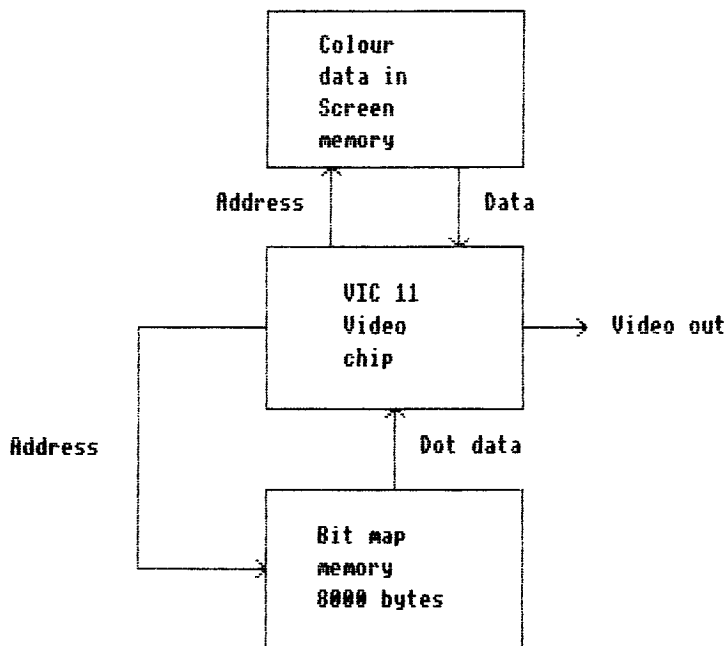


Fig. 1.6. Display system arrangement for the bit mapped graphics mode.

### Adding colour

The Commodore 64 allows the display to be presented in colour. The screen memory used to define which symbol is displayed simply calls up a pattern of '1's and '0's to represent the dots making up the symbol. Here a '1' indicates that the dot is in the foreground colour, and a '0' indicates it is the background colour. When the Commodore 64 is turned on, the foreground colour will be set to light blue and the background to blue. The Commodore 64 does allow a choice of sixteen different colours, and any pair of these may be chosen as the foreground and background colours.

To allow individual symbols to be set to any colour combination, a separate section of memory is used to hold the colour information for each character space on the text screen. This is known as the *colour memory*. It is the same size as the screen memory (1000 bytes) and laid out in the same way relative to the text display. The colour memory is located at a fixed position high in the memory address range. As each symbol code is written into the screen memory, a corresponding colour code goes into the colour memory and this determines the colour of the displayed symbol.

The border around the main display area on the screen can also be set to any one of the sixteen colours, and this is controlled by one of the registers in the video chip. The background colour is also controlled by a register in the VIC11 chip, and may be set to any of the sixteen available colours. For more colourful results it is possible to select another display mode which permits the use of four different background colours.

When we come to the high resolution mode the text screen memory is used to store the colour data. This means that the colour of all lit dots within a text character space will be the same, so the *colour resolution* is only  $40 \times 25$ .

In both high resolution and text modes we can select a multicolour mode which allows the use of four different colours within a symbol space. The penalty here is that the resolution is reduced to  $160 \times 200$  but the advantage is much greater flexibility in the use of colour. This mode allows the production of multicoloured text or graphics symbols when the character graphics mode is in use.

## Sprite graphics

A variation on the user-defined symbol is the 'sprite'. In some ways this is a bit like a larger version of the symbol, but with a dot array of  $24 \times 21$ . Like user-defined symbols, the dot patterns for sprites are held in the RAM, although not in the same area. Unlike symbols, the sprites are not placed on the screen by using PRINT or POKE commands, but are controlled directly by registers within the video chip. As a result a sprite can be placed anywhere on the screen.

The Commodore can handle up to eight different sprites and the video chip can detect when two sprites overlap, which can be useful for detecting hits and collisions in games programs. Sprites can also be arranged so that as one sprite passes another it will blank the other sprite, as if it were passing in front; alternatively it may be

## 12 Commodore 64 Graphics & Sound

blanked out by another sprite as if it had passed behind that sprite. These features can be particularly useful when producing graphics for arcade-style games, where several objects may be moving around the screen at the same time.

Like the text and graphics symbols, sprites can be multi-coloured, and can be turned on or off at will by placing appropriate data into the registers of the video display chip. In Chapter 6 we shall take a look at how sprites are created and controlled on the Commodore 64.

Sprites are entirely independent of the text and bit-mapped display systems, and are controlled by the VIC11 chip itself. The dot patterns making up a sprite are, however, stored in memory in much the same way as the dot patterns for user-defined symbols. It is possible to display sprites with either text or bit-mapped graphics displays with no difficulty.

### **The video display chip**

The key to the Commodore 64 display capabilities is the VIC11 video interface chip itself. This contains all the logic that controls the generation of the text and graphics displays. This chip works in parallel with the central processor chip in calling up data from the screen or bit map memories, and dot patterns from the character generator, and using them to produce the video signals needed to build up the picture on the TV screen. The computer itself has to put the required data into the screen, colour, and bit map memories, and then leaves the VIC11 chip to get on with producing the display on the screen. While the computer goes on executing the rest of its program, the VIC11 chip steals small amounts of time from the central processor when it needs access to the computer memory, but this has little effect on the execution of the computer's main program task.

The VIC11 chip's addressing system limits its access to the main computer memory so that only one 16k byte bank of the main memory is seen by the VIC11 chip at any time. Normally this will be the section of memory from address 0 up to address 16383. It is possible to alter the section of memory accessed by the video chip to any one of the four possible 16k byte banks of memory available in the 64k main memory of the computer. For most purposes it is best to leave the VIC11 chip with its normal access to the first 16k block of memory.

## **Sound generation**

The Commodore 64 has quite a versatile sound generating capability. This is also controlled by a special chip called the Sound Interface Device or SID chip, which is sometimes referred to by its type number 6581.

The sound chip provides three independent sound generator channels, and each of these can be controlled in frequency and amplitude. There are also envelope generators to shape the sound, and a filter system which allows even more complex sound processing.

In Chapters 9 and 10 we shall explore some of the aspects of the sound chip and see how it can be used to produce sound effects and to play music.

## Chapter Two

# Character Graphics

Let us make a start by looking at the text and character graphics display mode of the Commodore 64. As we saw in Chapter 1 the basic text display mode provides 25 rows of text symbols with 40 symbols in a row, and this is the mode which will be selected when you switch on the computer.

The data for the text display is held in a screen memory area, which takes up 1000 words of the main computer memory. This character memory normally occupies memory locations 1024 to 2023, but it is possible to move it to any convenient location within the 16k memory block that is being accessed by the video display chip.

The 1000-word memory area used for text contains one word for each displayed symbol, and this memory word contains a character code, with a value from 0 to 255, corresponding to the character to be displayed on the screen. An important point to note here is that the character code stored in the screen memory is not the same as the ASCII character code used in a PRINT statement. The layout of the memory is arranged so that successive characters across a text row are stored in 40 sequential memory locations, and are followed by the codes for the symbols in the next display row. Thus the symbol at the top left corner of the screen has its code stored in memory location 1024, and the one at the top right is held in memory location 1063. The layout of the screen memory is as shown in Fig. 2.1.

### Placing text on the screen

The Commodore 64 has a text cursor which indicates to the machine, and the user, where the next symbol is to be displayed on the screen. This is normally shown as a flashing block.

We can control the position of the cursor by using various keys on

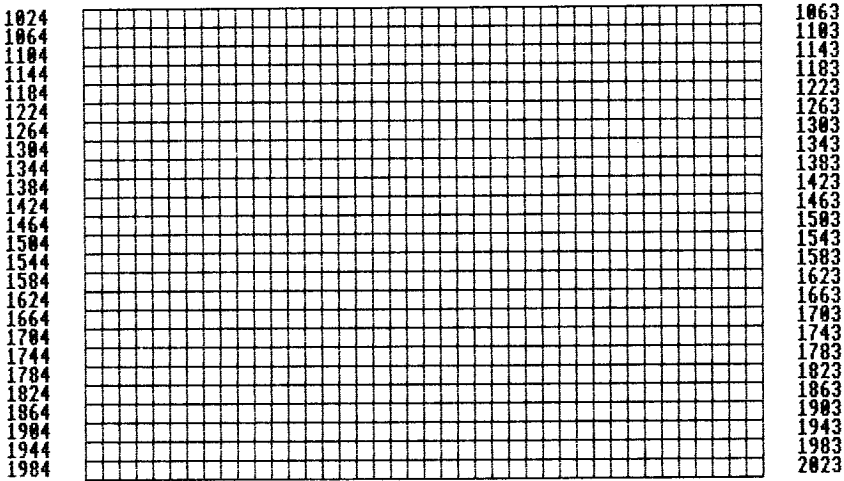


Fig. 2.1. The layout of the screen memory in the Commodore 64.

the keyboard. If you press the key in the top row marked CLR/HOME the cursor will immediately move to the top left corner of the screen. This is the HOME operation. Note that the text already being displayed on the screen is unaffected. If you hold down the SHIFT key while you press the CLR/HOME key a different action occurs. Now any text that was being displayed is cleared off the screen, and the cursor once again moves to the top left corner. This is the CLR (screen clear) operation. A point to note here is that the statement CLR in BASIC has a different function, and should not be confused with screen clear.

At the bottom right of the keyboard are two adjacent keys with arrows on them. One has arrows pointing up and down, and it moves the cursor up and down the screen. The other has arrows pointing left and right, and it controls left/right movement of the cursor.

Now if you press the cursor up/down key the cursor will move down the screen one row at a time. Pressing the left/right cursor key moves the cursor to the right. To move the cursor up the screen the SHIFT key must be held down while the cursor up/down key is pressed. Similarly using the SHIFT key with the left/right cursor key moves the cursor to the left.

At the start of a program it is useful if we clear the screen display and place the cursor at the top left corner. This can be achieved inside the program by using the CLR/HOME key as if it were a normal letter key, and the result is placed between quotes after a PRINT command as follows:

```
1000 PRINT "[SHIFT CLR]"
```

The required control code is keyed in by using the **SHIFT** and **CLR** keys together. Here the key names have been included between square brackets to indicate that these are the names of the keys to be pressed and not an ordinary text string.

If you just want to move the cursor to the top right corner, without clearing the screen, then the **CLR/HOME** key must be used by itself as follows:

```
1000 PRINT "[HOME]"
```

The cursor can be moved up, down, left or right by using the cursor arrow keys, either with or without **SHIFT**, in a **PRINT** statement in the same way.

Each of the cursor control keys does in fact produce a symbol code, and this appears on screen between the quote symbols when you type in the **PRINT** statement. This symbol will also appear in the program listing; however, when the instruction is executed by the computer the cursor moves, but the special symbol is not displayed on the screen. These special symbols are important, however, because they show that there are cursor control codes in the character string. Thus the **CLR** key produces a heart-shaped symbol and the **HOME** key produces an S symbol in blue on a light blue background. The other cursor keys also produce unique symbols and these are shown in Fig. 2.2. A point to note is that the cursor control symbols are not written to the screen when the program is actually run, and they do not take up any space in the screen memory.

Each of the cursor control keys does in fact produce a particular character code and an alternative way of placing them in a **PRINT** statement is to use the form

```
1000 PRINT CHR$(N)
```

where **N** is a number from 0 to 255 corresponding to the ASCII code for the cursor move that you want to execute. These ASCII code numbers for the various cursor control actions are listed in Fig. 2.2. Thus if you want to clear the screen you could use

```
1000PRINT CHR$(147)
```

To move the cursor to any desired position on the screen a series of cursor shift codes can simply be included in the **PRINT** statement before the text or graphics symbols, or even between them. It is usually best to start off with a **CLR** or **HOME** command at the start

of the program so that you know where the cursor is before starting to move it.







Key	Shift	ASCII	List
	key	code	Symbol
CRSR D	No	17	
CRSR U	Yes	145	
CRSR R	No	29	
CRSR L	Yes	157	
HOME	No	19	
CLR	Yes	147	

Fig. 2.2. The keys, character codes and listing symbols for cursor control on the Commodore 64.

### Simulating PRINT AT operations

Some personal computers have a PRINT AT statement included in their BASIC language, but in the case of the Commodore 64 this very useful command is not available. We can, however, devise alternative schemes which will produce the same results as a PRINT AT command.

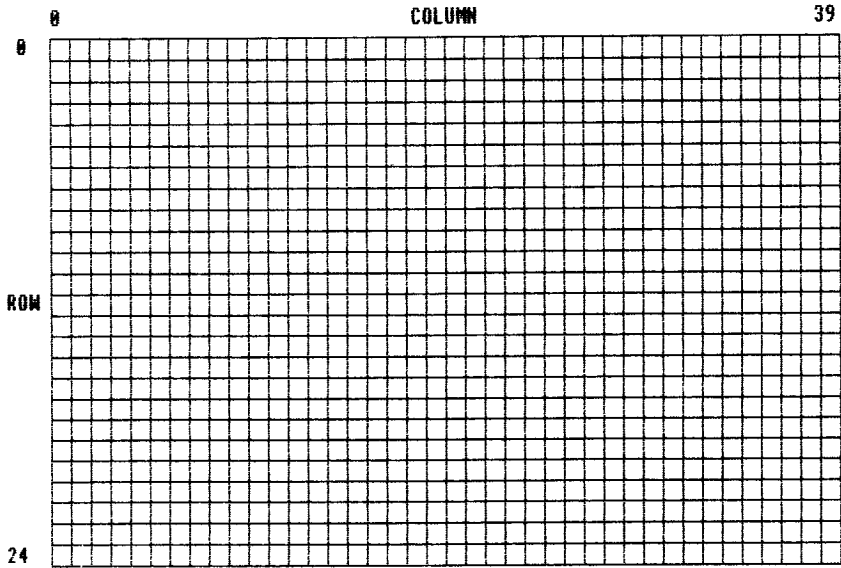
First let us see what PRINT AT does. The statement usually takes the form

```
PRINT AT R,C;"Text"
```

where R and C are numbers which define the row and column position on the screen where printing is to start. Sometimes R and C may be replaced by a single number which indicates the number of the character space starting from the top left corner and working across each row and down the screen. Of these forms the R,C version is probably easier to understand and use, although the version with a single variable N is easier to implement on the Commodore 64. In this book the R,C form of position co-ordinates will be used. The



range of R and C and their relationship to the display area on the screen are shown in Fig. 2.3.



*Fig. 2.3.* Screen layout for text displays showing the row and column numbering.

One simple approach to producing a PRINT AT operation is to start by using HOME to place the cursor at row 0 column 0. To select the required column, a count loop is set up running from 1 to C, where C is the column number. On each pass through the loop, a single CRSR RIGHT code is printed, moving the cursor one space to the right. At the end of the loop the cursor is at the required column. A problem occurs if the required position is in column 0. To deal with this a test for C=0 is included before the loop and if C=0 then the loop is bypassed and the program jumps to the statement number following the loop. A similar process is then used to set up the required row position. This gives a routine on the following lines:

```

500 IF C=0 THEN 530
510 FOR I=1 TO C
520 PRINT CHR$(29);:NEXT
530 IF R=0 THEN 560
540 FOR I=1 TO R
550 PRINT CHR$(17);:NEXT
560 RETURN

```

Here a subroutine has been used, and after C and R have been given values the cursor is positioned by using GOSUB500.

Although the Commodore 64 doesn't have a PRINT AT command it does keep track of the current position of the cursor on the screen. This information is stored in memory locations 209, 210 and 211.

The column number from 0 to 39 is held in location 211. If we want to know the current column position of the cursor this can be done by using

$$C = \text{PEEK}(211)$$

If we want to place the cursor at a particular column position across the screen this can be done by POKEing the column number into memory location 211 as follows:

$$\text{POKE } 211, C$$

where C is the column number from 0 to 39.

When we come to deal with the row number, things are a little more complicated. Here the Commodore 64 stores the memory address where the first character in the row will be stored in the screen memory. Since the screen memory starts at location 1024, the row address for the first row will be 1024, 1064 for the next row, and so on with each row adding 40 to the address. Thus the row address is given by

$$R1 = 1024 + 40 * R$$

where R is the row number from 0 to 24. This gives a number between 1024 and 1984, which is too big to fit into a single memory word, so the computer uses two words (209 and 210) to store this number. The storage in memory is in binary form, and one word (209) represents numbers up to 256 while the second, 210, represents the number of 256s. To find the number that goes into location 210 we take R1, divide by 256, and discard the fractional part as follows:

$$R2 = \text{INT}(R1 / 256)$$

Now to find the remainder that has to go into location 209 we can use the following:

$$R3 = R1 - 256 * R2$$

Thus to set the cursor locations in memory we can use the following little subroutine:

## 20 Commodore 64 Graphics & Sound

```
500 POKE211,C
510 R1 = SM + 40*R
520 R2 = INT(R1/256)
530 POKE210,R2
540 POKE209,R1-256*R2
550 RETURN
```

Here the variable SM (start of screen memory) has been included in case you have moved the position of the screen memory from its normal address of 1024.

If we want to read the cursor position this can be done by PEEKing the locations 209 to 211, and then calculating R and C as follows:

$$C = \text{PEEK}(211)$$
$$R = (256 * \text{PEEK}(210) + \text{PEEK}(209) - 1024) / 40$$

In this calculation 1024 could be replaced by SM if the screen memory start address was likely to be different from 1024.

### Another way of setting R and C

There is a useful machine code routine in the Commodore 64 computer's own operating system which we can use to set up the cursor position on the screen. This accepts the row and column numbers, and then automatically sets up the cursor position data in locations 209 to 211.

This routine requires that we transfer data to the A, X, and Y registers of the processor and then call the machine code subroutine to set up the cursor position. The data is actually transferred into three memory locations as follows:

```
A register = location 780
X register = location 781
Y register = location 782
```

The data is transferred by simply POKeIng the numbers into the appropriate memory locations as follows:

```
POKE 780,0
POKE 781,R
POKE 782,C
SYS65520
```

The SYS statement calls the machine code subroutine which starts at memory location 65520. When the cursor has been positioned, the BASIC program is resumed at the next line number. This routine has the advantage that it uses R and C as direct inputs and automatically takes care of the actual address of the screen memory.

## Drawing lines

For many graphics applications we shall want to draw lines on the screen, perhaps to separate the columns of a table of figures, or to make up simple diagrams. This can be done by using the graphics symbols containing line elements.

For horizontal lines there are eight basic symbols, each containing a horizontal line across the symbol space in one of the row positions, and these are shown in Fig. 2.4 together with their ASCII and screen memory codes. By using the appropriate symbol, and repeating it across a text row as required, we can draw a horizontal line at any desired position on the screen. The limitation is that the line must start and end at the edge of one of the character columns.









ASCII CODE	SCRN CODE		ASCII CODE	SCRN CODE	
163	99		96	64	
101	69		102	70	
100	68		114	82	
99	67		164	100	

Fig. 2.4. The horizontal line segment symbols and their codes.

Vertical lines can be handled by another set of eight graphics, each with a line of dots running from top to bottom in one of the columns of the symbol matrix. Here we can draw vertical lines by PRINTing the symbols one above the other on the screen. This can be done by adding 'cursor left' and 'cursor down' control codes after each symbol to move the cursor to the correct position for the next

graphics symbol. These symbols, and their ASCII and screen codes, are shown in Fig. 2.5.









ASCII CODE	SCRN CODE		ASCII CODE	SCRN CODE	
165	101		125	93	
116	84		104	72	
103	71		121	89	
98	66		167	103	

Fig. 2.5. The vertical line segment symbols and their codes.

When we come to sloping lines, things get a little more complicated. If you print the set of horizontal line symbols one after another, starting with the symbol with a line at the top of the symbols space, and with each successive line one row of dots lower, this will give a stepped line that slopes down the screen to the right. If you want the line to slope upwards, the symbols are simply printed in reverse order. After eight symbols have been printed, a 'cursor up or down' move is required, and then the sequence starts again to continue the sloping line. The same basic idea can also be applied using the vertical line segments to generate stepped lines that slope a little to the right or left of vertical. Here cursor moves are needed after each symbol, and an extra one is needed after every eight symbols.

Lines can be made to slope at steeper angles by using alternate symbols from the set of eight. Thus you might use the symbols with the line drawn in rows 1, 3, 5 and 7. For extended lines the cursor shift to the next row or column will need to be made after only four symbols have been printed. The possible variations of this technique are numerous, and it is worthwhile experimenting to see the range of different lines that can be produced by just using the horizontal and vertical line segment symbols.

There are two symbols to deal with diagonal lines, and these are shown in Fig. 2.6. When drawing a line using these symbols, one or more cursor control codes will be needed after each graphics symbol




ASCII CODE	SCREEN CODE	SYMBOL
109	77	
110	78	
118	86	

Fig. 2.6. The diagonal line symbols and their codes.

to place the cursor in the correct position for the next section of line to be printed. A third symbol gives crossed lines.

Crossing lines and junctions present something of a problem. There are four right-angle T junctions and eight corner shape symbols. These allow crossovers and junctions to be made at the corners or middle of character spaces. There are also some curved corner shapes.

The graphics symbols may be obtained from the keyboard by holding down either the SHIFT or LOGO keys while a letter key is pressed. The LOGO key is the one at the bottom right of the keyboard, with the Commodore trademark printed on it. The actual graphics symbols are printed on the front face of the key. You will note that there are two symbols on each key. To select the right-hand symbol use the letter key with the SHIFT key; for the left-hand symbol use the letter key with the LOGO key.



This set of line symbols can give remarkable flexibility in building up a diagram, but some care is needed in planning the layout of the diagram to ensure it will fit in with the available symbols. The basic technique involved in producing a picture on the screen is similar to that of building up a jigsaw puzzle.

## Reverse video

You will have noticed that some of the symbols displayed when you type in a cursor control code are shown in *reverse video*. This means the dots that would normally be lit are dark, and those that would normally be dark are lit. Thus the HOME key produces an S symbol in dark blue on a light blue background.

Any symbol or set of symbols can be displayed in this fashion by using [CTRL 9] to select the RVS ON mode, where all symbols typed in will be displayed in reverse video form. To return to the normal display mode [CTRL 0] (RVS OFF) is used. Note that it is possible to have just one or two symbols in a string in reverse video by just switching RVS on for the selected symbols. At the end of a PRINT line the RVS ON mode is automatically turned off.

Reverse video is useful with the mosaic block symbols since several patterns do not appear to be available. If you examine the symbol pattern, however, you will see that the missing symbols can be generated by using RVS with those that are directly available from the keyboard. The ASCII codes and listing symbols for the RVS ON and RVS OFF commands are shown in Fig. 2.7.

Keys used	Action	ASCII code	List symbol
CTRL 9	RVS ON	18	
CTRL 0	RVS OFF	146	

*Fig. 2.7.* The keys, character codes and listing symbols for switching to and from the inverse video symbols.

### **Making new characters**

So far we have produced drawings and pictures or text displays by using the character sets programmed into the character generator ROM built into the Commodore 64. For most purposes these may well be perfectly adequate, but there will be times when we may want to display some special symbols that are not available in the standard set. As an example, we might (in a mathematical program) want to introduce Greek letters or special signs, such as the integration and square root symbols.

For producing text displays in other languages we may require accented letters, or in the case of Russian we might use the Cyrillic alphabet. Once again, there is a need for some custom-designed symbols.

In graphics we have already seen that there are some limitations in using the standard set of graphics symbols. Here it might be useful to have some new graphics symbols to suit the needs of the particular

type of diagram we want to produce. Examples here might be the inclusion of special symbols, such as those used in electrical and other types of engineering drawing.

If, instead of taking the character dot patterns from the character generator ROM, we could persuade the VIC11 chip to use dot patterns from the RAM area of memory, then we could write in our own dot patterns to define the shapes of the displayed symbols. In fact this facility for producing *user-defined* text and graphics symbols is provided on the Commodore 64.

The character generator ROM is actually located at addresses starting at 53248 in the memory address map of the computer. As far as the VIC11 chip is concerned, however, the ROM *appears* to be located at addresses 4096 to 6143 in memory. This is because the video chip only looks at a 16k byte block of memory at any time. If the alternative (lower case) character set is in use then the ROM appears at locations 6144 to 8191. In fact there is also a section of RAM at this address, but it is disabled when the video chip calls up dot patterns for a symbol, and the data is read from the ROM instead. The address that the VIC11 chip looks at for its dot pattern data can, however, be altered, and if this is done the dot patterns will actually be read from the RAM itself. The address looked at is controlled by one of the registers in the video chip, so by changing the data in this register we can pick up our own set of symbol dot patterns for display on the screen.

The character data address register in the video chip is at memory location 53272, and the lower four data bits of the word are used to select a 2k block of memory where the video chip will read data for its symbol patterns. The number we feed into this register is from 0 to 14, and is always even. In fact, this number is simply the number of 1k bytes to the start of the character pattern memory, and will be either 0, 2, 4, 6, 8, 10, 12 or 14. Note that the video chip only looks at one 16k bank of memory at a time, so there is no need for addresses higher than 16k.

If we select blocks 4 or 6, the normal ROM will be used. Block 0 must be avoided, since this part of the memory is used by the computer itself to store system variables, and if overwritten will cause complete chaos. Similarly, block 2 should be avoided, since this is where the BASIC program starts. The safest place to start is at 12k (12288), and this can be done by POKEing 12 into location 53272. A point to watch is that the upper four bits of this register in the VIC chip control the location of the screen memory, so you do not want to disturb them. The solution is to use the following



command:

```
POKE 53272, (PEEK(53272)AND15)OR A
```

where A is the dot pattern start address in k bytes. Here the AND operation sets all the lower four bits at 0 without affecting the upper four bits. Then the OR operation sets the lower four bits to their new value, again without affecting the upper four bits.

### Copying the ROM character set

If we simply shift the location of the dot pattern memory being used by the video chip, and then PRINT characters to the screen, the result will be random patterns of dots. This is because the area of RAM that we have defined as the character generator contains random data: we haven't programmed any data into it. Now for most programs we shall still want to use some of the standard symbol set, so the first thing we need to do is to copy the character dot pattern data from the character generator ROM into the area of RAM we are going to use as a character generator.

Normally if we tried PEEKing the addresses at which the ROM appears to the VIC chip we would simply get the contents of the RAM at these addresses. This is because the ROM is normally disabled unless the VIC chip is addressing it. To switch on the ROM so the CPU can read data from it we need to change a data bit in location 1 of the memory.

An important point, however, is that memory address locations 0 and 1 in the Commodore 64 do in fact call up two registers in the CPU itself. These registers are used in interrupt operations such as the one involved in scanning the keyboard. To avoid problems it is advisable to disable the keyboard and other interrupts while you are copying data from the character generator ROM. This can be done by using

```
300 POKE 56334, PEEK(56334) AND 254
```

With the interrupt system disabled we can now use register 1 to switch on the character generator ROM. This is done by resetting bit 2 of the word, which has a value of 4, by using the POKE operation

```
310 POKE 1,PEEK(1) AND 251
```

When the ROM is enabled, it will appear to the CPU at addresses from 53248 to 57343, and all we need to do at this stage is set up a

simple loop operation to copy data words from the ROM addresses into the area of RAM we are going to use for our user-defined symbols. This can be done as follows:

```
320 FORJ=0TO4095
330 POKEJ+12288,PEEK(53248+J)
340 NEXT
```

After copying the dot data the ROM can be disabled by using

```
350 POKE 1,PEEK(1)OR 4
```

and then the interrupts reactivated by using

```
360 POKE 56334,PEEK(56334) OR 1
```

At this stage we now have all the standard dot patterns in memory, starting at location 12288, and if we now switch the character memory address of the VIC11 chip to 12288 by using

```
370 POKE 73272, (PEEK(53272)AND240)OR12
```

then the computer will appear to behave normally as we type in data or PRINT characters to the screen.

Like the other text characters, each of the user-defined symbols has eight rows with eight dots in each row, and each dot may be either 'on' or 'off'. In the computer, each memory word has eight bits, each of which may be set as a 1 (on) or a 0 (off), so it is convenient to store one row of dots from the character pattern into one memory word. The eight rows of dots making up the character are then stored in eight successive memory words. If we want to create a new symbol then the new dot pattern must be written into a set of eight memory locations in the user-defined graphics area of the memory, replacing one of the standard dot patterns that we have just copied from ROM.

Of course we may just want to rearrange the existing set of symbols, and this can be done by copying the required dot patterns from the ROM into the RAM at a different position in the code table. We could also mix symbols from the two sets that are available in the ROM. The second set of symbols is located at addresses 2048 higher in the ROM, that is from 55296 upward.

## Programming a new symbol

The first step in creating a new symbol is to work out the dot pattern

that is needed to build up the symbol. This can easily be done by drawing a grid with eight rows of squares and eight squares in each row, as shown in Fig. 2.8. Squares are then shaded in to pick out the shape of the desired symbol.

128	64	32	16	8	4	2	1	DATA BIT PATTERN	DECIMAL
								0 1 1 1 1 1 1 1	127
								0 0 1 0 0 0 0 1	33
								0 0 0 1 0 0 0 0	16
								0 0 0 0 1 0 0 0	8
								0 0 0 1 0 0 0 0	16
								0 0 1 0 0 0 0 1	33
								0 1 1 1 1 1 1 1	127
								0 0 0 0 0 0 0 0	0

Fig. 2.8. Dot matrix and data coding for a user-defined symbol.

Once the dot pattern has been worked out, the next step is to work out the numbers that have to be stored in the memory. Figure 2.8 shows the layout of a typical user-defined graphics character. Here the pattern is of the Greek letter sigma. In the diagram, the black dots are in the foreground colour and will be represented by '1's in the computer word, while the remaining dots are in background colour and will be '0's. Each data bit in the word has a numerical value starting with 1 for the right-hand end bit, and working up in the sequence 2, 4, 8, 16 and so on for successive bits as we move to the left through the data word. The actual value of each bit is shown at the top of the diagram.

To find the decimal number that has to be fed into the computer, we can simply add together the numerical values for all the bits in the word that are set at '1'. This gives a number in the range 0 to 255.

To set up the dot pattern in memory we now have to write the sequence of eight numbers into eight successive memory locations, and this can easily be done by using a series of POKE commands as follows:

```
500 FOR K = 0 TO 7
510 POKE A+K,D(K)
520 NEXT K
```

where A is the starting address in the computer memory for the dot

pattern of the character we are creating, and D is an array of eight numbers representing the dot pattern.

All we have to do now is put the data words into the right place in memory. The first step here is to decide what screen code you want the new symbol to have. Let us say that this is some variable C. Now we have to pick up the memory address  $C*8$  words up from the start of the character RAM area, and we write our eight data words into the next eight successive memory locations. Suppose we wanted to replace the standard character 'C'. This has a screen code of 3, so the dot data words would go into memory starting at location  $12288+(3*8) = 12312$ , and going up to location 12319.

```

100 REM CREATING A NEW SYMBOL
105 REM SELECT UPPER CASE SYMBOLS
110 PRINT CHR$(142)
115 REM PROTECT NEW CHARACTER MEMORY
120 POKE52,48:POKE56,48
130 UG=12288:CG=53248
135 REM TURN OFF KEYBOARD INTERRUPT
140 POKE56334,PEEK(56334)AND254
145 REM ENABLE CHARACTER ROM
150 POKE1,PEEK(1)AND251
155 REM COPY CHARACTER SET
160 FORK=0 TO 2047
170 POKE UG+K,PEEK(CG+K)
180 NEXT
185 REM DISABLE CHARACTER ROM
190 POKE1,PEEK(1)OR4
195 REM RESTORE KEYBOARD INTERRUPT
200 POKE56334,PEEK(56334)OR1
205 REM SELECT USER GRAPHICS RAM
210 POKE53272,(PEEK(53272)AND240)+12
220 CC=1
225 REM LOAD NEW SYMBOL DOT PATTERN
230 FOR J=0 TO 7
240 READ A:POKE UG+8*CC+J,A
250 NEXT
260 DATA 127,33,16,8,16,33,127,0
270 PRINT CHR$(147)
280 FOR N=64 TO 80
290 PRINT CHR$(N);" ";
300 NEXT

```

*Fig. 2.9.* Program demonstrating copying of the character set and creation of a user-defined symbol.

In Fig. 2.8 the dot pattern for the Greek letter sigma is shown and alongside it are the required data words that will represent the dot pattern in RAM. The program listed in Fig. 2.9 demonstrates how this new character can be set up in the user graphics area and used on the display screen.

In line 110 the upper case character set is selected. This is the normal symbol set with the graphics symbols. In line 120 the character RAM area of memory is protected from BASIC by altering the top address available to BASIC programs. Lines 140 and 150 turn off keyboard interrupts and enable the character generator ROM so that we can copy its dot patterns. The complete set of character dot patterns is then copied into the user-defined character RAM area of memory. Next the ROM is disabled and the interrupt restored.

Line 210 now selects the user graphics RAM dot patterns for use by the display. The new pattern for the sigma symbol is then POKEd into the character RAM in place of the dot pattern for the A symbol. Finally the screen is cleared and the set of letter symbols is printed out. You will notice that the A is now displayed as a sigma. When the program completes, the word READY will also contain the sigma symbol instead of an A and in fact all As will be displayed as sigmas while the user graphics RAM is selected. The new dot pattern will remain in the user-defined graphics set as long as the computer remains switched on. Of course you can define as many new symbols as you like, and use them to replace any of the existing symbols in the standard character set.

## Chapter Three

# High Resolution Graphics

So far the pictures on the screen have been produced by printing patterns of graphics symbols. As we have seen, there are limitations in using the standard set of graphics symbols, but these can often be overcome by creating user-defined symbols. The lack of flexibility of character graphics becomes important when we want to draw shapes such as circles. Much greater flexibility in drawing shapes is possible by making use of the high resolution bit-mapped graphics mode provided on the Commodore 64. This gives a screen resolution of 320 by 200 dots and permits the state of each dot to be individually controlled.

One penalty that has to be paid for this extra flexibility is loss of memory space: the bit map display mode uses up an 8000-word area of memory compared with only 1000 words for the character graphics mode. With some 30k of memory available, this need not be a serious problem. Another and perhaps more significant disadvantage is that the normal PRINT command no longer produces a text display when bit-mapped graphics are selected. As a result, the combination of text and graphics on the high resolution displays becomes a more complex operation than it is with the simple character graphics display.

### Memory allocation

When the bit map graphics mode of the Commodore 64 is selected it uses an 8000-word section of memory, which starts at the same address as the user graphics memory. As with the user-defined graphics memory, the start address of the memory used in the bit map mode has to be set up in the VIC11 chip. Here the start address must be chosen to avoid conflict with memory used by the processor itself, or by the BASIC interpreter. One possible start address for the

bit mapped display memory is at a point 8k (8192) bytes up from the start of memory.

The choice of memory position is rather more limited than for the user-defined symbol set, because now we have to fit in an 8000-word block of memory which must fall within the 16k bank of memory being accessed by the VIC11 chip. If you set the bit map at an address between 8k and 16k then part of the map will fall outside this 16k block, and will not be displayed. The section of memory used for the bit map mode is governed by register 24 in the VIC11 chip. This is in fact the same register that specified the start of the user-defined character patterns in memory, and the start address may be set in steps of 2k bytes at a time.

To set the bit map memory to the required position we need to POKE a new number into register 24 of the video chip, which is at memory address 53272. This register has only 3 bits (bits 1, 2 and 3) allocated for setting the bit map address; the remaining bits control other functions. A point to note is that the least significant bit of the register is not used; this is why the selected memory start address increases in steps of 2k bytes. At switch-on the bit map address bits are usually set at 0. To avoid altering the other bits in the register it is best to PEEK the contents, then add in the memory address bits and POKE the result back into the register. This can be done by using

```
POKE53272,PEEK(53272) OR MS
```

where MS is the bit map start address in kilobytes. So to place the bit map at address 8192 (8k), MS would have the value 8. If the address in the register might have altered from 0, perhaps because you have been using custom symbols, then it is as well to AND the result of the PEEK with 240 before adding in the bit map address. This will set the lower four bits of the register to 0.

It is a good idea to protect the bit map area of memory from BASIC. This can be done by the following two POKE operations:

```
POKE 52, 32
```

```
POKE 56, 32
```

These may be included as the first instructions of your BASIC program, or they may be carried out directly by just typing them in without line numbers. The data to be POKEd into locations 52 and 56 will be the top address you want BASIC to use in kilobytes, multiplied by 4. If the number POKEd into these locations is 32, BASIC will 'think' that the top of the available memory is at 8192, which will leave you about 6000 bytes of memory for your BASIC program and its variables.

## **Setting the bit map mode**

Once you have placed the display memory in an appropriate position, the next step in achieving bit-mapped high-resolution graphics is to select the bit map mode. This is done by setting a control bit in register 17 of the video display chip. This register is located at memory address 53265, and the particular bit we need to set is bit 5, which has a numerical value of 32.

The control register which governs the bit map mode also controls other functions within the video display chip, so we want to alter the state of bit 5 without altering any of the other bits in the register. To discover what is already in the register we can use PEEK(53265). If we want to set bit 5, the simplest approach is to OR the contents of the register with the number 32. For the number 32 the only bit which is at 1 will be bit 5, and this is the only bit that will be affected by the OR function. So to select the bit map mode we can use the statement

```
POKE 53265,PEEK(53265) OR 32
```

If we want to switch off the bit-mapped display mode and return to normal text displays then we need to reset bit 5 of the control register to 0. This can be done by using the AND function. In this case the contents of the register are ANDed with a mask pattern word where all bits except bit 5 are set at '1'. This pattern is produced by the number 255-32, or 223, and the required statement becomes

```
POKE 53265,PEEK(53265) AND 223
```

This is how the mode would be switched during a program. Another and perhaps easier way of restoring the normal display mode after the program has stopped is to hold down the RUN/STOP key and press the RESTORE key. If you want to switch back and forth in a program, however, the POKE commands must be used.

## **Clearing the screen**

When the bit map mode has been selected the normal 'screen clear' function of the text mode no longer operates, so a different technique for clearing the display has to be used. Remember that in the bit map mode the state of the individual bits in the words of the display memory determines whether the dots are turned on or not. To clear the screen we simply have to set all the words in the bit map



memory to zero. This can be done in BASIC by using a simple loop which POKES zero successively into each location of the bit map display memory. This might be done as follows:

```
1000 BM = 8192
1100 FOR N=BM TO BM+8000
1200 POKE N,0
1300 NEXT
```

Here the loop starts at the start address of the bit map (BM) and steps through the next 8000 memory locations placing 0 into each one.

When you have cleared the display memory there will still be a random selection of coloured blocks on the screen. This is because when the bit map mode is selected the text display screen memory is used to control the colour of the dots on the bit-mapped screen. We shall be looking at colour more closely in the next chapter. For the moment we shall set the background colour to blue and the foreground colour to white. To do this we simply have to POKE the number 22 into every location of the text screen memory.

### Setting individual dots

To draw effectively on the high resolution screen the first thing we need to be able to do is to set individual dots 'on' or 'off' as required. To do this we need to POKE a 1 into the appropriate bit of one of the words in the display memory. The problem here is to convert from a set of X and Y co-ordinates to a word address and a bit pattern to go into that word.

It is convenient to arrange the X and Y co-ordinate system as shown in Fig. 3.1. Here the value of X increases from 0 at the left edge of the screen to 319 at the right edge; Y starts at 0 at the top of the screen and increases to 199 at the bottom of the screen.

The bit map memory is arranged in what seems at first to be a rather strange layout. One might expect the words to be allocated along each row of dots on the screen, with each word representing eight successive dots along the line. In fact, the memory is arranged so that the eight rows of each character space are stored successively in memory, and these sets of eight rows of dots are taken in sequence across the screen as shown in Fig. 3.2.

The lines across the screen are effectively stacked in sets of 8, so that after the first 8 lines have been stored the next memory location

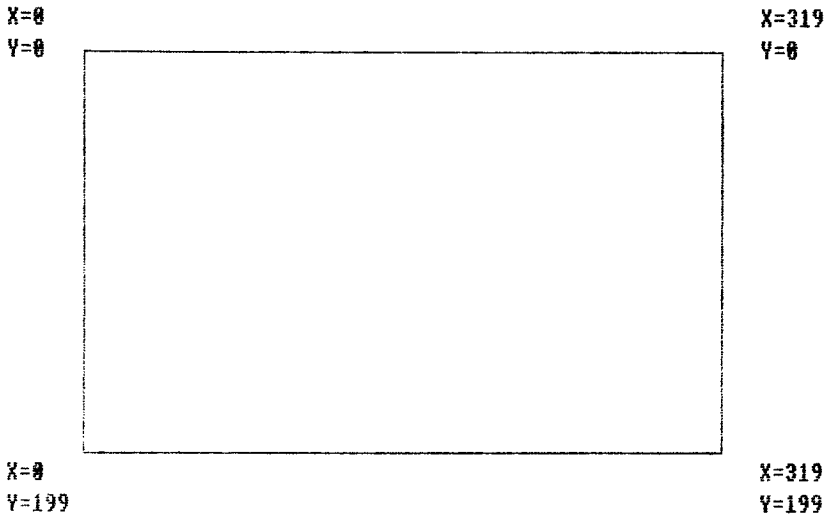


Fig. 3.1. Layout of the X and Y co-ordinates on the bit map screen.

is at the start of the next block of 8 lines. This arrangement is in fact the same as that used for the user-defined graphics patterns. There are some advantages to this arrangement when text has to be added to a bit-mapped picture, as we shall see later.

```

byte0      byte8      byte16     ....  byte304  byte312
byte1      byte9      byte17     ....  byte305  byte313
byte2      byte10     byte18     ....  byte306  byte314
byte3      byte11     byte19     ....  byte307  byte315
byte4      byte12     byte20     ....  byte308  byte316
byte5      byte13     byte21     ....  byte309  byte317
byte6      byte14     byte22     ....  byte310  byte318
byte7      byte15     byte23     ....  byte311  byte319
byte320    byte328    byte336     ....  byte624  byte632
byte321    byte329    byte337     ....  byte625  byte633
byte322    byte330    byte338     ....  byte626  byte634

```

Fig. 3.2. Layout of the bit map screen data in the computer memory.

Let us start with the X direction. Each successive word across a single line is spaced 8 locations on from the last, so we can find part of the address by dividing X by 8 and taking the integer value. Each group of 8 scan lines is equivalent to a row of text symbols, and the next step is to find out which row we are in. This can be done by simply dividing Y by 8 and then taking the integer to remove any fractional parts. Now in each row there are 8 scan lines with a total of

$40 \times 8$  or 320 individual words. So the second block of 8 lines starts 320 memory locations from the start of memory. If we multiply the row number by 320 this gives the start address for the block of 320 memory locations that we want to access. The next step is to find out which scan line of the set of 8 the dot to be set is in. This is related to the Y position, and must be a number from 0 to 7. For this we can simply take the lowest part of the Y value by using  $Y \text{ AND } 7$ , which selects out the three least significant bits (value 0 to 7) of the Y number.

Assembling these steps together to find the particular word to be altered we get the equation

$$P = BM + 320 * INT(Y/8) + 8 * INT(X/8) + (Y \text{ AND } 7)$$

where  $BM$  is the start address of the bit map memory, and  $P$  is the address of the location we need to alter.

The next step is to set the required bit in the word. The bit number is obtained by taking the three least significant bits of the X number, which tells us which of the 8 bits is to be set. This is done by simply ANDing X with 7. Now in the display memory word each bit has a value which is 2 raised to a power from 0 to 7 according to the bit position in the word. So bit 0 (the right-hand one) has a value of 1 ( $2^0$ ), the next bit is 2 ( $2^1$ ), and so on to the left-hand bit which has the value 128 ( $2^7$ ). The value of X, however, runs from 0 to 7, moving left to right through the word, so to set the required bit we have to calculate  $2^{(7-(X \text{ AND } 7))}$ . The result of this calculation is Ored with the existing contents of the memory location at address P, and then the result is POKEd back into that memory location.

This process of setting a dot may seem rather complex, but in fact boils down to a simple three-line subroutine. Now all we have to do to plot a point is to set X and Y to the co-ordinates of the point on the screen, and then call the subroutine to light up the required dot. To reset a dot to its 'off' state we need to AND the contents of the required memory location with  $255 - DV$ , where DV is the value of the data bit representing the dot we want to reset.

The program listed in Fig. 3.3 sets up the bit map mode, clears the screen, and then plots a series of randomly-positioned dots all over the screen. The dot setting subroutine starts at line 2000. A point to note here is that the screen clearing operation takes several seconds, during which time nothing much will appear to happen.

```

100 REM HI-RES RANDOM DOTS
105 REM CLEAR BIT MAP AREA
110 MS=8192
120 PRINT"SETTING UP"
130 FOR M=MS TO MS+8000:POKE M,0:NEXT
135 REM MOVE BIT MAP ADDRESS
140 POKE53272,PEEK(53272) OR 8
145 REM SELECT BIT MAP MODE
150 POKE 53265,PEEK(53265) OR 32
155 REM SET COLOURS
160 FOR I=1024 TO 2023:POKE I,22:NEXT
170 GOTO1000
195 REM DOT SET ROUTINE
200 P=MS+320*INT(Y/8)+8*INT(X/8)+(YAND7)
210 POKE P,PEEK(P) OR(2*(7-(X AND 7)))
220 RETURN
995 REM MAIN PROGRAM
1000 FOR N=1 TO 200
1010 Y=200*RND(0)
1020 X=320*RND(0)
1030 GOSUB200
1040 NEXT
1050 END

```

*Fig. 3.3.* Simple BASIC program to select the bit map mode and plot random dots.

### Machine code screen clear

Waiting for the computer to clear the screen using a BASIC program is a rather tedious business, as you will just have discovered. The solution is to use a machine code routine to perform this task, and to call it from BASIC whenever the bit map screen needs to be cleared.

Figure 3.4 shows the list of assembly language instructions for a short machine code routine which will clear the bit map screen. The machine code itself is set up in the computer by using a short piece of BASIC program to POKE the data representing the machine code into the computer memory. This is shown in Fig. 3.5. Here we have placed the machine code in an area of memory starting at location 49152 by using a loop to read the required data numbers and POKE them into successive memory locations. This area of memory is above that used by BASIC, so it will not be overwritten by your

START	LDA	#0	Start address
	STA	251	
	LDA	#32	in 251/2
	STA	252	
	LDX	#32	No. of blocks
BLOCK	LDY	#0	
BYTE	LDA	#0	
	STA	251,Y	Set byte to 0
	DEY		Next byte
	BNE	BYTE	End of block?
	INC	252	Update address
	DEX		to next block
	BNE	BLOCK	All done?
	RTS		Return

Fig. 3.4. Machine code program to clear the bit map screen.

```

100 REM BIT MAP CLEAR M/C ROUTINE
105 REM CLEARS 8000 BYTES FROM 8192 UP
110 REM CALL USING SYS49152
120 T=0:FOR N=49152 TO 49176
130 READ A:POKE N,A:T=T+A:NEXT
140 READA:IF T>A THEN PRINT"ERROR":END
150 PRINT"LOADED OK"
160 DATA 169,0,133,251,169,32,133,252
170 DATA 162,32,160,0,169,0,145,251
180 DATA 136,208,251,230,252,202,208
190 DATA 246,96,3887

```

Fig. 3.5. BASIC program to load machine code screen clear routine.

BASIC programs or their data and variables. Once the data has been POKED into the memory the machine code routine will remain available all the time the machine remains switched on. Typing NEW doesn't affect the machine code area, so you could use this short BASIC program to load the screen clear routine at the start of a computing session, then type NEW and load the BASIC programs that will use the routine.

Be very careful in entering the DATA items: an error here can cause unpredictable results when the machine code routine is executed, and will probably cause the computer to become totally locked-up. A simple checking routine has been built into the program which should pick up most data entry errors. This check adds up all of the DATA items and compares the result with a final check DATA word. If the results match, the message 'LOADED

OK' is printed, but if a mismatch occurs the message 'DATA ERROR' is printed. This simple check is not totally infallible, since two errors in succession could cancel one another and give the correct check answer, but the machine code will still be in error.

```

100 REM RANDOM BIT MAPPED DOTS
105 REM SET BIT MAP ADDRESS TO 8192
110 BM=8192
120 POKE53272,PEEK(53272)OR8
125 REM SELECT BIT MAP MODE
130 POKE53265,PEEK(53265)OR32
135 REM SET COLOURS
140 FOR I=1024 TO 2023:POKEI,22:NEXT
145 REM USE CLEAR ROUTINE AT 49152
150 SYS49152
160 GOTO1000
195 REM DOT SETTING SUBROUTINE
200 P=BM+320*INT(Y/8)+8*INT(X/8)+(YAND7)
210 POKE P,PEEK(P)OR(2↑(7-(XAND7)))
220 RETURN
995 REM MAIN PROGRAM
1000 FOR N=1 TO 200
1010 X=320*RND(0)
1020 Y=200*RND(0)
1030 GOSUB200
1040 NEXT
1050 END

```

*Fig. 3.6.* Random bit-mapped dots program using machine code screen clear routine.

The machine code screen clear operation is called from a BASIC program by using the instruction SYS49152 as a BASIC statement. In the program shown in Fig. 3.6 the screen is set to bit map mode, then cleared, and then the random dots are plotted as before. This time the process repeats 20 times, and you will notice that the screen clears almost instantly.

## Drawing lines

Now that we can plot individual points on the screen, the next step is to draw lines. Suppose we want to draw a horizontal line across the screen from a point  $X_1, Y$  to another point  $X_2, Y$ . Since the line is horizontal, all the points along it will have the same  $Y$  value. To

draw the line on the screen we have to set all the dots along the line from point X1 to point X2 to the lit or '1' state. This can easily be done by using a simple loop operation where the X value is stepped from X1 to X2 and a dot is set at each step as follows:

```

300 FOR X = X1 TO X2
310 P=320*INT(Y/8)+8*INT(X/8)+(YAND7)
320 B=21(7-(XAND7))
330 POKE BM+P,PEEK(BM+P) OR B
340 NEXT

```

Here it is assumed that the bit map mode has been set up and values have been given to X1, X2 and Y earlier in the program.

Drawing vertical lines on the screen follows a similar procedure to the drawing of horizontal lines, except that now the X co-ordinate of the points remains constant and the Y co-ordinate is stepped from Y1 at one end of the line to Y2 at the other, with a point being plotted at each step.

A general line-drawing routine should be able to draw lines at an angle as well as horizontal and vertical lines. This requires a more complex routine, since both X and Y co-ordinates now have to be changed as the line is drawn.

If the line is to be a diagonal one, at 45 degrees, the X and Y changes are equal, and the direction of the line will simply be determined by the signs of the steps in X and Y. Thus if both X and Y increase by one for each successive dot the line is drawn diagonally downward and to the right. If X decreases by one and Y increases by one the line is drawn down and to the left. If Y decreases by one at each step then the line will be drawn up the screen and to left or right according to whether X decreases or increases.

If the line is not at 45 degrees then the steps in X and Y will be different from one another. The technique here is to choose the larger co-ordinate change from one end of the line to the other and make that the total number of steps to be plotted. Now some steps will need to be made diagonally, while between them there will be vertical or horizontal steps according to which direction has the greater change in position.

Let us consider an example. Suppose we want to draw a line from point X1,Y1 = 50,50 to point X2,Y2 = 150,80. First we check whether X or Y has the largest change in value. In this case it is X which changes by 100 while Y changes by only 30. So here we make the total number of steps equal to 100.

In the program this can be done by using an IF statement to

compare  $\text{ABS}(X_2 - X_1)$  with  $\text{ABS}(Y - Y_1)$ . The absolute values must be used since either  $X_2 - X_1$  or  $Y_2 - Y_1$  may be negative, and this would give the wrong answers. We are only interested here in the size of the difference in the X and Y co-ordinates of the line.

If  $X_2$  is larger than  $X_1$  the line is drawn from left to right, whereas if  $X_2$  were smaller than  $X_1$  the line would extend to the left and the X value would have to be reduced by one for each successive dot. The direction in which the steps have to be plotted is governed by the signs of the  $(X_2 - X_1)$  and  $(Y_2 - Y_1)$  differences, and these are determined by using  $\text{SGN}(X_2 - X_1)$  and  $\text{SGN}(Y_2 - Y_1)$ . The results are held as variables SX and SY, which will have values of -1, 0 or +1.

To draw the line we shall now plot a total of 100 dots along the line. Since the change in Y is only 30 there will be 30 diagonal steps, and the remaining 70 steps are in the X direction only. To give a smooth line the diagonal steps must be spread out evenly along the line.

To do this we set up a running total ND which is initially set to the integer value of half the larger of the X or Y differences. In this case

$$\text{ND} = \text{INT}(\text{NX})/2 = 50$$

Now we set up a loop running from 1 to NX, in this case. If NY were greater it would determine the size of the count. On each pass through the loop the lower of the two differences (NY) is added to ND, and the result is compared to the larger difference (NX). Normally a horizontal or vertical step is made, but when the new total becomes larger (i.e. here  $\text{ND} > \text{NX}$ ) a diagonal step is made. After the diagonal step the larger difference, in this case NX, is subtracted from ND. In the example, ND starts at 50 and we add NY (30) giving 80. This is less than NX, so the first step is horizontal. On the next pass ND becomes 110, so we make a diagonal step and reduce ND by NX to make it 10. The next three steps are horizontal, with ND at 40, 70 and 100, then another diagonal step is made, and so on along the line.

We can now produce a general line drawing routine as shown in Fig. 3.7. The program shown draws a series of lines of random length and with random angles all over the screen. The line drawing subroutine starts at line 300 and this calls a dot-setting routine at line 200. Now when the line is drawn it will have a stepped appearance depending upon its angle relative to the horizontal or vertical axis.



```

100 REM RANDOM HIGH RES LINES
105 REM SET BIT MAP ADDRESS TO 8192
110 BM=8192
120 POKE53272,PEEK(53272)OR8
125 REM SELECT BIT MAP MODE
130 POKE53265,PEEK(53265)OR32
135 REM SET COLOURS
140 FOR I=1024TO2023:POKEI,22:NEXT
145 REM USE CLEAR ROUTINE AT 49152
150 SYS49152
160 GOTO1000
195 REM DOT SETTING SUBROUTINE
200 P=BM+320*INT(Y/8)+8*INT(X/8)+(YAND7)
210 POKE P,PEEK(P)OR(2^(7-(XAND7)))
220 RETURN
295 REM LINE DRAWING SUBROUTINE
300 SX=SGN(X2-X1):SY=SGN(Y2-Y1)
310 NX=ABS(X2-X1):NY=ABS(Y2-Y1)
320 X=X1:Y=Y1:GOSUB200
330 IF NY>NX THEN390
340 ND=INT(NX/2)
350 FORK=1TO NX:ND=ND+NY
360 IF ND<NX THEN X=X+SX:GOTO380
370 ND=ND-NX:X=X+SX:Y=Y+SY
380 GOSUB200:NEXT:GOTO440
390 ND=INT(NY/2)
400 FORK=1TO NY:ND=ND+NX
410 IF ND<NY THEN Y=Y+SY:GOTO430
420 ND=ND-NY:X=X+SX:Y=Y+SY
430 GOSUB200:NEXT
440 RETURN
995 REM MAIN PROGRAM
1000 FOR N=1 TO 15
1010 X1=320*RND(0):Y1=200*RND(0)
1020 X2=320*RND(0):Y2=200*RND(0)
1030 GOSUB300
1040 NEXT
1050 END

```

*Fig. 3.7.* General purpose line drawing routine for use with the bit map graphics mode.

### Screen edge limiting

In the line drawing program of Fig. 3.7 the values of X1, X2, Y1 and

Y2 are calculated so that they are within the limits 0 to 319 (for X) and 0 to 199 (for Y). In some programs where the line length and position are being calculated by the program it is possible that one or both ends of the line will fall outside the permitted range of values for X and Y, so we need some scheme which can deal with this situation.

If X or Y were negative, or greater than the screen limits, the data for the points would be written into computer memory which is outside that allocated to the high resolution screen, and this could have disastrous effects on the program. In a computer program, however, it may be that the values calculated for the end point of a line are outside the screen area, so we need to build into our subroutine some checks for the screen edge. This can be done in the dot plotting subroutine by testing the values of X and Y before trying to plot the dot.

The simplest approach is to arrange that if  $X < 0$  or  $X > 319$  the dot plotting operation is skipped altogether and the subroutine returns to the main program. A second test for  $Y < 0$  or  $Y > 199$  also causes the dot plotting operation to be skipped. These two tests are placed at the start of the dot plotting subroutine. This scheme effectively clips off any part of the line that goes off the screen, and the new subroutine becomes

```

200 IF X<0 OR X>319 THEN 250
210 IF Y<0 OR Y>199 THEN 250
220 P=320*INT(Y/8)+8*INT (X/8)+(YAND7)
230 B=2↑(7-(XAND7))
240 POKE BM+P,PEEK(BM+P)OR B
250 RETURN

```

Here BM is the start address of the bit map memory, and B is the value of the bit that has to be set to light the particular dot being plotted.

### Screen wraparound

An alternative approach to dealing with off-screen points is to apply *screen wraparound*. This treats the screen as if it were a sheet of paper wrapped around a cylinder so that the left and right edges of the paper touch. Now if we draw a line which runs off the right edge of the paper it continues by running in from the left edge of the paper. Thus on our screen a point moving off the right edge of the

screen will reappear at the left edge of the screen. A line running off the right side of the screen would have its off-screen segment drawn at the left of the screen. We can also apply vertical wraparound so that a line running off the bottom of the screen continues to be drawn running down from the top of the screen.

This wraparound effect can be achieved by successively adding or subtracting 320 to or from X until X falls within the screen limits. Similarly Y is changed by 200 at a time, until the Y value is on the screen. Of course if there are a lot of points off the screen this can be a slow process. The dot plotting routine could now be

```

200 IF X<0 THEN X=X+320:GOTO200
210 IF X>319 THEN X=X-320:GOTO210
220 IF Y<0 THEN Y=Y+200:GOTO 220
230 IF Y>199 THEN Y=Y-200:GOTO230
240 P=320*INT(Y/8)+8*INT(X/8)+(YAND7)
250 B=2^(7-(X AND 7))
260 POKE BM+P,PEEK(BM+P) OR B
270 RETURN

```

Here if  $X < 0$  then 320 is added to X to bring X into the correct range, and the test is repeated in case X is still negative. If  $X > 319$  then 320 is subtracted from X and the test is repeated. A similar series of tests is carried out on the Y values, and then the dot is set. Fig. 3.10 shows a program where the dot subroutine is arranged to give wraparound.

### Drawing simple figures

So far we have looked at plotting dots and drawing lines, but for many purposes we may want to draw complete figures such as triangles, rectangles, polygons and circles. A sketching program could be used, but it is usually more convenient to let the computer draw the figures itself. We shall now look at some of the techniques involved in drawing shapes on the screen.

To draw a triangle the simplest approach is to draw three straight lines linking the corners of the triangle. A rectangle can also be drawn by working out the X,Y co-ordinates for each of its four corners, and then drawing four lines which link the points together to form the sides of the rectangle.

The program listed in Fig. 3.8 draws a series of random-shaped triangles around the screen. Here the corner co-ordinates of the triangles are set up as variables XA,YA, XB,YB and XC,YC, and

```

100 REM RANDOM TRIANGLES
105 REM SET BIT MAP ADDRESS TO 8192
110 BM=8192
120 POKE53272,PEEK(53272)OR8
125 REM SELECT BIT MAP MODE
130 POKE53265,PEEK(53265)OR32
135 REM SET COLOURS
140 FOR I=1024TO2023:POKEI,22:NEXT
145 REM USE CLEAR ROUTINE AT 49152
150 SYS49152
160 GOTO1000
195 REM DOT SETTING SUBROUTINE
200 P=BM+320*INT(Y/8)+8*INT(X/8)+(YAND7)
210 POKE P,PEEK(P)OR(2^(7-(XAND7)))
220 RETURN
295 REM LINE DRAWING SUBROUTINE
300 SX=SGN(X2-X1):SY=SGN(Y2-Y1)
310 NX=ABS(X2-X1):NY=ABS(Y2-Y1)
320 X=X1:Y=Y1:GOSUB200
330 IF NY>NX THEN390
340 ND=INT(NX/2)
350 FORK=1TO NX:ND=ND+NY
360 IF ND<NX THEN X=X+SX:GOTO380
370 ND=ND-NX:X=X+SX:Y=Y+SY
380 GOSUB200:NEXT:GOTO440
390 ND=INT(NY/2)
400 FORK=1TO NY:ND=ND+NX
410 IF ND<NY THEN Y=Y+SY:GOTO430
420 ND=ND-NY:X=X+SX:Y=Y+SY
430 GOSUB200:NEXT
440 RETURN
995 REM MAIN PROGRAM
1000 FOR N=1 TO 5
1005 REM SELECT CORNER POINTS
1010 XA=320*RND(0):YA=200*RND(0)
1020 XB=320*RND(0):YB=200*RND(0)
1030 XC=320*RND(0):YC=200*RND(0)
1035 REM DRAW TRIANGLE
1040 X1=XA:Y1=YA:X2=XB:Y2=YB:GOSUB300
1050 X1=XB:Y1=YB:X2=XC:Y2=YC:GOSUB300
1060 X1=XC:Y1=YC:X2=XA:Y2=YA:GOSUB300
1070 NEXT
1080 END

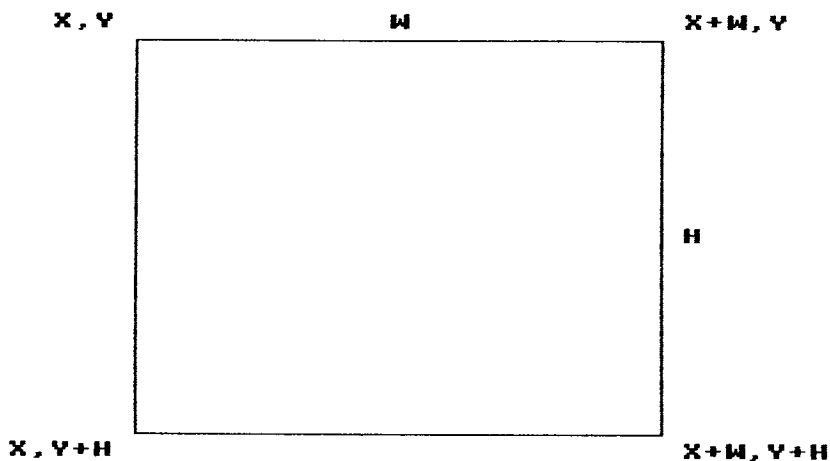
```

Fig. 3.8. Random triangle drawing program.

then lines are drawn linking these points. This program needs the machine code clear routine loaded into memory before it is used.

### Drawing rectangles

We could draw a rectangle in the same way as a triangle by specifying the corner points and linking them with lines, but there is a simpler approach which can be used. Rectangles have a width  $W$  and a height  $H$ , and knowing these and the  $X, Y$  position of one corner we can let the computer calculate the  $X, Y$  co-ordinates for the other corners and draw in the lines. The values for the other corners are shown in Fig. 3.9.



*Fig. 3.9.* Diagram showing drawing co-ordinates for a rectangle based on width and height.

To draw the rectangle we start by setting  $X1$  and  $Y1$  to the  $X$  and  $Y$  values for the top left corner.  $X2$  will now be  $X1+W$  but  $Y2$  is the same as  $Y1$ , since the line is horizontal. We can now call the line drawing subroutine to draw in the top side of the rectangle. Now  $X1$  and  $Y1$  are set equal to  $X2$  and  $Y2$ . For the right side of the rectangle a new value of  $Y2$  equal to  $Y1+H$  is calculated, and another line drawn. For the third side  $W$  is subtracted from  $X1$  to give the new value for  $X2$ , and on the final side  $H$  is subtracted from  $Y1$  to give the final value for  $Y2$ . This sequence is shown in the program listed in Fig. 3.10. Here the rectangle-drawing operation is set up as a subroutine, and random size rectangles are drawn at random points

```

100 REM RANDOM RECTANGLES
102 REM WITH SCREEN WRAPAROUND
105 REM SET BIT MAP ADDRESS TO 8192
110 BM=8192
120 POKE53272,PEEK(53272)OR8
125 REM SELECT BIT MAP MODE
130 POKE53265,PEEK(53265)OR32
135 REM SET COLOURS
140 FOR I=1024TO2023:POKEI,22:NEXT
145 REM USE CLEAR ROUTINE AT 49152
150 SYS49152
160 GOTO1000
195 REM DOT PLOT WITH SCREEN WRAPAROUND
200 X=INT(X):Y=INT(Y)
210 IF X<0 THEN X=X+320:GOTO210
220 IF X>319 THEN X=X-320:GOTO220
230 IF Y<0 THEN Y=Y+200:GOTO230
240 IF Y>199 THEN Y=Y-200:GOTO240
250 P=BM+320*INT(Y/8)+8*INT(X/8)+(YAND7)
260 POKE P,PEEK(P)OR(2↑(7-(XAND7)))
270 RETURN
295 REM LINE DRAWING SUBROUTINE
300 SX=SGN(X2-X1):SY=SGN(Y2-Y1)
310 NX=ABS(X2-X1):NY=ABS(Y2-Y1)
320 X=X1:Y=Y1:GOSUB200
330 IF NY>NX THEN390
340 ND=INT(NX/2)
350 FORK=1TO NX:ND=ND+NY
360 IF ND<NX THEN X=X+SX:GOTO380
370 ND=ND-NX:X=X+SX:Y=Y+SY
380 GOSUB200:NEXT:GOTO440
390 ND=INT(NY/2)
400 FORK=1 TO NY:ND=ND+NX
410 IF ND<NY THEN Y=Y+SY:GOTO430
420 ND=ND-NY:X=X+SX:Y=Y+SY
430 GOSUB200:NEXT
440 RETURN
495 REM RECTANGLE SUBROUTINE
500 X2=X1+W:Y2=Y1:GOSUB300
510 X1=X2:Y2=Y1+H:GOSUB300
520 X2=X1-W:Y1=Y2:GOSUB300
530 X1=X2:Y2=Y1-H:GOSUB300
540 RETURN
995 REM MAIN PROGRAM
1000 FOR N=1 TO 10

```

```

1005 REM SELECT CORNER POINT
1010 X1=480*RND(0):Y1=300*RND(0)
1020 W=30+30*RND(0):H=25+50*RND(0)
1025 REM DRAW RECTANGLE
1030 GOSUB500
1040 NEXT
1050 END

```

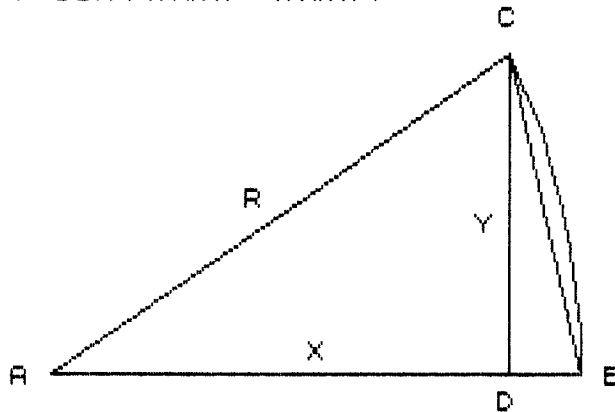
*Fig. 3.10.* Program to draw random rectangles with screen wraparound.

on the screen. The dot plotting routine used in this program features screen wraparound, so that when a rectangle is near the edge of the screen and part of it goes over the edge the overspill will be drawn at the other side of the screen.

### Drawing circles

One figure we shall often require is the circle. There are several methods for drawing circles. These build up the circle by plotting a large number of individual dots, or drawing a number of short lines to build up the circular figure. The positions of the dots or lines are calculated using one of the mathematical formulae for a circle.

$$Y = \text{SQR}((R \# R) - (X \# X))$$



*Fig. 3.11.* Derivation of the equations for a circle by the squares method.

Figure 3.11 shows a segment of a circle. Here point A is at the centre of the circle while points B and C are on the circle itself. The length of the line AB is equal to that of line AC, and is called the radius (R) of the circle. Let us now drop a vertical line down from point C to meet

side AB at point D. This produces a right-angled triangle ACD.

To place dots at points B and C we need to know the X and Y co-ordinates for each of those points. It is convenient to calculate these relative to point A which will therefore have co-ordinates X and Y both equal to 0. If we move to point B, its X co-ordinate must be equal to R, the length of line AB. Since the line is horizontal the Y co-ordinate must be the same as that of point A, and is therefore 0. When we look at point C, the X value is the length of side AD, and the Y value is equal to the length of side CD of the triangle ACD.

For a right-angled triangle the square of the length of the longest side is the sum of the squares of the lengths of the other two sides. We can write this down as follows:

$$(AC)*(AC) = (AD)*(AD) + (CD)*(CD)$$

Now AC = radius R while AD and CD are the X and Y co-ordinates for point C, so putting these in our equation we get

$$R * R = (X * X) + (Y * Y)$$

which can be rearranged to give us

$$Y * Y = (R * R) - (X * X)$$

from which we can get Y by simply taking the square root. The calculation for Y now becomes

$$Y = \text{SQR}((R * R) - (X * X))$$

The values of X and Y in this equation are measured with reference to the centre point of the circle. Note that for point B the equation for Y is still true, since in this case X=R, so the term on the right becomes zero and therefore Y=0.

To place the circle at some particular point on the screen we shall have to add in the X and Y co-ordinates for the point where the circle is to be drawn. To avoid confusion we shall call these co-ordinates XC and YC.

In order to plot all the points around the circle we need to calculate values of Y for a series of values of X ranging from -R to +R, and the more points we calculate the better the circle will look.

When we take the square root of a number there are in fact two possible answers with the same numerical value, one being positive and the other negative. For each value of X we shall have two values for Y, and be able to plot a pair of points, one on the upper half of the circle, and the other on the lower half. To plot the first point the result of the square root calculation is added to the Y value for the



centre of the circle (YC) to give a point below the centre line of the circle. The second point has the square root subtracted from YC, and will lie above the centre line of the circle. This is because, on the bit map screen, Y increases as we move down the screen.

Since we are plotting single points, and the circumference of the circle is just over 6 times radius R, it is convenient to have a total number of points equal to 4 times R to make up the circle. Remember that we plot two points for each calculation, so a good value for the number of calculations is twice the value of radius R. This is easily achieved by taking all of the X values from  $X=-R$  to  $X=+R$ . Thus a circle with a radius of 50 screen units would calculate 100 steps and plot a total of 200 points around the circle.

The program shown in Fig. 3.12 draws a circle of radius 50 units with its centre point at  $XC=160$ ,  $YC=100$ , which is roughly at the centre of the screen. With this routine the number of calculations depends upon the size of the circle. By changing the values of XC,

```

100 REM CIRCLE DRAWING USING SQUARES
110 BM=8192
115 REM SET BIT MAP START
120 POKE53272,PEEK(53272)OR8
125 REM SELECT BIT MAP MODE
130 POKE53265,PEEK(53265)OR32
132 REM CLEAR SCREEN
134 REM NEEDS M/C ROUTINE AT 49152
140 SYS49152
145 REM SET COLOURS
150 FORI=1024TO2023:POKEI,22:NEXT
160 GOTO1000
195 REM DOT PLOT ROUTINE
200 P=BM+320*INT(Y/8)+8*INT(X/8)+(YAND7)
210 FOKEP,PEEK(P)OR(2*(7-(XAND7)))
220 RETURN
495 REM CIRCLE ROUTINE
500 FOR X1=-R TO R
510 Y1=SQR(R#R-X1*X1)
520 X=XC+X1:Y=YC+Y1:GOSUB200
530 Y=YC-Y1:GOSUB200:NEXT:RETURN
995 REM MAIN PROGRAM
1000 XC=160:YC=100:R=50
1010 GOSUB500
1020 END

```

*Fig. 3.12.* Circle drawing program using squares method.

YC and R, other circles may be drawn, or you could draw a series of random-size circles all over the screen. It will be seen that the larger size circles take a noticeable time to draw. This is because the computer has quite a lot of calculations to carry out. The square root function itself is rather slow in BASIC. If we want to draw circles faster we will need to look at other ways of calculating the points around the circle.

You will note that at the right and left sides of the circle the first few points tend to be rather spread out, especially on the larger radius circles. This can be overcome by plotting more points – for instance by increasing X by steps of 0.5 instead of 1 – but the circle takes even longer to draw.

### The trigonometric method

Instead of plotting a series of dots we can draw a series of short lines, which when joined together will form the outline of the circle. For the second method of drawing circles we need to get involved in some simple trigonometry.

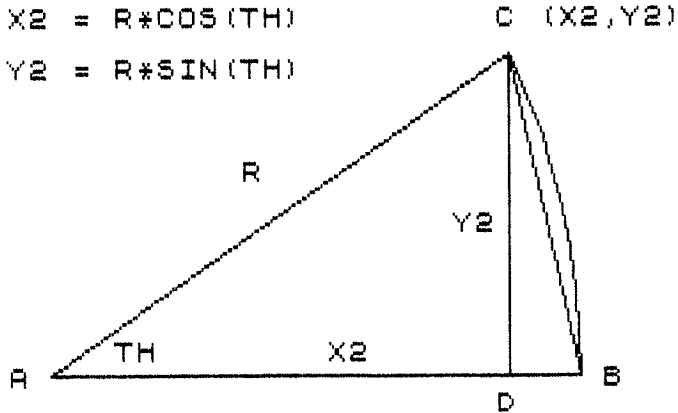


Fig. 3.13. Derivation of equation for the trigonometric method of circle drawing.

Figure 3.13 shows a segment of the circle with two points B and C on the circle itself and point A at the centre. The angle at point A of the triangle is given the variable name TH which is a shortened version of THETA, the name of the Greek letter normally used for labelling angles. The co-ordinates X2,Y2 for point C can now be calculated using the radius R and the angle TH.

To find the length of side CD the function we need to use is SIN(TH). SIN(TH) is the ratio of the length of the side of the triangle opposite angle TH to the length of the hypotenuse (the side opposite the right angle), which is side AC. So in our triangle

$$\text{SIN(TH)} = \text{CD/AC}$$

We already know that AC = radius R. The length of side CD is the Y2 value for point C, assuming that at point A the value of Y=0. Substituting these new terms in the equation we get

$$\text{SIN(TH)} = \text{Y2/R}$$

and if we multiply both sides by R the result becomes

$$\text{Y2} = \text{R} * \text{SIN(TH)}$$

Having found Y2 we need to find a value for side AD, which is the X2 value for point C. Now it just happens that COS(TH) is the ratio of the length of the adjacent side (AD) of the triangle to the length of the hypotenuse (AC) so we get

$$\text{COS(TH)} = \text{AD/AC}$$

and substituting the values X2 and R gives

$$\text{X2} = \text{R} * \text{COS(TH)}$$

To find the co-ordinates for the next point we apply the same equations, but now the angle TH has a different value.

To draw the first segment of the circle we must draw a line from point B to point C. This line starts at point B, where TH=0 and the required values for X1, Y1 are

$$\text{X1} = \text{XC} + \text{R}$$

$$\text{Y1} = \text{YC}$$

where XC and YC are the co-ordinates for the centre of the circle.

The next step is to calculate the co-ordinates of point C, which will be

$$\text{X2} = \text{XC} + \text{R} * \text{SIN(TH)}$$

$$\text{Y2} = \text{YC} + \text{R} * \text{COS(TH)}$$

and the line BC can be drawn from X1, Y1 to X2, Y2.

For the next line segment of the circle, the values of X1 and Y1 are set equal to the values of X2 and Y2. The angle TH is then increased, and new values are calculated for X2, Y2 using the new value for the angle TH. This process continues until the angle TH reaches 360

degrees, when a complete circle will have been drawn.

While angles in degrees are familiar to us, the computer doesn't work in degrees; it uses *radians* instead. All we need to know here is that there are  $2*\pi$  radians in 360 degrees. The number  $\pi$  (pronounced PI) is a constant whose value is approximately 3.14, and it is the ratio of the circumference of a circle to its diameter. We do not need to remember the value for  $\pi$  because the Commodore 64 has a special key which allows us to insert  $\pi$  into a program statement as required. This key is marked with the Greek letter  $\pi$  on its front face, and must be used with the shift key held down. It produces the number 3.14159265.

A convenient number of steps for drawing the circle is given by R, the radius in screen units. Drawing the circle involves using a simple loop to repeat the calculations and draw a short line segment R times. After each segment of the circle has been drawn, the value of the angle TH is increased by  $2*\pi/R$  radians, and the values of X1 and Y1 are updated to point to the end of the line that has just been drawn, ready for the next drawing step.

A program to draw random-sized circles is shown in Fig. 3.14.

```

100 REM CIRCLES USING TRIG METHOD
110 BM=8192
115 REM SET BIT MAP START
120 POKE53272,PEEK(53272)OR8
125 REM SELECT BIT MAP MODE
130 POKE53265,PEEK(53265)OR32
132 REM CLEAR SCREEN
134 REM NEEDS M/C ROUTINE AT 49132
140 GY949132
145 REM SET COLOURS
150 FORI=1024TO2023:POKEI,22:NEXT
160 GOTO1000
195 REM DOT PLOT ROUTINE WITH CLIPPING
200 IF X<0 OR X>819 THEN 240
210 IF Y<0 OR Y>199 THEN 240
220 P=BM+320*INT(Y/8)+8*INT(X/8)+(YAND7)
230 POKEP,PEEK(P)OR(2+(7-(XAND7)))
240 RETURN
295 REM LINE DRAWING ROUTINE
300 SX=SGN(X2-X1):SY=SGN(Y2-Y1)
310 NX=ABS(X2-X1):NY=ABS(Y2-Y1)
320 X=X1:Y=Y1:GOSUB200
330 IF NY>NX THEN 360

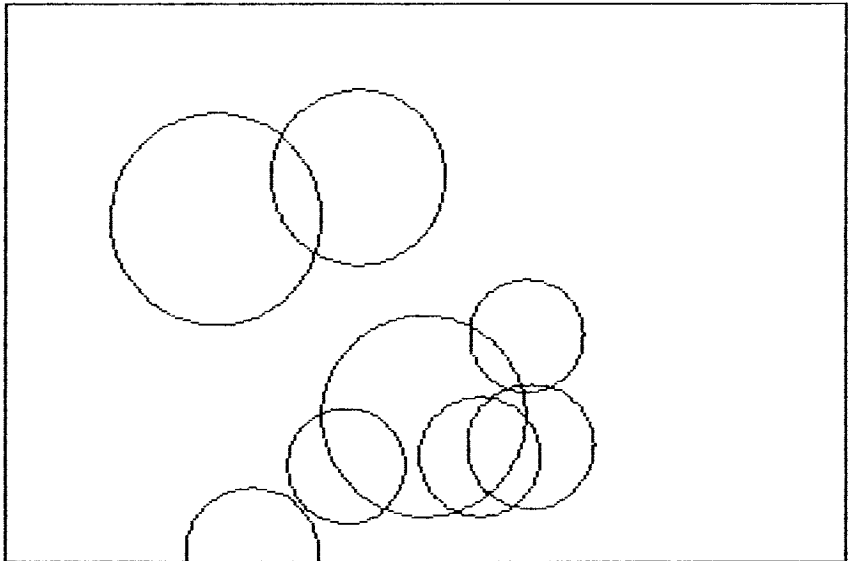
```

```

340 ND=INT(NY/2):FORK=1TOMX:ND=ND+NY
350 IF ND<NX THEN X=X+SX:GOTO370
360 ND=ND-NX:X=X+SX:Y=Y+SY
370 GOSUB200:NEXT:GOTO420
380 ND=INT(NY/2):FORK=1TONY:ND=ND+NX
390 IFND<NY THEN Y=Y+SY:GOTO410
400 ND=ND-NY:X=X+SX:Y=Y+SY
410 GOSUB200:NEXT
420 RETURN
495 REM CIRCLE ROUTINE
500 DT=2*PI/R:TH=0:X1=XC+R:Y1=YC
510 FOR I=1 TO R
520 TH=TH+DT:X2=XC+R*COS(TH)
530 Y2=YC+R*SIN(TH):GOSUB300
540 X1=X2:Y1=Y2:NEXT:RETURN
995 REM MAIN PROGRAM
1000 FOR N=1TO8
1010 XC=INT(300*RND(0))+10
1020 YC=INT(170*RND(0))+10
1030 R=15+INT(15*RND(0))
1040 GOSUB500:NEXT
1050 END

```

*Fig. 3.14.* Random circle program using the trigonometric method.



*Fig. 3.15.* Typical display of random circles.

This program gives a result similar to that shown in Fig. 3.15. The actual circle-drawing section is written as a subroutine starting at line 500. If several circles are to be drawn, it is best to use a subroutine for drawing the circle, and call it from the main program whenever it is needed. Before calling the subroutine, the values for R, XC and YC must be set up for the required circle.

This circle-drawing program uses a dot plotting routine which clips points at the edge of the screen. Thus any points falling outside the screen area are not plotted at all.

### The rotation method

A variation of the trigonometric method for a circle bases the calculations upon the angle through which the radial line is rotated at each step. In this case the new values for X2 and Y2 are calculated from the values for the previous point (X1, Y1) rather than from the radius and the total angle.

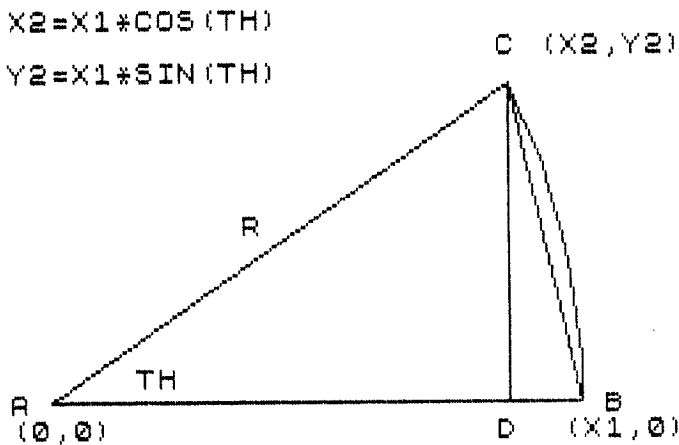


Fig. 3.16. Equations for rotation of a point with Y1 initially at 0.

If we look at Fig. 3.16 the value of Y1 is zero so that only the X1 value, which also happens to be equal to R, affects the results. Here we get

$$X2 = X1 * \cos(\text{TH})$$

$$Y2 = X1 * \sin(\text{TH})$$

Now consider the situation where the radial line is vertical and is moved through angle TH. This is shown in Fig. 3.17. Here the value

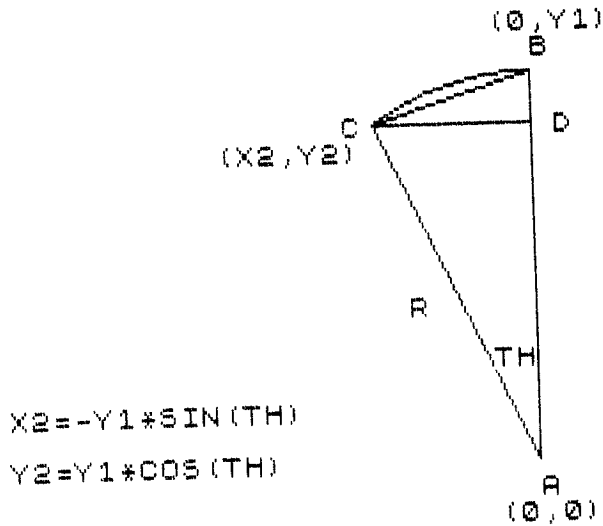


Fig. 3.17. Equations for rotation of a point with X1 initially at 0.

of X1 is 0, and only the Y1 value affects the results. In this case the value of X2 is negative, since the point has been shifted to the left of the line where X1=0. Here we get the results

$$X2 = -Y1 * \text{SIN}(TH)$$

$$Y2 = Y1 * \text{COS}(TH)$$

If we combine these two results we can produce a general expression for calculating X2 and Y2 for any initial values of X1 and Y1. The two new equations are

$$X2 = X1 * \text{COS}(TH) - Y1 * \text{SIN}(TH)$$

$$Y2 = X1 * \text{SIN}(TH) + Y1 * \text{COS}(TH)$$

Now the big advantage of this approach is that the value of TH is constant, so we can work out the values of SIN(TH) and COS(TH) before entering the co-ordinate calculation and line drawing loop. This eliminates virtually all the trigonometric calculations, which tend to be slow. The program for drawing a circle now becomes as shown in the listing of Fig. 3.18. As for the trigonometric method, this program produces random circles with edge clipping.

```

100 REM CIRCLES USING ROTATION
110 BM=8192
115 REM SET BIT MAP START
120 POKE53272,PEEK(53272)ORS
125 REM SELECT BIT MAP MODE

```

```

130 POKE53265,PEEK(53265)OR32
132 REM CLEAR SCREEN
134 REM NEEDS M/C ROUTINE AT 49152
140 SYS49152
145 REM SET COLOURS
150 FOR I=1024 TO 2023:POKE I,22:NEXT
160 GOTO1000
195 REM DOT PLOT ROUTINE WITH CLIPPING
200 IF XC0 OR X>319 THEN 240
210 IF YC0 OR Y>199 THEN 240
220 P=BM+320*INT(Y/8)+8*INT(X/8)+(YAND7)
230 POKEP,PEEK(P)OR(2*(7-(YAND7)))
240 RETURN
295 REM LINE DRAWING ROUTINE
300 SX=SGN(X2-X1):SY=SGN(Y2-Y1)
310 NX=ABS(X2-X1):NY=ABS(Y2-Y1)
320 X=X1:Y=Y1:GOSUB200
330 IF NY>NX THEN 380
340 ND=INT(NY/2):FORK=1TONX:ND=ND+NY
350 IF ND<NX THEN X=X+SX:GOTO370
360 ND=ND-NX:X=X+SX:Y=Y+SY
370 GOSUB200:NEXT:GOTO420
380 ND=INT(NY/2):FORK=1TONY:ND=ND+NX
390 IF ND<NY THEN Y=Y+SY:GOTO410
400 ND=ND-NY:X=X+SX:Y=Y+SY
410 GOSUB200:NEXT
420 RETURN
495 REM CIRCLE ROUTINE
500 SN=SIN(2*pi/R):CN=COS(2*pi/R)
510 X1=R:Y1=0:FOR I=1 TO R
520 X2=XC+X1*CN-Y1*SN
530 Y2=YC+X1*SN+Y1*CN
540 X1=X2-XC:Y1=Y2-YC:GOSUB300
550 X1=X2-XC:Y1=Y2-YC:NEXT:RETURN
995 REM MAIN PROGRAM
1000 FOR N=1TOS
1010 XC=INT(300*RN(0))+10
1020 YC=INT(170*RN(0))+10
1030 R=15+INT(15*RN(0))
1040 GOSUB500:NEXT
1050 END

```

Fig. 3.18. Program to draw circles using the rotation method.



Note here that although XC and YC are added in to give the coordinates for the line drawing they are subtracted from X1,Y1 before calculating the new value of X2,Y2. This is important, since the X1 value used to calculate X2 must always be measured relative to the centre of the circle for correct results. If desired you could use another variable for the value of X1, Y1, without the offset component XC,YC.

## Drawing polygons

If we reduce the number of steps used in the circle drawing routine, the result will be a figure with a number of equal straight sides. Such a figure is called a *regular polygon*.

To make a general polygon drawing routine we could introduce a new parameter NS (number of sides) and then modify the program to make the required number of drawing steps. Thus the rotation angle TH will be given by

$$TH = 2*\pi/NS$$

To see how this works try running the program shown in Fig. 3.19 which draws random sized regular polygons with from three to eight equal length sides. Figure 3.20 shows the sort of display produced by this program.

```

100 REM RANDOM POLYGONS
110 BM=8192
115 REM SET BIT MAP START
120 POKE53272,PEEK(53272)OR8
125 REM SELECT BIT MAP MODE
130 POKE53265,PEEK(53265)OR32
132 REM CLEAR SCREEN
134 REM NEEDS M/C ROUTINE AT 49152
140 SYS49152
145 REM SET COLOURS
150 FORI=1024TO2023:POKEI,22:NEXT
160 GOTO1000
195 REM DOT PLOT ROUTINE WITH CLIPPING
200 IF X<0 OR X>319 THEN 240
210 IF Y<0 OR Y>199 THEN 240
220 P=BM+320*INT(Y/8)+8*INT(X/8)+(YAND7)
230 POKEP,PEEK(P)OR(2+(7-(XAND7)))
240 RETURN

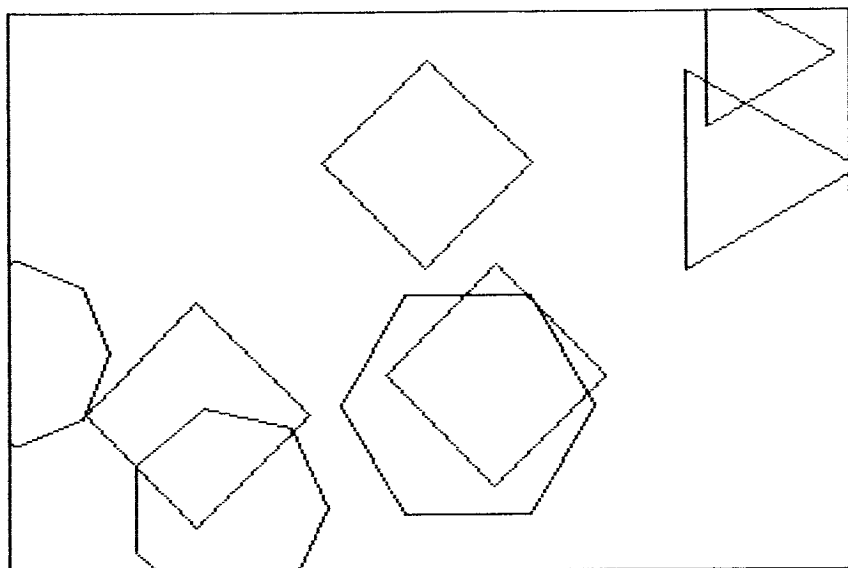
```

```

295 REM LINE DRAWING ROUTINE
300 SX=SGN(X2-X1):SY=SGN(Y2-Y1)
310 NX=ABS(X2-X1):NY=ABS(Y2-Y1)
320 X=X1:Y=Y1:GOSUB200
330 IF NY>NX THEN 380
340 ND=INT(NY/2):FOR K=1 TO NX:ND=ND+NY
350 IF ND<NX THEN X=X+SX:GOTO370
360 ND=ND-NX:X=X+SX:Y=Y+SY
370 GOSUB200:NEXT:GOTO420
380 ND=INT(NY/2):FOR K=1 TO NY:ND=ND+NX
390 IF ND<NY THEN Y=Y+SY:GOTO410
400 ND=ND-NY:X=X+SX:Y=Y+SY
410 GOSUB200:NEXT
420 RETURN
495 REM POLYGON ROUTINE
500 SN=SIN(2*PI/NS):CN=COS(2*PI/NS)
510 X1=R:Y1=0:FOR I=1 TO NS
520 X2=XC+X1*CN-Y1*SN
530 Y2=YC+X1*SN+Y1*CN
540 X1=X2-XC:Y1=Y2-YC:GOSUB300
550 X1=X2-XC:Y1=Y2-YC:NEXT:RETURN
995 REM MAIN PROGRAM
1000 FOR N=1 TO 6
1010 XC=INT(300*RND(0))+10
1020 YC=INT(170*RND(0))+10
1030 R=25+INT(25*RND(0))
1040 NS=3+INT(5*RND(0))
1050 GOSUB500:NEXT
1060 END

```

Fig. 3.19. Polygon drawing program using the rotation method.



*Fig. 3.20.* Typical random polygon display.

## Chapter Four

# Adding Colour

So far, the text and character graphics displays we have generated on the Commodore 64 have been in light blue on a blue background, which are the default display colours. In the bit map displays, however, the colours were changed to white on a blue background by POKEing numbers into the screen memory area. The computer is in fact able to present much more colourful displays, where the symbols may be set to any one of sixteen different colours. The colours of the border around the display area, and of the background, can also be set to any of the sixteen available colours. In the bit mapped mode any pair of colours from the sixteen available may be chosen for the drawing and background colours.

More advanced techniques allow the creation of multi-coloured text or graphics symbols. In this chapter we shall look at the various ways in which the colours can be controlled.









### Setting text colours

Let us start by experimenting with changes in the text colour. The computer normally starts up with the text symbols in a light blue colour on a blue background. The colour of the displayed text can be changed by making use of the CONTROL key (marked CTRL) at the upper left side of the keyboard.

If you press the CTRL and 1 keys together (we shall indicate this in the text as [CTRL 1]), you will notice that the colour of the flashing cursor changes to black. If you now type in some text or graphics symbols they will be displayed in black. Notice that only new symbols are affected: the text that was already on the screen remains unchanged. When using the CTRL key it is best to hold it down and then press the number key. The CTRL key works in much the same way as the SHIFT key to alter the function of other keys on the

keyboard, but doesn't by itself produce any effect on the screen.

Now try using CTRL with the 2 key. This time you will see that any new text is now displayed in white. The CTRL key can also be used with the other number keys from 3 to 8, and each of these causes the new text to be printed in a different colour, giving eight possible colours for the displayed text. Figure 4.1 shows the set of colours



Colour	Keys Used	ASCII Code	Listing Symbol
Black	CTRL 1	144	
White	CTRL 2	5	
Red	CTRL 3	28	
Cyan	CTRL 4	159	
Purple	CTRL 5	156	
Green	CTRL 6	30	
Blue	CTRL 7	31	
Yellow	CTRL 8	158	

*Fig. 4.1.* Colours produced by using the CTRL key, with their ASCII codes and listing symbols.

produced by using the CTRL key with the number keys from 1 to 8.

There are, as we have seen, sixteen possible colours available for text on the Commodore 64. So how do we manage to select the remaining eight? The answer lies in the use of the Commodore Logo key, which is at the left end of the bottom row of keys. This is the key that carries the 'C'-shaped trade mark (logo) of Commodore computers. We shall call it the LOGO key.

To select text colours the LOGO key is used with the number keys in the same way as we used the CTRL key. Now if you press the LOGO and 1 keys at the same time the colour of the flashing cursor will change to orange, and all new text symbols entered will also be orange. Like the CTRL key, the LOGO key can be used with the numbers 1 to 8 to produce eight different text colours. The eight colours produced by using the LOGO key are shown in Fig. 4.2.

Colour	Keys Used	ASCII code	List Symbol
Orange	LOGO 1	129	
Brown	LOGO 2	149	
Lt red	LOGO 3	150	
Grey 1	LOGO 4	151	
Grey 2	LOGO 5	152	
Lt Green	LOGO 6	153	
Lt Blue	LOGO 7	154	
Lt Grey	LOGO 8	155	

*Fig. 4.2.* Colours produced by using the LOGO key, with their ASCII codes and listing symbols.

### The colour memory map

As we saw in Chapter 1, a separate area of memory is set aside to hold the colour information for each of the text symbol positions on the screen. Within each symbol position all the lit dots will be displayed in the foreground colour, while the unlit dots are displayed in the background colour. If the foreground colour is changed, then all the lit dots change to the new colour.

The colour memory occupies the 1000 memory locations from 55296 to 55319, and these correspond to the screen memory locations 1024 to 2023. Thus the colour data for the symbol in location 1024 is held in location 55296, and so on.

In the colour memory each location specifies the colour for a particular symbol space on the screen. There are sixteen colours altogether, and these can be represented by the numbers 0 to 15. The computer stores binary numbers in its memory, and the numbers 0 to 15 can be represented by a four-bit binary data word. In the colour memory the lower four bits of the data word specify the text colour.

If a PEEK command is used to read the contents of a colour

memory location, the result will be a number between 240 and 255. This may seem odd since the numbers representing the colour only run from 0 to 15. The reason is that the upper four bits of the colour memory word are not used, and to the processor these four bits will all appear to be set at '1'. If you want to PRINT out the colour data for a particular symbol space, then the upper four bits need to be masked off by using an AND operation as follows:

```
PRINT PEEK(N) AND 15
```

where N is the location in the colour memory. The AND operation slices off the upper four data bits of the word to leave the colour number from 0 to 15.

The data we have to write into the colour memory is simply a number from 0 to 15 corresponding to the colour we want. The colour numbers and their associated colours are shown in Fig. 4.3. You will notice that although the colour sequence is the same, the numbers are different from those we used with the CTRL and LOGO keys.

For the first eight colours the number required is one less than the number of the key used with the CTRL key. For the other eight colours the number is seven greater than the number of the key used with the LOGO key.

### **Changing colour using PRINT**

In a program we put the symbols on the screen by using a PRINT statement and enclosing the character string in quotes. We can also insert the [CTRL n] or [LOGO n] codes into a PRINT statement in the same way.

Try typing in the following lines:

```
PRINT "[CTRL3] RED TEXT"  
PRINT "[CTRL7] YELLOW TEXT"
```

When you hit the return key the message RED TEXT will be printed in red, and for the second line the message YELLOW TEXT is printed in yellow. The LOGO key together with a number can be inserted into a PRINT text string in the same way to switch to one of the other eight colours.

You will now notice when the [CTRL n] is inserted in a PRINT statement a strange graphics symbol appears on the screen where the colour control code was entered, but when the command is actually

Display Colour	Colour memory POKE code	ASCII CHR\$ code
Black	0	144
White	1	5
Red	2	28
Cyan	3	159
Purple	4	156
Green	5	30
Blue	6	31
Yellow	7	158
Orange	8	129
Brown	9	149
Light Red	10	150
Dark Grey	11	151
Medium Grey	12	152
Light Green	13	153
Light Blue	14	154
Light Grey	15	155

*Fig. 4.3.* The set of available colours and their numbers as used for colour memory, bit map and background colour selection.

executed this symbol is not printed on the screen. The advantage of having a symbol shown in the PRINT command line is that we can see there is a control code in the test string. Figures 4.1 and 4.2 show the various symbols displayed to indicate these colour control codes.

In a program, the control codes are inserted into the text string in much the same way. The only difference is that the statement now has a line number at the start. When the program is LISTED, the



control codes will be displayed on the screen using their special symbols. If you have a Commodore printer the graphics symbols for these codes will also be presented on the printout.

### Using CHR\$ codes

All symbols and control codes can also be represented by an ASCII code, and this can be used in a PRINT statement by using the CHR\$(n) form. One might have expected the codes for the colours to run in sequence, but in fact they are scattered through the complete set of codes between the groups of codes used for graphics and text. The codes corresponding to the sixteen colours available are listed in Figs. 4.1 and 4.2.

Now let us see how these colour control codes might be used. Suppose we wanted to change the text colour to white while printing a text string. We could use the statement

```
PRINT CHR$(5);"WHITE TEXT"
```

and this would produce the same effect as including[CTRL I] in the text string. Note here that a semicolon is included after the CHR\$(N).

If we want to make several changes of colour it is best to set up an array variable to hold the complete set of codes. Suppose we use the variable TC (text colour) and set it with an array dimension of 8. Now we can set up a single loop to read in the set of colour codes. We can arrange that the sequence of colours is the same as that of the CTRL or LOGO sequence. Now to set a colour we merely have to set the value in the CHR\$ term equal to TC(N) where N is the number that would have been used with the CTRL or LOGO key. This is shown in the following small program:

```
100 FOR N=1 TO 8
110 READ TC(N)
120 NEXT
130 DATA 144,5,28,159,156,30,31,158
140 PRINT CHR$(147);
150 FOR N=1 TO 8
160 PRINT CHR$(TC(N));"****";
170 NEXT
```

The other eight colours could equally well have been set up in the C array, or you could even have all sixteen colour codes set up in the

```

100 REM COLOURS USING CTRL KEY
110 DIM CC(8),C$(8)
120 FOR N=1TO8
130 READ CC(N),C$(N)
140 NEXT
150 DATA 144,"BLK",5,"WHI",28,"RED"
160 DATA 159,"CYN",156,"PUR",30,"GRN"
170 DATA 31,"BLU",158,"YEL"
180 PRINT CHR$(147);
190 FOR R=1TO15
200 PRINT CHR$(18);
210 FOR B=1TO8
220 PRINT CHR$(CC(B));" ";
230 NEXT:NEXT
240 PRINT CHR$(146)
250 FOR N=1TO8
260 PRINT" ";C$(N);" ";
270 NEXT
280 PRINT
290 FOR N=1TO8
300 PRINT"CTRL ";
310 NEXT
320 FOR N=1 TO 8:PRINT STR$(N);" ";
330 NEXT
340 END

```

*Fig. 4.4.* Colour bar program using the CTRL colours.

array. You can also allocate the colours in any sequence you like, since the colour selected by the number  $C(N)$  depends upon the data you choose to feed into the array at the start.

The program listing of Fig. 4.4 uses the  $CHR\$$  codes to produce a pattern of vertical coloured bars, using the eight colours produced by the CTRL key. The display also shows the corresponding CTRL numbers. The program listing of Fig. 4.5 produces a similar colour bar display but this time uses the set of eight LOGO key colours.

### Changing the background colour

The Commodore 64 starts up with a blue background, cyan text and a cyan border around the display area. The background colour can be changed to any one of the sixteen colours we can use for the text.

In most computers there is a BASIC command to set up the

```

100 REM COLOURS USING LOGO KEY
110 DIM CC(8),C$(8)
120 FOR N=1TO8
130 READ CC(N),C$(N)
140 NEXT
150 DATA 129," ORG ",149," BRN "
160 DATA 150,"L.RD ",151,"GRY1 "
170 DATA 152,"GRY2 ",153,"L.GR "
180 DATA 154,"L.BL ",155,"GRY3 "
190 PRINT CHR$(147);
200 FOR R=1TO15
210 PRINT CHR$(18);
220 FOR B=1TO8
230 PRINT CHR$(CC(B));"      ";
240 NEXT:NEXT
250 PRINT CHR$(146)
260 FOR N=1TO8
270 PRINT C$(N);
280 NEXT
290 PRINT
300 FOR N=1TO8
310 PRINT"LOGO ";
320 NEXT
330 FOR N=1 TO 8:PRINT STR$(N);"  ";
340 NEXT
350 END

```

*Fig. 4.5.* Colour bar program using the LOGO colours.

background colour of the display, but on the Commodore 64 this is done by setting up a register in the VICII chip. The VICII video display chip contains several registers which are used to control the various display modes, and also some of the colour features of the display. Figure 4.6 shows the registers used for colour and mode control together with their memory addresses.

Background colour is controlled by register 33 in the chip, and this is located at memory address 53281. The number that has to be POKEd into this register ranges from 0 to 15, and is the same as the corresponding text colour number shown in Fig. 4.3. As an example, if we want a red background then the instruction would be

```
POKE 53281,2
```

since 2 is the colour number for red. As with the colour memory, only the lower four bits of the register are used, with the upper 4 bits

Register	Address	Function
17	53265	Extd. Background (Bit 6)
22	53270	Multicolour mode (Bit 4)
32	53280	Border colour
33	53281	Background colour 0
34	53282	Background colour 1
35	53283	Background colour 2
36	53284	Background colour 3

*Fig. 4.6.* The colour control registers of the VIC11 chip with their memory addresses and functions.

set at '1'. If you PEEK the register in a program remember that you need to AND the result with 15 to get the correct colour number.

Unlike the text colour, which can be different for each symbol space, the background colour all over the screen will change when the colour number in the background register of the VIC11 chip is altered.

### Setting the border colour

The border colour on the screen is controlled by register number 32 in the VIC11 chip, and this has the memory address 52380. As with the background colour, we can change the border colour by POKEing a number from 0 to 15 into this register. The colours produced are the same as those produced by the background register. The whole border area will change colour when a new number is placed in the border control register.

```

100 REM RANDOM COLOURED BLOCKS
110 DIM CC(16)
115 REM READ COLOUR CHR$ CODE DATA
120 FOR N=1TO16:READ CC(N):NEXT
130 DATA 144,5,8,159,156,30,31,158
140 DATA 129,149,150,151,152,153,154,155
145 REM CLEAR SCREEN
150 PRINT CHR$(147)
155 REM SET REVERSE VIDEO ON
160 PRINT CHR$(18);
170 FOR N=1 TO 2000
175 REM POSITION CURSOR
180 R = INT(24* $\text{RND}(\emptyset)$ )
190 C = INT(40* $\text{RND}(\emptyset)$ )
200 POKE780,0:POKE781,R:POKE782,C
210 SYS65520
215 REM SELECT RANDOM COLOUR
220 K=INT(16* $\text{RND}(\emptyset)$ )+1
230 PRINT CHR$(CC(K));" ";
240 NEXT

```

*Fig. 4.7.* Program to display a pattern of random coloured blocks.

### **Colour patterns**

The program listed in Fig. 4.7 uses random values of R and C with the direct cursor positioning routine. It PRINTs solid blocks of different colours in a random pattern all over the screen. Here reverse video is used with a space symbol to produce the coloured blocks, and colours are selected at random from the whole set of sixteen colours and set up on the screen using the CHR\$ colour codes.

Another type of pattern that can produce colourful displays makes use of mirror imaging. A program to produce this type of display is listed in Fig. 4.8. Here the screen is effectively divided into four quadrants around a centre point where  $R=12$  and  $C=20$ . A random point is chosen in one quadrant and a coloured block is PRINTed at that point. Next the mirror images of that point are plotted in the other three quadrants and blocks of the same colour are PRINTed. As the dots increase, a symmetrical pattern will develop, similar to the pattern produced by a kaleidoscope.

```

100 REM KALEIDOSCOPE PATTERN
110 DIM CC(16)
115 REM READ COLOUR CHR# CODE DATA
120 FOR N=1TO16:READ CC(N):NEXT
130 DATA 144,5,8,159,156,30,31,158
140 DATA 129,149,150,151,152,153,154,155
145 REM CLEAR SCREEN
150 PRINT CHR$(147)
155 REM SET REVERSE VIDEO ON
160 PRINT CHR$(18);
170 FOR N=1 TO 2000
180 Y = INT(12*RND(0))
190 X = INT(20*RND(0))
195 REM SELECT COLOUR
200 K=INT(16*RND(0))+1
210 R=12+Y:C=20+X:GOSUB400
220 PRINT CHR$(CC(K));" ";
230 R=12-Y:C=20+X:GOSUB400
240 PRINT CHR$(CC(K));" ";
250 R=12-Y:C=20-X:GOSUB400
260 PRINT CHR$(CC(K));" ";
270 R=12+Y:C=20-X:GOSUB400
280 PRINT CHR$(CC(K));" ";
290 NEXT
300 END
395 REM CURSOR POSITION SUBROUTINE
400 POKE780,0:POKE781,R:POKE782,C
410 SYS65520:RETURN

```

*Fig. 4.8.* Program to produce kaleidoscope patterns using the mirror image technique.

## Coloured wallpaper

To demonstrate the effect of changing colour on the character graphics display, we can now try a program which produces simple wallpaper-style patterns in a range of colours. The program is listed in Fig. 4.9.

In this program the computer generates a random string of five to twenty characters, and then repeatedly prints the string until the screen is filled with a pattern. Between each symbol in the string a random colour code is inserted, which selects one of the first eight colours for the following symbol. The length of the string is chosen so that it is not a submultiple of 40. As a result, the pattern is printed

```

100 REM COLOURED WALLPAPER
110 DIM C(8)
115 REM SET UP COLOUR CODES
120 FOR N=1 TO 8
130 READ C(N)
140 NEXT
150 DATA 5,28,30,31,144,156,158,159
160 FOR J=1TO500
170 PRINT"J";
175 REM SET BACKGROUND COLOUR
180 POKE 53281-8*RND(0)+8
190 REM CHOOSE STRING LENGTH
190 NC=INT(15*RND(0)+5)
200 IF INT(40/NC)=40/NC THEN 190
210 A$=""
215 REM SET UP TEXT STRING
220 FOR N=1TONC
225 REM SELECT GRAPHICS COLOUR
230 K=8*RND(0)+1
240 A#=A#+CHR$(C(K))
245 REM SELECT GRAPHICS SYMBOL
250 A#=A#+CHR$((54*RND(0))+161)
260 NEXT
265 REM DISPLAY PATTERN
270 FOR N=1TO1000-NC STEP NC
280 PRINT A$)
290 NEXT
300 FOR T=1TO2000:NEXT
310 NEXT
320 END

```

*Fig. 4.9.* Program to produce coloured wallpaper patterns.

at a slightly shifted position on the next row of symbols, and this produces diagonal patterns on the screen.

Each pattern is held on the screen for a few seconds by using a simple counting loop, and then the screen is cleared and a new pattern is drawn. The program generates a set of 50 patterns, but would go on producing patterns almost indefinitely if the size of the main loop using the variable J were increased.

To provide more variety in the patterns the background colour is set to a random colour by POKEing a number between 8 and 15 into the address 53280. This sets up one of the second eight colours for the background. You could, of course, set the background to any colour between 0 and 15, which would produce more variations, and

it would also be possible to use all sixteen colours for the graphics symbols. Remember that you will need to dimension the array C to 16 and add the other eight data codes to the DATA statement as well as increasing the number of items read.

### Using POKEs to display symbols

Instead of using the PRINT command to write text symbols to the screen we can adopt a different approach. You will remember that the character codes of the symbols being displayed are held in the screen memory, and their position in that area of memory is directly related to their position on the screen. Thus the code for the symbol at the top left corner position is in the first location in the screen memory. If we write the character code of the symbol we wish to display into the screen memory, it will be displayed on the screen. The position of the symbol on the screen now depends upon where we place it in the screen memory.

It is convenient to denote our symbol positions in terms of the row R of text that the symbol is in, and its column position C across that row starting from the left side of the screen. Since there are 25 rows we can number them from 0 to 24, and this will be the range of values for R. Similarly, with 40 characters per row the value of the column number, C, will range from 0 to 39. The layout of the screen memory is shown in Fig. 4.10, and you will notice that memory locations are allocated in sequence working across the rows from left to right, and then row by row down the screen from top to bottom.

To find the required memory location in the screen memory, given a row and column position on the screen, we can use the calculation

$$M = 1024 + 40 * R + C$$

Note here that the rows are numbered starting from the top of the screen.

Having found the memory location we can insert the required screen symbol code by using a POKE command as follows:

```
POKE N,SC
```

where SC is the screen symbol code for the character to be displayed, and should be in the form of a decimal number.

There is an important difference between POKEing a character to the screen and PRINTing it. The Commodore 64 does not use the normal ASCII character coding scheme when it stores data



representing the displayed symbols in its screen memory. For PRINTing to the screen using the CHR\$(N) format the standard ASCII coding is used, but for POKEing symbols to the screen a different set of codes is used. These are Commodore 64 *screen codes*. In the screen codes there are no control codes, such as for colour setting or cursor movement, and some blocks of symbols have their codes changed from their normal ASCII values.

The screen codes 0 to 31 correspond to ASCII codes 64 to 95. The codes from 32 to 63 are the same for both ASCII and screen memory. Codes 64 to 95 on the screen are equivalent to ASCII codes 96 to 127, and the screen codes from 96 to 127 produce the symbols with ASCII codes from 160 to 191.

The screen codes from 128 to 255 produce reverse video versions of the characters with screen codes from 0 to 127. Thus a space which has a code of 32 becomes a filled-in block when screen code 160 (128+32) is used. Reverse video on the screen is obtained by simply adding 128 to the basic screen code for the symbol. The particular set of symbols displayed will still depend on whether the upper or lower case symbol set has been selected. Normally upper case is selected at switch-on, and this provides the set of character graphics symbols.

When a PRINT command is used to put symbols on the screen the colour memory locations for those symbols are automatically set up to the current text colour. An important difference when POKEing symbols to the screen is that, unlike the PRINT command, a POKE does not set up the symbol colour. Thus POKEing a screen code into the screen memory may not always produce a visible symbol unless the corresponding colour memory location is set to the required display colour code. To deal with this situation, a colour code must be POKEd into the equivalent position in the colour memory. The simplest way to do this is to set up two variables SM and CM to give the start addresses of screen and colour memory areas. Normally these would be set up as SM=1024 and CM=55296. Now if the position in the memory P is calculated from

$$P = 40 * R + C$$

then the symbol and colour can be dealt with using

```
POKE SM+P,SC:POKE CM+P,CC
```

where SC is the symbol's screen code, and CC is the colour code (0 to 15) for the colour you want the symbol to have.

## Extended background colour mode

In the normal text or character graphics display mode of the Commodore 64, we have seen that changing the background colour causes the background of all displayed symbols to change at the same time. More colourful results would be possible if we could select different background colours for some areas of the screen. This can be achieved by using the Extended Background mode.

In the extended background colour mode we can set up four different background colours, and it becomes possible to allocate any one of the four colours as the background colour for each individual character on the screen.

The display chip needs to know which background colour to display in each character space, and to do this it uses two of the data bits in the character code that is stored in the screen memory. Here a trade-off has to be made, because normally all the bits in the screen memory word are used to define the symbol to be displayed. In the extended background mode, however, the upper two bits of the code are used to define the background colour, leaving only 6 bits for the character itself. The result is that we can only have 64 displayable characters when the extended background is selected. These symbols will, in fact, be the ones with codes from 0 to 63. Remember that here we are talking about *screen* data codes and not ASCII codes. The same restriction will apply if you set up a table of user-defined symbols.

To select the extended background mode we need to set bit 6 of register 17 of the VIC11 chip to '1', and to return to normal operation that bit must be reset to its normal '0' state. To set up the mode we can use the command

```
POKE 53265,PEEK(53265)OR 64
```

and to reset the system to the normal character display mode we use

```
POKE 53265,PEEK(53265)AND 191
```

The PEEK is included here because register 17 also controls other functions, so the states of the other bits should be preserved when switching the extended background colour bit. Note that when you switch back to the normal mode, all kinds of odd characters are likely to be displayed, because now the bits used to select the extra background colours become part of the character code.

Using two data bits there are four possible combinations of '1' and '0' states, and these will give the four alternative background

MS bits of screen code		Screen codes	Background colour	
Bit7	Bit6		(reg. address)	
0	0	0 - 63	No.0	(53281)
0	1	64 - 127	No.1	(53282)
1	0	128 - 191	No.2	(53283)
1	1	192 - 255	No.3	(53284)

Extended background colour mode is selected by setting bit 6 to '1' in register 17 (53265) of the VIC11 chip.

*Fig. 4.10.* Colours selected by the most significant bit pair of the symbol code when extended background mode is used.

colours. When both bits are at '0' the normal background colour, which is referred to as Background No 0, will appear in the symbol space just as in a normal character display.

The other three combinations of bits call up Background No. 1, Background No. 2 and Background No. 3 as shown in Figure 4.10.

Each of the four background colours is controlled by the contents of a register in the VIC11 chip, and each may be set to any of the sixteen colours by simply placing the required colour number into the appropriate background colour register.

To set up the background colour in a character space, all we have to do is set the corresponding combination in the two upper bits of the screen memory word for that character position on the screen.

This is done by firstly ANDing the data already in the memory with 63, which effectively sets the top two bits at '0'; then the result is ORed with the required background colour number multiplied by 64, as shown below.

```
POKE M,(PEEK(M)AND63)OR(BC*64)
```

When you use the CHR\$(N) and PRINT operation to place characters on the screen, remember that you can only use the symbols with ASCII codes from 32 to 95, which correspond to screen codes 32 to 63 and 0 to 31. Any other displayable character

```

100 REM EXTENDED BACKGROUND DEMO
105 REM CLEAR SCREEN AND POKE SYMBOLS
110 PRINT CHR$(147)
120 FOR I=0 TO 255
130 POKE1024+I,I:POKE55296+I,7
140 NEXT
145 REM SELECT EXTENDED BACKGROUND MODE
150 POKE53265,PEEK(53265)OR64
155 REM SET EXTENDED BACKGROUND COLOURS
160 POKE53282,2
170 POKE53283,4
180 POKE53284,8
190 END

```

*Fig. 4.11.* Demonstration of the extended background mode.

codes will produce these same symbols, but with a different background colour.

The program listed in Fig. 4.11 demonstrates this mode. This program starts by POKEing the complete set of 256 character patterns on to the screen in light blue. At this point you will note that graphics and reverse video symbols are displayed. The extended background colours are then set up and the extended background colour mode is selected. At this point you will note that all of the graphics symbols disappear, and four different background colours are presented in the four groups of 64 characters. If you want to use extended background with graphics symbols, then the required symbols should be set up as a user-defined character set, with screen codes running from 0 to 63. This is easily done by copying the required dot patterns from the ROM character generator.

### **Multicolour symbol mode**

Another mode which can be used to produce more colourful displays is the Multicolour mode. This allows us to choose four different colours for the dots making up the symbol.

Once again, a trade-off has to be made in order to find somewhere to store the extra colour information. This time the dots in each row of the character space are grouped in pairs, so that the character dot matrix now becomes effectively four wide by eight high. Now the two bits representing each of the adjacent dot pairs are used to select the display colour for the pair of dots.

Two of the available colours are the text colour, as read from the

Bit pair in dot pattern	Colour	Register address
00	Background No. 0	53281
01	Background No. 1	53282
10	Background No. 2	53283
11	Foreground	Colour RAM

**Note.** Only colours 0 - 7 can be set in colour RAM when this mode is used.

*Fig. 4.12.* Colours produced by character bit pairs in the multicolour mode.

colour memory, and the Background No. 0 colour. This is the normal overall screen background colour. The other two colours are the Background No. 1 and Background No. 2 colours specified by registers 22 (53282) and 23 (53283) in the VIC11 chip. These two colours can be set to any of the sixteen available colours by simply POKEing the required colour number into the registers.

The actual colours presented by a pair of dots in the symbol space is governed by the bit state combination of the pair of bits, and is shown in Fig. 4.12. Thus if the bits are set to the combination 01 they will both be displayed in Background colour No. 1.

A point to note is that in this mode only the first eight colour numbers can be used in the colour memory. Bit 3 of the colour memory is used for control purposes when this mode is selected. The program listed in Fig. 4.13 shows the effect on standard symbols when the multicolour mode is selected. Generally this mode will be used with user-defined symbols so that the dot patterns can be selected to pick out the desired colour combinations within the symbol space.

### **Multicolour bit map mode**

Normally in the bit map mode we have only two colours, one for the dots and one for the background. As for the text mode, there is a multicolour version of the bit map display mode. This allows a

```

100 REM DEMO OF MULTICOLOUR SYMBOLS
105 REM CLEAR SCREEN
110 PRINT CHR$(147)
115 REM SET GREY BACKGROUND
120 POKE 53281,11
130 FOR I=0 TO 900 STEP 2
140 POKE 1024+I,90:POKE55296+I,8
150 NEXT
155 REM SELECT MULTICOLOUR MODE
160 POKE 53270,PEEK(53270)OR16
170 FOR J=1 TO 7
180 FOR K=0 TO 7
185 REM SET MULTICOLOUR REGISTERS
190 POKE53282,J
200 POKE53283,K
210 FOR T=1TO 1500:NEXT
220 NEXT
230 NEXT
235 REM RESTORE NORMAL MODE
240 POKE 53270,PEEK(53270)AND239
250 POKE 53281,6

```

Fig. 4.13. Multicolour character mode demonstration program.

Bit pair in dot pattern	Colour	Register address
00	Background No. 0	53281
01	Upper 4 bits	Screen RAM
10	Lower 4 bits	Screen RAM
11	Lower 4 bits	Colour RAM

**Note.** This mode is selected by setting bit 5 of 53265 and bit 4 of 53270.

Fig. 4.14. Colours produced by bit pair combinations in the multicolour bit-mapped mode.

choice of four colours on the screen at any time. To cater for this, the horizontal resolution is reduced to 160. The scheme here is that the dot positions across the screen are taken in pairs, and each pair of dot bits indicates the colour for the pair of dots.

The technique therefore works in much the same way as for multicolour characters. Selection of the multicolour bit map mode involves setting the multicolour mode bit (bit 4 in register 22, address 53270) and also setting the bit map mode bit (bit 5 in register 17, address 53265). Colours are set up in the screen memory, colour memory and background register, and appear as shown in Fig. 4.14.

The dot plotting routine must be changed slightly to cater for the multicolour bit map mode. This is shown in the program listing of Fig. 4.15. Here the value of X is made even, by first dividing X by 2, then taking the integer value to remove any fractional part, and then multiplying by 2 to restore X to its normal range of values.

When setting the dots a colour variable C is used with a value from 0 to 3. This is used as a multiplier when setting the dot pattern, and effectively sets a pair of dots to the required combination. The program shown here plots random dots in three different colours. The normal line drawing routine can also be modified, if desired, so that it plots only alternate dots along the X axis. The normal line routine should still work satisfactorily with the modified dot routine, but it will plot the same pair of dots twice as it moves in the X direction.

```

100 REM MULTICOLOUR BIT MAP DOTS
105 REM NEEDS M/C SCREEN CLEAR ROUTINE
110 BM=8192
115 REM BIT MAP AT 8192 IN MEMORY
120 POKE53272,PEEK(53272)OR8
130 POKE53265,PEEK(53265)OR32
140 FOR I=1024 TO 2023:POKEI,37:NEXT
150 SYS49152
160 GOTO1000
195 REM PLOT MULTICOLOR DOT
200 X=2*INT(X/2)
210 P=BM+320*INT(Y/8)+8*INT(X/8)+(YAND7)
220 CP=C*2+(6-(XAND7))
230 POKE P,PEEK(P) OR CP
240 RETURN
1000 FOR I=0 TO 999:POKE55296+I,7:NEXT
1005 REM SELECT MULTICOLOUR MODE
1010 POKE53270,PEEK(53270)OR16
1020 FOR N=1 TO 200
1030 X=INT(318*RND(0))
1040 Y=INT(198*RND(0))
1050 C=1+INT(3*RND(0))
1060 GOSUB200
1070 Y=Y+1
1080 GOSUB200
1090 NEXT
1100 END

```

*Fig. 4.15.* Program to plot random coloured dots using the multicolour bit map mode.



## Chapter Five

# Setting Objects in Motion

For most computer games, particularly the arcade type, an important feature is the display of moving objects on screen. Computer displays of this type make use of the basic principles of the cartoon film. A sequence of pictures is produced, and the individual pictures or 'frames' are presented in rapid succession on the screen. The moving object is drawn in a slightly different position on each successive picture, and because the viewer's eyes retain each image for a short time after it has disappeared, successive pictures merge together and the object appears to move smoothly across the screen.

The television pictures producing the computer display are also being presented in rapid succession, so if an object on the screen is moved to a new and slightly different position on successive display scans then the object will appear to move over the screen. To achieve reasonably smooth movement, the changes between pictures are made at least once every  $1/10$  second. If the movement steps are small, the movement will appear to be smooth but slow. With larger steps between successive pictures the motion becomes faster but may tend to appear jerky.

In the simplest form of animation an object such as a ball, alien invader or spaceship is moved from one character position to the next either left, right, up, down or possibly diagonally. For more realistic results, the object on the screen may need to change shape as it moves. An example of this would be a man walking across the screen. If the image of the man remained constant he would appear to glide across the screen rather like an ice skater. To give the impression of walking or running the position of the legs, and perhaps the arms too, must be changed regularly. In effect a series of 'snapshots' of the action of walking are presented in rapid succession. Most actions such as walking are repetitive, so perhaps three or four different images may be used, and the sequence is

simply repeated at different screen positions as the man moves across the display. When animating an object, and particularly a familiar real-life object, it is important to study how it moves carefully if you want to get realistic animation.

### A simple moving ball

For many games-type programs a simple object such as a ball moves around the screen. This is fairly easy to achieve by using character graphics on the normal text display screen. On the Commodore 64 there is a convenient symbol which displays a circle giving a good representation of a ball. This has the display code 87, or 81 if you want the circle filled in with text colour.

Let us start by placing the ball somewhere near the middle of the screen, by POKEing its code into screen memory location 1524. Remember that screen memory runs from 1024 to 2023. Remember, too, that you need to POKE a colour number into the corresponding position in the colour memory. A variable P may be used to keep track of the position of the ball. It is convenient to let P run from 0 to 999 and add P as an offset to either the colour memory address (CM) or the screen memory start address (SM). These two variables are set to 1024 and 55296 at the start of the program.

To move the ball to the right we want to POKE it into memory location 1525. Having POKEd the symbol into the new position we must now erase the original symbol, otherwise we shall end up with a trail of balls across the screen. This is easily done by POKEing 32 (a space symbol) into location P. Now the value of P is updated by making it equal to PN so that it still points to the current ball position. This process is then repeated to make the ball move to the right across the screen. This can be done by using a routine such as the following:

```
500 PN=P+1
510 POKE SM+PN,87:POKE CM+PN,7
520 POKE SM+P,32
530 P=PN
540 RETURN
```

Here the ball colour is 7 (yellow), which shows up well on the blue background. Each time the ball is to be moved we simply use a GOSUB500 statement in the main program. To move the ball to the

left we need to make PN equal to P-1 when calculating the new position for the ball in line 500.

For vertical motion of the ball the calculations have to be a little different. In the screen memory each row of text takes up 40 memory locations. Now if we want to move the ball to the space immediately below its current position, P, we shall have to add 40 to P to get the required value for PN. Apart from this change the routine would be the same as for moving the ball horizontally. If we want to move to the position immediately *above* the current one then PN is obtained by subtracting 40 from P. As for the sideways motion, the ball is blanked from position P by POKEing 32 to that location before P is updated to its new PN value, ready for the next move.

Moving diagonally is a little more complicated because we have a combination of left-right and up-down movements. To move up to the right we have a right move (+1) and an up move (-40), so P needs to have 39 subtracted from it to get the new position. Figure 5.1 shows the change in the value of P required for making a single step in each of the eight basic directions. This figure assumes that the ball is at location P, which would be at the centre point of the diagram.

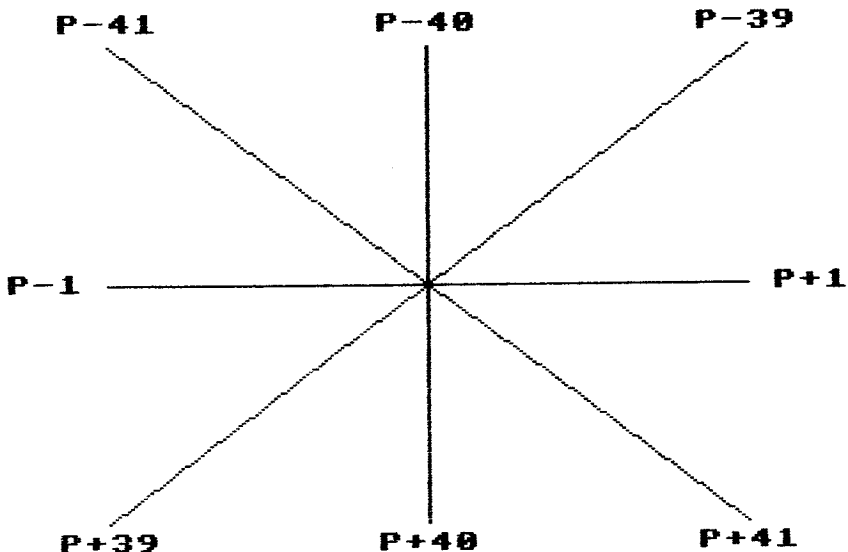


Fig. 5.1. Required changes to the POKE address in the screen memory for one symbol space movement in the eight main directions of motion.

## Movement using PRINT and cursor set

Instead of POKEing the ball symbol to the screen we could, of course, PRINT it. Here the cursor can be set directly to a row R and column C position by using the direct cursor set routines described in Chapter 2. To move to the right C is increased by 1, and to move left C is reduced by 1. For an upward step the value of the row R is reduced by 1, and to move down, the row is increased by 1. Diagonal motions involve combinations of these steps as shown in Fig. 5.2. In the PRINT statement a semicolon should be included after the string or CHR\$ code for the ball symbol.

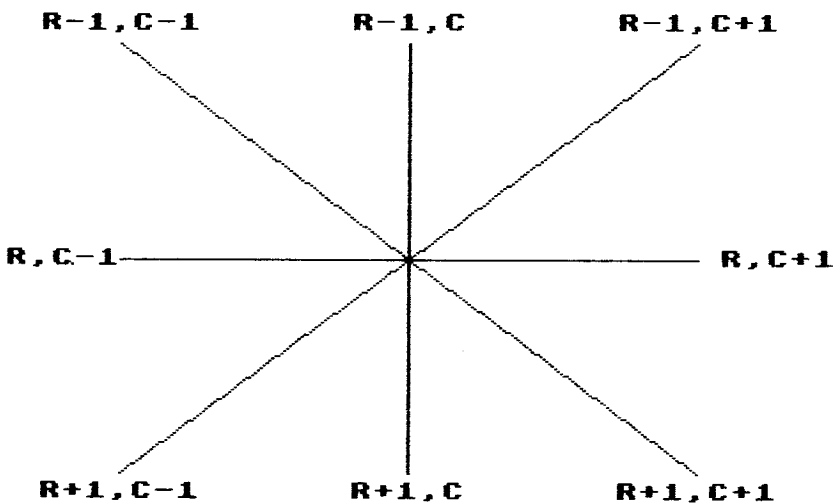


Fig. 5.2. Changes in R,C values needed for different directions of motion when using PRINT.

The previously displayed ball has to be blanked out, of course. Here the cursor is set to the current ball position and a blank space is printed. Then the new values for R, C may be calculated and the cursor repositioned to allow the ball to be printed.

## Bouncing off the walls

If you tried moving the ball using the simple program sequence we have just discussed problems would soon arise. Suppose we start with the ball at the centre of the screen and then move it step by step to the right. All will be well until the ball reaches the edge of the screen and then, on its next step, the ball will go off the right edge and reappear at the left side of the screen on the next line down. If the ball is moving to the left it will reappear at the right, one line

higher. If the ball were to run off the top or bottom of the screen the symbol would be POKEd into memory outside the screen memory, and could cause the program to fail; worse still, it could lock up the whole computer system. To avoid this state of affairs we can arrange that when the ball reaches one of the edges of the screen this is detected, and the direction of motion is reversed. This way when the ball reaches a screen edge it is reflected back towards the middle, as if it had bounced off a wall.

This bouncing action is fairly easy to achieve. The technique is to draw a border around the screen using four different graphics symbols. At the right we might use symbol 97, which has its left half lit, while at the left we would use symbol 225, which has its right half lit. For the top and bottom the symbols with codes 98 and 226 respectively are used. A problem occurs in the corners, and here a solid block (code 160) is used. If this were not done the ball might escape through the corner and result in numbers being POKEd into memory areas outside the screen memory.

Now we can allocate two new variables, DX and DY, which give the change in PC required for horizontal or vertical steps. Thus DY would be 40 for moving down, and DX would be 1 for moving right. Now PN is calculated by adding the current values for DX and DY to PC. When we want to reverse the sideways motion we simply invert the sign of DX by using

```
700 DX=-DX
```

and to reverse the up-down motion we apply the same process to DY. When only up-down motion is required then DX is set to zero, and for sideways motion DY is set at zero.

Having calculated PN we can detect a screen edge by PEEKing the contents of location PN. The value we get is now compared with the character codes for the edges of the screen. If a match is detected with one of the right or left edge symbols the sign of DX is reversed, and the movement of the ball symbol is skipped so the ball stays where it is. On the next move, however, the left-right motion is reversed, and the ball will move back away from the edge of the screen. If the symbol detected is one of those at the top or bottom of the screen then the sign of DY is changed, and here the up-down motion of the ball is reversed on the next step. If a corner is detected both DX and DY are reversed in sign.

The result is that the ball appears to bounce off the walls, and this is demonstrated by the program listed in Fig. 5.3. In this program the first section from line 110 to line 280 simply clears the screen and

```

100 REM BOUNCING BALL
110 SM=1024:CM=55296
115 REM CLEAR SCREEN
120 PRINT"J"
125 REM DRAW BORDER
130 FOR J=1 TO 38
140 POKESM+J,98:POKECM+J,5
150 NEXT
160 FOR J=79 TO 983 STEP 40
170 POKESM+J,97:POKECM+J,5
180 NEXT
190 FOR J=961 TO 999
200 POKESM+J,226:POKECM+J,5
210 NEXT
220 FOR J=40 TO 920 STEP 40
230 POKESM+J,225:POKECM+J,5
240 NEXT
245 REM FILL CORNERS
250 POKESM,160:POKECM,5
260 POKESM+39,160:POKECM+39,5
270 POKESM+960,160:POKECM+960,5
280 POKESM+999,160:POKECM+999,5
285 REM SET START POSITION
290 PC=200+300*RND(0)
295 REM SET START DIRECTION
300 DX=1:DY=40
305 REM BALL MOVEMENT LOOP
310 FOR N=1 TO 1500
320 GOSUB500
330 NEXT
340 END
495 REM BALL MOVE SUBROUTINE
500 PN=PC+DX+DY
505 REM CHECK FOR EDGE OF SCREEN
507 REM AND APPLY BOUNCE ACTION
510 IF PEEK(SM+PN)<>160 THEN 530
520 DX=-DX:DY=-DY:PN=PN+DX+DY:GOTO610
530 IF PEEK(SM+PN)<>98 THEN 550
540 DY=-DY:PN=PN+DY:GOTO610
550 IF PEEK(SM+PN)<>97 THEN 570
560 DX=-DX:PN=PN+DX:GOTO610
570 IF PEEK(SM+PN)<>226 THEN 590
580 DY=-DY:PN=PN+DY:GOTO610
590 IF PEEK(SM+PN)<>225 THEN 610
600 DX=-DX:PN=PN+DX

```

```

605 REM MOVE BALL
610 POKESM+PN,87:POKECM+PN,7
620 POKESM+PC,32:PC=PN
630 RETURN

```

*Fig. 5.3.* Program to demonstrate a bouncing ball animation routine.

draws in the border. The ball starts off at a random point somewhere around the centre of the screen memory, and is initially sent off diagonally downward and to the right. The ball movement and edge detection are carried out in the subroutine starting at line 500.

If PRINT is used to draw the ball then the bounce can be initiated if  $C=0$  OR 39 for the X direction reversal. Similarly if  $R=0$  OR  $R=24$  the up-down (Y) motion could be reversed. Here it is easier to compare row and column values with the edges of the screen rather than checking for a border symbol.

Thus the check and bounce operation becomes as follows:

```

600 IF R=24 OR R=0 THEN DY=-DY
610 IF C=39 OR C=0 THEN DX=-DX

```

This assumes that the movement is stepping one symbol space at a time. For faster movement, of course, the ball could be moved two or three spaces at a time. In this case the test needs to be altered to detect  $<0 \text{ OR} >N$  where N is either 24 or 39 according to whether the row or column limit is being checked. To prevent PRINTing off the screen, the actual PRINT step should be skipped after a limit is detected.

## Detecting collisions

In many games the ball has to be hit by a bat. In others missiles are fired, and we need to know if they have hit their target. The basic principle for detecting collisions or hits is the same as that used for detecting the border in the ball routine. The new location of the object is checked by using a PEEK command before the object itself is moved. The contents of the location are then compared with possible objects that could be there, and then appropriate action is taken.

Sometimes we may want to move an object over a background scene without erasing the background. In this case, before the moving object symbol is POKEd into its new position the character code at the new location PN is PEEKed and stored as a variable,

such as CC. On the next move this code is POKEd back into its original position again.

If PRINT is used to display the moving object, any collisions that occur may be detected by PEEKing as described above. An alternative approach is to keep track of the positions of all objects that are moving on the screen and these are compared for possible matches at each movement step. If there are several objects this can become a complex exercise.

In general, if collisions and hits are to be detected it is probably much easier to use sprite graphics, where the VIC11 chip does the checking for you. We shall be exploring sprite graphics in the next chapter.

### Scrolling the screen

One method of producing movement in the screen display makes use of an action called *scrolling*. If you are in the normal text mode, for example while typing in a program, and the display has reached the bottom line of the screen, you will notice that when the bottom line is completed the whole display moves up by one line. This leaves a blank line at the bottom of the screen ready to accept new input. This action is known as 'scrolling'; the effect is like a paper scroll moving past a window represented by the screen. Each time the display is scrolled upward the original top row of text is lost, and a new row may be entered at the bottom.

By writing some routines to shift the data stored in the screen and colour memories it is possible to make the display scroll down the screen so that new data may be entered on the top line. It is also possible to carry out sideways scrolling from left to right or vice versa. With sideways scrolling a complete column of symbols is moved off one side of the screen and a new column of symbols is moved in from the opposite side.

The actual process of scrolling involves transferring the contents of the screen memory one position to the right, left, up or down. Suppose we want to scroll to the right. Starting with the top row of the display the 38th character is copied into the 39th position, then the 37th is moved to the 38th position, and so on until the whole row has moved one space to the right. Then all the other rows are moved in the same way. You could write a BASIC program to perform this action by PEEKing locations in the 38th column, then POKEing the result into column 39, and so on. You will find, however, that it takes



START	LDY #39	Column count
	LDA #0	From address
CLOOP	STA 251	
	LDA #4	in 251/252
	STA 252	
	LDA #1	To address
	STA 253	
	LDA #4	in 253/254
	STA 254	
	LDX #25	Row count
RLOOP	LDA 251,Y	Move one byte
	STA 253,Y	screen memory
	CLC	
	LDA 252	Update
	ADC #212	pointers
	STA 252	to
	CLC	colour
	LDA 254	memory
	ADC #212	
	STA 254	
	LDA 251,Y	Move colour
	STA 253,Y	memory byte
	SEC	
	LDA 252	Restore
	SBC #212	pointers
	STA 252	to
	SEC	screen
	LDA 254	memory
	SBC #212	
	STA 254	
	CLC	
	LDA 251	Update
	ADC #40	pointers
	STA 251	to
	LDA 252	next
	ADC #0	row
	STA 252	
	CLC	
	LDA 253	
	ADC #40	
	STA 253	
	LDA 254	
	ADC #0	
	STA 254	
	DEX	
	BEQ NEXTC	Column done?
	JMP RLOOP	
NEXTC	DEY	
	BMI DONE	All done?
	JMP CLOOP	
DONE	RTS	

*Fig. 5.4.* Assembly code routine to scroll the screen to the right by one column position.

several seconds to scroll the entire screen one column to the right. Scrolling the screen using BASIC is too slow to be of much use for animation. To produce faster scrolling the movement of data in the screen memory is usually carried out by machine code routines. The Commodore 64 already has a machine code routine for scrolling the screen up and this can automatically be brought into play by printing to the bottom row on the screen.

```

100 REM SCROLL RIGHT ROUTINE
110 REM USING MACHINE CODE
120 REM CALL WITH SYS49152
130 T=0
140 FOR N=49152 TO 49246
150 READ A:POKE N,A:T=T+A
160 NEXT
170 READ C
180 IF T<>C THEN PRINT"DATA ERROR"
190 PRINT"LOADED OK"
200 DATA 160,38,169,0,133,251,169,4
210 DATA 133,252,169,1,133,253,169,4
220 DATA 133,254,162,25,177,251,145,253
230 DATA 24,165,252,105,212,133,252,24
240 DATA 165,254,105,212,133,254,177,251
250 DATA 145,253,56,165,252,233,212,133
260 DATA 252,56,165,254,233,212,133,254
270 DATA 24,165,251,105,40,133,251,165
280 DATA 252,105,0,133,252,24,165,253
290 DATA 105,40,133,253,165,254,105,0
300 DATA 133,254,202,240,3,76,20,192
310 DATA 136,48,3,76,2,192,96,14210

```

Fig. 5.5. BASIC program to load the machine code scroll right routine.

If you want to scroll the screen sideways or downwards a machine code routine has to be written for this function. The assembly language program listed in Fig. 5.4 provides a machine code routine to scroll the screen to the right. This routine in fact scrolls both the screen memory and the colour memory at the same time. To get the routine into the Commodore 64 the data representing the machine code instructions is loaded into the computer memory using the short BASIC program listed in Fig. 5.5. This machine code routine starts at location 49152 and takes up 96 bytes of memory. It may be called from a BASIC program by using the statement

SYS49152

Since the machine code is stored above the memory area used by BASIC you can type NEW and load another BASIC program without affecting the scrolling routine, which remains in memory until the machine is switched off.

As usual, some care is needed when typing the DATA statements. Any error can produce unpredictable results when the machine code is run, and will probably cause the whole computer system to lock up. A simple check routine has been built in which should pick up most errors and give a warning message. This adds up all of the DATA items and compares the result with a test total which is stored immediately after the machine code data. If the test checks out the computer will print 'LOADED OK'. The check is not infallible, since two errors may cancel one another to give the correct check result or one of the data items with a zero value may have been omitted, so it is as well to check the DATA statements anyway. Once a correct version is obtained, save it on tape for future use.

```
1000 REM SCROLL RIGHT DEMO
1005 REM NEEDS M/C ROUTINE LOADED
1007 REM INTO MEMORY AT 49152
1010 FOR N=1 TO 100
1020 C=N AND15
1030 FOR K=1TO8:SYS49152:NEXT
1040 FOR J=0TO960 STEP 40
1050 POKE1024+J,42:POKE55296+J,C
1060 NEXT
1070 NEXT
1080 END
```

*Fig. 5.6.* BASIC program to demonstrate the action of the scroll right routine.

The program listed in Fig. 5.6 can be used to demonstrate the scroll right routine. It may be added to the end of the loading program of Fig. 5.5 or loaded separately. When RUN, this program should produce a display of vertical coloured bands of \* symbols scrolling to the right across the screen. Press RUN/STOP to stop the program and then RUN/STOP with RESTORE to return to normal BASIC operation. Note that when a new column of stars is POKEd into column 0 of the screen memory a column of colour codes must be POKEd into the colour memory as well. If you only want to load the scroll routine, just run the program listed in Fig. 5.5.

## Scrolling to the left

For games programs such as *Defender* a landscape moves across the bottom of the screen from right to left, and this effect may be achieved by scrolling the entire screen to the left and inserting new data at the right-hand side.

The program listed in Fig. 5.7 will load a machine code routine to scroll the screen to the left. This routine is loaded into memory just above the scroll right routine. The machine code starts at location 49254 and uses 96 bytes of memory. It may be called from BASIC by using the statement

```
SYS49254
```

In this routine the symbols in column 2 of the screen are moved to column 1, then column 3 moves to column 2, and so on across the screen. New data may then be placed in column 39 ready to scroll across the screen. Once again, a careful check should be made for

```
100 REM SCROLL LEFT ROUTINE
110 REM USING MACHINE CODE
120 REM CALL WITH SYS49254
130 T=0
140 FOR N=49254 TO 49350
150 READ A:POKE N,A:T=T+A
160 NEXT
170 READ C
180 IF T<>C THEN PRINT"DATA ERROR"
190 PRINT"LOADED OK"
200 DATA 160,0,169,1,133,251,169,4
210 DATA 133,252,169,0,133,253,169,4
220 DATA 133,254,162,25,177,251,145,253
230 DATA 24,165,252,105,212,133,252,24
240 DATA 165,254,105,212,133,254,177,251
250 DATA 145,253,56,165,252,233,212,133
260 DATA 252,56,165,254,233,212,133,254
270 DATA 24,165,251,105,40,133,251,165
280 DATA 252,105,0,133,252,24,165,253
290 DATA 105,40,133,253,165,254,105,0
300 DATA 133,254,202,240,3,76,122,192
310 DATA 200,192,39,240,3,76,104,192
320 DATA 95,14863
```

Fig. 5.7. Program to load a machine code routine to scroll the screen to the left.

errors in the DATA statements although a check routine is built in. This routine can be in memory at the same time as the scroll right routine since it is in a different section of memory. A point to note is that the routines are separated by a gap of several memory locations.

```

1000 REM SCROLL LEFT DEMO
1010 REM M/C ROUTINE MUST BE LOADED
1020 REM AT MEMORY LOCATION 49254
1030 PRINT CHR$(147)
1040 L=959:D=-40:SM=1024:CM=55296
1050 FOR N=1TO 500
1060 I=INT(7*RND(0))
1070 IF D>0 THEN S=77
1080 IF D<0 THEN S=78
1090 FOR J=0TO I
1100 POKESM+L,S:POKECM+L,3
1110 SYS49254:POKESM+L,32:L=L+D
1120 IF L>959 THEN L=959:S=100
1130 IF L<719 THEN L=719:S=99
1140 NEXT
1150 IF S=77 OR S=78 THEN L=L-D
1160 D=-D:NEXT
1170 END

```

*Fig. 5.8.* Program to demonstrate a simple moving landscape display using the machine code scroll left routine.

The BASIC program of Fig. 5.8 produces a moving landscape display using the machine code scroll left routine. In this program a single symbol is POKEd into column 39 and a colour code is POKEd into the corresponding position in the colour memory. The screen is then scrolled left by one column and the character in column 39 is blanked by POKEing a space (code 32) symbol in its place. This process then repeats continuously. The position of the symbol in column 39 is then stepped up or down for random numbers of steps to produce the effect of hills moving past on the landscape. Note the test to prevent the POKE from going outside the screen memory area. Don't forget to load the scroll routine into memory if you want to use it in a BASIC program.

### Scrolling down the screen

The third machine code routine presented here (Fig. 5.9) will scroll

```

100 REM SCROLL DOWN ROUTINE
110 REM USING MACHINE CODE
120 REM CALL WITH SYS49360
130 T=0
140 FOR N=49360 TO 49453
150 READ A:POKE N,A:T=T+A
160 NEXT
170 READ C
180 IF T<>C THEN PRINT"DATA ERROR":END
190 PRINT"LOADED OK"
200 DATA 169,152,133,251,169,7,133,252
210 DATA 169,192,133,253,169,7,133,254
220 DATA 162,24,160,39,177,251,145,253
230 DATA 24,165,252,105,212,133,252,24
240 DATA 165,254,105,212,133,254,177,251
250 DATA 145,253,56,165,252,233,212,133
260 DATA 252,56,165,254,233,212,133,254
270 DATA 136,48,3,76,228,192,56,165
280 DATA 251,233,40,133,251,165,252,233
290 DATA 0,133,252,165,253,233,40,133
300 DATA 253,165,254,233,0,133,254,202
310 DATA 240,3,76,226,192,96,15511

```

*Fig. 5.9.* Program to load a machine code routine to scroll the display down the screen.

the entire screen display down one row at a time. In this case the routine starts with the bottom two rows of text and transfers the symbols from the 24th row into the 25th row. Then the 23rd row is shifted to the 24th row position and so on, working up the screen to the top. New data may now be inserted into the top row and the process repeated. The result on the screen is a picture that appears to move down the screen.

This machine code routine starts at location 49360 and takes up 94 bytes of memory. It is called by using SYS49360. A data check has been built in but it is still a good idea to check the DATA statements for possible errors before attempting to run the machine code routine.

This type of scrolling action is often used for road race games where the player's car is displayed at the bottom of the screen and the road scrolls down the screen past the car. In simple versions the car can be moved from side to side and the object is to stay on the road. Collisions with the sides of the road are counted as crashes. Sometimes, in more complex versions, cars and other objects scroll down the screen as well as the road.

You could in fact load all three machine code routines into memory at the same time to allow scrolling to the left, right or down at will. Scrolling up would use the built-in routine if required. Each routine is called by a SYS command, and these are as follows:

SCROLL RIGHT	SYS49152
SCROLL LEFT	SYS49254
SCROLL DOWN	SYS49360

When scrolling in any direction you will need to use BASIC to set up the new row or column that is to move on to the screen into the space left by the scrolling action. This will probably impose a limit on how fast the movement can be.

### **The 38-column and 24-row modes**

The Commodore 64 has some special display modes available for use with scrolling animation. One has only 38 columns displayed. One column at each side of the screen is blanked off by the VIC11 chip although there may in fact be data stored in the appropriate positions in the screen memory. This scheme allows new data to be written into the side column, but it will not become visible until the data in the screen memory is scrolled.

This 38-column mode can be selected by resetting bit 3 in register 16 of the VIC11 chip to a '0' state. The memory address for the register is 53270, and to select the required mode the following POKE can be used:

POKE53270,PEEK(53270) AND 247

There will now be one blanked-off column at each side of the screen, where new data can be POKEd without being visible until it is scrolled on to the screen.

To get back to the normal 40-column display mode the bit in register 16 must be set as follows:

POKE53270,PEEK(53270) OR 8

If you are scrolling up or down the screen a 24-row mode can be selected. There is only one blank row in this mode, and it may be placed either at the top or the bottom of the display as required. For a blank line at the top the following POKE can be used:

POKE53265,PEEK(53265)AND247

If the blank line is to be at the bottom of the screen then use

POKE53265,PEEK(53265) OR 7

To restore the normal 25-row screen display use

POKE53265,PEEK(53265) OR 8



# Chapter Six

## Sprite Graphics

Perhaps the most interesting graphics display feature on the Commodore 64 is the sprite graphics capability provided by the VDP11 chip. As we saw in Chapter 1, sprites are in some ways like super-size user-defined symbols, since their actual dot pattern is set up by the user. Sprite graphics, however, provide more than the simple user-defined symbols used in character graphics, as we shall see in this chapter.

### Defining a sprite pattern

Like a user-defined character, a sprite is made up of a pattern of dots, but instead of using an  $8 \times 8$  dot matrix the sprite is much bigger, and is laid out over a 24-dot wide by 21-dot high matrix. This is shown in Fig. 6.1.

As in the case of a user-defined symbol, the data which defines the dot pattern for the sprite is stored in the RAM part of the Commodore 64 memory, but each sprite pattern takes up a rather larger chunk of memory than a symbol pattern. The 24 dots across each row in the sprite are grouped together to form three 8-bit memory words or *bytes*. Since there are 21 rows of dots in the sprite, we shall need  $21 \times 3$  or 63 bytes altogether to define the dot pattern. In fact an extra unused byte is added to make a total of 64 bytes, since this fits in conveniently with the binary addressing scheme used by the computer itself.

Defining a sprite pattern involves much the same process as defining a text or graphics symbol, except that it is on a larger scale. We can start by laying out a grid of  $24 \times 21$  squares as shown in Fig. 6.1. Each row is now divided into three sections, with eight squares in each, so that effectively the whole matrix is split up into three eight-dot wide columns. Each of the eight dots across a column has a

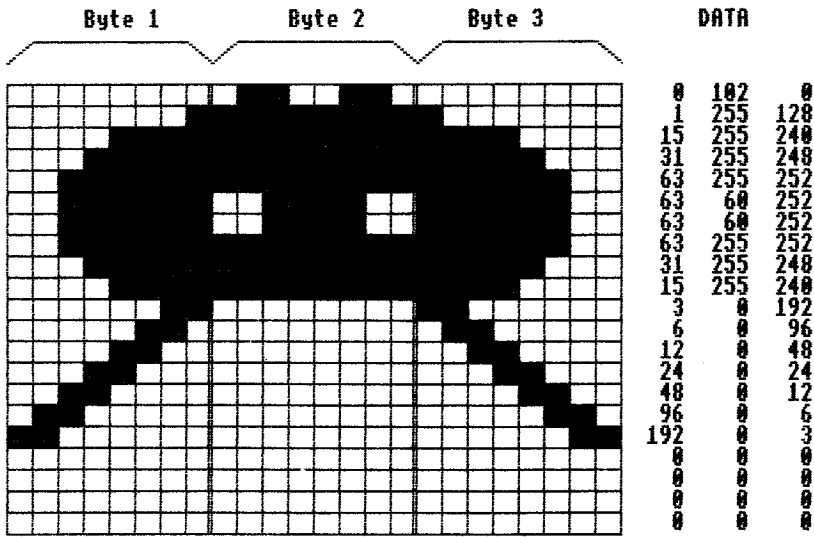


Fig. 6.1. Layout of the dot matrix for a sprite showing the data values for a typical sprite pattern.

numerical value from 1 to 128, just like the dots in a symbol matrix. Working from left to right the sequence of values for the dots is 128, 64, 32, 16, 8, 4, 2 and 1.

The dots in a standard sprite, like those in a text symbol, can be either 'on' (1) or 'off' (0). In the case of a symbol the '0' dots are displayed in the background, but for a sprite the '0' dots are treated as if they were transparent, so that whatever is already on the screen at that point will be displayed. At this stage all the squares corresponding to lit dots can be shaded in and are treated as '1's. Now we can take each group of eight dots in the top row and add together the numerical value of all the dots at '1' to give a value for the memory byte. Working from left to right the three bytes for the top row of dots in the sprite are stored in successive memory locations. The remaining bytes making up the sprite definition are then stored in groups of three, working down row by row from the top to the bottom of the sprite, and they will occupy the next 60 memory locations to give a total of 63 memory locations for the sprite. The 64th memory location can either be set to 0 or ignored, according to which is more convenient in the program. If you are setting up three or four different sprite patterns it may be more convenient to include the 64th byte as a zero in the data array so that the whole set of data for all the sprites can be read into memory in a single read loop.

Figure 6.1 shows the dot patterns for an alien invader figure, and alongside are shown the data values required for the 63 bytes of data that define the dot pattern for the sprite.

### **Locating the sprite definitions**

We have said that the sprite pattern definitions are stored in memory, but now we must consider where they go and how we tell the VIC11 chip where to find them.

As far as the position in memory is concerned, the sprite data can be placed anywhere within the 16k byte bank of memory that is being seen by the video chip. Normally this will be bank 0, which starts at location 0 at the bottom of memory and goes up to memory location 16383. In this area, as we have already seen, there are some sections which are used by the computer itself, while others are reserved for the screen display and the character generator ROM. The BASIC program is also stored in this area, starting at location 2048.

For a small number of sprites it is possible to use the cassette buffer area which runs from location 828 to 1019. Of course, if you use this area of memory you will not be able to use the cassette tape while working with the sprites since the cassette data will overwrite your sprite patterns. This piece of memory is 192 bytes long and gives enough space for three sprite patterns. If you need more space for sprite dot patterns then a convenient place is at 12k (12288), provided you have not already used it for user-defined characters or the bit map memory.

There are 64 bytes in each sprite definition block, so it is convenient to divide the memory up into 64-byte blocks and number these 0, 1, 2 etc starting from location 0 of the memory. Block number 13 will have an address of 832 ( $13 \times 64$ ) which is four bytes after the start of the tape buffer area. Fortunately the four bytes following the tape buffer area are not used, so we can still get 192 bytes for sprite definitions starting from location 832 in memory. Three sprite dot patterns can now be located in blocks 13, 14 and 15 of the memory.

Wherever the sprite data pattern is located, the video chip will need to know where to find it, and this is achieved by a series of sprite pointers. These are single bytes located at memory positions 2040 to 2047, just above the screen memory area. One byte is used for each sprite as shown in Fig. 6.2. Each sprite pointer can have a value from

Memory Address	Points to dot data for
2040	Sprite 0
2041	Sprite 1
2042	Sprite 2
2043	Sprite 3
2044	Sprite 4
2045	Sprite 5
2046	Sprite 6
2047	Sprite 7

Fig. 6.2. The sprite pointer locations in computer memory.

0 to 255, and if we use this to denote which 64-byte block of memory contains the dot pattern then we can locate the sprite pattern anywhere within  $256 \times 64$  or 16k bytes of memory – which just happens to be the section of memory that can be seen by the video chip. So if we want to have the dot pattern for sprite 0 located at, say, 12288 (block number 192), the number 192 would be placed into the sprite 0 pointer at location 2040 in memory. Now when the video chip wants the dot pattern for sprite 0 it will read the 192 from the pointer at location 2040 and then pick up the dot data starting from location 12288 ( $192 \times 64$ ) in memory. If we changed the pointer data to 14, then the dot data would be picked up starting from location 896 ( $14 \times 64$ ) and so on.

There are eight independent sprites, and they can all have different dot patterns. Of course, you can have two or more sprites with the same pattern. For example, if the memory pointers for, say, sprites 1 and 4 are the same, then the two sprites will have the same dot pattern. You could, in theory, have up to 256 different sprite patterns stored in the memory at the same time, and select up to eight of these at a time for the sprites that are displayed on screen. To change the shape of a sprite all you need do is change the number in its pointer location to point to a different block in memory where the new dot pattern is stored.

In the program listed in Fig. 6.3, the data for the invader figure has

```

100 REM SPRITE GENERATION AND CONTROL
105 REM CLEAR SCREEN
110 PRINT CHR$(147);"SPRITE CONTROL DEMO"
115 REM VIC CHIP START ADDRESS
120 VC=53248
125 REM SET SPRITE POINTERS
130 FOR I=0 TO 7:POKEVC+I,13:NEXT
135 REM READ SPRITE DATA
140 FOR N=0 TO 62
150 READ A:POKEVC+N,A
160 NEXT
165 REM TURN ON SPRITE 0
170 POKEVC+21,255
175 REM SET SPRITE 0 TO YELLOW
180 POKEVC+39,7
185 REM SET SPRITE 0 X,Y POSITION
190 POKEVC,150:POKEVC+1,100
195 REM TURN SPRITE 1 ON
200 POKEVC+21,PEEK(VC+21)OR2
205 REM SET SPRITE 1 TO PURPLE
210 POKEVC+40,4
215 REM POSITION SPRITE 1
220 POKEVC+2,155:POKEVC+3,95
225 REM SET SPRITE 2 ON AND CYAN
230 POKEVC+21,PEEK(VC+21)OR4
240 POKEVC+41,5
250 POKEVC+4,50:POKEVC+5,120
255 REM SET SPRITE 2 TO DOUBLE WIDTH
260 POKEVC+23,4
265 REM SET UP SPRITE 3
270 POKEVC+21,PEEK(VC+21)OR8
280 POKEVC+42,3
290 POKEVC+6,200:POKEVC+7,120
295 REM SET SPRITE 3 TO DOUBLE HEIGHT
300 POKEVC+29,8
310 FOR T=1 TO 2000:NEXT
315 REM SET SPRITE 1 TO DOUBLE SIZE
320 POKEVC+23,PEEK(VC+23)OR2
330 POKEVC+29,PEEK(VC+29)OR2
340 FOR T=1 TO 10000:NEXT
350 POKE VC+23,0:POKEVC+29,0
360 FOR T=1 TO 2000:NEXT
365 REM SWITCH OFF SPRITES 0, 1 AND 3
370 POKEVC+21,2
380 FOR T=1 TO 2000:NEXT

```

```

385 REM SWITCH OFF ALL SPRITES
390 POKEVC+21,0
400 END
495 REM SPRITE DATA
500 DATA 0,102,0,1,255,128
510 DATA 15,255,240,31,255,248
520 DATA 63,255,252,63,60,252
530 DATA 63,60,252,63,255,252
540 DATA 31,255,248,15,255,240
550 DATA 3,0,192,6,0,96
560 DATA 12,0,48,24,0,24
570 DATA 48,0,12,96,0,6
580 DATA 192,0,3,0,0,0
590 DATA 0,0,0,0,0,0
600 DATA 0,0,0

```

*Fig. 6.3.* Program demonstrating the creation of a sprite, X,Y positioning, colour selection, sprite priority and expansion in X and Y directions.

been read in and set up in block 13, and it will be used for four different sprites. In fact all eight sprite pointers are set to 13 in line 130.

### Turning sprites on and off

Most of the operations for controlling sprites are governed by a bank of 34 registers in the VIC11 chip. In fact most of the registers in this chip are involved in the control of sprites. The sprite control registers and their functions are shown in Fig. 6.4.

Even when a sprite pattern definition has been set up in the memory, and the sprite pointer has been set up to tell the VIC11 chip where the dot pattern data is, the sprite will not appear on the screen. This is because the sprite needs to be 'turned on', and this action is controlled by register 21 in the video chip, which is at memory address 53269.

Register 21 actually controls all eight sprites, with one data bit being allocated to each sprite as shown in Fig. 6.5. Setting the appropriate bit at '1' will cause the associated sprite to be switched on, and it will appear on the screen. Resetting the bit to '0' turns the sprite off again. Suppose we want to turn on sprite No. 1, which is controlled by bit 1 of the register. This bit has a numerical value of 2, so we simply have to POKE a 2 into location 53269. We have to be careful here since some of the other bits may already be set up for the

Register	Address	Function
SPRITE X,Y POSITION		
0	53248	Sprite 0 X position
1	53249	Sprite 0 Y position
2	53250	Sprite 1 X position
3	53251	Sprite 1 Y position
4	53252	Sprite 2 X position
5	53253	Sprite 2 Y position
6	53254	Sprite 3 X position
7	53255	Sprite 3 Y position
8	53256	Sprite 4 X position
9	53257	Sprite 4 Y position
10	53258	Sprite 5 X position
11	53259	Sprite 5 Y position
12	53260	Sprite 6 X position
13	53261	Sprite 6 Y position
14	53262	Sprite 7 X position
15	53263	Sprite 7 Y position
SPRITE CONTROL REGS. (1 bit per sprite see Fig 6.5 for layout)		
16	53264	X position MS bit
21	53269	Enable sprite
23	53271	Y (height) expansion
27	53275	Sprite-data priority
28	53276	Set multicolour mode
29	53277	X (width) expansion
30	53278	Sprite-sprite contact
31	53279	Sprite-data contact
SPRITE COLOUR CONTROL		
37	53285	Multicolour No. 0
38	53286	Multicolour No. 1
39	53287	Sprite 0 colour
40	53288	Sprite 1 colour
41	53289	Sprite 2 colour
42	53290	Sprite 3 colour
43	53291	Sprite 4 colour
44	53292	Sprite 5 colour
45	53293	Sprite 6 colour
46	53294	Sprite 7 colour

Fig. 6.4. The sprite registers of the VIC11 chip, their functions and memory locations.

Bit Value	128	64	32	16	8	4	2	1
	Sprite 7	Sprite 6	Sprite 5	Sprite 4	Sprite 3	Sprite 2	Sprite 1	Sprite 0
Bit No.	7	6	5	4	3	2	1	0

Fig. 6.5. Allocation of the sprite control bits in registers 16, 21, 23, 27, 28, 29, 30 and 31 of the VIC11 chip.

other sprites, so we want to leave those as they are. The required POKE command therefore includes a PEEK at the existing state of the register; then the bit that is to be set is ORed with the existing data, which is then POKEd back into the register as follows:

```
400 POKE53269,PEEK(53269)OR2
```

In fact there is no need to work out the numerical value for the bit: it is simply 2 raised to the power N, where N is the sprite number from 0 to 7. Thus the POKE command becomes

```
400 POKE53269,PEEK(53269)OR(2↑SN)
```

where SN is the sprite number. If we want to turn off the control bit for a sprite then we need to use an AND expression which assumes all bits are set at '1' except the one we want to set at '0'. This is done by subtracting 2↑SN from 255 as follows:

```
400 POKE53269,PEEK(53269)AND(255-2↑SN)
```

You can of course turn two or more sprites on or off at a time: just use the appropriate pattern of bits to make up the number with which the current contents of the register are ORed and ANDed.

### Sprite positioning

Having turned on a sprite, we want to be able to place it in some desired position on screen. Unlike the text symbols we do not POKE the sprite to the screen. The sprite position is actually controlled by a bank of registers in the video chip. These are registers 0 to 16, which are at addresses 53248 to 53264.

For a sprite, the X position across the screen can range from 0 to 511 and the Y position from 0 to 255. The Y direction is quite



straightforward, since a single byte can represent the numbers from 0 to 255. For the X direction, however, we need 9 data bits to define the X location. This is dealt with by using two data words, one with all eight bits used and the second with just the most significant (9th) bit.

Each sprite is allocated a pair of registers for its Y location and the lower 8 bits of its X location. The most significant X bit for all eight sprites is placed in register number 16 (53264) where one data bit is allocated to each sprite as shown in Fig. 6.5. Positioning a sprite on the screen involves POKeIng the required X and Y address data into the corresponding VICII sprite position registers.

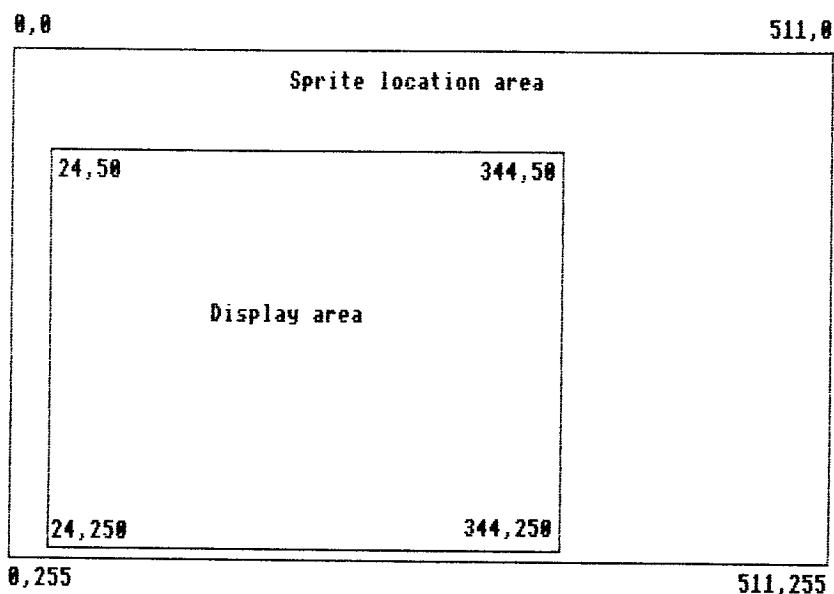


Fig. 6.6. Relationship between the sprite X,Y grid and the display screen area.

The position of the sprite on the screen is always measured relative to the top left corner dot in the sprite matrix. When a sprite is positioned at X,Y=0,0 it will actually be off the display area of the screen. This is shown in Fig. 6.6, where the display screen area is shown relative to the sprite position grid. To place a sprite at the top left corner of the screen, its X and Y position will need to be X=24 and Y=50. When X is increased to 344, the sprite will be just off the right edge of the screen.

To set up X and Y for any sprite is fairly straightforward. Suppose we set a variable VC=53248, which is the start of the video chip

addresses in memory. The X and Y registers are in pairs, so that for sprite 0 they are at VC (X) and VC+1 (Y), for sprite 1 they become VC+2 (X) and VC+3 (Y), and so on. To select the registers, therefore, the X register will be

$$VC + 2*SN$$

and the Y register will be

$$VC + 2*SN + 1$$

For convenience, however, the register addresses are listed in Fig. 6.4. The register containing the most significant bits for the X values of all eight sprites is always 53264. Here you will need to carry out an OR or an AND operation, in the same way as we did for the register which turns sprites on and off.

To deal with the X addressing we can use the following arrangement:

```

600 XM=255-2*SN
610 POKEVC+16,(PEEK(V C+16)AND XM
620 IF X<256 THEN 650
630 X=X-256
640 POKE VC+16,PEEK(V C+16)OR 2*SN
650 POKE VC+2*SN,X
660 POKE VC+2*SB+1,Y
    
```

Here the most significant X bit for the sprite we want to position is initially set to '0' in lines 600 and 610. This is done by setting up a variable XM, where all bits are '1' except the bit we are using. This bit has a value  $2^{SN}$  where SN is the sprite number. By subtracting this value from 255 (all bits set to 1) we get the required value for XM. Variable XM is then ANDed with the current contents of register 16 (line 610) to reset the bit for sprite SN without affecting those of the other sprites.

In line 620 X is tested, and if it is lower than 256 the program goes to line 650, and the X and Y numbers are simply stored into the sprite position registers. If X is greater than 255 then a '1' is POKED into the most significant X position and 256 is subtracted from X before the X and Y values are written to the position registers. The required registers are selected using the sprite number SN to calculate the offset from the start of the video chip addresses (VC).

Remember, however, that the sprite position is still measured from the top left corner of the sprite dot matrix. If you have produced a small sprite pattern at the centre of the dot matrix then

you will need to allow for the offset within the dot matrix if you want to position, say, the centre of your sprite image. This might be dealt with by subtracting 12 from X and 10 from Y before positioning the sprite. This will place the centre of the sprite at the required X,Y position on the screen.

In the program of Fig. 6.3 the four sprites are set up at four different positions on the screen in lines 190, 220, 250 and 290.

### **Colouring the sprite**

So far we have told the video chip what pattern of dots to use for the sprite, and where to place it on the screen; but we also need to tell it what colour the sprite should be.

Like the text and graphics symbols a sprite may be displayed in any one of sixteen colours, but these are controlled by data in a series of registers in the video chip rather than by the colour memory.

The registers that control sprite colour are numbers 39 to 46 in the VIC11 chip, and they have addresses 53287 to 53294. One register is allocated to each sprite as shown in Fig. 6.4. As in the case of the colour memory, these registers use only the lower four bits to give the sixteen possible colours. The colour numbers are exactly the same as those POKEd into the colour memory when we set up character colours. If you PEEK one of the sprite colour registers, however, you will actually get a number between 240 and 255, because the upper four bits of the data are all set permanently at '1'. This can be overcome by ANDing the result of the PEEK with 15 to give the actual colour number from 0 to 15.

The dots in the sprite pattern that are set at '0' (off) will act as if they were transparent, and in their place will be displayed whatever colour is already set up at that spot on the screen. If the sprite is over an area of background, the dots are in background colour. If the sprite is over some graphics symbols, then parts of the symbols will show through the sprite as if it had holes in it.

Setting up the colour for a sprite is quite straightforward, since all we have to do is POKE the required colour number into the colour register for that particular sprite. So if we want sprite number 1 to be green, we would need to POKE the number 5 (green) into register 39 of the VIC11 chip, which is at memory location 53288, as follows:

```
550 POKE 53288,5
```

In the program of Fig. 6.3 the sprites are set to different colours, and this is done in lines 180, 210, 240 and 290.

### Multicoloured sprites

In the normal mode, all the lit dots in the sprite pattern will have the colour specified by that sprite's colour register. As with the graphics symbols, we can select a multicolour mode which allows us to specify any one of four different colours at different parts of the sprite. In a multicolour sprite the dots are dealt with in pairs, so that effectively the sprite becomes only 12 dots wide; each pair of dots across the sprite will have the same colour. The actual colour is determined by the combination of the pair of bits.

If the pair of bits are both '0' then the result is a transparent colour, exactly as for a normal sprite. When the combination is '10', with the first bit of the pair at '1', the normal sprite colour applies to both dots. Note that this is different from the arrangement for multicolour characters, where a '11' combination produces the normal symbol colour.

The other two combinations of dot states in the pair of dots, '01' and '11', produce two new colours which are called Sprite Multicolour #1 and #2. These two colours are defined by numbers in registers 37 and 38 of the VIC11 chip, at addresses 53286 and 53287

Bit pair in dot pattern	Colour	Register address
00	Transparent	
01	Multicolour No.0	53285
10	Sprite colour	53287-94
11	Multicolour No.1	53286

Note. This mode is selected by setting  
sprite bit in register 53276.

Fig. 6.7. Bit patterns and associated colour selection for multicolour mode sprites.

respectively. Once again, we can choose any of the sixteen colours for these two registers. An important difference here is that these two colours will apply to all sprites which have been set in the multicolour mode. The four-bit combinations and the colours they produce in a multicolour sprite are shown in Fig. 6.7.

When we select the multicolour mode with text or graphics characters then all the characters on the screen are displayed in the multicolour mode. With the sprites we have much more flexibility, since each individual sprite may be set for either normal or multicolour display mode at will. This is controlled by register 28 of the VIC11 chip, at address 53276. In this register one data bit is allocated to each sprite, starting with sprite 0 at the right-hand end and working up to sprite 7 at the left-hand (most significant) end. This works in the same way as the register that turns the sprites on and off. When a data bit is at '0' the corresponding sprite will be in normal mode, and if the bit is set at '1' the sprite switches to multicolour mode.

### **Stretching a sprite**

One rather neat facility provided by the VIC11 sprite graphics is that we can easily double the width or height of a sprite. This is dealt with by two registers, one controlling the expansion in the X direction and the other controlling Y expansion. They are register 29 (53277) for X, and 23 (53271) for Y expansion. As with the multicolour mode, each bit in these registers controls one sprite, and if it is at '0' the sprite is displayed at the normal size. Setting the bit to '1' will cause the sprite to double in size in the X or Y direction. If the expansion bit for a sprite is set at '1' in both X and Y registers, the sprite expands to double size in both directions. Using this facility we can stretch sprites horizontally or vertically or make them twice as big all round.

In the program of Fig. 6.3 sprite 1 has been set up with Y expansion and sprite 2 has X expansion. Sprite 3 has both X and Y expansion bits set, and is therefore displayed twice full size.

### **Sprite priority**

With graphics symbols, if we place one symbol on top of another it simply replaces that symbol on the screen. The sprites work in a

different way. Each sprite is given a priority, with the highest level going to sprite 0 and the level decreasing in sequence down to the lowest, which is sprite 7. If two sprites overlap on the screen, the sprite with the higher priority will overlay the other sprite. However, where the dots of the higher priority sprite are set at '0' the lower priority sprite will show through. Normally sprites have priority over text or bit map data, but register 27 may be used to change this for individual sprites. If the sprite bit in this register is set at '1' then the sprite will appear to be behind the text or bit map data. Note that all sprites have priority over the background colour. Sprites are, however, masked by the screen edge, and will disappear as they move into the border region.

If you want a sprite to appear in front of another sprite, you must give it a higher priority by arranging that its sprite number is *lower* than that of the other sprite. Conversely, if you want the sprite to appear *behind* another then it should be given a *higher* sprite number. This can be seen in the display produced by the program listed in Fig. 6.3, where the double size sprite 3 is partially masked by sprite 0, and therefore appears to be behind sprite 0.

The use of sprite priority becomes important when several sprites are being animated, as in a game, and where some sprites must appear to pass in front of or behind others on the screen.

## Animation using sprites

Animating an object such as a ball, rocket, bomb or racing car follows the same basic principles when using sprites as were involved when animating graphics symbols. With sprites we can position the sprite by using X, Y co-ordinates, and the directions produced will be the same as for a symbol with R, C co-ordinates; R becomes Y, and C becomes X. In the program listed in Fig. 6.8 two invader figures move from opposite sides of the screen to meet in the middle. Here the X value for the right-hand sprite starts at 255, which is about two-thirds of the way across the screen from the left.

Movement with sprites can be very much smoother than with symbols because X and Y can change by just one dot position at a time. In the case of the X co-ordinates, which can vary from 0 to 511, account has to be taken of the most significant X bit when POKING the X values into the sprite position registers. One way of doing this is to test X to see if it is greater than 255. If not, then the X value is

POKEd into the sprite X register and a '0' is POKEd into the appropriate bit of register 16. If  $X > 255$  then  $X - 256$  is POKEd into the sprite X register, and a '1' is POKEd into the sprite bit of register 16. The program of Fig. 6.8 could be modified so that the right-hand sprite starts off from  $X = 300$ , which will place part of the sprite off the right edge of the screen. In this case the most significant bit of X will have to be taken into account in setting the sprite position.

For more rapid motion the steps of X and Y position may be made greater than 1, but eventually the motion will tend to appear jerky if the steps are made too large.

### Sprite collision detection

We saw in the last chapter that for games and similar applications involving animation there is a need to detect collisions between different objects, or between an object and the screen boundary. This normally involves either PEEKing the screen (to see what is already there before moving the object) or comparing the positions of two objects to see if they are about to reach the same position on the screen.

When we use sprite graphics, however, the detection of collisions between sprites is handled by the VIC11 chip itself. Register 30 is used to note collisions between sprites. Here each sprite is allocated one data bit. If two sprites overlap so that one of the lit dots in one sprite occupies the same position as a lit dot in the other sprite then two bits will be set to '1' in the collision register. These will be the bits corresponding to the two sprites that have overlapped.

Reading the collision register by PEEKing its contents will reset all the bits to '0' again. Thus after the register has been read, the information regarding collisions will be lost unless you immediately save it as another variable by using a statement such as

```
5000 CD = PEEK (53278)
```

Now it is possible to carry out a series of tests on the collision state CD to determine which sprites collided and to decide what action to take.

Suppose we want to check if sprite 1 has collided with another sprite. We can AND CD with the bit value for sprite 1, which is 2. This effectively blanks off all the other bits. Then we can check to see if the result is 2, which would indicate that the sprite 1 collision bit is set. This gives a statement as follows:

510 IF (CD AND 2) = 2 THEN 'Action'

where 'Action' is whatever you want to do when a collision occurs.

There may, of course, be collisions between the sprite and symbols on the text screen. For instance, the symbols on the text screen might represent, say, the sides of a road, and the sprite is your racing car. Here the sprite-to-sprite collision register will not be affected, but the VIC11 chip caters for this situation by having another register (number 31) which detects collisions between sprites and either text or bit map graphics data. Thus if sprite 1 passes over a graphics symbol on the text screen, its bit will be set in register 31 of the VIC11 chip.

Once again reading the sprite-to-data collision register will reset all the bits to '0', so you need to save the contents as a variable so you can process the collision bits to see which sprite or sprites was involved in the collision.

For a bouncing ball using sprites the detection of the screen edge is best done by using simple comparison of the sprite X and Y registers with the edges of the screen, based on the sprite positioning co-ordinates. Remember that the sprite X,Y grid extends outside the screen area, and you must also allow for the width and height of the sprite; its position co-ordinates refer to its *top left corner*.

### Animation involving shape changes

For many animated displays we shall want not only to move the sprite around the screen, but to have it change shape as it moves. A simple example of this is the invader figure from a typical computer game. As the invader moves across the screen, his legs move as well. This is a very simple movement: in one position the legs are spread apart, and in the next the feet are tucked in and the knees bent.

Figure 6.8 shows a listing of a program which demonstrates this type of action. Here three different sprite patterns have been set up in the memory at blocks 13, 14 and 15. Two of these are for the invader figure; one has the legs spread apart, and the other has them bent.

Four different sprites are defined for the invader figures. Two of these (sprites 0 and 1) are for the invader at the left of the screen, and sprites 2 and 3 are used for the invader at the right of the screen. The left-hand sprites are set to cyan colour, and those for the right-hand invader to the purple colour. Sprites 0 and 2 have the dot pattern with the legs apart, while sprites 1 and 3 have the legs bent.



```
100 REM SPRITE MOVEMENT AND COLLISIONS
105 REM CLEAR SCREEN
110 PRINT CHR$(147)
115 REM VIC CHIP START ADDRESS
120 VC=53248
125 REM SET SPRITE POINTERS
130 POKE2040,13:POKE2041,14
140 POKE2042,13:POKE2043,14
150 POKE2044,15
155 READ SPRITE DATA
160 FOR N=0 TO 191
170 READ A:POKE832+N,A
180 NEXT
185 REM SET Y POSITIONS
190 POKE VC+1,150:POKEVC+3,150
200 POKE VC+5,150:POKEVC+7,150
205 REM SET SPRITE COLOURS
210 POKEVC+39,4:POKEVC+40,4
220 POKEVC+41,3:POKEVC+42,3
230 POKEVC+43,7:POKEVC+21,0
240 FOR N=1 TO 10
250 FOR X=0 TO 130 STEP 8
255 REM MOVE SPRITES 0 AND 2
260 POKEVC,X:POKEVC+4,255-X
265 REM CHECK FOR COLLISION
270 CI=PEEK(VC+30)
280 IF CI<>0 THEN 370
285 REM DISPLAY SPRITES 0 AND 2
290 POKEVC+21,5
300 FOR T=1 TO 200:NEXT
305 REM MOVE SPRITES 1 AND 3
310 POKEVC+2,X+4:POKEVC+6,251-X
315 REM CHECK FOR COLLISION
320 CI=PEEK(VC+30)
330 IF CI<>0 THEN 370
335 REM DISPLAY SPRITES 1 AND 3
340 POKEVC+21,10
350 FOR T=1 TO 200:NEXT
360 NEXT X
365 REM DISPLAY EXPLOSION SPRITE 4
370 POKEVC+8,X:POKEVC+9,130
380 POKEVC+23,16:POKEVC+29,16
390 POKEVC+21,16
400 FOR T=1 TO 1000:NEXT
410 POKEVC+21,0:CI=PEEK(VC+30)
```

```

420 FOR T=1TO500:NEXT
430 NEXT N
440 END

795 REM SPRITE 0 AND 2 DATA
800 DATA 0,102,0,1,255,128
810 DATA 15,255,240,31,255,248
820 DATA 63,255,252,63,60,252
830 DATA 63,60,252,63,255,252
840 DATA 31,255,248,15,255,240
850 DATA 3,0,192,6,0,96
860 DATA 12,0,48,24,0,24
870 DATA 48,0,12,96,0,6
880 DATA 192,0,3,0,0,0
890 DATA 0,0,0,0,0,0
900 DATA 0,0,0,0

905 REM SPRITE 1 AND 3 DATA
910 DATA 0,102,0,1,255,128
920 DATA 15,255,240,31,255,248
930 DATA 63,255,252,63,60,252
940 DATA 63,60,252,63,255,252
950 DATA 31,255,248,15,255,240
960 DATA 3,0,192,6,0,96
970 DATA 12,0,48,24,0,24
980 DATA 12,0,48,6,0,96
990 DATA 3,0,192,0,0,0
1000 DATA 0,0,0,0,0,0
1010 DATA 0,0,0,0

1015 REM SPRITE 4 DATA
1020 DATA 96,24,12,48,24,24
1030 DATA 24,24,48,12,24,96
1040 DATA 6,24,192,3,25,128
1050 DATA 1,155,0,0,255,0
1060 DATA 0,102,0,255,231,255
1070 DATA 0,102,0,0,255,0
1080 DATA 1,155,0,3,25,128
1090 DATA 6,24,192,12,24,96
1100 DATA 24,24,48,48,24,24
1110 DATA 96,24,12,0,0,0
1120 DATA 0,0,0,0

```

*Fig. 6.8* Program to demonstrate sprite animation with shape changes and collision detection.

The process of animation consists of alternately switching on sprites 0 and 2 or sprites 1 and 3 while making an X movement between each step. As each invader moves across the screen his legs bend and straighten.

The collision register is also checked at each step, and when a collision is detected the program jumps out of the animation loop. At this point the invader sprites are turned off, and an explosion sprite displayed in double size is superimposed where the invaders were. In a game this would, of course, be accompanied by explosion sounds.

In this program five sprites were used to show how several sprites might be handled. But the shape change animation can be achieved using just two sprites for the invaders. These have to be separate sprites because they are at different points on the screen. To carry out the shape change, however, we need only alter the sprite pointers so that they point to the second dot pattern. Thus (on one step) memory locations 2040 and 2041 will both contain the number 13, and on the next step they will be changed to point to the dot pattern at block 14. Since the VIC11 chip now reads the alternate dot pattern to display the sprite, the shape change is instantaneous. After the collision, sprite 1 can be turned off and the pointer for sprite 0 in location 2040 is changed to 15 to point to the explosion effect. Now, of course, the scale expansion must be applied to sprite 0 to produce the required explosion effect.

For more complex animation, such as a little man walking across the screen, four or five different sprite shapes might be used. Firstly the action of taking a step is broken down into four or five stages, and a 'snapshot drawing' is made of the figure at each stage through the action of taking the step. These pictures of the man are then set up as separate sprite dot patterns. In the animation sequence the series of dot patterns is used in sequence, and the sprite is moved in the X direction a small amount each time. At the end of the step the whole process is repeated for the next step and so on. This can take some effort; for realistic results the action of the figure has got to be just right, and include movement of the body and arms as well as leg movement.

## Chapter Seven

# Graphs and Charts

Any computer can readily carry out lots of calculations or measurements and end up by printing out or displaying enormous arrays of numbers. This presents problems for the average computer user, who has to make some sense of this mass of output data. One technique for dealing with a mass of numbers is to present them as a list or table. Unfortunately this may not always be particularly helpful when we come to interpret the results.

When examining a list or table of results the computer user is generally more interested in the way the results are changing than in the precise numbers. Often we shall be interested in seeing the *trend* of changes in the results, since this may enable us to make a rough prediction of the way things will change in the immediate future. In a set of production results from a factory it is more interesting to see whether output is rising or falling than to consider the exact output figures for each week.

When producing displays such as gauges or meters we could, of course, simply print up the numerical value of the quantity being measured. An example of this form of readout is a digital watch or clock. Although the digital clock provides a very accurate readout, the old familiar clock face with two or three hands provides a readout which is much easier to use for telling the time when we are not interested in time to the nearest second. Here the position of the hands, even if we cannot read the figures on the dial, gives us a reasonably accurate sense of time at a glance; most people have to think for a moment when reading a digital display to convert the reading into something they understand. We tend to think of time in terms of 'a quarter to three' rather than 14.45.

When we come to displaying computer results it is often better to use a graphics display or a chart than a table of figures. Such a graph or chart usually shows each result either as a line of varying length or

perhaps as a dot whose height above some reference line is proportional to the quantity being displayed.

### **A simple meter display**

When the computer is used to monitor or simulate some technical activity we shall often need to display gauges or meters which will show readings of, for instance, temperature, pressure, voltage and so on.

With the Commodore 64 it is fairly easy to create a meter display where a pointer moves along a calibrated scale in sympathy with changes in the quantity being measured. As with a clock, we can judge the state of a reading merely by looking at the pointer's position along this scale.

Figure 7.1 lists a program for drawing a simple meter display. Here the scale is horizontal, which makes the display easy to produce. The program selects random voltage readings from 0 to 100 volts and places the pointer at the nearest 5 V point on the scale. Three subroutines are used to perform the various operations required.

Line 110 clears the screen using CHR\$(147), which is equivalent to CLR/HOME. The program then jumps to the subroutine at 200 which draws the scale. In this routine the first step is to position the cursor, and this is achieved by the subroutine starting at line 300. This uses a machine code routine in the Commodore 64 ROM and is described in Chapter 2.

With the cursor positioned at the left end of the scale a simple PRINT is used to draw the scale. The PRINT string in line 210 uses the symbols produced by the keys [SHIFT O], [LOGO G] and [SHIFT P]. The cursor is then moved down one row, and the scale calibration is printed. Finally, on the next line down, the legend 'VOLTS' is printed.

An integer value of voltage V with 5 V steps is generated in line 130 and then in line 140 a subroutine is called to draw the pointer. The first step is to erase the previous pointer position at C1. On the first reading there is no pointer to erase but C1 is set at 10, the zero point of the scale.

The pointer can move only one character space at a time, and from our scale each character space represents 5 volts. A value V1 is calculated, which is the integer of  $V/5$ , and this gives the number of spaces to the right of zero where the pointer should be. The column position C is calculated by adding 10 (the column for 0V) to V1, and

```

100 REM SIMPLE METER DISPLAY
105 REM CLEAR SCREEN AND DRAW SCALE
110 PRINT CHR$(147):GOSUB200:C1=10
120 FOR N=1TO50
125 REM SELECT VOLTAGE READING
130 V=5*INT(100*RNND(0)/5)
135 REM PRODUCE METER DISPLAY
140 GOSUB400
145 REM TIME DELAY FOR VIEWING
150 FOR D=1TO1000:NEXT:NEXT
160 END
195 REM DRAW SCALE
200 R=14:C=10:GOSUB300
210 PRINT"┌───┬───┬───┬───┬───┐"
220 R=15:C=10:GOSUB300
230 PRINT"0   25   50   75   100"
240 R=16:C=10:GOSUB300
250 PRINT"          VOLTS"
260 RETURN
295 REM POSITION CURSOR TO R,C
300 POKE780,0:POKE781,R:POKE782,C
310 SYS65520:RETURN
395 REM DRAW POINTER SUBROUTINE
400 R=12:C=C1:GOSUB300
405 REM ERASE LAST READING
410 PRINT" "
420 V1=INT(V/5)
425 REM SET NEW POINTER POSITION
430 C1=10+V1
440 R=12:C=C1:GOSUB300
445 REM DRAW POINTER
450 PRINT"┌"
460 R=20:C=15:GOSUB300
470 PRINT" "
480 R=20:C=15:GOSUB300
485 REM PRINT VOLTAGE READING
490 PRINT"V = ";STR$(V)
500 RETURN

```

Fig. 7.1. Program to produce horizontal meter display.

the cursor is then placed in position and the pointer printed. To avoid ending up with a series of pointers along the scale the previous pointer position is erased by printing a space over it in line 410. The pointer itself uses the graphics symbol produced by [LOGO G].

As a guide, the value of V is printed out below the gauge and a time delay is built into the main program loop to allow the scale to be read for each new voltage.

### **Vertical meter display**

It is common to find gauges and meters where the scale is vertical instead of horizontal. Figure 7.2 shows a program to produce such a display. The technique used is similar to but a little more complicated than that used for the horizontal meter.

The scale is drawn using two loops, one giving the four main sections of the scale and the other dealing with individual steps in each section. The  $\emptyset$  graduation is drawn first, using [LOGO X]. Then four [SHIFT B] symbols are printed, one above the other. Here R is decremented and the cursor repositioned after each PRINT. Next the scale calibration and graduation mark are drawn. The mark is [LOGO W]. On the final step, the [LOGO W] symbol is overprinted with a [LOGO S] symbol to give the proper termination at the top of the scale.

The pointer is positioned by moving it up by V1 spaces from the zero position by resetting the cursor row position in line 43 $\emptyset$ . In this program each symbol space represents 2V, but a refinement allows the cursor to be positioned in 1V steps. In line 45 $\emptyset$  V1 is checked to see if it is odd or even. If V1 is even a line is drawn across the middle of the space using a [SHIFT \*] symbol, but if V1 is odd the line is placed at the top of the character space by printing a [LOGO T] symbol.

This technique of providing half-division steps by choosing a different symbol for the pointer can also be applied to a horizontal meter display. The idea could be extended further to give four or even eight possible positions of the pointer in the symbol space by adding further tests and choosing appropriate symbols. In this program the pointer is displayed in yellow, with the colour being switched on and off in lines 44 $\emptyset$  and 48 $\emptyset$ . Lines 49 $\emptyset$  and 50 $\emptyset$  use a [HOME] command at the start of the text string to move the cursor to the top left of the screen.

### **Thermometer display**

Another popular style of gauge uses a variable length strip to

```

100 REM VERTICAL METER DISPLAY
105 REM CLEAR SCREEN AND DRAW SCALE
110 PRINT CHR$(147):GOSUB200
120 R1=22
130 FOR V=0T040
135 REM DRAW POINTER
140 GOSUB400
145 REM TIME DELAY FOR VIEWING
150 FOR D=1T01000:NEXT:NEXT
160 END
195 REM DRAW SCALE
200 R=22:C=14:GOSUB300
210 PRINT " 0 4";
220 FOR I=1 TO 4
230 FOR J=1 TO 4:R=R-1:GOSUB300
240 PRINT"    I":NEXT
250 R=R-1:GOSUB300
260 PRINT STR$(10*I);" 4":NEXT
270 C=C+5:GOSUB300:PRINT"  ";
280 R=12:C=8:GOSUB300:PRINT"VOLTS";
290 RETURN
295 REM POSITION CURSOR TO R,C
300 POKE780,0:POKE781,R:POKE782,C
310 SYS65520:RETURN
395 REM DRAW POINTER SUBROUTINE
400 R=R1:C=20:GOSUB300
405 REM ERASE LAST READING
410 PRINT" ";
420 V1=INT(V/2)
430 R=22-V1:GOSUB300:R1=R
440 PRINT CHR$(158);
450 IF V/2=INT(V/2) THEN 470
460 PRINT"~":GOTO480
470 PRINT"~";
480 PRINT CHR$(154);
485 REM PRINT VOLTAGE AT TOP OF SCREEN
490 PRINT"§
500 PRINT"§ V = ";STR$(V);
510 RETURN
520 RETURN

```

*Fig. 7.2.* Program to produce vertical meter display.

indicate the variable being measured. A familiar example is the everyday mercury thermometer. In the thermometer, the length of "



the mercury column is directly proportional to the temperature being measured. We can represent the mercury column on the Commodore 64 by drawing a simple horizontal bar whose length is also proportional to the measurement it represents.

To make sense of a thermometer reading we need some sort of scale. On a real thermometer the scale may be a set of graduation marks etched into the glass of the thermometer tube, or the marks may be made on the frame alongside the glass tube. On the computer display it is more convenient to draw the scale alongside the measurement column, and we can draw this in much the same way as for a meter type display.

The horizontal bar type display is easiest to produce, since the bar itself is just a series of space symbols displayed in reverse video. The program listed in Fig. 7.3 produces a display of this type. In this program the bar drawing step is preceded by an erasure of the previous reading. This is done by printing spaces over the whole length of the bar. The reading is also checked for  $T1=0$ . If this is detected, a single line at the left of the scale is drawn to indicate a zero reading.

The scaling is arranged so that each character space represents 5 degrees F. To find the number of character blocks required for the bar,  $T1$  is calculated as the integer of  $T/5$ . More precision can be achieved by the technique used in the last program, which checks for a half symbol space step at the end of the bar. If a half step is required, then a symbol with the left half lit is added to the end of the bar.

### **Vertical bar type displays**

Normally we expect to see thermometers, and many other gauges of the bar type, mounted with the bar running vertically. This form of bar display can readily be produced on the Commodore 64, but involves a slightly more complex routine for producing the bar and the scale. A program for drawing this type of display is shown in Fig. 7.4.

The main difference here is that after each symbol making up the bar has been printed on the screen we need to move the cursor up one space and then left one space to place it in the correct position for printing the next section of the bar. This is done in lines 410 and 460 where the space is followed by [CRSR U] and [CRSR L].

The scale drawing step is similar to that used for the vertical

```

100 REM HORIZONTAL BAR GAUGE
105 REM CLEAR SCREEN AND DRAW SCALE
110 PRINT CHR$(147):GOSUB200
120 FOR N=1TO50
125 REM SELECT TEMPERATURE
130 T=5*INT(100*RND(0)/5)
135 REM PRODUCE BAR DISPLAY
140 GOSUB 400
145 REM TIME DELAY FOR VIEWING
150 FOR D=1TO1000:NEXT:NEXT
160 END

195 REM DRAW SCALE
200 R=14:C=10:GOSUB300
210 PRINT CHR$(158);
220 PRINT" |-----| "
230 R=15:C=10:GOSUB300
240 PRINT"0   25   50   75   100"
250 R=16:C=10:GOSUB300
260 PRINT"         DEG F"
270 PRINT CHR$(154);
280 RETURN

295 REM POSITION CURSOR TO R,C
300 POKE780,0:POKE781,R:POKE782,C
310 SYS65520:RETURN

395 REM DRAW BAR SUBROUTINE
400 R=12:C=10:GOSUB300
405 REM ERASE LAST READING
410 FOR K=1TO20:PRINT" ";:NEXT
420 PRINT CHR$(156);
430 T1=INT(T/5)
440 R=12:C=10:GOSUB300
450 IF T1=0 THEN PRINT" | ";:GOTO500
460 PRINT CHR$(18);
470 FOR K=1 TO T1
480 PRINT" ";
490 NEXT
500 PRINT CHR$(146);
505 REM PRINT TEMPERATURE READING
510 R=20:C=15:GOSUB300
520 PRINT"         "
530 R=20:C=15:GOSUB300
540 PRINT CHR$(154);
550 PRINT"T = ";STR$(T);
560 RETURN

```

Fig. 7.3. Program to produce horizontal moving bar display.

```

100 REM VERTICAL BAR GAUGE
105 REM CLEAR SCREEN AND DRAW SCALE
110 PRINT CHR$(147):GOSUB200
120 FOR V=0TO40
125 REM DRAW POINTER
130 GOSUB400
135 REM TIME DELAY FOR VIEWING
140 FOR D=1TO1000:NEXT:NEXT
150 END
195 REM DRAW SCALE
200 R=22:C=14:GOSUB300
210 PRINT " 0  |";
220 FOR I=1 TO 4
230 FOR J=1 TO 4:R=R-1:GOSUB300
240 PRINT "    |";NEXT
250 R=R-1:GOSUB300
260 PRINT STR$(10*I);"  |";NEXT
270 C=C+5:GOSUB300:PRINT" _ ";
280 R=12:C=8:GOSUB300:PRINT"VOLTS";
290 RETURN
295 REM POSITION CURSOR TO R,C
300 POKE780,0:POKE781,R:POKE782,C
310 SYS65520:RETURN
395 REM DRAW POINTER SUBROUTINE
400 R=22:C=21:GOSUB300
405 REM ERASE LAST READING
410 FOR I=1TO20:PRINT"  |";NEXT
420 V1=INT(V/2)
430 GOSUB300
440 PRINT CHR$(18):CHR$(156);
450 IF V1=0 THEN 470
460 FOR I=1 TO V1:PRINT"  |";NEXT
470 PRINT CHR$(146);
480 IF V/2=INT(V/2) THEN 500
490 PRINT" _ ";
500 PRINT CHR$(154);
505 REM PRINT VOLTAGE AT TOP OF SCREEN
510 PRINT"⌘      "
520 PRINT"⌘ V = ";STR$(V);
530 RETURN

```

*Fig. 7.4. Program to produce vertical thermometer display.*

meter, but different symbols are used so the graduation steps occur at the bottom of character spaces rather than in the middle. This fits

in more conveniently with drawing the bar, since the latter is built up from reverse video character spaces.

In this program a check is made for half steps, and a half-filled symbol is added to the top of the bar as required in lines 480 and 490. This could be extended to provide quarter or eighth size steps if desired, by adding further tests and printing different symbols at the top of the bar.

## Bar charts

While the 'thermometer' display is useful to show the current state of some measurement, a more useful arrangement is to show how the situation has varied over a period of time. We could perhaps measure the temperature at noon on each day of the week. A display showing this information can easily be arranged by drawing the thermometer displays for the days of the week alongside one another. For this display only the variable length bar is drawn for each day, and a single scale is included at the left-hand side. To improve visibility, the bars may be drawn with a gap between adjacent bars. This type of display is sometimes referred to as a *histogram* but is more commonly called a *bar chart*.

Bar charts are not normally intended to give particularly accurate displays: their main application is to show the general trend of the variable being displayed. They are frequently used in business to show the trend in sales over a year, or perhaps the stock level, number of orders or income over a period. It is very easy to see the trend of the results on such a chart.

A useful enhancement of the bar chart is to arrange for the colour of the bar to be changed if its level goes above, or perhaps below, some predetermined limit. This can provide an easily-recognised warning that a situation is becoming dangerous or needs attention. In such cases either the whole bar changes colour or the part above the limit line might change colour.

The Commodore 64 character graphics can be used to draw a bar chart. Although the resolution is relatively coarse, the resultant display can be quite effective for this type of chart.

## Horizontal bar charts

The easiest type of bar chart for use on the Commodore 64 is one

```

100 REM HORIZONTAL BAR CHART
110 DIM D$(7),T(7)
115 REM READ IN DATA
120 FOR I=1TO7:READ D$(I),T(I):NEXT
130 DATA "SU",25,"MO",46,"TU",60,"WE"
140 DATA 50,"TH",70,"FR",80,"SA",55
145 REM DRAW SCALES
150 PRINT CHR$(147):GOSUB 200
155 REM DRAW BARS
160 FORN=1 TO 7:R=2*N+1:GOSUB400:NEXT
165 REM PRINT LEGEND
170 GOSUB 600
180 END
195 REM DRAW SCALES
200 R=17:C=10:GOSUB350
210 PRINT" | | | | | "
220 R=18:GOSUB350
230 PRINT"0 25 50 75 100"
240 R=19:GOSUB350
250 PRINT"          DEG F"
260 C=9
270 FOR R=16 TO 2 STEP -1
280 GOSUB350:PRINT" |":NEXT
290 C=5:R=0
300 FOR N=1 TO 7:R=2*N+1
310 GOSUB350:PRINT D$(N):NEXT
320 RETURN
345 REM POSITION CURSOR TO R,C
350 POKE780,0:POKE781,R:POKE782,C
360 SYS65520:RETURN
395 REM DRAW BAR SUBROUTINE
400 T1=INT(T(N)/5)
410 PRINT CHR$(156);
420 C=10:GOSUB350
430 IF T1=0 THEN PRINT"| ":GOTO500
440 PRINT CHR$(18);
450 FOR K=1 TO T1:PRINT" ":NEXT
460 PRINT CHR$(146);
470 PRINT CHR$(154);
480 RETURN
595 REM PRINT LEGEND
600 R=21:C=12:GOSUB350
610 PRINT"DAILY TEMPERATURES"
620 RETURN

```

*Fig. 7.5.* Program to draw simple bar chart with horizontal bars.

where the bars are horizontal and run across the screen from left to right.

The bar itself is created by printing reversed space symbols, which cause each symbol space in the bar to be filled with text colour. The length of the bar is selected by the number of space symbols printed. This is basically the same technique as that used for the bar gauge. For slightly better accuracy we could use the half-filled character block as the last symbol in the bar to provide an extra half size step when required.

Figure 7.5 lists a program to draw a bar chart of daily temperatures. A scale is included up the left-hand side which shows the days of the week, and another scale along the bottom shows the temperature reading. In this case the temperature data is read in as an array. The program could be rearranged so that the temperatures are input from the keyboard.

### **Vertical bar charts**

Although the horizontal bar chart is easy to produce on the Commodore 64, the more common form of bar chart has its bars running vertically. Charts of this type can be produced on the Commodore 64, but they require a slightly more complex program.

Figure 7.6 gives a listing for a program to draw a bar chart with vertical bars built up by using graphics symbols. In this program a separate bar is drawn for each day of the week and each bar is drawn using the same technique as for the mercury column in the vertical bar gauge program. The data in this program is read into an array so that the drawing of the bars can use a common drawing loop. It could easily be arranged that the temperature data is typed in from the keyboard by using an INPUT statement instead of READ to set up the temperature values.

The display produced on the screen is similar to Fig. 7.7. By altering the scales and legends this program can readily be adapted to display any desired variable on the chart.

### **Multiple bar charts**

When two different variables are to be displayed on the same chart the bars are drawn in pairs so that they become interleaved. To provide a clearer distinction between the sets of bars a different

```

100 REM VERTICAL BAR CHART
110 DIM D$(7),T(7)
115 REM READ IN DATA
120 FOR I=1TO7:READ D$(I),T(I):NEXT
130 DATA "SU",10,"MO",18,"TU",24,"WE"
140 DATA 20,"TH",16,"FR",24,"SA",12
145 REM CLEAR SCREEN DRAW SCALES
150 PRINT CHR$(147):GOSUB 200
155 REM DRAW BARS
160 FOR N=1TO7:C=3*N+10:GOSUB400:NEXT
165 REM PRINT LEGEND
170 GOSUB 600
180 END
195 REM SCALES SUBROUTINE
200 R=18:C=5:GOSUB350
210 PRINT" 0 J";
220 FOR I=1 TO 3
230 FOR J=1 TO 4:R=R-1:GOSUB350
240 PRINT"    I":NEXT
250 R=R-1:GOSUB350
260 PRINT STR$(10*I);" J":NEXT
270 C=C+5:GOSUB350:PRINT" _ ";
280 R=19:C=11:GOSUB350
290 PRINT"-----"
300 R=20:C=12:GOSUB350
310 FOR I=1TO7:PRINT D$(I);" ";:NEXT
320 RETURN
345 REM POSITION CURSOR TO R,C
350 POKE780,0:POKE781,R:POKE782,C
360 SYS65520:RETURN
395 REM DRAW BAR SUBROUTINE
400 R=18:GOSUB350
410 T1=INT(T(N)/2)
420 IF T1=0 THEN 460
430 PRINT CHR$(18);CHR$(156);
440 FOR I=1 TO T1:PRINT" JI":NEXT
450 PRINT CHR$(146);
460 PRINT CHR$(154);
470 RETURN
595 REM PRINT LEGEND
600 R=11:C=1:GOSUB350
610 PRINT"DEG C";
620 R=22:C=12:GOSUB350
640 PRINT"DAILY TEMPERATURES";
650 RETURN

```

Fig. 7.6. Program to draw a conventional vertical bar chart.

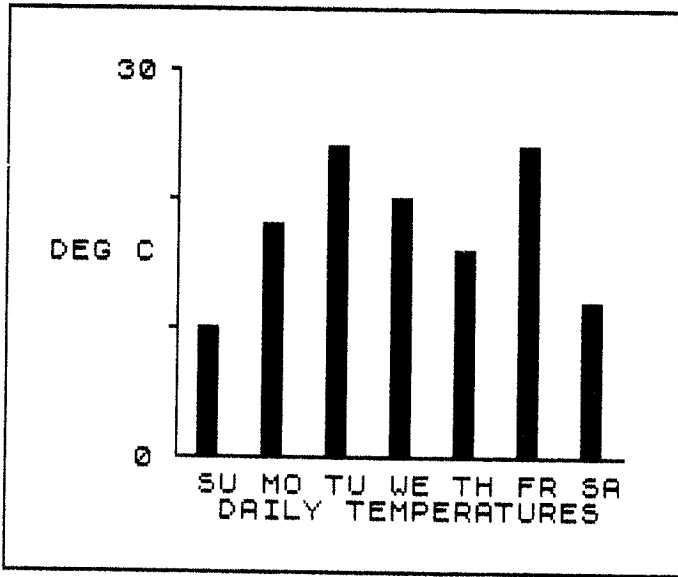


Fig. 7.7. Display produced by program listed in Fig. 7.6.

colour may be used for each set of bars. Three or perhaps four graphs could be interleaved in this way if desired. Some bars could be drawn as open boxes, but with different coloured outlines. A typical application for a multiple bar chart might show the income and expenditure on a single chart. It might be useful to show the predicted income and expenditure as well, to see how the actual values compare with predicted trends.

Multiple bar charts with horizontal bars are quite easily produced using a technique similar to that for a simple horizontal bar chart. The bars for the second quantity being displayed are interleaved between those of the first set of data. Two different colours may be used for the two sets of bars so that they can easily be picked out on the chart.

An example of a multiple bar chart with vertical bars is shown in the program listed in Fig. 7.8. This shows the maximum and minimum temperatures for the seven days of a week. In this case one set of bars is drawn in purple and the others are in the yellow colour.

You could have three or four interleaved sets of bars, each in a different colour. On such a chart a legend may be included to show what each set of bars represents.



```

100 REM MULTIPLE BAR CHART
110 DIM D$(7),TH(7),TL(7)
115 REM READ IN DATA
120 FOR I=1TO7:READ D$(I),TH(I),TL(I):NEXT
130 DATA "SU",14,10,"MO",16,12
140 DATA "TU",24,20,"WE",28,16,"TH"
150 DATA 16,14,"FR",14,10,"SA",12,8
155 REM CLEAR SCREEN DRAW SCALES
160 PRINT CHR$(147):GOSUB 200
165 REM DRAW BARS
170 FOR N=1TO7:GOSUB400:NEXT
175 REM PRINT LEGEND
180 GOSUB 600
190 END
195 REM SCALES SUBROUTINE
200 R=18:C=5:GOSUB350
210 PRINT"  @  J";
220 FOR I=1 TO 3
230 FOR J=1 TO 4:R=R-1:GOSUB350
240 PRINT"      I";:NEXT
250 R=R-1:GOSUB350
260 PRINT STR$(10*I):"  J";:NEXT
270 C=C+5:GOSUB350:PRINT"  _  ";
280 R=19:C=11:GOSUB350
290 PRINT"-----"
300 R=20:C=12:GOSUB350
310 FOR I=1TO7:PRINT D$(I):" ";:NEXT
320 RETURN
345 REM POSITION CURSOR TO R,C
350 POKE780,0:POKE781,R:POKE782,C
360 SYS65520:RETURN
395 REM DRAW BAR SUBROUTINE
400 C=9+3*N:R=18:GOSUB350
410 T1=INT(TL(N)/2)
420 IF T1=0 THEN 460
430 PRINT CHR$(18):CHR$(156);
440 FOR I=1 TO T1:PRINT"  █";:NEXT
450 PRINT CHR$(146);
460 C=C+1:GOSUB350
470 T1=INT(TH(N)/2)
480 IF T1=0 THEN 530
490 PRINT CHR$(18):CHR$(158);
510 FOR I=1 TO T1:PRINT"  █";:NEXT
520 PRINT CHR$(146);
530 PRINT CHR$(154);

```

```

540 RETURN
595 REM PRINT LEGEND
600 R=11:C=1:GOSUB350
610 PRINT"DEG C";
620 R=22:C=12:GOSUB350
640 PRINT"DAILY TEMPERATURES";
650 RETURN

```

*Fig. 7.8.* Program to draw a multiple bar chart.

## Scientific graphs

Although the bar chart is well suited for business use, when we come to scientific or mathematical graph-plotting a slightly different arrangement is used. Here the graph is required to give a more accurate display of results.

The layout is similar to that of a bar chart, with the results of the calculation or experiment plotted vertically on the screen and the measurement steps horizontally. In this case, however, the value of Y is simply shown as a dot at a point equivalent to the top of the bar on a bar chart. Sometimes, to make the point easier to see, a small + sign, triangle or circle may be used as a marker instead.

In a bar chart the variables are normally positive, but in a scientific graph the variables X and Y may be either positive or negative. To cater for this, the X and Y axes are drawn as shown in Fig. 7.10. Positive values of X are drawn to the right of the vertical Y axis, and negative values of X to the left. Similarly, positive values of Y are drawn above the X axis, and negative values below. When there are no negative values for X, the left-hand half of the graph is not drawn, so the Y axis appears at the left side of the diagram. Similarly, if there are no negative values of Y only the upper part of the graph, above the X axis, would be drawn. Sometimes only a quarter of the complete graph need be drawn to display all the points required. The advantage of drawing only part of the complete X, Y axis system is that the required part of the chart can be expanded to fill the screen, thus giving better resolution.

This type of graph can be produced using character graphics, by using, say, an asterisk or diamond symbol to indicate the plotted points; but better results are obtained by using the higher resolution of the bit map graphics mode, so this is the approach we shall adopt.

To see how this type of graph is produced, let us take as an example the equation  $Y = \text{SIN}(X)$  and plot the value of Y for values

of  $X$  ranging from  $-1\theta$  to  $+1\theta$ . The value of  $\text{SIN}(X)$  will always lie between the limits  $-1$  and  $+1$  for all values of  $X$ . To produce a reasonable size graph we shall need to multiply the resultant  $Y$  values by a scaling factor  $YS$ . A convenient size is produced by setting  $YS = 6\theta$  at the start of the program. A second multiplier,  $XS$ , is used to scale the  $X$  values. In this case  $XS$  is set at  $1\theta$  to give a graph which is a total of 200 units wide on the screen. The values for  $XS$  and  $YS$  are chosen to give the largest graph that will fit on the screen, based on the expected range of values of  $X$  and  $Y$  to be plotted.

The first step in constructing the graph is to produce the  $X$  and  $Y$  axis lines and scales. The centre point of the axes where  $X=0$  and  $Y=0$  is defined by two variables  $XC$  and  $YC$ . These are set at  $16\theta$  and  $1\theta\theta$  respectively, to place the graph in the centre of the screen area. You could, of course, choose other values for  $XC, YC$  to place the graph in a different position if desired.

To draw the  $X$  axis we start by setting  $X1 = XC - 1\theta * XS$  and  $Y1 = YC$ . These values give the starting co-ordinates for drawing the  $X$ -axis line across the screen. The end of the line ( $X2, Y2$ ) is set four units down the screen in the  $Y$  direction ( $YC + 4$ ) and a short graduation line is drawn. Next, a loop is used to draw alternately a horizontal line  $XS$  units long and a vertical mark four units high, and this is done 20 times to give the  $X$  axis and scale.

The  $Y$  axis and its scale marks are drawn in a similar fashion. In this case a new variable  $YA$  has been introduced, which is the number of screen units between scale marks on the  $Y$  axis, and is set equal to  $YS / 1\theta$  since there are 10 divisions in the  $Y$  scale. The  $Y$  axis is drawn down the screen from a point  $Y1 = YC - 1\theta * YA$ . The process of drawing the  $X$  and  $Y$  axes is dealt with as a subroutine starting at line  $5\theta\theta$ , although it could equally well be included as a sequence of instructions in the main program if desired.

Having drawn the axes, the next step is to plot the graph itself. Here the calculations are carried out in a loop with the angle ( $XP$ ) being stepped in increments of  $0.05$  from  $-1\theta$  to  $+1\theta$ . Note that the units of  $XP$  will be *radians* in this calculation. The  $X$  co-ordinate for each point to be plotted is calculated from

$$X = XC + \text{INT}(XS * XP)$$

and the  $Y$  co-ordinate is given by,

$$Y = YC - \text{INT}(YS * \text{SIN}(XP))$$

Note the minus sign here, which is required because the  $Y$  co-

```

100 REM SINE GRAPH PLOT
105 REM SET BIT MAP ADDRESS TO 8192
110 BM=8192
120 POKE53272,PEEK(53272)OR8
125 REM SELECT BIT MAP MODE
130 POKE53265,PEEK(53265)OR32
135 REM SET COLOURS
140 FOR I=1024TO2023:POKEI,22:NEXT
145 REM USE CLEAR ROUTINE AT 49152
150 SYS49152
160 GOTO1000
195 REM DOT PLOT WITH SCREEN WRAPAROUND
200 X=INT(X):Y=INT(Y)
210 IF X<0 THEN X=X+320:GOTO210
220 IF X>319 THEN X=X-320:GOTO220
230 IF Y<0 THEN Y=Y+200:GOTO230
240 IF Y>199 THEN Y=Y-200:GOTO240
250 P=BM+320*INT(Y/8)+8*INT(X/8)+(YAND7)
260 POKE P,PEEK(P)OR(2^(7-(XAND7)))
270 RETURN
295 REM LINE DRAWING ROUTINE
300 SX=SGN(X2-X1):SY=SGN(Y2-Y1)
310 NX=ABS(X2-X1):NY=ABS(Y2-Y1)
320 X=X1:Y=Y1:GOSUB200
330 IF NY>NX THEN 390
340 ND=INT(NX/2)
350 FOR K=1 TO NX:ND=ND+NY
360 IF ND<NX THEN X=X+SX:GOTO380
370 ND=ND-NX:X=X+SX:Y=Y+SY
380 GOSUB200:NEXT:GOTO440
390 ND=INT(NY/2)
400 FOR K=1 TO NY:ND=ND+NX
410 IF ND<NY THEN Y=Y+SY:GOTO430
420 ND=ND-NY:X=X+SX:Y=Y+SY
430 GOSUB200:NEXT
440 RETURN
495 REM AXIS DRAWING ROUTINE
500 X1=XC-XS*10:Y1=YC
510 Y2=Y1+4:X2=X1:GOSUB300
520 FOR S=1TO20
530 X2=X1+XS:Y2=YC:GOSUB300
540 Y2=Y2+4:X1=X2:GOSUB300
550 NEXT
560 X1=XC:YA=INT(YC/10):Y1=YC-YA*10
570 X2=X1-4:Y2=Y1:GOSUB300

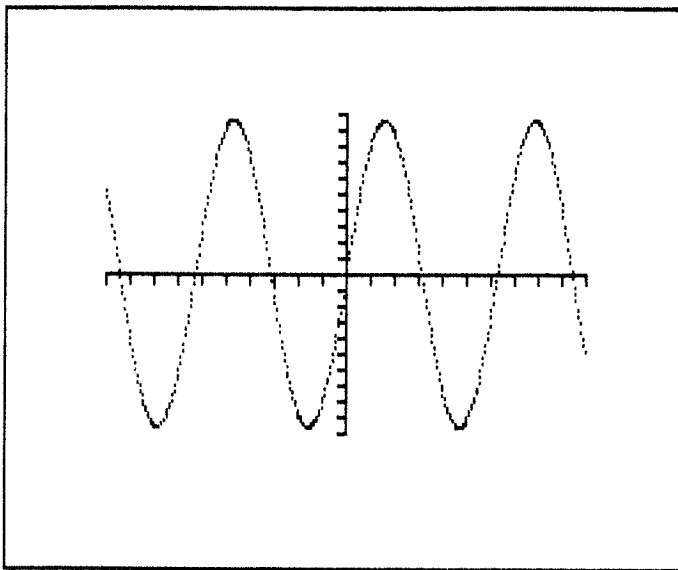
```

```

580 FOR S=1TO20
590 Y2=Y1+YA:X2=XC:GOSUB300
600 X2=X2-4:Y1=Y2:GOSUB300
610 NEXT
620 RETURN
995 REM MAIN PROGRAM
1000 XC=160:YC=100:XS=10:YS=60
1005 REM DRAW AXES AND SCALES
1010 GOSUB500
1015 REM DRAW SINE CURVE
1020 FOR XP=-10 TO 10 STEP 0.05
1030 Y=YC-INT(YS*SIN(XP))
1040 X=XC+INT(XS*XP)
1050 GOSUB200
1060 NEXT
1070 END

```

*Fig. 7.9.* Program to plot a sine graph with axes and scales.



*Fig. 7.10.* Display produced by program listed in Fig. 7.9.

ordinates of the screen increase as we move down the screen whereas the conventions of the graph require that Y should increase as we move up the screen.

The program listing is given in Fig. 7.9, and the result produced on

the screen is similar to that shown in Fig. 7.10. In this program the display colours are white on a blue background. If you want to use different colours, the number POKEd into the screen memory in line 140 must be altered.

### Joining the points

In order to obtain a reasonable picture of the curve produced by the sine function a large number of values must be plotted so that the points are closely spaced. If there were fewer values for X and Y the points would tend to be spread apart, giving a less clear impression of the function shape.

Sometimes we may wish to find the probable value for Y at a value of X that was not included in the points used for the graph. By using a technique known as *interpolation* we can obtain an approximate value for such an intermediate point on the curve.

The simplest technique for interpolation is to join successive points on the curve with straight lines. This is generally known as *linear interpolation*. We can in fact join the points with a straight line as the graph is plotted. This gives a curve that is easier to follow when the number of points available is limited. Some care is needed when using interpolation, because if too few points are used the straight line interpolation technique can become wildly inaccurate.

To join the points, the graph plotting routine is altered. Instead of setting a single dot for each point, the line drawing subroutine is used to draw a short line from one point to the next. Variables X1 and Y1 are used to specify the start of the line and X2, Y2 are used for the end of the line. After each line is drawn, X1 and Y1 are updated to equal the co-ordinates X2, Y2 and new values are chosen for X2, Y2 which will be the co-ordinates of the next point on the graph. Note that before entering the plotting loop the values X1 and Y1 are set up for the first point on the graph where  $XP = -10$ .

This variation of the graph plot program using linear interpolation to join the dots is shown in Fig. 7.11. In this program a cosine curve is plotted. This has the same shape as a sine curve but is shifted in position along the X axis, as shown in Fig. 7.12.

### Adding text in the bit map mode

In the cosine graph drawing program an additional feature is that a

```

100 REM COSINE GRAPH WITH LINKED POINTS
105 REM SET BIT MAP ADDRESS TO 8192
110 BM=8192
120 POKE53272,PEEK(53272)OR8
125 REM SELECT BIT MAP MODE
130 POKE53265,PEEK(53265)OR32
135 REM SET COLOURS
140 FOR I=1024TO2023:POKEI,22:NEXT
145 REM USE CLEAR ROUTINE AT 49152
150 SYS49152
160 GOTO1000
195 REM DOT PLOT WITH SCREEN WRAPAROUND
200 X=INT(X):Y=INT(Y)
210 IF X<0 THEN X=X+320:GOTO210
220 IF X>319 THEN X=X-320:GOTO220
230 IF Y<0 THEN Y=Y+200:GOTO230
240 IF Y>199 THEN Y=Y-200:GOTO240
250 P=BM+320*INT(Y/8)+8*INT(X/8)+(YAND7)
260 POKE P,PEEK(P)OR(2↑(7-(XAND7)))
270 RETURN
295 REM LINE DRAWING ROUTINE
300 SX=SGN(X2-X1):SY=SGN(Y2-Y1)
310 NX=ABS(X2-X1):NY=ABS(Y2-Y1)
320 X=X1:Y=Y1:GOSUB200
330 IF NY>NX THEN 390
340 ND=INT(NX/2)
350 FOR K=1 TO NX:ND=ND+NY
360 IF ND<NX THEN X=X+SX:GOTO380
370 ND=ND-NX:X=X+SX:Y=Y+SY
380 GOSUB200:NEXT:GOTO440
390 ND=INT(NY/2)
400 FOR K=1 TO NY:ND=ND+NX
410 IF ND<NY THEN Y=Y+SY:GOTO430
420 ND=ND-NY:X=X+SX:Y=Y+SY
430 GOSUB200:NEXT
440 RETURN
495 REM AXIS DRAWING ROUTINE
500 X1=XC-XS*10:Y1=YC
510 Y2=Y1+4:X2=X1:GOSUB300
520 FOR S=1TO20
530 X2=X1+XS:Y2=YC:GOSUB300
540 Y2=Y2+4:X1=X2:GOSUB300
550 NEXT
560 X1=XC:YA=INT(YC/10):Y1=YC-YA*10
570 X2=X1-4:Y2=Y1:GOSUB300

```

```

580 FOR S=1TO20
590 Y2=Y1+YA: X2=XC: GOSUB300
600 X2=X2-4: Y1=Y2: GOSUB300
610 NEXT
620 RETURN

995 REM MAIN PROGRAM
1000 XC=160: YC=100: XS=10: YS=60
1005 REM DRAW AXES AND SCALES
1010 GOSUB500
1015 REM DRAW COSINE CURVE
1020 X1=XC-10*XS: Y1=YC-INT(YS*COS(-10))
1030 FOR XP=-10 TO 10 STEP 0.4
1040 Y2=YC-INT(YS*COS(XP))
1050 X2=XC+INT(XS*XP)
1060 GOSUB300: X1=X2: Y1=Y2
1070 NEXT
1075 REM PRINT LEGEND Y=COS(X)
1080 POKE56334, PEEK(56334)AND254
1090 POKE1, PEEK(1)AND251
1100 R=1: C=16: CG=53248
1110 P=BM+320*R+8*C
1120 FOR N=1TO8
1130 READ A: CP=CG+8*A
1140 FOR J=0TO7
1150 POKE(P+J), PEEK(CP+J)
1160 NEXT
1170 P=P+8
1180 NEXT
1190 POKE 1, PEEK(1)OR4
1200 POKE56334, PEEK(56334)OR 1
1210 DATA 25, 61, 3, 15, 19, 40, 24, 41
1220 END

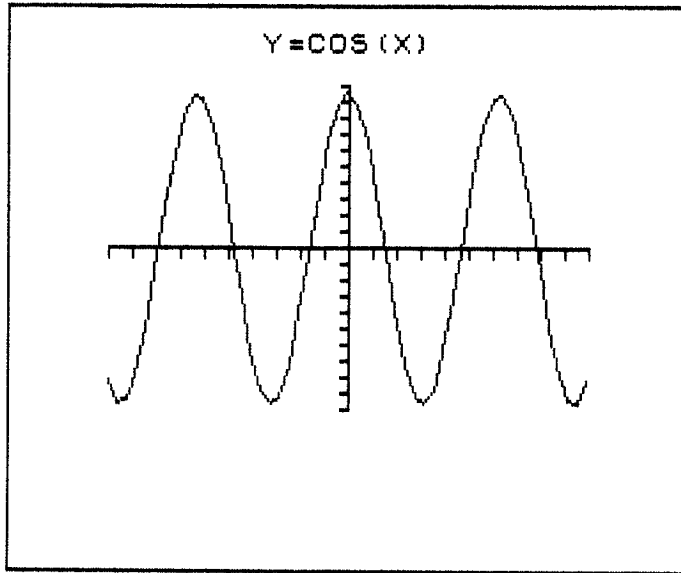
```

Fig. 7.11. Program to draw a cosine curve using linked points and including a text heading on the bit map screen.

text legend 'Y=COS(X)' has been inserted into the high resolution picture at the top of the screen. This uses a technique which may be useful in other programs where text is to be combined with bit-mapped graphics.

As soon as the bit map display mode is selected the normal PRINT function no longer operates, because the text screen memory is now being used to store colour information for the bit-mapped picture. You will notice that at the end of the program





*Fig. 7.12.* Display produced by the program listed in Fig. 7.11.

execution a short row of coloured blocks appears to the left of the picture. This is in fact the word 'READY', which has been written into the screen memory but is being interpreted by the VIC11 chip as a set of colour codes for the bit map screen.

Adding text to the bit map display involves copying the dot patterns of the required symbols from the character generator ROM into the appropriate positions in the bit map memory. This is a similar operation to that used to create a new character set. The rather odd layout of the bit map memory now becomes an advantage, since the layout of the bit pattern words in the bit map memory is the same as that of the dot patterns in the character generator ROM. To copy a symbol we simply have to copy the eight successive data words of the dot pattern from the ROM into eight successive words in the bit map memory.

The first step is to choose the desired symbol dot pattern. In the ROM the symbols are stored in screen code order. The @ symbol is number 0, A is number 1 and so on. To find the start address of a dot pattern we simply multiply the screen code of the required letter or other character by eight and add this to the start address of the ROM (53248).

The next step is to work out the position in the bit map memory where the data must be placed. We can start with a row and

column position based on the normal 25-row by 40-column text screen layout. To find the address in the bit map memory the row number is multiplied by 320 and added to the column number multiplied by eight, then the result is added to the start address of the bit map memory.

To set up the text symbol we have to copy its dot pattern from the character generator into the bit map memory. First the keyboard interrupt timer is turned off and the ROM enabled. At this point the character pattern is selected and eight successive words are PEEKed from the character ROM and POKEd into successive locations in the bit map memory. This is repeated for all the symbols to be transferred. After the dot patterns for all the symbols have been copied, the ROM is enabled and the interrupt timer turned on again.

It is convenient to set up the required text as a data array of screen codes which are read in and processed one at a time. In this case a single string of symbols is transferred. You could, of course, transfer several strings of symbols and place them at different points on the screen if desired by altering the row and column numbers used to select the address in the bit map before each new set of symbols is transferred.

## Chapter Eight

# Depth and Perspective

The graphs and charts we have drawn so far have had just two variables, X and Y, which were plotted horizontally and vertically on the screen. In the real world there are many situations where three variables are involved. The third variable is usually given the name Z. An example of this would be a display showing the height of various points in a small area of land. In this case the X and Y coordinates would be used to define the location of a particular point in the area, and might represent, say, length and width, or perhaps the north-south and east-west position. In this case the third term, Z, is the height of the land surface at the point X, Y. Here the value of Z depends upon both X and Y, since a change in either X or Y takes us to another point on the land surface with a different value for Z.

Drawing a three-axis graph requires slightly different techniques from those needed for a two-axis graph, since we have to find a way of fitting in the Z axis. If X and Y are plotted as usual on the screen, the Z ordinates should theoretically be plotted out from the surface of the screen. This is obviously impractical, so we need to consider other arrangements.

Suppose we were building a cardboard model of the three-axis plot. The first step would be to plot a series of graphs of Z against X. Once the graphs had been plotted, the next step might be to stand the graphs one behind the other. How can this be done on our display screen?

One solution might be to draw them so that the graph for each new value of Y is displaced to the left and up on the screen. This helps to separate the individual graphs for the different values of Y. In effect we have tilted the Y axis so that it becomes a sloping line which runs upwards and to the left of the X, Y, Z origin point, where X, Y and Z are all zero. The next step is to tilt the X axis so that it now slopes up to the right of the origin point. The Z axis can now be drawn vertically up the screen in the same way that the Y axis is drawn on a two-axis graph or chart.

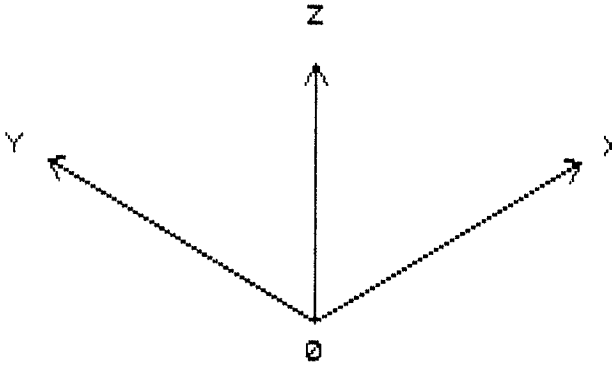


Fig. 8.1. Layout of the X, Y and Z axes of a three-axis graph or chart.

The usual arrangement for displaying a three-axis plot is to draw both the X and Y axes at about 30 degrees to the horizontal axis of the screen, with the Z axis vertical as shown in Fig. 8.1. Now as X increases the plotted point moves upwards and to the right, and as Y increases the plotted point moves upwards and to the left. Finally Z displaces the point vertically on the screen.

Drawing a three-axis graph or chart is not readily achieved by using the character graphics mode of the Commodore 64, so for this type of display we need to make use of the high resolution and drawing flexibility that is available in the bit-mapped mode.

### Three-axis bar charts

One type of display that looks impressive in a three-axis version is a bar chart. The first step in constructing such a chart is to choose an origin point where the values of X, Y and Z are all at zero. This point determines where the bar chart is positioned on the screen, and also acts as a reference point around which the plot will be constructed. The next step might be to draw a grid showing the X and Y co-ordinates in the plane where  $Z = 0$ .

To draw the X axis at about 30 degrees to the horizontal there will be changes in both X and Y screen co-ordinates as we move along the X axis. The Y movement required is half the X movement, so our screen co-ordinates for points along the X axis (Y and Z both = 0) will be

$$\begin{aligned} X1 &= XC + XP \\ Y1 &= YC - XP/2 \end{aligned}$$

where  $XP$  is the  $X$  co-ordinate along the  $X$  axis. When  $X$  and  $Z$  are both at 0 the line representing the  $Y$  axis goes up and to the left. Since the movement is to the left of the origin point  $(XC, YC)$  this means that the screen  $X$  co-ordinates for points along the  $Y$  axis must be less than  $XC$ . To get the 30-degree angle to the left the change in screen  $X$  position is made negative, and equals the  $Y$  value on the graph. To get the upward slope, the screen  $Y$  position changes by half as much, and the change is subtracted from  $YC$  so that the line moves upwards. The screen co-ordinates here become

$$\begin{aligned} X1 &= XC - YP \\ Y1 &= YC - YP/2 \end{aligned}$$

where  $YP$  is the  $Y$  co-ordinate along the  $Y$  axis on the graph being plotted.

For any other point on the  $Z = 0$  plane then the position of  $X1, Y1$  will be produced by combining the two results we obtained above to give

$$\begin{aligned} X1 &= XC + XP - YP \\ Y1 &= YC - XP/2 - YP/2 \end{aligned}$$

The  $Z$  term is plotted vertically, so it will only affect the screen  $Y$  value of a point on the chart. Since we are going to draw a vertical line to represent the  $Z$  ordinate we need to know the co-ordinates for the top of the line. Now the  $X$  value is the same as  $X1$  and for the new  $Y$  value  $Z$  is simply subtracted from  $Y$  so the values for co-ordinates  $X2, Y2$  become

$$\begin{aligned} X2 &= X1 = XC + XP - YP \\ Y2 &= Y1 - Z = YC - XP/2 - YP/2 - Z \end{aligned}$$

The  $Z$  ordinates can now be produced by drawing lines starting at  $X1, Y1$  and running to point  $X2, Y2$  using a bit map line drawing subroutine.

A three-axis bar chart produced in this way would have a simple vertical line for each  $Z$  ordinate, and would look like a bed of nails. The chart can be made to look more attractive by turning the simple vertical line into a bar aligned along, say, the  $X$  axis. This involves drawing another  $Z$  ordinate of the same height but at a different position along the  $X$  axis. The top and bottom of these two ordinates are then joined with short lines, and the area within the bar is filled with colour.

The program of Fig. 8.2 produces a three-axis chart with wide bars aligned along the  $X$  axis. Variables  $XC$  and  $YC$  give the

```

100 REM 3 AXIS CHART WITH WIDE BARS
105 REM SET BIT MAP ADDRESS TO 8192
110 BM=8192
120 POKE53272,PEEK(53272)OR8
125 REM SELECT BIT MAP MODE
130 POKE53265,PEEK(53265)OR32
135 REM SET COLOURS
140 FOR I=1024TO2023:POKEI,22:NEXT
145 REM USE CLEAR ROUTINE AT 49152
150 SYS49152
160 GOTO1000
195 REM DOT PLOT ROUTINE
200 X=INT(X):Y=INT(Y)
210 IF X<0 OR X>319 THEN 280
220 IF Y<0 OR Y>199 THEN 280
230 P=BM+320*INT(Y/8)+8*INT(X/8)+(YAND7)
240 IF ER=1 THEN270
245 REM SET DOT
250 POKE P,PEEK(P)OR(2↑(7-(XAND7)))
260 GOTO280
265 REM ERASE DOT
270 POKEP,PEEK(P)AND(255-2↑(7-(XAND7)))
280 RETURN
295 REM LINE DRAWING ROUTINE
300 SX=SGN(X2-X1):SY=SGN(Y2-Y1)
310 NX=ABS(X2-X1):NY=ABS(Y2-Y1)
320 X=X1:Y=Y1:GOSUB200
330 IF NY>NX THEN 390
340 ND=INT(NX/2)
350 FOR K=1 TO NX:ND=ND+NY
360 IF ND<NX THEN X=X+SX:GOTO380
370 ND=ND-NX:X=X+SX:Y=Y+SY
380 GOSUB200:NEXT:GOTO440
390 ND=INT(NY/2)
400 FOR K=1 TO NY:ND=ND+NX
410 IF ND<NY THEN Y=Y+SY:GOTO430
420 ND=ND-NY:X=X+SX:Y=Y+SY
430 GOSUB200:NEXT
440 RETURN
495 REM DRAW AXES
500 FOR XP=0 TO 100 STEP 20
510 X1=X0+XP:Y1=Y0-XP/2
520 X2=X1-100:Y2=Y1-50:GOSUB300
530 NEXT

```

```

540 FOR YP=0 TO 90 STEP 15
550 X1=XC-YP:Y1=YC-YP/2
560 X2=X1+110:Y2=Y1-55:GOSUB300
570 NEXT
580 RETURN
595 REM DRAW BAR ROUTINE
600 FOR N=0 TO Z-1 STEP 2
610 X1=XC+XP-YP:Y1=YC-XP/2-YP/2-N
620 X2=X1+XA:Y2=Y1-YA:GOSUB300
630 NEXT
635 REM ERASE BAR OUTLINE
640 ER=1
650 X1=XC+XP-YP+XA:Y1=YC-XP/2-YP/2-YA
660 X2=X1:Y2=Y1-Z-1:GOSUB300
670 X1=X2:X2=X-XA:Y1=Y2:Y2=Y2+YA
680 GOSUB300
690 X1=X2:Y1=Y2:Y2=Y2+Z+1:GOSUB300
700 ER=0
710 RETURN
995 REM MAIN PROGRAM
1000 XC=160:YC=180:ER=0
1010 XA=10:XB=8:YA=5:YB=4
1015 REM DRAW AXES
1020 GOSUB500
1030 FOR XP=100 TO 0 STEP -20
1040 FOR YP=90 TO 0 STEP -15
1050 Z=INT((4+2*COS(XP/20))*(YP/10+1))
1055 REM DRAW BARS
1060 GOSUB600
1070 NEXT
1080 NEXT
1090 END

```

Fig. 8.2. Program to produce a three-axis bar chart with bars aligned along the X axis.

position of the origin of the graph, where X, Y and Z values are all 0. Since variables X and Y are used by the dot plotting routine, new variables XP and YP have been used for the co-ordinates of the bars. Variables XA and YA determine the width of the bars. To draw the bar, a series of short lines is drawn along the direction of the X axis but stacked one above the other to build up the bar. The total number of lines is equal to Z/2. Here only the alternate lines were drawn in to save time. Line 600 could be changed by deleting STEP 2 so that all lines up the bar are drawn to give a solidly-filled bar.

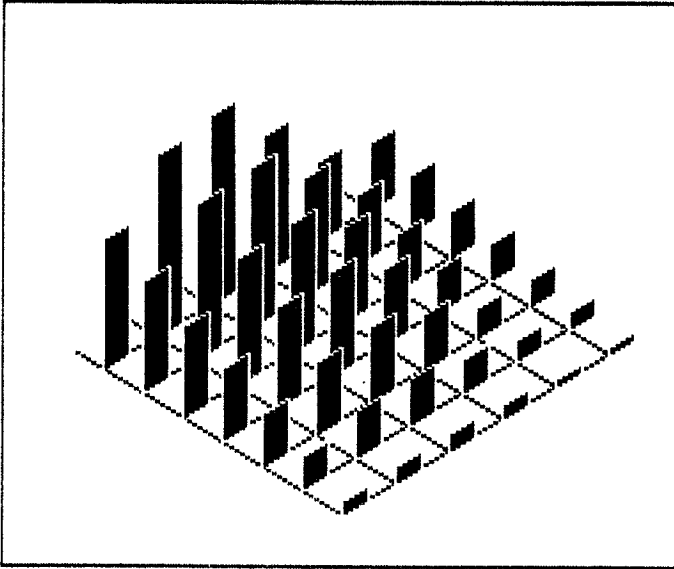


Fig. 8.3. Typical display produced by program listed in Fig. 8.2.

This program will take longer to draw the chart and will produce results similar to that shown in Fig. 8.3.

In this program the bars at the rear of the graph are drawn first and then the drawing progresses toward the front. With solid bars being drawn where a bar in front overlaps one at the back it will mask it off. To separate the bars that overlap, the outline of each bar is erased after that bar has been drawn.

Here you will see that the bit mode dot plotting routine has been extended. If the variable ER is at  $\emptyset$  the dots are plotted in the normal way. When ER=1 this is tested in the dot plotting routine and a jump is made to line 27 $\emptyset$ . Here instead of using an OR function to set the dot an AND function is used to reset the dot to the 0 or 'off' state.

### Producing solid bars

A further development is to draw the bars so that they appear to be solid. In effect, one side of the bar is aligned along the X axis and another aligned with the Y axis for each Z ordinate. A diamond-shaped top is drawn to complete the bar. One side of the bar may then be filled with colour as desired.

The program listing shown in Fig. 8.4 produces an example of this



```

100 REM 3 AXIS CHART WITH SOLID BARS
105 REM SET BIT MAP ADDRESS TO 8192
110 BM=8192
120 POKE53272,PEEK(53272)OR8
125 REM SELECT BIT MAP MODE
130 POKE53265,PEEK(53265)OR32
135 REM SET COLOURS
140 FOR I=1024TO2023:POKEI,22:NEXT
145 REM USE CLEAR ROUTINE AT 49152
150 SYS49152
160 GOTO1000
195 REM DOT PLOT ROUTINE
200 X=INT(X):Y=INT(Y)
210 IF X<0 OR X>319 THEN 280
220 IF Y<0 OR Y>199 THEN 280
230 P=BM+320*INT(Y/8)+8*INT(X/8)+(YAND7)
240 IF ER=1 THEN270
245 REM SET DOT
250 POKE P,PEEK(P)OR(2^(7-(XAND7)))
260 GOTO280
265 REM ERASE DOT
270 POKEP,PEEK(P)AND(255-2^(7-(XAND7)))
280 RETURN
295 REM LINE DRAWING ROUTINE
300 SX=SGN(X2-X1):SY=SGN(Y2-Y1)
310 NX=ABS(X2-X1):NY=ABS(Y2-Y1)
320 X=X1:Y=Y1:GOSUB200
330 IF NY>NX THEN 390
340 ND=INT(NX/2)
350 FOR K=1 TO NX:ND=ND+NY
360 IF ND<NX THEN X=X+SX:GOTO380
370 ND=ND-NX:X=X+SX:Y=Y+SY
380 GOSUB200:NEXT:GOTO440
390 ND=INT(NY/2)
400 FOR K=1 TO NY:ND=ND+NX
410 IF ND<NY THEN Y=Y+SY:GOTO430
420 ND=ND-NY:X=X+SX:Y=Y+SY
430 GOSUB200:NEXT
440 RETURN
495 REM DRAW AXES
500 FOR XP=0 TO 100 STEP 20
510 X1=XC+XP:Y1=YC-XP/2
520 X2=X1-100:Y2=Y1-50:GOSUB300
530 NEXT
540 FOR YP=0 TO 90 STEP 15

```

```

550 X1=XC-YP:Y1=YC-YP/2
560 X2=X1+110:Y2=Y1-55:GOSUB300
570 NEXT
580 RETURN
595 REM DRAW BAR ROUTINE
600 FOR N=0 TO Z-1 STEP 2
610 X1=XC+XP-YP:Y1=YC-XP/2-YP/2-N
620 X2=X1+XA:Y2=Y1-YA:GOSUB300
630 NEXT
635 REM ERASE SIDE OF BAR
640 FOR N=0 TO Z-1
650 X1=XC+XP-YP:Y1=YC-XP/2-YP/2-N
660 X2=X1-XB:Y2=Y1-YB:ER=1:GOSUB300
670 NEXT:ER=0
680 X1=XC+XP-YP:Y1=YC-XP/2-YP/2
690 X2=X1-XB:Y2=Y1-YB:GOSUB300
700 X1=X2:Y1=Y2:Y2=Y2-Z:GOSUB300
710 X1=X2:X2=X2+XB:Y1=Y2
720 Y2=Y2+YB:GOSUB300
730 X1=X2:Y1=Y2:Y2=Y2+Z:GOSUB300
735 REM ERASE TOP OF BAR
740 FOR N=0 TO XB-1
750 X1=XC+XP-YP-N:Y1=YC-Z-(XP+YP+N)/2
760 X2=X1+XA:Y2=Y1-YA:ER=1:GOSUB300
770 NEXT:ER=0
775 REM DRAW TOP OF BAR
780 X1=XC+XP-YP:Y1=YC-XP/2-YP/2-Z
790 X2=X1+XA:Y2=Y1-YA:GOSUB300
800 X1=X2:X2=X2-XB:Y1=Y2
810 Y2=Y2-YB:GOSUB300
820 X1=X2:X2=X2-XA:Y1=Y2:Y2=Y2+YA
830 GOSUB300
840 X1=X2:X2=X2+XB:Y1=Y2:Y2=Y2+YB
850 GOSUB300
860 RETURN
995 REM MAIN PROGRAM
1000 XC=160:YC=180:ER=0
1010 XA=10:XB=8:YA=5:YB=4
1020 GOSUB500
1030 FOR XP=100 TO 0 STEP -20
1040 FOR YP=90 TO 0 STEP -15
1050 Z=INT((2+COS(XP/20))*(YP/10+1))
1060 GOSUB600
1070 NEXT
1080 NEXT
1090 END

```

Fig. 8.4. Program to produce three-axis bar chart with solid bars.

type of display and the result on the screen is similar to that shown in Fig. 8.5. In the program the front face of the bar is drawn first and filled. Next the side of the bar along the Y axis is erased by setting  $ER=1$  and drawing a series of short parallel lines one above the other across the bar. This erases any parts of other bars that lie behind the one being drawn and might otherwise show through the bar. The outline of the side of the bar is then drawn. Finally the diamond-shaped top of the bar is erased by drawing lines across it with  $ER$  set at 1, and then its outline is drawn in. Here the bars are filled by drawing alternate lines to save time. For better results you could fill the whole of the X face of the bars to get results similar to those shown in Fig. 8.5.

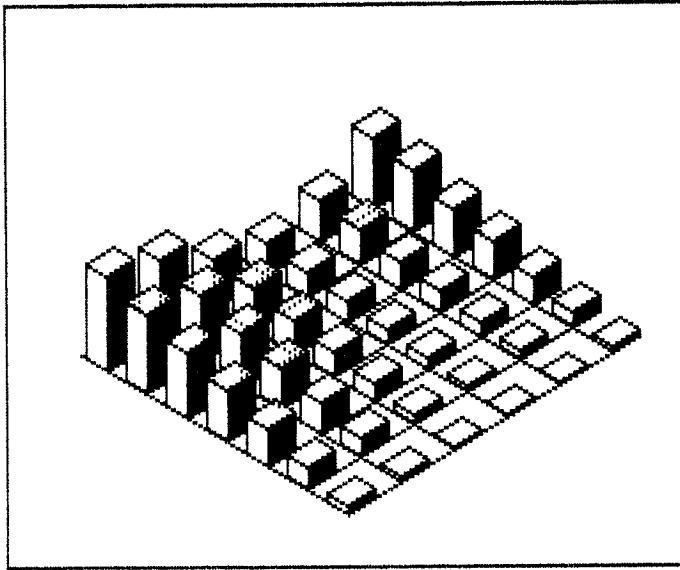


Fig. 8.5. Typical display produced by program listed in Fig. 8.4.

### Circular three-axis plots

A rather interesting variation of three-axis plotting is the *circular three-axis plot* which can produce some rather attractive patterns.

In this version of the three-axis graph the X, Y plane of the chart is made elliptical so that the resultant plot is displayed on the screen as a sort of ridged disc viewed from an angle with elliptical ridges produced by the Z ordinates.

The technique of plotting this type of graph makes use of the quadratic method for drawing a circle to produce the X,Y axes. The X scale is set at perhaps two or three times the Y scale to produce an elliptical figure on the screen. Z values are simply added to the calculated Y co-ordinates, and points are plotted at the tops of the Z ordinates. The program of Fig. 8.6 gives an examples of this type of plot. The function to be plotted is set up by using a Define Function (DEF FN) statement in line 1020. The actual function used here will determine the contour shape of the display, and you could experiment with a variety of functions.

The loop starting at line 1030 scans across the X axis from  $X = -100$  to  $X = +100$ . Then, in line 1050, an inner loop which scans the

```

100 REM CIRCULAR 3-D PLOT
105 REM NEEDS M/C SCREEN CLEAR ROUTINE
110 BM=8192
120 POKE53272,PEEK(53272)OR8
130 POKE53265,PEEK(53265)OR32
140 FORI=1024TO2023:POKEI,22:NEXT
150 SYS49152
160 GOTO1000
195 REM DOT PLOTTING ROUTINE
200 X=INT(X):Y=INT(Y)
210 IF X<0 OR X>319 THEN 250
220 IF Y<0 OR Y>199 THEN 250
230 P=BM+320*INT(Y/8)+8*INT(X/8)+(YAND7)
240 POKE P,PEEK(P)OR(2+(7-(XAND7)))
250 RETURN
995 REM MAIN PROGRAM
1000 K=π/2000
1010 M=1/SQR(2)
1020 DEF FN A(Z)=10*COS(K*(XP*XP+YP*YP))
1030 FOR XP=-100 TO 100
1040 Y1=5*INT(SQR(10000-XP*XP)/5)
1050 FOR YP=Y1 TO -Y1 STEP -5
1060 Z=FN A(SQR(XP*XP+YP*YP))-M*YP
1070 IF YP=Y1 THEN 1090
1080 IF Z<Z1 THEN 1110
1090 X=160+XP:Y=100-INT(Z/2):GOSUB200
1100 Z1=Z
1110 NEXT YP
1120 NEXT XP

```

Fig. 8.6. Program to produce a circular three-axis plot.

Y axis of the plot is started. Here the limit value Y1 is based upon the value of X with the characteristic circle equation in line 1040, and this gives the elliptical shape to the plot. Inside these loops the Z value is calculated using function FN A and the corresponding point is plotted using the bit map plot routine.

A typical display appears as in Fig. 8.7. Note that this type of display takes quite a long time to generate because of the large number of points and the relatively complex calculations for each point.

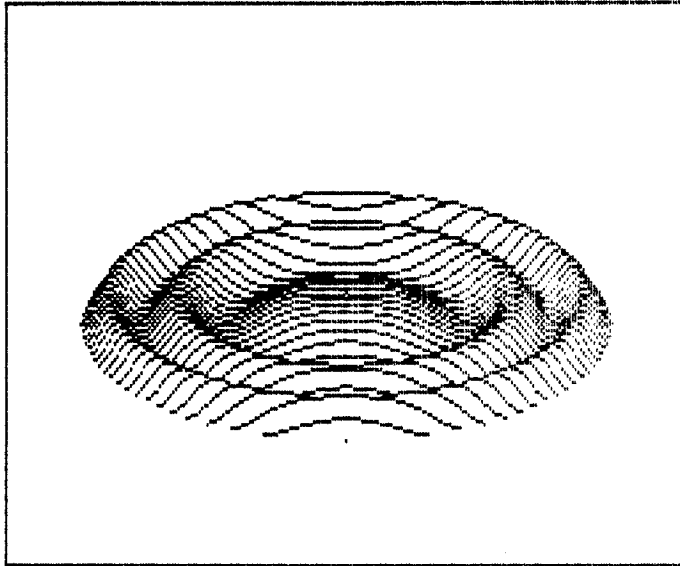


Fig. 8.7. Typical display of a circular three-axis plot.

### **Perspective drawings**

To create the illusion of depth an artist uses a technique known as *perspective*. Artists discovered many centuries ago that as an object is moved further away from the viewer it appears to get smaller, and as it moves closer it seems to get bigger. They also found that by applying this idea to drawings and paintings they could produce a much more realistic picture. This technique is known as perspective drawing and over the years mathematicians have evolved formulae that allow us to calculate the shape and size objects should be drawn to give a correct perspective view. This technique can be applied in computer graphics to produce pseudo-three-dimensional displays on the screen.

To see how perspective works, imagine you are standing on a flat plain, with a road in front of you that stretches away to the horizon. Although the sides of the road are actually parallel, the road will appear to get narrower as it approaches the horizon. The cars and trucks travelling along the road also appear to get smaller as they move away from your position towards the horizon. In fact, the optical image they produce *does* get smaller as they move away. If we apply this basic rule to our pictures on the screen we can also produce an illusion of depth, despite the fact that our display is really a flat screen.

Firstly we need to decide on some system of co-ordinates by which we can measure the positions of points on the objects being viewed and the corresponding points needed to produce the screen image. We shall assume that the X axis runs across from left to right as usual. The Z axis is normally the vertical direction, as we had it in our three axis graphs. This leaves the Y axis, and the best arrangement is to have the Y axis along the direction of view.

If you looked at the road across the desert from actual road level (that is with your eye at the road surface) the view would be rather uninspiring, because every point on the road and the desert would lie along a single line through the X axis. In order to see the road properly we need to be located above it.

Figure 8.8 shows a side view of the situation where we are viewing the road from an altitude Z. In order to project the image on our flat screen we shall assume that we are looking through a window at distance D.

Suppose we take a point on the road at distance Y1. This will

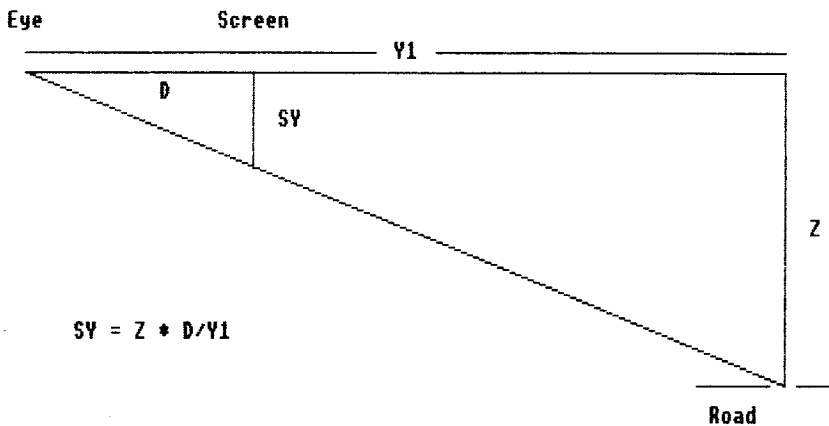


Fig. 8.8. Projection of a distant point on a road to a point on the display screen.

appear to be below the horizon line on the screen by an amount SY. We have assumed here that the horizon is effectively at eye level, which it will be if we are looking along a line parallel to the Y axis. Now the small triangle between the screen and eye is of the same shape as the large triangle passing through point Y1. This means that the sides of the two triangles have the same proportions, so we can say that

$$SY/D = Z/Y1$$

and rearranging this we get

$$SY = Z*D/Y1$$

Now you will note that the size of the image on the screen is inversely proportional to the distance Y1. If we took another point on the road at a different distance, Y2, then it would produce a different line length on the screen, giving a new value for SY of

$$SY = Z*D/Y2$$

Now suppose there were a series of equal length lines drawn along the centre line of the road. Each line will produce a short vertical line on the screen, and as the point on the road gets farther away the image produced on the screen by each line gets shorter.

If we drew a similar view of the road looking down on it we would find that the same basic formula applies for the SX image size on the screen, which represents the road width. As Y1 becomes greater the width of the image on the screen decreases according to the formula

$$SX = W*D/Y1$$

where W is the width of the road.

Vertical objects alongside the road will also produce images that follow this general rule of being inversely proportional to distance Y.

Let us start with the road. Suppose we place the horizon halfway up the screen at position Y=99. If we draw lines from the bottom two corners of the screen to a point halfway across the horizon line we have a road.

Suppose the road is 25 feet wide, and we are viewing a 14-inch TV screen from, say, a distance of 4 feet. The TV screen will be 1 foot wide and this is equal to SX. D will be 4, and X will be 25. Now

$$SX = D*X/Y = 4*25/Y = 1$$

therefore

$$Y = 4 \times 25 \text{ or } 100 \text{ feet}$$

The scaling factor for the X direction in our screen drawing can now be worked out. There are 320 units of X across the screen on the Commodore 64 and when  $Y=100$  the road fills the screen width, so SX must be 320. If we rewrite our equation with a multiplier XS we get

$$SX = XS \times X/Y$$

and inserting our calculated values we get the following

$$320 = XS \times 25/100$$

now extracting the XS term we get

$$XS = 320 \times 100 / 25 = 1280$$

To calculate values of SX we would use the equation

$$SX = 1280 \times X/Y$$

We can apply a similar process to calculate the Y scale factor, assuming that our viewpoint is at a height of 10 feet, and that for a distance of 100 feet the point plotted is at the bottom of the screen. We shall assume that the horizon line is at a point 100 units up from the bottom of the screen. From this

$$SY = YS \times 10/100 = 100$$

and

$$YS = 1000$$

so to calculate the Y points on the screen we would use

$$SY = 1000 \times Z/Y$$

To plot a point on the road itself,  $Z=10$ , and the actual Y value for use in the drawing instruction will be  $99+SY$ , since as we get closer the point moves down the screen. If there is a vertical pole then to plot the top of the pole we subtract the height of the pole from 10 to obtain the value for Z. Thus a 10-foot high pole will always produce a Y value which lines up with the horizon, since we are actually looking along a line 10 feet above the road. If the pole is higher than 10 feet the value of SY is negative, and the top of the pole goes above the centre line of the screen.

To draw a perspective view of a road with, say, trees alongside,



you will need to set up values of height for the tree trunk and the top of the tree, and also the width of the tree. Knowing the distance of each tree from the viewer, its X,Y co-ordinates for the base, top and side points can be calculated using

$$\begin{aligned} SX &= 160 + 1280 \times X/Y \\ SY &= 99 + 1000 \times (10-Z)/Y \end{aligned}$$

where SX and SY are the screen drawing co-ordinates, X is the distance measured from the centre of the road with positive values to the right, and Z is the height of the object. Y is the distance measured from the viewing position. Once the co-ordinates are known the tree can be drawn by making it up from a small set of lines. The road markings are dealt with in a similar fashion, except that here Z will be 0.

In the program, X1,Y1,X2 and Y2 are used by the line drawing routine, and so are SX and SY, so some new variables have been introduced to avoid conflict between the various parts of the program. Here YR is used for the distance from the viewer to an object being drawn. For the road markings W is used for actual width, and L for actual length. W1 and W2 are used as the screen widths of the near and far ends of a displayed road marking, and YA and YB are used for the Y screen ordinates for the two ends of a road marking.

For drawing the trees, TR is the height of the trunk, TT is the total height of the tree, and TW is the width of the branches at the bottom of the tree. The trees themselves are very simplified images consisting of a triangular area for the branches and a vertical line for the trunk.

Figure 8.9 shows a program listing to draw a perspective view of a road, and Fig. 8.10 shows the result on the screen.

```

100 REM PERSPECTIVE VIEW OF A ROAD
105 REM SET BIT MAP ADDRESS TO 8192
110 BM=8192
120 POKE53272,PEEK(53272)OR8
125 REM SELECT BIT MAP MODE
130 POKE53265,PEEK(53265)OR32
135 REM SET COLOURS
140 FOR I=1024TO2023:POKEI,22:NEXT
145 REM USE CLEAR ROUTINE AT 49152
150 SYS49152
160 GOTO1000
195 REM DOT PLOT WITH SCREEN WRAPAROUND

```

```

200 X=INT(X):Y=INT(Y)
210 IF X<0 THEN X=X+320:GOTO210
220 IF X>319 THEN X=X-320:GOTO220
230 IF Y<0 THEN Y=Y+200:GOTO230
240 IF Y>199 THEN Y=Y-200:GOTO240
250 P=BM+320*INT(Y/8)+8*INT(X/8)+(YAND7)
260 POKE P,PEEK(P)OR(2^(7-(XAND7)))
270 RETURN
295 REM LINE DRAWING ROUTINE
300 SX=SGN(X2-X1):SY=SGN(Y2-Y1)
310 NX=ABS(X2-X1):NY=ABS(Y2-Y1)
320 X=X1:Y=Y1:GOSUB200
330 IF NY>NX THEN 390
340 ND=INT(NX/2)
350 FOR K=1 TO NX:ND=ND+NY
360 IF ND<NX THEN X=X+SX:GOTO380
370 ND=ND-NX:X=X+SX:Y=Y+SY
380 GOSUB200:NEXT:GOTO440
390 ND=INT(NY/2)
400 FOR K=1 TO NY:ND=ND+NX
410 IF ND<NY THEN Y=Y+SY:GOTO430
420 ND=ND-NY:X=X+SX:Y=Y+SY
430 GOSUB200:NEXT
440 RETURN
495 REM DRAW TREE SUBROUTINE
500 X1=XA:X2=XA:Y1=YA
510 Y2=YB:GOSUB300
520 X2=XA+XW:Y1=Y2:GOSUB300
530 X1=X2:X2=XA-XW:GOSUB300
540 X1=X2:X2=XA:Y2=YC:GOSUB300
550 X1=X2:Y1=Y2:X2=XA+XW:Y2=YB:GOSUB300
560 RETURN
995 REM MAIN PROGRAM
1000 X1=160:X2=0:Y1=99:Y2=199:GOSUB300
1010 X2=319:GOSUB300
1020 XS=1280:YS=1000:W=1:L=50
1025 REM DRAW ROAD MARKINGS
1030 FOR YR=100 TO 2000 STEP 100
1040 W1=INT(XS*W/YR)
1050 W2=INT(XS*W/(YR+L))
1060 YA=99+INT(YS*10/YR)
1070 YB=99+INT(YS*10/(YR+L))
1080 X1=160-W1:X2=160+W1
1090 Y1=YA:Y2=YA:GOSUB300
1100 X1=X2:Y2=YB:X2=160+W2:GOSUB300

```

```

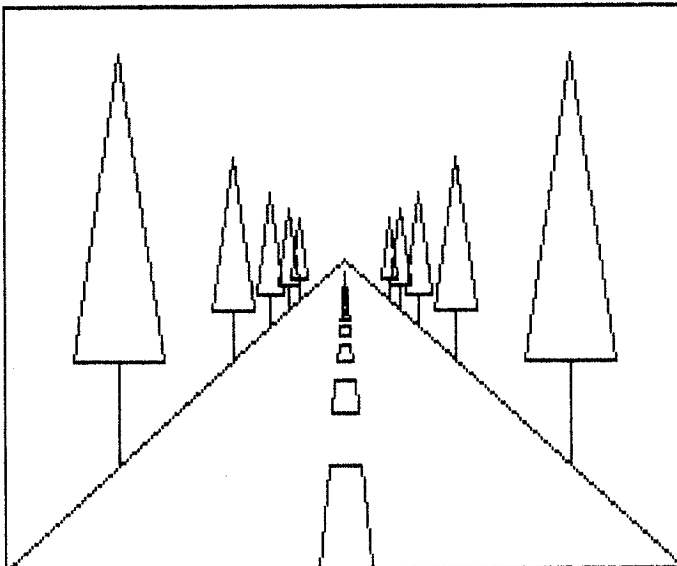
1110 X1=X2:Y1=Y2:X2=160-W2:GOSUB300
1120 X1=X2:Y2=YA:X2=160-W1:GOSUB300
1130 NEXT
1140 TR=5:TT=20:TW=3
1145 REM DRAW TREES
1150 FOR YR=150 TO 1200 STEP 150
1160 YA=99+INT(YS*10/YR)
1170 YB=99+INT(YS*(10-TR)/YR)
1180 YC=99+INT(YS*(10-TT)/YR)
1190 XT=INT(XS*12.5/YR)
1200 XW=INT(XS*TW/YR)
1210 XA=160+XT:GOSUB500
1220 XA=160-XT:GOSUB500
1230 NEXT

```

*Fig. 8.9.* Program to draw a perspective view of a road.

One problem with this type of perspective drawing is that it tends to be very slow using BASIC plotting and drawing routines in the bit map mode. Better results could, of course, be achieved by using machine code routines for dot plotting and line drawing, but the writing of such routines is not a particularly simple task.

Perspective views can be produced using character graphics by firstly sketching the perspective view and then assembling it in jigsaw fashion using character graphics symbols or even user-defined symbols.



*Fig. 8.10.* Display produced by the program listed in Fig. 8.9.

## Chapter Nine

# The Sound Generator

So far our Commodore 64 computer has remained silent, although it is in fact capable of generating a wide variety of sounds. These sounds are produced by a special chip within the machine called the 6581 Sound Interface Device or 'SID' chip.

Many of the home computers currently available use a small loudspeaker mounted within the computer case to produce sound output. While this type of loudspeaker is adequate for producing simple 'beep' sounds it is rather unsatisfactory if more complex sound effects or music are to be produced. The technique adopted by Commodore is to take the sound signals generated by the computer system and add them to the television output signal, in the same way that sound is added to a broadcast television programme. As a result, the sound signals pass through the sound section of the television receiver and are reproduced by the loudspeaker in the TV set. This provides several advantages. Firstly the volume and quality of the sound signals are determined by the television receiver, and the results will invariably be better than those that can be achieved by a speaker built into the computer. Another advantage is that the sound comes from the same position as the picture, which is more realistic when games programs are being used.

The sound generator used in the Commodore is a very comprehensive one, allowing three independent sound channels and providing full control of the characteristics of the sounds produced by each of the channels. This allows the computer to produce a wide range of sound effects and quite impressive musical performance. One drawback, however, is that there are no special BASIC commands to handle the operation of the sound generator, and this has to be achieved by POKEing numbers into the internal registers of the chip. At first glance this may seem rather a complicated process, but it is actually fairly straightforward once the functions of the various registers are understood.

## The nature of sound

The sounds that we hear are produced by rapid changes in the air pressure against a membrane, known as the eardrum, within our ears. These pressure changes are converted into nerve impulses which, when they reach the brain, produce the sensation that we recognise as sound.

The sound waves themselves can be likened to the ripples produced on a pond when a stone is thrown into it. When the stone enters the water it starts off a series of circular ripples or waves which travel outwards across the surface of the water. If we took a sectional view through the waves they would look similar to those shown in Fig. 9.1. The crests, or tops, of the waves are equally spaced, and the distance between them is known as the *period* while the height of the wave is called its *amplitude*. In this case the waves have a shape which is similar to a sine curve, and they are known as sinusoidal waves.

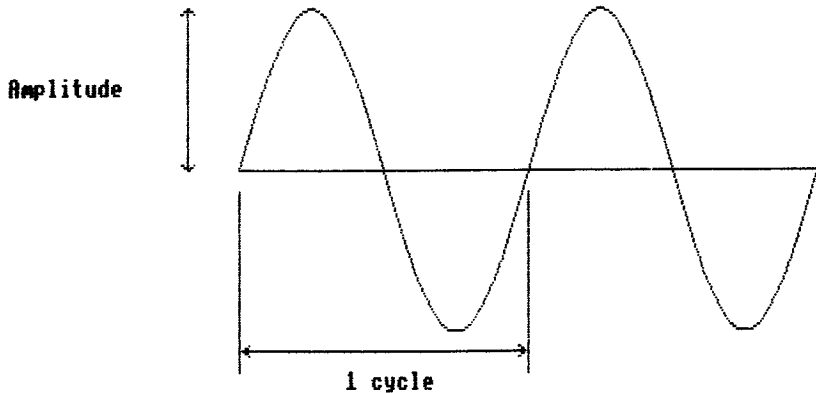


Fig. 9.1. A pure note sound wave showing the cycle and amplitude.

A further characteristic of waves is their *frequency*, which is a measure of the number of complete cycles of the wave that occur at a given point on the pond during a period of one second. The frequency may therefore be quoted in *cycles per second*. In the case of sound and electrical waves a special name is given to the units of frequency, and this is *hertz* (Hz). In effect hertz and cycles per second mean the same thing, so a wave having a frequency of 50 cycles per second can also be referred to as a wave of frequency 50 Hz.

In the case of waves on the water the wave shape is produced by changes in the height of the water as the wave travels across the

pond. In practice the water itself does not move across the pond but is merely displaced up or down as the wave passes by. In the case of sound the wave is transmitted as a change in the air pressure, so that rings of air around the sound source are alternately compressed and expanded as the wave radiates outward. Period, frequency and amplitude still have the same meaning, though. For sound signals, however, the amplitude of the sound is usually referred to as the *volume* and the frequency may be referred to as the *pitch* of the sound.

The human ear responds only to a limited range of sound frequencies with the lower end typically at frequencies of 30 to 50 Hz and the highest audible frequencies from about 10000 to 15000 Hz (10 kHz to 15 kHz). Most of our sound communication, for instance in speech, uses basic frequencies in the range 300 to 3000 Hz, which is the typical frequency range handled by a public telephone system. In music the basic musical notes normally range in frequency from about 50 to 4000 Hz. The higher frequencies may be present in a sound and add character to it as we shall see in a moment. The Commodore 64 sound generator can produce basic sound frequencies in the range 0 to about 4000 Hz.

## The sound generator

All the sounds produced by the Commodore 64 are generated by a special chip known as the 6581 Sound Interface Device or SID chip. This is a very complex chip, but from the point of view of programming it appears as a set of 29 registers, each of which occupies one memory location (as far as the Commodore 64 is concerned). By using PEEK and POKE commands we can manipulate the contents of these registers just as if they were part of the main computer memory. You may wonder what happens to the actual computer memory at the addresses occupied by the sound chip: the answer is that the computer memory is disabled whenever that particular group of addresses is called, so there is no interaction between the sound chip and the main memory.

The sound chip registers occupy the memory addresses from 54272 to 54300. Of the 29 registers seven are allocated to each of the sound channels, thus allowing independent control of each channel. Four registers provide a common control of volume, and allow control of various sound filter characteristics. The remaining four permit signals from the sound generators to be read and used as

control signals. Thus the output from one sound channel may be used to control the characteristics of another sound channel to produce complex sound effects. The addresses and functions of the various registers are shown in Fig. 9.2.

Register	Address	Function
0	54272	Ch.1 Frequency FL
1	54273	Ch.1 Frequency FH
2	54274	Ch.1 Pulse width PWL
3	54275	Ch.1 Pulse width PWH
4	54276	Ch.1 Control
5	54277	Ch.1 Attack/decay
6	54278	Ch.1 Sustain/release
7	54279	Ch.2 Frequency FL
8	54280	Ch.2 Frequency FH
9	54281	Ch.2 Pulse width PWL
10	54282	Ch.2 Pulse width PWH
11	54283	Ch.2 Control
12	54284	Ch.2 Attack/decay
13	54285	Ch.2 Sustain/release
14	54286	Ch.3 Frequency FL
15	54287	Ch.3 Frequency FH
16	54288	Ch.3 Pulse width PWL
17	54289	Ch.3 Pulse width PWH
18	54290	Ch.3 Control
19	54291	Ch.3 Attack/decay
20	54292	Ch.3 Sustain/release
21	54293	Filter frequency low
22	54294	Filter frequency high
23	54295	Filter control
24	54296	Ch.3 mute/Volume
25	54297	Potentiometer X
26	54298	Potentiometer Y
27	54299	Output sig. Ch.3
28	54300	Envelope sig. Ch.3

*Fig. 9.2.* The set of control registers in the SID sound chip with their memory addresses and functions.

## Selecting the frequency

One of the important characteristics of sound is its frequency or pitch. In the sound generator chip this is controlled by POKEing a number into registers 0 and 1 of the desired sound channel. For the moment let us assume we are going to use sound channel 1. The registers controlling frequency for channel 1 are in fact registers 0 and 1 of the sound chip, and correspond to memory locations 54272 and 54273 respectively. To save having to work out actual addresses all the time it is convenient to set a variable SD equal to 54272. Now the address of register 0 is simply SD, and that of register 1 becomes SD+1.

The frequency is selected as a 16-bit number ranging from 0 to 65535, and this gives a corresponding output frequency which ranges from 0 to just under 4 kHz. The actual frequency is related to the number by a simple mathematical formula, as follows:

$$F = 0.06097 * FN \text{ Hz.}$$

where F is the output frequency in Hz and FN is the decimal value of the 16-bit number placed in registers 0 and 1 for the sound channel in use. Thus a number of 10000 gives a frequency

$$F = 0.06097 * 10000 = 609.7 \text{ Hz.}$$

Usually we shall want to calculate the number FN that is required to set a specific frequency F. In this case the equation can be rearranged to give the number that has to be set up in the registers. The equation now becomes

$$FN = F/0.06097$$

Thus for a 1000 Hz tone we need

$$FN = 1000/.06097$$

which gives the number 16402 to be written into the registers.

To insert the number into the sound chip we need to split the 16-bit number into two 8-bit numbers and then POKE one into each register. Let us start by taking the more significant part of the number (the upper 8 bits). This can be found by dividing the number by 256 and then chopping off the fractional part using the INT function, as follows:

$$FH = \text{INT}(FN)$$

where FH is the upper 8-bit part of the number. This can be inserted



into register 1 by using

```
POKE SD+1,FH
```

To get the lower 8 bits of the number (FL) we can simply subtract  $FH*256$  from FN and poke it into register 0 as follows:

```
POKE SD,FN-FH*256
```

To see how this works, and to demonstrate the range of frequencies available, try running the program listed in Fig. 9.3.

```
100 REM RANGE OF SOUND FREQUENCIES
105 REM FROM SOUND CHIP
110 SD=54272
115 REM CLEAR SID REGISTERS
120 FOR N=SD TO SD+24:POKE N,0:NEXT
125 REM SET VOLUME AT MAX
130 POKESD+24,15
135 REM SET SUSTAIN LEVEL
140 POKESD+5,9:POKESD+6,240
145 REM TURN ON SOUND
150 POKESD+4,17
160 FOR F=256 TO 65000 STEP 128
165 REM SELECT FREQUENCY
170 FH=INT(F/256):FL=F-256*FH
175 REM SET FREQUENCY
180 POKESD,FL:POKESD+1,FH
185 REM TONE DURATION DELAY LOOP
190 FOR T=1 TO 100:NEXT
200 NEXT
210 POKESD+4,16
220 END
```

*Fig. 9.3.* Program to demonstrate the range of frequencies produced by the sound generator.

In this program line 120 is a loop to set the internal registers of the sound generator to zero. Note that only 25 of the 29 registers are reset since the last four (registers 26 to 29) are read-only types, and cannot be written into.

Lines 130 to 150 set up some of the other registers within the chip to ensure that we do in fact get sound output. We shall be looking at the operation of these registers a little later.

The tone generation loop starting at line 160 increases the value of F from 256 to 65000 in steps of 128, and for each value of F the corresponding values for FL and FH are calculated and POKEd

into registers 0 and 1. A simple counting loop is then used to give a time delay, and then the loop repeats with the next frequency.

When the program is run a range of tones rising in frequency from a low-pitched buzz to a fairly high-pitched note will be produced.

## Volume control

Register 24 in the sound chip controls the volume of sound produced. Only the four lower bits of the register are used for this purpose, giving sixteen different levels of volume. The control is common to all three sound channels.

For most purposes the upper four bits of this register will be set at zero, so generally, for setting the volume, it is sufficient to POKE a number from 0 to 15 into this register.

The effect of varying the volume is demonstrated by the program listed in Fig. 9.4. Here a sequence of tone frequencies is set up by

```
100 REM DEMO OF SOUND VOLUME CONTROL
110 DIM FL(8),FH(8)
115 REM READ TONE FREQUENCY DATA
120 FOR N=1 TO 8:READ FL(N),FH(N):NEXT
130 DATA 195,16,209,18,31,21,96,22
140 DATA 30,25,49,28,165,31,135,33
150 SD=54272
155 REM CLEAR SID REGISTERS
160 FOR I=SD TO SD+24:POKE I,0:NEXT
165 REM SET VOLUME LEVEL
170 FOR V=1 TO 15:POKESD+24,V
175 REM SET SUSTAIN LEVEL
180 POKESD+5,9:POKESD+6,16*V
190 FOR J=1 TO 8
195 REM SET FREQUENCY
200 POKESD,FL(J):POKESD+1,FH(J)
205 REM SOUND TONE
210 POKESD+4,33
220 FOR T=1 TO 250:NEXT
225 REM GENERATE SILENT PAUSE
230 POKESD+4,32
240 FOR T=1 TO 50:NEXT
250 NEXT
260 FOR T=1 TO 500:NEXT
270 NEXT
```

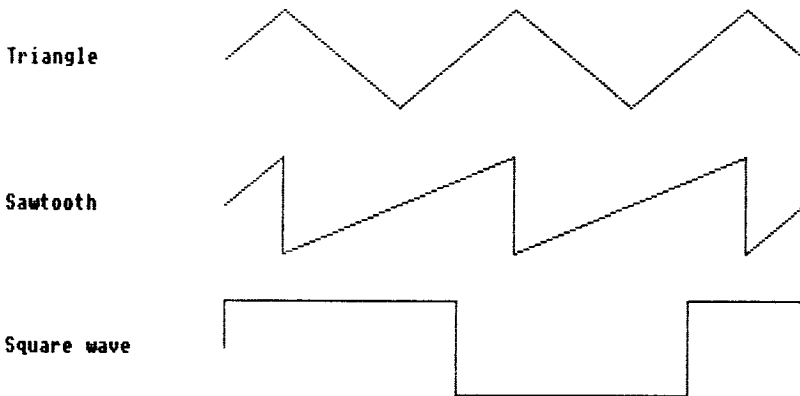
Fig. 9.4. Program demonstrating effect of volume setting.

reading in the values for FH and FL down a data statement. In fact these were chosen to give a musical scale, but any other frequencies would do equally well to demonstrate the effect of volume. Line 18 $\emptyset$  is important, as we shall see later, since it causes the volume level to be maintained while the tone is being generated. Line 21 $\emptyset$  turns on the sound output, and line 23 $\emptyset$  turns it off so the individual tones are separated by a short period of silence.

### Different wave shapes

A sine wave produces a pure tone of just one frequency. In practice it is rather difficult to produce a pure sine wave with electronic circuits, and it is much easier to generate a *square wave* where the signal is switched on for one half of the cycle and off for the other half. The square wave, if analysed, turns out to be a combination of a basic sine wave plus a number of smaller amplitude harmonic signals. These harmonics have frequencies which are multiples of the basic or fundamental note frequency. Thus the second harmonic has twice the frequency, the third harmonic three times the frequency, and so on. The effect of adding these harmonics is to produce a richer sound than the pure tone.

The Commodore 64 can produce three basic waveshapes: triangular, sawtooth and square wave. These are shown in Fig. 9.5.



*Fig. 9.5.* Diagram showing the three basic waveforms produced by the sound chip.

The one we have used in the frequency demonstration is the triangular wave, which is perhaps the closest to the pure sinusoidal waveform. The different output waveforms are selected by the

control register for each sound channel, which is register 4 of each group (SD+4, SD+11 and SD+18).

In the frequency demonstration program the sound is turned on in line 150 by setting bits 0 and 4 of the control register for sound channel 1. Here bit 0 controls whether sound is produced or not, and bit 4 selects the triangular output waveform. To set bit 4 we POKE in the number 16 (17 when bit 0 is also to be set at '1'). If instead of setting bit 4 of the control register for the selected sound channel we set bit 5, and leave bit 4 at 0, a different output waveform is produced by the sound generator. Bit 5 is selected by POKEing 32 into the register (33 if bit 0 is also to be set 'on'). Now the waveform is a sawtooth which rises linearly through the cycle period and then falls rapidly to zero at the end of the cycle. This waveform produces a different sound quality which is richer in harmonics than the triangular wave. The fundamental frequency of the sound is, however, unaffected.

We can select the third basic waveform by setting bit 6 instead of bit 5. This means that we need to POKE 65 into register 4 to turn on the sound, and POKE in 64 to stop the sound. The waveform produced by this action is a square waveform where the signal is

```

100 REM DEMO OF DIFFERENT WAVEFORMS
110 DIM FL(8),FH(8)
120 FOR I=1 TO 8:READ FL(I),FH(I):NEXT
130 DATA 195,16,209,18,31,21,96,22
140 DATA 30,25,49,28,165,31,135,33
150 SD=54272
160 FOR I=SD TO SD+24:POKE I,0:NEXT
170 POKESD+24,15:POKESD+3,8
180 POKESD+5,37:POKESD+6,240
190 W=16
200 FOR WS=1 TO 3
210 FOR J=1 TO 8
220 POKESD,FL(J):POKESD+1,FH(J)
225 REM SELECT OUTPUT WAVEFORM
230 POKESD+4,W+1
240 FOR T=1 TO 300:NEXT
250 POKESD+4,W
260 FOR T=1 TO 50:NEXT
270 NEXT
280 W=W*2
290 NEXT

```

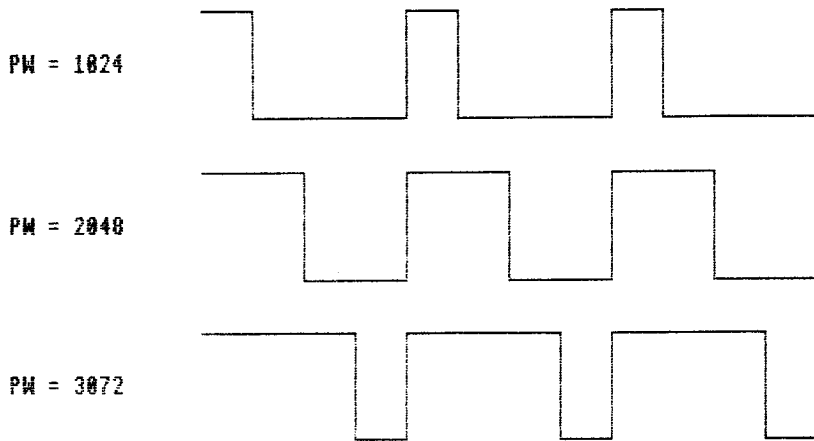
Fig. 9.6. Program showing tone quality produced by the different waveforms.

switched on for one half-cycle and off for the other half-cycle. The result is a louder and much brighter sound, with lots of harmonics. For correct operation of the square wave we also need to POKE the number 8 into register 3 of the sound channel to set up the pulse width. We shall look more closely at this in a moment.

The short program listed in Fig. 9.6 demonstrates the different sound quality produced by the triangular, sawtooth and square waves.

### Pulse width variation

When we produced the square wave output in the program of Fig. 9.6 the 'on' and 'off' periods of the wave were set equal, and each covered one half of the wave cycle. In the Commodore 64 we can in fact control the ratio of the 'on' time relative to the 'off' time to give a wide variety of different wave shapes.. This is shown in Fig. 9.7. Each



*Fig. 9.7.* Diagram showing waveforms with different pulse widths.

of these wave shapes will in fact produce a different quality of sound.

The pulse width – the part of the cycle where the output signal is high – can be varied from zero up to the full cycle period by altering the number contained in registers 2 and 3 of the sound channel in use.

Only 12 of the 16 bits in registers 2 and 3 are actually used to set the pulse width. The upper four bits of register 3 are not used. The result is that pulse width is given by a 12-bit binary number which can have a value from 0 to 4095. For an equal on-off square wave the

value for pulse width should be 2048. This is achieved by setting register 3 to a value of 8 and register 2 to zero.

To see how the variation of pulse width affects the tone quality, try running the program listed in Fig. 9.8. In this program a series of tones is played, and after each sequence the pulse width is increased to give a different tone quality. Altering the pulse width alters the mixture of harmonics in the output sound. The shorter pulses give a rather raspy sort of sound, and longer pulse widths tend to give a hollow sound. These pulse waveforms are similar to those produced by many woodwind instruments.

```

100 REM DEMO OF EFFECT OF PULSE WIDTH
110 DIM FL(8),FH(8)
115 REM READ TONE FREQUENCY DATA
120 FOR N=1 TO 8:READ FL(N),FH(N):NEXT
130 DATA 195,16,209,18,31,21,96,22
140 DATA 30,25,49,28,165,31,135,33
150 SD=54272
155 REM CLEAR SID REGISTERS
160 FOR I=SD TO SD+24:POKE I,0:NEXT
165 REM SET VOLUME
170 POKESD+24,15
175 REM SET ATTACK DECAY SUSTAIN LEVELS
180 POKESD+5,54:POKESD+6,240
190 FOR PW=0 TO 9
200 PRINT "PULSE WIDTH = ",15*(2↑PW)
210 FOR J=1 TO 8
215 REM SET FREQUENCY
220 POKESD,FL(J):POKESD+1,FH(J)
225 REM SET PULSE WIDTH
230 P=15*(2↑PW):POKESD+3,INT(P/256)
240 POKESD+2,P-256*INT(P/256)
245 REM SOUND TONE
250 POKESD+4,65
260 FOR T=1 TO 500:NEXT
265 REM GENERATE SILENT PAUSE
270 POKESD+4,64
280 FOR T=1 TO 50:NEXT
290 NEXT
300 FOR T=1 TO 500:NEXT
310 NEXT

```

Fig. 9.8. Program showing the effect on tone quality of varying the pulse width of a square wave.

## Noise generation

So far the waveforms we have produced from the sound channel have had a regular cyclic variation. We could, however, produce a signal where the instantaneous level of the signal varies randomly with time. This is known as *noise*, and if the variations are truly random the result is what is known as 'white noise'. If we play the noise through a speaker the result is a harsh hissing sound similar to that produced by a television receiver when it is not tuned in to a station.

The Commodore 64 can produce a noise signal if we set bit 7 of the control register and leave bits 4, 5 and 6 at the 0 state. This is done by POKEing 128 or 129 into register 4 of the selected sound channel. The actual noise signal produced by the Commodore 64 is not truly random but will repeat its pattern at intervals, and this produces what is known as 'coloured noise' since it does repeat from time to time. The quality of the noise is also governed by the frequency set up in registers 0 and 1.

It may seem odd that we should want to generate noise at all, but in fact, as we shall see in a moment, noise is an important requirement for many types of sound effect. To get some idea of the sounds produced when a channel is set for noise output try running the program listed in Fig. 9.9. The lower frequencies of noise output produce buzzing sounds which may be useful as sound effects.

```

100 REM DEMO OF NOISE OUTPUT
110 SD=54272
120 FOR I=SD TO SD+24:POKEI,0:NEXT
130 POKESD+24,15:POKESD+6,240
140 POKESD+5,34
150 FOR N=0 TO 15
160 POKESD,64:POKESD+1,N
170 POKESD+4,129
180 FOR T=1 TO 1500:NEXT
190 POKESD+4,128
200 FOR T=1 TO 50:NEXT
210 NEXT

```

Fig. 9.9. Program demonstrating noise output from channel 1.

## The sound envelope

We have seen that the type of sound produced can be altered by

changing the shape of the sound waveform. Much more dramatic effects on the character of the sounds produced can be achieved by controlling the way in which the amplitude of the sound signal varies with time. This variation with time is known as the sound *envelope*. In fact we can have an *amplitude envelope* where the sound volume changes with time and also a *pitch envelope* where the pitch or frequency of the sound varies with time. For the moment we'll consider the amplitude envelope.

In our experiments so far we have simply turned on the sound, held it for a while, and then turned it off. In real life the amplitude of a sound does not always switch off abruptly; it tends to die away gradually. In the same way the sound may build up slowly. Changes in the way that the sound builds up and dies away can completely alter the character of a sound. The Commodore 64 sound generator allows us to control the rates at which the sound output will build up and die away, and by carefully manipulating these rates we can produce a wide range of interesting sound effects.

The pattern of changes in sound volume as a sound is produced falls into four separate phases, which are known as *attack*, *decay*, *sustain* and *release*. These four phases are shown in Fig. 9.10. This type of envelope control is usually referred to as an ADSR envelope system.

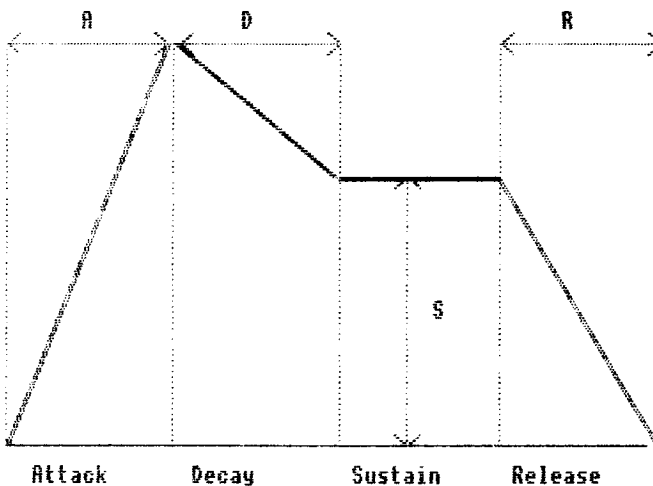


Fig. 9.10. The amplitude envelope showing attack, decay, sustain and release phases.

The attack phase governs the rate at which the sound level builds up to the preset maximum volume level. With a fast or high rate of



attack the volume rises very rapidly to the set level; a slow attack rate causes the level to build up more gradually. In the Commodore sound generator there are sixteen different levels of attack, which are specified in terms of the length of time that the sound will take to build up to a maximum volume. An attack level of 0 gives virtually instantaneous rise of volume; an attack rate of 15 allows the volume to build up over a period of about eight seconds. The attack times for the range of attack rates from 0 to 15 are listed in Fig. 9.11.

<b>ATTACK NUMBER</b>	<b>RISE TIME millisecs</b>
0	2
1	8
2	16
3	24
4	38
5	56
6	68
7	80
8	100
9	250
10	500
11	800
12	1 sec
13	3 secs
14	5 secs
15	8 secs

*Fig. 9.11.* The attack rates available on the SID chip.

Once the volume reaches the preset level the attack phase ends, and the sound generator moves on into the decay phase where the sound level falls again to a second preset level. As with the attack phase, the decay rate is specified by a number from 0 to 15 which selects the length of time over which the sound dies away. Here a decay number of 0 gives the most rapid switch-off, while 15 causes the sound to die away gradually. The decay times are longer than the attack times and are listed in Fig. 9.12.

DECAY/RELEASE NUMBER	FALL TIME milli secs
0	6
1	24
2	48
3	72
4	114
5	168
6	204
7	240
8	300
9	750
10	1.5 secs
11	2.4 secs
12	3 secs
13	9 secs
14	15 secs
15	24 secs

Fig. 9.12. The decay and release rates for the SID chip.

The third phase of the sound envelope is called the sustain phase, and during this period the sound volume remains constant. For this phase the computer does not specify a rate of rise or fall but merely selects a volume level at which the sound will be held. The sustain phase starts when the volume during the decay phase reaches the value specified for the sustain phase. Unlike the attack and decay, the sustain phase continues until the specified sound channel is turned off. The values for the sustain parameter are from 0 to 15, as for volume.

The final phase is the release, which starts when the sound channel is turned off by resetting bit 0 in its control register. The sound now dies away at a rate determined by the number specified for the release rate, which is the same as that for the decay phase. The difference is that in the release phase the volume always decays away to zero.

The attack and decay rate parameters are combined together in a single register which is register number 5 in the group for each sound channel. The register has eight bits, of which the upper four are used for the attack rate and the lower four for the decay rate. To set up this register we simply multiply the attack rate number by 16, to move it to the upper four bits of the word, and then add in the decay rate number and POKE the result into the required register as follows:

```
1000 AD = 16*A + D
1010 POKE SD+5,AD
```

or more simply

```
1000 POKE SD+5,(16*A+D)
```

where A and D are the required rates for attack and decay and are both in the range 0 to 15.

Let us try an experiment using the short program shown in Fig. 9.13. Here the attack rate has been set at 0 to give a very rapid rise to full volume, and the decay rate has been set at 11 to give a very slow decay. In this particular case the sustain level is set at 0 so the sound should decay away to silence rather slowly. The effect produced is very much like a bell or chime provided that the sounds generated are separated by enough time to allow the previous sound to die away.

A better way of producing bell or chime sounds is to make use of the sustain and release phases as well as attack and decay. For the sustain number we can simply use the same number as for volume,

```

100 REM DEMO OF BELL OR CHIME EFFECT
110 DIM FL(8),FH(8)
120 FOR I=1 TO 8:READ FL(I),FH(I):NEXT
130 DATA 135,33,162,37,62,42,193,44
140 DATA 60,50,99,56,75,63,15,67
150 SD=54272
160 FOR I=SD TO SD+24:POKE I,0:NEXT
170 POKE SD+24,15:POKE SD+3,8
180 POKE SD+5,11:POKE SD+6,11
210 FOR J=1 TO 8
220 POKE SD,FL(J):POKE SD+1,FH(J)
225 REM SELECT OUTPUT WAVEFORM
230 POKE SD+4,65
240 FOR T=1 TO 1000:NEXT
250 POKE SD+4,64
260 FOR T=1 TO 50:NEXT
270 NEXT

```

Fig. 9.13. Program to show the production of bell or chime sounds.

but in this case it has to be shifted to the upper four bits of the register, so we need to multiply it by 16 before adding in the release rate number. Now the duration of the sound, which was governed by a simple time delay loop, is made quite short. Here we simply have to allow enough time for the attack phase to be completed. When the sound is turned off, the release phase will start and the sound slowly dies away. If the next tone is started before the release phase ends a reverberation effect similar to that of a bell or chime is produced.

## Drums and explosions

Percussive instruments, such as drums and cymbals, have similar envelope characteristics to those of a bell or chime, since the sound is produced by hitting the instrument. The result is that there is a very rapid attack and a medium to slow decay or release.

When we come to a side drum or a cymbal the required sound is produced by using noise instead of a tone for the sound waveform. For a bass drum or an instrument such as a triangle, however, a tone waveform might be used.

Try running the program listed in Fig. 9.14, which uses noise with a fast attack and slow release envelope to produce sounds which might be used for drum, gunshot or explosion effects according to the frequency of the noise generated. Try experimenting with the

```

100 REM DEMO OF EXPLOSION EFFECTS
110 SD=54272
120 FOR I=SD TO SD+24:POKEI,0:NEXT
130 POKESD+24,15
140 FOR D=7TO13:POKESD+5,D:PRINT"D= ",D
150 FOR N=0 TO 15 STEP 3
160 POKESD,255:POKESD+1,N
170 POKESD+4,129
180 FOR T=1TO1500:NEXT
190 POKESD+4,128
200 FOR T=1TO50:NEXT
210 NEXT
220 NEXT

```

*Fig. 9.14.* Program demonstrating percussion and explosion sounds.

values for attack, decay, release and frequency to see the range of effects that can be produced.

The piano is a rather special case of a percussion instrument, and requires a slightly more complex sound envelope. In this case the attack rises to maximum volume; then the decay falls fairly rapidly to a lower volume of about 60% maximum during the sustain period, after which there is a slow release. Here you need to set the sustain level to about 10 if you have volume set to 15. Try using a slow release and see how that affects the sound produced.

For plucked string instruments the attack is fast but the decay and release times are not as long as for a bell or chime, so the sound dies away fairly rapidly. These instruments may also use the reduced sustain characteristics of a piano. This type of sound is often used for the sound of a ball bouncing off a bat or wall in a game.

For an organ sound a moderate attack rate (say 3 or 4) is used, with similar decay and release rates and sustain at about 75% full volume.

## **Siren sounds**

A type of sound effect which may be useful in a game is the siren. The basic characteristic of this type of sound is that the frequency changes while the sound is being produced. This effect can be achieved by using a counting loop to change the value of the sound frequency. This uses the same general idea as the program listed in Fig. 9.3, except that the sound channel is turned on and left on

throughout the program, and the delay loop within the main frequency change loop is removed. With this arrangement the frequency will increase fairly rapidly through the range. If the whole loop is set to repeat, then the frequency will sweep up through the range then instantly fall and repeat the cycle.

There is another way of controlling the frequency of, say, channel 1 as it produces its sound output. Channel 3, unlike the others, has a facility by which its output can be read from register number 27. This gives a number which varies in the range 0 to 255 and follows the wave shape being generated by oscillator 3.

If we set oscillator 3 to a triangle wave, then PEEK register 27 and add the result to the frequency number for channel 1, the channel 1 frequency will rise and fall in sympathy with the output from channel 3. This produces a siren-like sound from channel 1 as demonstrated by the program listed in Fig. 9.15. Try changing the

```

100 REM SIREN TYPE SOUNDS
110 SD=54272
120 FOR N=SD TO SD+24:POKEN,0:NEXT
130 POKESD+24,15
140 POKESD+5,17:POKESD+6,246
150 FOR K=16 TO 255 STEP 16
160 POKESD+19,17:POKESD+20,246
170 POKESD+14,K:POKESD+15,0
180 POKESD+18,17:POKESD+4,33
190 FOR J=1 TO 100
200 F=8000+4*(PEEK(SD+27))
210 FH=INT(F/256):FL=F-256*FH
220 POKESD,FL:POKESD+1,FH
230 NEXT J
240 NEXT K
250 POKESD+18,16:POKESD+4,32
260 END

```

Fig. 9.15. Program for producing siren sounds using the output of channel 3 to control channel 1.

waveform produced by channel 3 to a sawtooth wave by altering the number POKEd to SD+18 from 17 to 33 in line 180. This will produce different siren-type effects. Another experiment to try is subtracting the PEEK of register SD+27 from the frequency number instead of adding it.

Increasing the frequency of channel 3 will alter the siren sound, giving a vibrato or tremolo effect where the tone frequency varies

rapidly over a small range. This can be combined with an organ envelope to give a vibraphone effect.

It is also possible to read out the envelope of channel 3 by PEEKing register 28. This means that you can apply an ADSR-type variation to the frequency of, say, channel 1 to get even more varied effects. To see the technique here try running the program listed in Fig. 9.16. Here the frequency of channel 1 is being altered by the

```

100 REM BOMB DROP SOUND
110 SD=54272
120 FOR N=SD TO SD+24:POKE N,0:NEXT
130 POKE SD+24,136
140 POKE SD+5,17:POKE SD+6,246
150 POKE SD+19,10:POKE SD+20,0
170 POKE SD+14,1:POKE SD+15,4
180 POKE SD+18,33:POKE SD+4,33
190 E=PEEK(SD+28):IF E<240 THEN 190
200 F=24000+64*E
210 FH=INT(F/256):FL=F-256*FH
220 POKE SD,FL:POKE SD+1,FH
230 E=PEEK(SD+28):IF E>32 THEN 200
250 POKE SD+18,16:POKE SD+4,32
260 POKE SD+5,11:POKE SD+6,9
270 POKE SD,1:POKE SD+1,3
280 POKE SD+24,15:POKE SD+4,129
290 FOR T=1 TO 1000:NEXT
300 POKE SD+4,128
310 FOR T=1 TO 1500:NEXT:GOTO 130
320 END

```

*Fig. 9.16.* Bomb drop effect using channel 3 envelope to control channel 1 frequency.

envelope of channel 3 to produce a descending note which might be used to simulate the sound of a falling bomb. At the end of the envelope from channel 3 an explosion sound is generated by channel 1 to complete the effect. This type of frequency control might be used to produce the so-called laser gun sounds for games.

When channel 3 is used to control another channel, either by using its output or envelope signals, we will usually want to keep the sound from channel 3 turned off. The output must be set 'on' for the desired waveform in register 18, otherwise no output will appear from registers 27 and 28. The actual sound output from channel 3 can, however, be switched independently by setting bit 7 of register 24. This is the volume register, and all we have to do is add 128 to the

required volume number and POKE the result into register 24. In the program of Fig. 9.16 this is carried out in line 130. In the siren program of Fig. 9.15 channel 3 was left on, since its frequency was set very low and does not produce an effective sound. If desired, line 130 in this program could be made the same as in Fig. 9.16.

The possible combinations of the various waveforms, envelope shapes and varying degrees of control of frequency via the channel 3 signals are almost limitless. Some of the possibilities have been suggested here, but much of the fun in playing with the sound facility will come from trying various combinations and noting interesting effects when they occur. When you get an effect something like one that you want, try small variations and gradually alter it until it sounds right.



# Chapter Ten

## Making Music

So far we have looked at the generation of various kinds of sound effects using the Commodore 64, but a more interesting application of the computer is in producing music. This may be just to provide simple jingles in a game program (to indicate victory or defeat) or it might be extended to make the computer play a piece of music. Another application could turn the Commodore 64 into a musical instrument that is played by using the keyboard in much the same way as the keyboard of a piano or organ. In order to understand how the computer is programmed for these applications we shall need to learn a little about the principles of music itself.

### The musical scale

Sound can have an enormous number of different frequencies within the range 30 Hz to 10000 Hz. In music, this vast range of sounds is reduced to a series of specific frequencies which are called *tones* or *notes*. The actual set of notes used for producing music depends upon the cultural background of the musicians: the set of notes used by a musician in a western country would be different from those used in Asian or Chinese music. As a result, western music sounds different from that of the eastern countries of Asia. For the moment we shall consider the musical scheme used in western countries but, of course, the principles could equally well be applied to eastern music.

In western music there are seven named notes, referred to by the letters A to G, and these correspond to a group of seven successive white notes on a piano keyboard. However, the full range of notes used in music consists of more than seven notes. For those notes beyond the first seven in the series the letters A to G are repeated, so that the note following the G is once again labelled A. This process

continues up the range with the letters A to G being repeated for each set of seven notes. Each group of notes running from A to G is called an *octave*. This name comes from the fact that the A note in one octave is the eighth note above the corresponding A in the octave below. In order to distinguish a note such as B in one octave from the B notes in other octaves a number may be written after the note letter to show which octave the note is in.

The set of frequencies used for the musical notes is arranged so that the frequency of the A note in one octave has approximately twice the frequency of the A in the octave below. This two-to-one frequency ratio also applies to other notes when they are compared with the equivalent note in the octave below.

Each octave is actually divided up into a series of 12 intervals called *semitones*. The pitch or frequency ratio between successive semitones is the 12th root of two, or about 1.06 to 1. Five of the so called 'natural' notes (A to G) are spaced two semitones, or one complete tone, apart. With only 12 semitones available in an octave, two pairs of notes (B,C and E,F) are spaced by only one semitone. This may seem rather an odd scheme, but it works quite well in practice. On a piano keyboard there are five black notes in each octave, and these occupy the semitone positions between those white notes that are spaced a full tone apart. Figure 10.1 shows a section of a piano keyboard with the notes labelled. The black notes

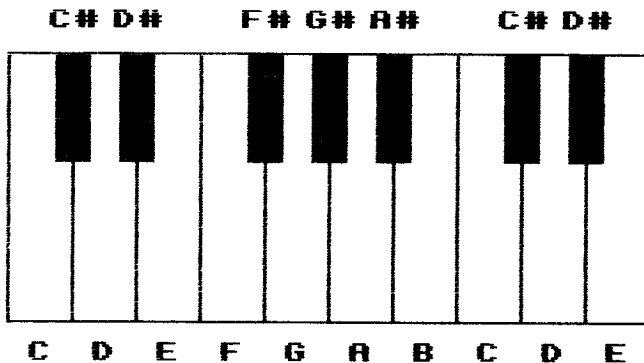


Fig. 10.1. The layout of notes on a piano keyboard.

are referred to as 'sharp' notes and are indicated by placing a # sign after the normal note letter. Thus A sharp is one semitone above note A and would be written as A#.

Sometimes in music there will be references to 'flat' notes. The flat notes are really the same as the sharps, but are simply referred to the

NOTE	NUMBER	FH	FL
C-1	2145	8	97
C#-1	2273	8	225
D-1	2408	9	104
D#-1	2551	9	247
E-1	2703	10	143
F-1	2864	11	48
F#-1	3034	11	218
G-1	3215	12	143
G#-1	3406	13	78
A-1	3608	14	24
A#-1	3823	14	239
B-1	4050	15	210
MIDDLE C			
C 0	4291	16	195
C# 0	4547	17	195
D 0	4817	18	209
D# 0	5103	19	239
E 0	5407	21	31
F 0	5728	22	96
F# 0	6069	23	181
G 0	6430	25	30
G# 0	6812	26	156
A 0	7217	28	49
A# 0	7647	29	223
B 0	8101	31	165
C 1	8583	33	135
C# 1	9094	35	134
D 1	9634	37	162
D# 1	10207	39	223
E 1	10814	42	62
F 1	11457	44	193
F# 1	12139	47	107
G 1	12860	50	60
G# 1	13625	53	57
A 1	14435	56	99
A# 1	15294	59	190
B 1	16203	63	75
C 2	17167	67	15

*Fig. 10.2.* Table of note frequency values and FL and FH values for the sound generator.

next higher natural note rather than the next lower. Thus a B flat note is one semitone below B and is exactly the same as an A sharp

which is one semitone above A. Remember that B is a whole tone above A. In music, flat notes are denoted by using a symbol like a small letter b after the note letter.

Western musical scales are tuned relative to a standard frequency of 440 Hz, and this is an A note. The A note one octave lower has a frequency of 220 Hz and the A note one octave higher is 880 Hz. Other note frequencies are then calculated relative to these, using the 12th root of two ratio for each semitone. Another reference note in the musical scale is called 'Middle C' and this is the C note in the octave below the standard pitch A note. The frequency of middle C is about 261 Hz. One reason for choosing middle C as a reference is that the musical scale starting with C uses only the white keys on a piano. A musical scale is simply a succession of notes covering an octave. Scales starting from other notes than C follow the same sequence of tones and semitones between successive notes, and will contain one or more sharp notes to make the scale sound right.

To produce the musical notes on the Commodore 64 we need to know their frequency numbers. The actual frequencies of musical notes are not nice round figures, and in some cases it is not possible to select the exact musical frequency on the SID chip, but the result is close enough not to be noticed. To save you the trouble of calculating the frequency numbers these are given in Fig. 10.2, for the range of notes from the C below middle C to the C two octaves above middle C. This figure also lists the FH and FL values required for each note.

### Written music

Although music could be written by using strings of letters, actual written music uses a more graphical scheme. The notes themselves are shown as large dots, and these are drawn on or between a set of

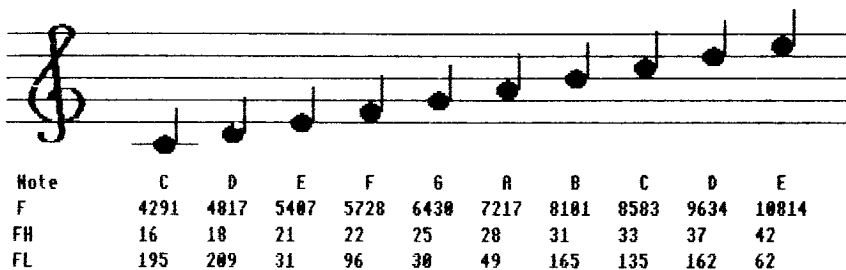


Fig. 10.3. The treble staff showing the frequency values for the treble notes.

five parallel horizontal lines called a *music stave*. This is shown in Fig. 10.3, where the sequence of notes from middle C up to the E in the next higher octave is shown. Middle C is the note below the stave which sits on its own short horizontal line. The frequency numbers and the values for FL and FH are also given for the range of notes shown.

The stave shown in Fig. 10.3 is known as the *treble stave*, and this is indicated by the *treble clef* symbol at the start or left-hand side of the stave. For notes below middle C there is another stave, called the *bass stave*, and this is shown in Fig. 10.4. Here a different symbol, known as the *bass clef*, appears at the start of the stave to identify it. Again the frequency numbers and FL, FH values are given for the notes shown.



Note	C	B	A	G	F	E	D	C
F	2145	2408	2703	2864	3215	3608	4050	4291
FH	8	9	10	11	12	14	15	16
FL	97	104	143	48	143	24	210	195

Fig. 10.4. The bass stave and its associated frequency values.

The natural notes are placed alternately on or between the lines of the stave, and the vertical position of the dot representing a note defines exactly which note it is. Sharp or flat notes are shown by placing a sharp or flat symbol before the note, as we did when writing the notes as letters. For notes that lie above the top line of the stave, one or more short stave lines are drawn above the top line of the stave in the same way as for the Middle C note below the stave.

Sometimes the music may require that a note on a particular line of the stave is always played as a sharp note. In such cases, to avoid having a sharp sign before every note on that line, the sharp sign is placed after the clef symbol at the start of the line of music, as shown in Fig. 10.5. When this is done the note symbols on that line of the stave are drawn normally, but are interpreted as sharp notes when the music is played. The same basic principle may also be used for flat notes by placing flat symbols after the clef symbol on the row of the stave containing the flat note.

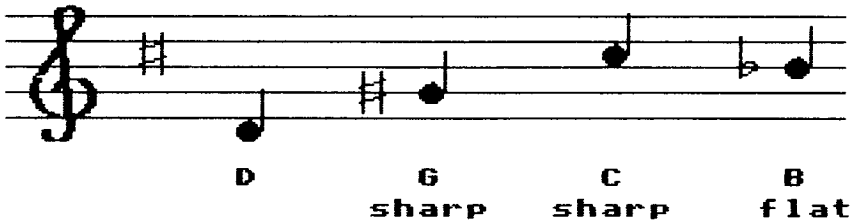


Fig. 10.5. The sharp and flat notes.

## Musical timing

Although a sequence of different notes will produce a tune, much of the character sound of music comes from the *tempo* and *rhythm* of the sounds, as well as their pitch or frequency. Tempo is the rate at which the tune is played; an average tempo might be about 150 beats per minute. The rhythm is produced by a pattern of varying note duration, and by inserting quiet pauses between some of the successive notes making up the tune. By varying tempo and rhythm the same sequence of note frequencies can be made to sound completely different.

The duration of notes in music is organised on a binary system. The basic note length is called a *crotchet*, and typically corresponds to a duration of about 0.4 seconds. The crotchet is shown as a black filled circle with a vertical tail. Usually the tail is drawn upwards at the right side of a note, but it may also be drawn downwards at the left side if required. Shorter notes than the crotchet are the *quaver*, which has half the duration of a crotchet, the *semiquaver* (1/4) and *demi-semiquaver* (1/8). These shorter notes are drawn like crotchets, but have one, two, or three ticks on the tail respectively. The *minim* is a longer note with twice the duration of a crotchet. It is drawn like a crotchet, but the circle is not filled in. Finally there is the *semibreve* which is four times as long as a crotchet and is shown with no tail. These note symbols are shown in Fig. 10.6.

In the Commodore 64 the note length is determined by using a simple counting loop to produce a delay between the time the sound output is turned on and the time it is turned off to end the note. This is done as follows:

```
220 FOR T=1 TO DR:NEXT
```

where DR is a number that determines the length of the note. For a crotchet, the value for DR will be 250, and the other note lengths are







SYMBOL	NAME	RATIO	DR
	Demi-semi-quaver	1/8	31
	Semi-quaver	1/4	63
	Quaver	1/2	125
	Crotchet	1	250
	Minim	2	500
	Semibreve	4	1000

Fig. 10.6 Note symbols and duration count values used in creating music.

directly proportional to this: a quaver has a DR value of 125, and so on. The required count durations for the various note lengths are indicated in Fig. 10.6.

Now we can apply these new duration values together with a set of note frequencies to produce a simple tune-generating program, as shown in the listing of Fig. 10.7. In this program three data arrays are used; one gives the note duration and the others are used for the FH and FL values for each note. Alternatively you could use one array to store duration and a second for the frequency numbers (FN). In this case the FH and FL terms would be derived by splitting FN into two bytes, as shown in Chapter 9.

The binary series of note durations does not always satisfy the needs of composers and musicians so sometimes, in written music, you may also come across notes with a dot alongside the note symbol as shown in Fig. 10.8. These notes have the duration increased by half. Thus a dotted crotchet would have an effective duration loop count of 375 instead of the normal 250.

## Rests and tempo control

So far in playing a tune the varying length notes have been played one after another with perhaps a very brief pause between them as the next note is set up. In the rhythms of actual music, however,

```

100 REM TUNE PLAYING PROGRAM
110 SD=54272
120 FORN=SD TO SD+24:POKEN,0:NEXT
130 POKESD+24,15
140 POKESD+5,34:POKESD+6,240
150 FOR J=1 TO 26
160 READ D,FH,FL
170 POKESD,FL:POKESD+1,FH
180 POKESD+4,33
190 FOR T=1 TO D:NEXT
200 POKESD+4,32
210 FOR T=1 TO 50:NEXT
220 NEXT
230 END
235 REM DURATION AND NOTE DATA
240 DATA 250,31,165,125,37,162
250 DATA 250,31,165,250,31,165
260 DATA 125,37,162,250,31,165
270 DATA 125,33,135,125,37,162
280 DATA 125,33,135,250,28,49
290 DATA 125,31,165,125,33,135
300 DATA 125,31,165,250,25,30
310 DATA 250,31,165,125,37,162
320 DATA 250,31,165,250,31,165
330 DATA 125,37,162,250,31,165
340 DATA 125,33,135,125,37,162
350 DATA 125,33,135,250,28,49
360 DATA 125,31,165,250,25,30

```

READY.







Fig. 10.7. A program to play a simple tune.

there are frequently deliberate silent periods between notes which act a little like written punctuation marks to separate phrases in the music.

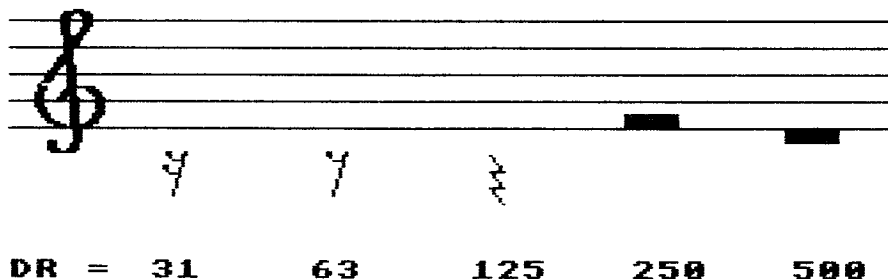
These silent periods are called *rests* or *pauses* and they, too, have a duration structure similar to that of the notes. The symbols used to denote the rests and their duration times are shown in Fig. 10.9.

To incorporate rests into a music program we could set up a separate array for them, but for most of the time the data in this array would be zero. Another possibility might be to make the frequency value zero when a rest is required. A simple test on the F value from the tune data stream will then tell the computer what to do. If the F value is greater than zero the note is played, but if  $F = 0$



SYMBOL	NAME	RATIO	DR
	Demi-semi-quaver dotted	1/8	46
	Semi-quaver dotted	1/4	94
	Quaver dotted	1/2	188
	Crotchet dotted	1	375
	Minim dotted	2	750
	Semibreve dotted	4	1500

*Fig. 10.8.* The dotted notes and their duration values.



*Fig. 10.9.* The rests and their duration values.

then the computer just counts off a period of silence equal to the note duration specified.

Another approach to the playing of rests might be to make the duration number negative when a rest is required. When a negative duration value is detected by testing DR the program would turn off the sound generator and then perform a delay using the ABS value of the duration number.

Changing the tempo of the music is equivalent to changing the duration of the crotchet on which the other note lengths are based. One way of handling this might be to make the note duration number (ND) for a crotchet equal to 1, and then the other note lengths would have durations based on this. Now a multiplier factor (TM) can be set up to determine the tempo of the music, and this might be set as 250 at the start of the program. To get the duration

number (DR) for the sound generating loop the note duration number (ND) is multiplied by the tempo number (TM), so that for a crotchet we would get

$$DR = ND \times TM = 1 \times 250 = 250$$

and for a quaver the sound duration would be,

$$DR = ND \times TM = 0.5 \times 250 = 125$$

This gives the same count durations as we had before when the tempo number is set at 250. If the tempo multiplier is reduced, then the whole tune will be played faster, and if the tempo number is increased the tune will be played at a slower rate. To avoid fractions in the note duration numbers these could be based on a demi-semiquaver (the shortest note) having an ND value of 1, so that a crotchet would then have a value of 8. Here the normal value for the tempo multiplier would have to be set at 31 instead of 250.

## **Play the Commodore 64**

So far we have produced musical tunes by setting up the sequence of notes and their durations and then using a loop to read the note data and generate the required sound. There is another, rather more attractive, possibility for producing music with the Commodore 64. Instead of feeding in prearranged music data it is possible to turn the computer into a playable instrument, where the music is produced by pressing keys on the keyboard in much the same way as one might play a piano or organ.

To make the Commodore 64 keyboard act in a similar way to the keyboard of a piano the various musical notes to be played must be allocated to individual keys on the computer keyboard. When one of these keys is pressed the corresponding note will be played by the Commodore 64.

The most convenient arrangement would be to choose the keys on the computer so that they are in roughly the same physical layout as the black and white keys on a piano. This is done by using the middle row of letter keys from A to ; as the white piano keys and using the W, E, T, Y, U, O and P keys for the black piano keys. Figure 10.10 shows the keys used and their relationship to the keys on a piano keyboard.

When we come to playing music on the computer, the first step will be to detect which key is being pressed so that the appropriate

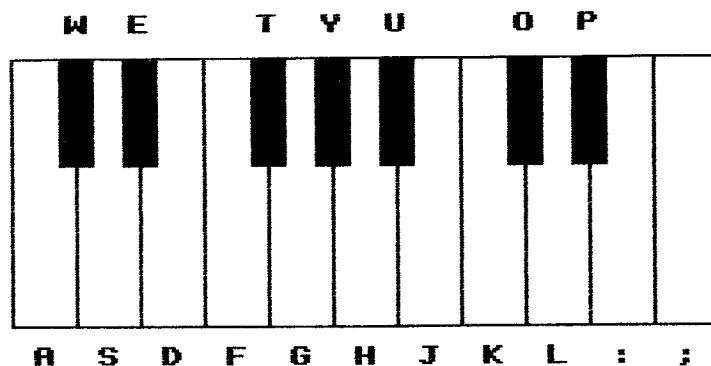


Fig. 10.10. Layout of keyboard and notes for using the Commodore 64 as a musical instrument.

note can be produced. One way of detecting which key has been pressed is to use the GET statement. This could be done using the following statement:

```
2000 GET A$:IF A$="" THEN 2000
```

Here if no key is pressed A\$ is a blank string (""), and the instruction loops continuously. If the S key were pressed then A\$ would become a string "S" and the program would move on to the next instruction. While GET will detect which key is pressed, it is not particularly suited to our purpose. This is because the Commodore stores the results of successive keypresses in an input buffer memory. This buffer can store up to 10 successive keystrokes, and each time GET is called it will read out one of the values from the buffer. As a result the key data that you are processing may not be for the key that is currently being pressed but for one that was pressed a few moments earlier.

The keys themselves are arranged in a matrix with eight rows and eight keys in each row. The rows and columns are scanned continuously by the computer, and when a key is detected as 'on' its code number is set up in memory location 197. The code number identifies the row and column position of the keyboard scan when a contact was detected, and therefore identifies one particular key. If we read the contents of location 197 they will indicate which key is being pressed at that instant in time. If no key is being pressed, the number in location 197 is set at 64. Now to detect a key press we can use the statement

```
2000 K=PEEK(197):IF K=64 THEN 2000
```

When no key is being pressed the statement just loops back to itself

KEY	NUMBER	KEY	NUMBER
A	10	W	9
B	28	X	23
C	20	Y	25
D	18	Z	12
E	14	1	56
F	21	2	59
G	26	3	8
H	29	4	11
I	33	5	16
J	34	6	19
K	37	7	24
L	42	8	27
M	36	9	32
N	39	0	35
O	38	@	46
P	41	*	49
Q	62	=	53
R	17	/	55
S	13	+	40
T	22	-	43
U	30	;	50
V	31	:	45

No key pressed gives the number 64.

Fig. 10.11. Values returned by keys to location 197 in memory.

since K will be 64. When a key is pressed the test fails, and the program execution moves on to the next statement line where we would start a piece of program to discover which key is being pressed.

The actual number values returned in location 197 by the various keys on the keyboard are shown in Fig. 10.11. As you will see, they do not conform to any convenient pattern which would make key decoding simple. To deal with this we can set up an array KV into which the numbers corresponding to the keys we want to detect are placed. Here it is convenient to arrange the set of key numbers in the same order as the musical notes that we want the keys to produce. Now when a key press is detected we can use a simple loop to

```

100 REM COMMODORE64 MUSICAL INSTRUMENT
110 DIM KV(18),FK(18),FL(18)
115 REM READ KEY AND FREQUENCY DATA
120 FOR N=1 TO 18
130 READ KV(N),FK(N),FL(N):NEXT
140 DATA 10,16,195,9,17,195,13,18,209
150 DATA 14,19,239,18,21,31,21,22,96
160 DATA 22,23,181,26,25,30,25,26,156
170 DATA 29,28,49,30,29,223,34,31,165
180 DATA 37,33,135,38,35,134,42,37,162
190 DATA 41,39,223,45,42,62,50,44,193
195 REM SET UP SOUND CHIP
200 SD=54272
210 FOR N=SD TO SD+24:POKE N,0:NEXT
220 POKE SD+24,15
230 POKE SD+5,17:POKE SD+6,246
235 REM READ KEYBOARD
240 K=PEEK(197):IFK<>64 THEN 260
245 REM STOP SOUND OUTPUT
250 POKE SD+4,32:GOTO240
255 REM FIND REQUIRED FREQUENCY
260 FORJ=1TO18:IFK=KV(J) THEN 260
270 NEXT:GOTO240
275 REM SET FREQUENCY
280 POKE SD,FL(J):POKE SD+1,FK(J)
285 REM START SOUND OUTPUT
290 POKE SD+4,33:GOTO240
300 END

```

*Fig. 10.12.* Program to turn the Commodore 64 into a playable musical instrument.

compare the value **K** with each of the values of **KV** in turn until a match is found.

We can set up a second data array, **F**, which is of the same size as array **KV**, and use it to hold the note frequency numbers for the notes we want to produce. Now when a match occurs in the key testing loop we can read the frequency number corresponding to the key that has been pressed and use this to set up the sound frequency. If you want to use the **FH** and **FL** values to set up the sound channel frequency then two arrays (**FH** and **FL**) will be needed as well as the **KV** array.

The program for converting the Commodore 64 into a musical instrument is listed in Fig. 10.12. This allows notes from the **C** above Middle **C** and the next octave and a quarter to be played. The timing of the music is now directly related to the way in which the keyboard is operated, since each note will be produced for as long as the key is held down, just as in an electronic organ. Thus the rhythm and tempo are governed primarily by the player.

In lines 120 and 130 the data arrays for **KV** (key value), **FH** and **FL** for the range of notes are set up from the data in lines 140 to 190.

In line 240 the keys are checked, and if no key is pressed (line 250) the sound output is turned off and the program loops back to check the keyboard again. When a key is detected the value of **K** from location 197 is checked against **KV** values in a loop, and if no match occurs the program loops back to the keyboard test again. When **K** does match a **KV** value the program jumps out of the comparison loop to line 280, the required values for **FH** and **FL** are set up in the sound chip registers, and the sound output is turned on.

This process of checking the keyboard and updating the state of the sound chip goes on continuously. If you hold down a key the note will continue to be produced until the key is released, just as it would on an organ.

In the program the sawtooth waveform has been selected. You could, of course, change the number **POKEd** into register 4 to select the triangular or square wave outputs as desired. Another possibility would be to allocate three other keys, such as number keys, to control the output waveform. These keys could be detected in the same way as the other keys used for the notes. In this case, however, instead of reading in frequency values the appropriate numbers would be **POKEd** into register 4 to turn on the required waveform.

The range of frequencies covered might be extended by building in an octave switching mechanism. Two or three keys might be allocated to selecting the octave. When one of these is pressed the

frequency range would be shifted to the specified octave range. For this type of scheme it may be better to work with complete frequency numbers rather than FH and FL numbers when selecting note frequency. The advantage here is that to move up an octave you simply have to double the frequency number, and to move down an octave the frequency number is divided by two. The values of FL and FH to go in the registers are easily calculated using the techniques described in Chapter 9.

Another possible development of the musical instrument scheme might be to have several different envelopes programmed, so that by selecting one of the number keys the sound envelope could be changed to give a different type of sound. This could range from a bell sound to organ or piano sounds. There are lots of possibilities for development here, and much of the fun of playing with the sound generator will be derived from experimenting with various combinations to discover the enormous variety of sounds that can be produced.

# Index

- amplitude of wave, 158
- animation, 82
- animation with shape changes, 113
- ASCII code, 14
- attack phase, 169
- attack rate, 170
  
- background colour, 1, 67, 75
- ball movement, 83
- bar chart, 125
- bar chart with solid bars, 145
- bar charts, three axis, 141
- bar gauge, 120
- bass clef, 182
- bass stave, 182
- bell sound, 172
- bit map memory, 31
- bit map memory layout, 35
- bit map mode, 9, 31
- bit map mode selection, 33
- bit map screen clear, 33
- bit map screen layout, 35
- bit map text display, 137
- bit mapped graphics, 8
- border colour, 69
- bouncing ball, 85
  
- character code, 5
- character generator, 4, 24
- character graphics, 5
- character set, 3, 4
- chime sound, 172
- CHR\$ codes for colours, 66
- circles by rotation method, 55
- circles by squares method, 48
- circles by trigonometry, 48
- circular three-axis plot, 148
- CLR key, 15
- collision detection, 88, 112
- colour ASCII codes, 65
- colour code, 11, 62
- colour code listing symbols, 62
- colour control registers, 58
- colour memory, 1, 63
- colour patterns, 70
- copying a symbol set, 26
- crotchet, 183
- CTRL key, 24, 61
- cursor keys, 15
- cursor position, 19, 20
  
- decay phase, 168
- decay rate, 170
- degrees, 53
- demi-semiquaver, 183
- dotted notes, 184
- drum type sounds, 173
  
- explosion effects, 173
- extended background colour, 75
  
- flat note, 179
- frequency of sounds, 158, 161
- frequency registers, 161
  
- graph axes, 131
- graph, interpolated, 135
- graph scales, 132
- graphics symbols, 6, 23
- graphs, 131
  
- high resolution graphics, 8, 31
- HOME key, 14
  
- interlaced scanning, 2
- interpolation, 135
  
- keyboard decoding, 189
- keyboard interrupt, 26



- line clipping, 42
- line drawing routine, 39
- line symbols, 21, 22
- LOGO key, 23, 62
- low resolution graphics, 7
  
- machine code screen clear, 37
- meter displays, 118
- minimum, 183
- mirror image patterns, 70
- mosaic graphics, 7
- movement of objects, 83
- moving ball, 83
- moving landscape, 93
- multicolour bit map mode, 78
- multicolour mode, 11, 77, 78, 109
- multicolour symbols, 77
- multiple bar chart, 127
- musical note frequencies, 180
- musical notes, 178
- musical scale, 178
- musical timing, 183
  
- noise signal, 168
  
- octave, 179
  
- perspective views, 150
- piano keyboard, 179
- plotting points, 34
- polygon drawing methods, 58
- polygon, regular, 58
- PRINT AT action, 17
- programming symbols, 27
- pulse width control, 166
  
- quaver, 183
  
- radians, 53
- Read Only Memory, 4, 25
- rectangles, 46
- release phase, 169
- release rate, 170, 172
- RESTORE key, 33
- rests in music, 184
- reverse video, 23
- ROM enable, 26
- rotation equations, 55
- RUN/STOP key, 33
- RVS ON/OFF commands, 24
  
- sawtooth wave, 164
- screen code, 29
- screen edge clipping, 42
- screen memory, 5, 14
- screen wraparound, 43
- scroll down routine, 94
- scroll left routine, 93
- scroll right routine, 91
- scrolling the screen, 89
- semibreve, 183
- semiquaver, 183
- semitone, 179
- setting bit map dots, 34
- sharp note, 179
- SHIFT key, 15, 23, 61
- SID chip, 13, 154
- SID registers, 160
- siren effect, 174
- sound envelope, 168
- Sound Interface Device, 13, 154
- sound pitch, 159
- sound waveforms, 164
- sound waves, 158
- sprite animation, 111
- sprite collision detection, 112
- sprite colour control, 108
- sprite control registers, 103
- sprite definitions, 99
- sprite dot matrix, 98
- sprite expansion, 110
- sprite graphics, 11, 98
- sprite, multicolour, 109
- sprite on/off control, 103
- sprite pointers, 100
- sprite position registers, 104
- sprite priority, 110
- square wave, 164
- sustain level, 172
- sustain phase, 169
- symbol dot matrix, 3, 27
- symbol space, 3
- SYS command, 39
  
- tempo, 184
- text colour, 61
- text cursor, 14
- text display, 3, 5
- text on bit map screen, 137
- thermometer display, 120
- three axis bar charts, 141
- three axis graphs, 140
- treble clef, 182
- treble stave, 182
- triangle drawing, 44
- triangular wave, 164
  
- user defined graphics, 8, 25, 28

VIC chip, 1, 5, 12

video display, 1

Video Interface Chip, 1, 5, 12

volume, 159, 163

volume control register, 163

wallpaper program, 71

wave period, 158

wave shapes for sound, 164

white noise, 168

written music, 181

