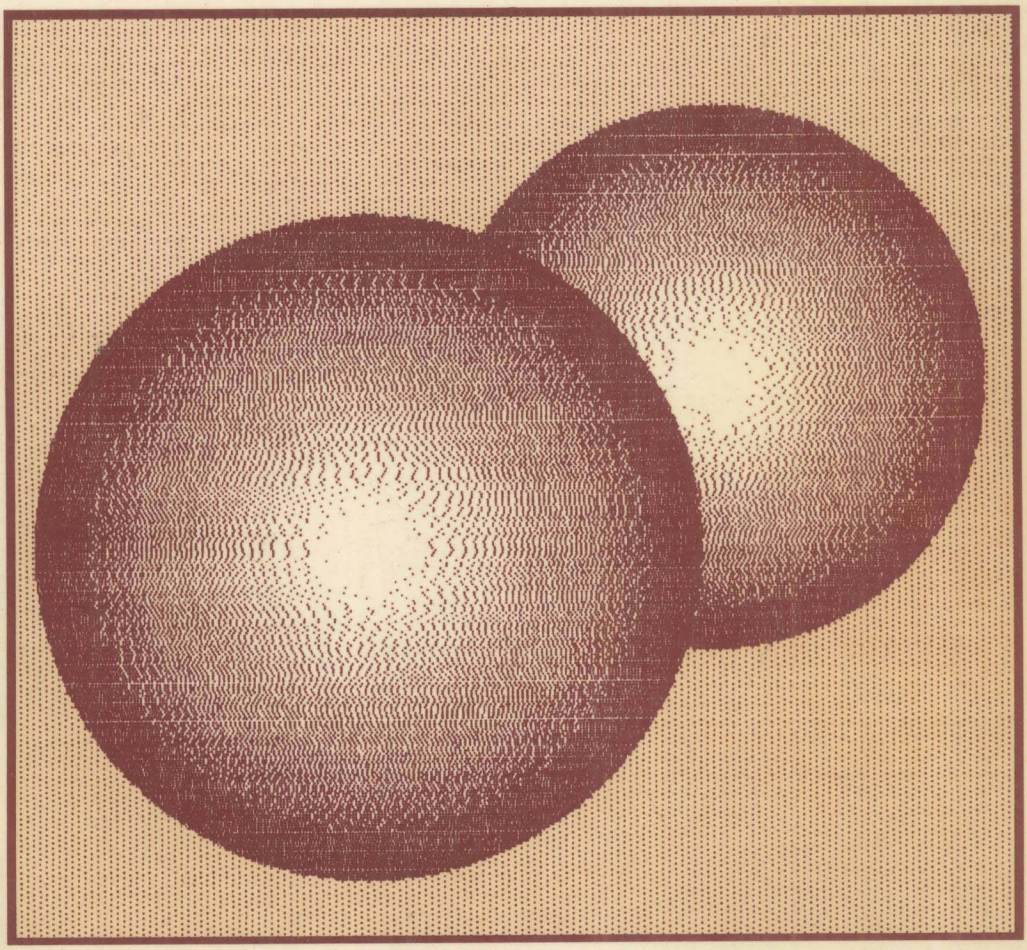
COMPUTER GRAPHICS FOR THE IBM PC

LEENDERT AMMERAAL



Computer Graphics for the IBM PC

Leendert Ammeraal

Hogeschool Utrecht
The Netherlands

JOHN WILEY & SONS
Chichester • New York • Brisbane • Toronto • Singapore

Reprinted February 1988

Library of Congress Cataloging-in-Publication Data:
Ammeraal, L. (Leendert)
Computer graphics for the IBM PC.
Bibliography: p.
Includes index.
1. IBM Personal Computer—Programming.
2. Computer graphics. I. Title.
QA76.8.I2594A48 1987 006.6'765 87-2024
ISBN 0 471 91501 7

British Library Cataloguing in Publication Data:
Ammeraal, L.
Computer graphics for the IBM PC.
1. Computer graphics 2. IBM Personal
Computer—Programming
I. Title
006.6 T385
ISBN 0 471 91501 7

Printed and bound in Great Britain by Anchor Brendon Limited, Tiptree, Essex

Preface

In my earlier book *Programming Principles in Computer Graphics* I used the four primitive routines *initgr, move, draw, endgr,* and I assumed that anyone who wishes to use these routines on a particular computer could easily make them available. In computer graphics, we should distinguish at least two levels of abstraction, or, in other words, two software layers. At the lower level some convenient but elementary routines, such as those mentioned, are implemented, and at the higher level, they are simply used. In my earlier book on graphics I focussed on the higher level, and, frankly speaking, the lower level then seemed less interesting to me. When that book was printed, the publisher asked me to supply its programs on a diskette for the IBM PC, and doing so, I had to deal with that lower level as well. I observed that for many users of the IBM PC the question:

"How to obtain graphics output using the C language?"

is not trivial at all, and, in short, it became clear to me that the subject of 'raster graphics' is worth writing another book.

Some other books on computer graphics discuss the difference between 'vector graphics' and 'raster graphics' in rather abstract terms and with emphasis on hardware characteristics. From this, one might erroneously draw the conclusion that my 'vector-oriented' elementary routines should not be suitable for raster graphics devices such as a video display and a matrix printer. In my opinion, once we have decided to deal with the lower level of graphics, the best way to avoid such a misconception is to discuss concrete examples of both hardware and software.

The software presented and explained in this new book was written for the IBM PC and compatible machines using PC DOS or MS DOS, with either the usual color graphics adapter or a monochrome graphics adapter such as the well-known 'Hercules Card' from Hercules Computer Technology. Another technical point to be mentioned is that I used Lattice C, version 3.00G. I hope that the book will be instructive also for users of other compilers, or even of other hardware. After all, the C program text contains many rather simple routines, which could easily be modified, if necessary.

Some peculiarities of hardware and software discussed in the book will sooner or later be outdated, so one might wonder whether it can be used as a *textbook*. In teaching, we always have to distinguish between fundamental aspects and technical details. However, the former are best exemplified by the latter. Students are interested in a video display of $M \times N$ pixels only if we mention some concrete values of M and N. As teachers, we had better do this, even though such values will not be valid forever. In textbooks on microcomputers, technical details must be

mentioned, and this book is not an exception to this. I hope that it will turn out to be useful in teaching practical computer science, both at universities and in polytechnic schools.

There are also many *professionals* who write programs to make a living. If they deal with graphics on an IBM PC or a compatible machine, then buying this book may save them money. They may either copy the graphics functions as they are listed, or use the underlying ideas for their own benefit. In particular, their attention will be drawn to the CAD program in Chapter 6.

Last but not least, the book might be interesting for the '*advanced hobbyist*'. These days a great many people use an IBM compatible PC at home, and, though a minority, a considerable number among them are familiar with some reasonable programming language, of which C is a good example. This book will show them that despite the absence of built-in graphics facilities, the C language is very suitable for obtaining graphics output.

L. Ammeraal

Contents

CHAPTER 1

Introduction

1.1. HISTORY AND SCOPE OF THIS BOOK

In my book *Programming Principles in Computer Graphics* (published by John Wiley, Chichester, 1986) I presented a number of programs, written in the C language, which used four primitive routines for graphics, namely:

initgr( )	to initialize graphics output;
move(x, y)	to move a (real or fictitious) pen to point (x, y);
draw(x, y)	to draw a line segment from the current position of the pen to point (x, y);
endgr( )	to perform any final actions.

It was easy for me to use these four functions on a PRIME 750, since I could express them into subroutines such as *PLOT,* available in DIPLOT, a device-independent plot library, well-known to many Fortran programmers who use CALCOMP plotters. Thanks to these subroutines, I could restrict myself to using them, not bothering about how they work internally. It makes sense to distinguish two software layers, or levels of abstraction. At the higher level we use device-independent routines, and at the lower level we implement them. Obviously, if such routines are available then the most practical thing to do is to use them, and this is why my earlier book was written at the higher level of abstraction.

Then why write a new book, which deals with the lower level? The simple answer to this question is that the primitive routines that people need are not always immediately available to them.

During the production process of my previous graphics book, the publisher asked me if I could deliver the programs listed in the book on a diskette to be used on the IBM Personal Computer and compatible machines. Somewhat embarrassed, I had to confess that though regarding myself as a computer professional, I had only occasionally used an IBM PC and was not sure that implementing my graphics programs on this machine would be a great success. (Incidentally, there are still many experienced programmers of main frame and mini computers who have not yet discovered the PC and who consequently will not appreciate this book; fortunately their number is decreasing.) Other IBM PC users told me that in the C language no handy primitive graphics functions such as I needed were available, and that I would have to write them myself. I realized, however, that it would be unwise to reject my publisher's request, so after some investigation I bought an IBM compatible PC, which, sooner or later, I would have done anyway. With some help from my publisher and other kind people, it was then relatively easy to implement the functions *initgr, move, draw, endgr,* and to make the programs of my earlier book run on the IBM PC.

Writing these four primitive functions turned out to be a nice low-level programming exercise, and there were so many people interested in it that I decided to extend the project somewhat and write a second graphics book about it. A separate treatment of high-level and low-level graphics programming in two books might be unusual, but it avoids the confusion of talking at two levels at the same time. In this low-level book, our first goal is to develop a tiny graphics package, which consists of only little more than the four primitive functions that I originally needed. If this is all you need, Chapters 1 and 2 will do. However, once we have decided to devote our attention to 'raster graphics', we also want to deal with its special features. For example, on the screen of a video display, drawn line segments can be erased, which is not possible with a pen plotter. This will be the subject of Chapter 3. In Chapter 4, we use matrix printers for graphics output. If you have such a printer, you will be able to produce a hard copy of your graphics results without buying a pen plotter. Chapter 5 deals with writing text in combination with graphics, and shows how to add special characters such as an integral sign. Finally, in Chapter 6 we shall develop a simple drawing system. Since this requires no special hardware, there are no obstacles to use it in practice, or, at least, to experiment with it. In general, the book encourages you to be active. It may disappoint you if you expect it to inform you about advanced research projects. Admiring achievements of others gives you less satisfaction than using your own brain, hands and PC.

1.2 SOME SPECIAL POINTS FOR THE C PROGRAMMER

I assume that you are familiar with the C language as it is presented in my book *C For Programmers*. As emphasized there, when using C on a specific machine with a specific compiler, we may need some additional information. We are now using the IBM PC or a compatible machine; the C programs and functions we are going to discuss were compiled with the Lattice C compiler, Version 3.0. If you use a different compiler, you should be aware of some special Lattice C characteristics, so that, if necessary, you can adapt the program text as required by your compiler. I shall therefore try to provide you with such information in the following subsections.

1.2.1 Type unsigned char

At various places you will notice the type *unsigned char*. In Lattice C, Version 3, the keyword *unsigned* before *char* prevents the leftmost bit of a character from being interpreted as a sign bit. Such a sign bit would be extended to the left if the character is converted to type *int*. For example, after

unsigned char $k = 0xC0$; /* *In binary*: 1100 0000 */

the result of the right shift operation

$$k \gg 4$$

will be $0x0C$ (=0 . . . 0 1100, in binary) which in most applications is what we want. But if k were of type *char* (instead of *unsigned char*), the 'sign bit' 1 would be used to 'widen' 1100 0000 to 1111 1111 1100 0000, and this, shifted right four positions, would yield 0000 1111 1111 1100. Note that now the value of the rightmost eight

bits is 0xFC. If this is undesirable, the keyword *unsigned* is a good remedy: it causes 1100 0000 to be widened to 0000 0000 1100 0000, which, after the right-shift, yields the correct value 0000 0000 0000 1100.

1.2.2 Direct input from the keyboard

We normally use the file pointer *stdin* for 'standard input' if we want to read something from the keyboard. For example,

scanf("*%d*", *&n*) is equivalent to *fscanf*(*stdin*, "*%d*", *&n*), and
getchar( ) is equivalent to *getc*(*stdin*).

These functions use a buffer, which implies that the characters we are typing are actually used only when we press the Enter key. This enables us to use the backspace key for corrections. Sometimes we wish the characters to be used the very moment we type them, without the obligation to press the Enter key. In Lattice C we have the following two functions for this purpose:

getch( ) Get a character from the keyboard, no echo.
getche( ) Get a character from the keyboard, echo.

Here 'echo' means that the character that we enter is displayed on the screen. With *getche* this happens in the same way as with *getchar*. We shall see that sometimes the machine is in 'graphics mode'; then echoing the entered character is not desired, so *getch* is suitable in that case.

Another useful non-standard function is

kbhit( ) Check if the keyboard is hit.

In contrast to normal input functions, *kbhit* will not wait until some key is pressed. It simply returns the value 1 if a key has been pressed, and 0 otherwise. If a character has been entered, *kbhit* will not skip over that character, in other words, the character can still be read in the normal way. The function *kbhit* enables us to terminate a loop when a key is pressed, as for example in

```
main()
{ int n=0; double x=1.0;
  while (1)
  { x *= 1.000000001; n++;
    if (kbhit()) break;
  }
  printf("n = %d   x = %f", n, x);
}
```

There is also a function *ungetch*, similar to the standard I/O function *ungetc*. We can use it after calling *getch* or *getche* to push a character onto a stack, so that it will be used when we call *getch* or *getche* again. The stack is only one level deep, so we should not push a second character onto it. In the example

```
ch1 = getch(); ungetch(ch1); ch2 = getch();
```

only one character will effectively be read from the keyboard, and this character is assigned to both *ch*1 and *ch*2. We shall use *ungetch* in subsection 1.2.4.

1.2.3 Memory models; peek and poke

The 8088/8086 processor employs a segmented addressing technique. Each address consists of two 16-bit components: a segment and an offset. The segment is shifted four bits to the left, that is, it is extended with four zero bits on the right-hand side, and then the offset is added to it. In this way we obtain a 20-bit address, sufficient for a 1-megabyte address space. There are four 'segment registers' to contain segments, namely

CS	'Code Segment'
DS	'Data Segment'
SS	'Stack Segment'
ES	'Extra Segment'

As long as our program fits into 64K bytes, it is possible to keep the segment constant in CS, and vary only the offset. Similarly, as long as our data area does not exceed 64K bytes, the segment in DS can be constant. In this way we use only 16-bit addresses, both for instruction fetching and for data access. This leads to more efficient code than when 20-bits addresses are used, and for a great many applications, a 64K program area and (another) 64K data area are sufficient. We call this way of using memory the S model (where S stands for Small). Besides the S model, we can use the P model if only the program size exceeds 64K, the D model if only the data exceeds 64K, or the L model (Large) if both program and data exceed 64K. We can specify to the compiler which model is desired; each model has its own library, so compiling and linking must be consistent wilth respect to the memory model. The default model is S, so this will be used if we do not tell the compiler anything about a model. We shall use this S model in all our programs; even the rather large drawing program to be discussed in Chapter 6 fits into 64K bytes.

The above discussion is rather machine-oriented; it could have been omitted if we had not to deal with graphics. However, in the next chapter, we shall directly access the graphics adapter, as if it were located in the normal memory, starting at address 0xB8000. Note that this is a 20-bit address, written in hexadecimal notation. We shall use the term 'screen memory' for the amount of memory located in the graphics adapter. It would be a pity if only for this special purpose we had to use the D or the L model. Fortunately, there are two functions in Lattice C to move data to and from such a memory area, respectively:

```
poke(segment, offset, source, nbytes)
peek(segment, offset, destination, nbytes)
```

For the first argument we use 0xB800. Note that this is only a 16-bit value, since, as mentioned above, it will be extended with four zero bits on the right-hand side. The first two arguments have type (unsigned) integer. The second argument is the offset to be used relative to the extended first argument. The third argument is a normal pointer to a character, so it may be the name of an array of characters. The fourth argument has type (unsigned) integer; it says how many bytes are to be copied.

1.2.4 Console break

We sometimes stop a running program using Ctrl Break, which means that we press the Break key while the Ctrl key is kept down. If there is no Break key, the combination Ctrl C is used for this purpose. Let us use the term 'console break' for either method. There are two kinds of problems with the console break, and especially for graphics programs it is important to solve them. The first problem is that the machine may not listen when we are trying to use the break facility. The operating system checks for a console break only on certain occasions. Depending on whether or not a 'break check flag' has been set, it performs this check either on any 'service request' or only on a console service request. If only computations are carried out, such as, for example, in the loop

```
for (i=0; i<30000; i++) s += 1 + 2 * (i / 2);
```

there is no such request, so the program will refuse to be interrupted by a console break. I actually encountered this problem in a more interesting computation than this one, and I then started looking for some innocent service request (preferably a console service request), which I could insert in the loop to make the operation system perform the desired check. I first tried a simple call of the function *kbhit* (see 1.2.2), which turned out to work satisfactorily in most situations. However, it did not work when I redundantly pressed some key before using the console break. This sometimes happens, for example, if only some letter should be entered, and we press not only the key with that letter, but also the Enter key. I therefore extended this solution, and used the following function:

```
checkbreak()
{ char ch;
  if (kbhit()) { ch = getch(); kbhit(); ungetch(ch); }
}
```

If *checkbreak* is called in the inner part of the above for-loop, the program will listen to our console break.

The second point is: What will happen if and when the machine listens to our console-break request? If we do not specify any action ourselves, the console break will activate the default interrupt handler, which simply stops program execution. In most programs this is what we want, but in graphics programs another action will be necessary, as we will see in Section 2.9. In general, we can 'plant a break trap' by writing a special function that says what should be done if and when a console break occurs. The address of that function, simply written as its name, is passed as an argument to the function *onbreak,* available in Lattice C. If our function returns a value of 0, the execution resumes at the interrupted point. Otherwise the program is aborted immediately. The function *onbreak* may also be given a null pointer as its argument; this may be written as the number 0. In that case the default interrupt handler will again be used when a console break occurs. If the argument of *onbreak* is not 0 but a function, it should be declared as such before it is used, otherwise the compiler would mistake it for a simple integer variable. Here is a demonstration of all this. It is based on the program in Subsection 1.2.2. Instead of pressing any key, we now use a normal console break.

```
/* BREAKDEMO.C: Console-break demonstration */
#include "dos.h"
int n=0, my_function();
double x=1.0;

main()
{ onbreak(my_function); /* Replace default interrupt handler */
  while (n<30000)       /* with  my_function.                */
  { x *= 1.000000001; n++;
    checkbreak();
  }
  onbreak(0);
  /* Any program text inserted here, when interrupted, would
     invoke the default interrupt handler.                    */
  printf("Normal program end, n = 30000   x = %f", x);
}

int my_function()
{ printf("n = %d     x = %f", n, x);
  exit(0);
}

checkbreak()
{ char ch;
  if (kbhit()) { ch = getch(); kbhit(); ungetch(ch); }
}
```

Note that *my_function* calls the standard function *exit*. It does not return to the main program, so it need not contain a return statement specifying whether or not the program is to resume, as mentioned above.

As a final remark we note that on an IBM PC there is a subtle difference between Ctrl Break and Ctrl C. If we run *BREAKDEMO.C* and we redundantly press any key more than once prior to a console break, Ctrl Break works properly but Ctrl C has no effect. If we press such a key only once or not at all before the console break, which is more likely, Ctrl Break and Ctrl C have identical effects.

1.2.5 Accessing the 8088 I/O ports

The lowest level at which input and output can be programmed is based on the elementary machine instructions *IN* and *OUT*. These instructions are normally used only by assembly-language programmers (and even for them higher-level I/O facilities are available, as Subsection 1.2.6 will show). The instruction *IN* reads data from an input port, and *OUT* writes data to an output port. These I/O ports are special hardware circuits used by the computer to communicate with external devices. All high-level routines for input and output eventually result in the execution of *IN* and *OUT* instructions. These instructions are also available in Lattice C. They are used as follows

```
v = inp(p);
outp(p, v);
```

where p and v have type (unsigned) int, p being the port address and v the port value. When using the functions *inp* and *outp*, we have to use the line

```
#include "dos.h"
```

in our program.

Direct accessing I/O ports incorrectly may cause all sorts of system problems, so it is strongly advised to use higher level I/O functions instead, wherever possible. The software in this book will not directly access I/O ports if it is running on a machine with a color graphics adapter. With a Hercules card (or a compatible monochrone graphics adapter), however, it will use the function *outp* to switch from text mode to graphics mode, and vice versa.

1.2.6 Registers and software interrupts

We shall now see how the computer can communicate with the outside world at a higher level than with directly accessing I/O ports. There is a set of routines, called Basic Input and Output System, or, briefly, *BIOS*. You will not expect that switching to a higher level requires more knowledge of the machine architecture, but curiously enough, to some extent this is the case. We cannot possibly use BIOS routines unless we know some general registers of the 8088 (or 8086) processor. We can understand this if we realize that these routines were designed to be called in assembly-language programs, not in a high-level language such as C. We need not discuss the entire 8088 register set, but we can restrict ourselves to the four Data Registers, shown in Fig. 1.1.

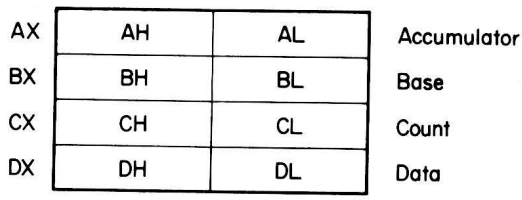


Fig. 1.1. 8088 Data Registers

Each of the registers AX, BX, CX, DX contains 16 bits. It is divided into a high-order and a low-order byte, each of which can be used as an eight-bit register. For example, if FA3B (hex.) is loaded in DX, the contents of DH and DL will be FA and 3B, respectively.

Another rather technical aspect is the way BIOS routines are called. A BIOS routine is not called as a subroutine, but as a so-called 'software interrupt'. This term is derived from the interrupt mechanism that is used for signals from the outside world. A running program is then really interrupted, but in such a way that normal program execution can be resumed later. When such an interrupt occurs, all register contents are pushed onto a stack, and a jump takes place to a location whose address is obtained from a table, the so-called interrupt vector. The program fragment starting at that location is called an interrupt routine. It ends with the instruction 'Return from Interrupt', which takes care that the saved register contents are popped from the stack, and that the execution of the interrupted program is resumed. In the same way as by external interrupts, our program can be 'interrupted' by a special instruction, called a software interrupt. Since the programmer decides where a software interrupt is to take place, it is conceptually similar to a subroutine call, but its implementation is similar to the way external interrupts are dealt with. Instead of the address of a subroutine, we only need a (usually small) number to identify the software interrupt. For all software interrupts

associated with the video display, this number is 16, usually written hexadecimally as 10. In an assembly-language program we use the instruction

```
INT 10H
```

to initiate any video software interrupt. Since interrupt 10H is used for many purposes, we have to use some parameter-passing mechanism. This is why we need the data registers of Fig. 1.1 in this context. Using Lattice C, we can write

```
int86(0x10, &regsin, &regsout);
```

The first argument is the interrupt number 10H. The second and the third arguments are the addresses of data structures which correspond to the registers that we would use in an equivalent assembly-language program. Any information to be passed to the routine is to be supplied in *regsin,* and any information returned by the routine will be available through *regsout.* To use all this, we have to write

```
#include "dos.h"
```

and to declare

```
union REGS regsin, regsout;
```

In the header file *dos.h,* the type *union REGS* is defined as in the third of the following three lines:

```
struct XREG { short ax,bx,cx,dx,si,di; };
struct HREG { byte al,ah,bl,bh,cl,ch,dl,dh; };
union REGS { struct XREG x; struct HREG h; };
```

In an object of type *union REGS,* the structures *x* and *h* occupy the same memory, which is just what we want. For example, *regsin.x.ax* shares memory with both *regsin.h.ah* and *regsin.h.al,* in the same way as the two-bytes register AX is divided into the two one-byte registers AH and AL, and so on. (Please, do not worry about *si* and *di* in the first of the above three lines or about the order in which *al* and *ah* occur in the second: I would rather not discuss the 8088 processor in more detail at this moment.) Before the software interrupt (number 10H) takes place, we have to place an unique code into register AH. In C, we achieve this as follows:

```
regsin.h.ah = code;
int86(0x10, &regsin, &regsout);
```

This code tells the interrupt routine which of several possible actions is required. For our purposes, there will be no need for two distinct arguments *regsin* and *regsout*: the same structure *regs* can be used for both purposes, so we will simply declare

```
union REGS regs;
```

and use *regs* instead of both *regsin* and *regsout.* Besides AH, some of the other registers may have to be filled to specify what we want in more detail, as we will see in Section 2.3.

1.2.7 The maximum stack size

Automatic variables and return addresses of functions are placed on a stack, which is a contiguous piece of memory. There is a default limitation imposed on the stack size, which may be insufficient, especially if we use recursive functions. Lattice C offers the possibility to choose a maximum stacksize larger than the default value. All we have to do is to write, for example,

```
unsigned int  _STACK = 15000;
main()
{ ...
  ...
}
```

if we want the value 15000 to override the default value (of 2048). The size may now be as large as 15000 bytes.

Incidentally, we can in turn override that new value when we give the command to execute the program. For example, if our program is called *MYPROG.EXE*, we can specify that the stack size limit is to be 20000 bytes by entering the command

```
MYPROG =20000
```

Since we assume that our programs will also be used by people who are not interested in such implementation details as stack sizes, we prefer the former method of specifying the maximum stacksize. If, in this example, the value 20000 is used for *_STACK* in the program text, then the user can simply enter the command

```
MYPROG
```

with the same effect as the more extensive *MYPROG* command above.

1.3 GRAPHICS ADAPTERS

On the screen of our video display, there are a great many points, and each of them can be made light or dark, or, as we sometimes say, white or black. The screen may be able to display colors, but we shall not deal with other colors than white and black. These white or black points are called picture elements, abbreviated as *pixels* or sometimes *pels*. The more pixels there are, the higher the resolution. The *graphics adapter,* also called *board,* or *card,* is the piece of hardware which determines this resolution. The three most popular graphics adapters, with their characteristics, are:

- Monochrome display adapter, used for text only; it displays 25 lines of 80 characters each.
- Monochrome graphics adapter (such as a Hercules Card), used both for text and graphics.
  When used for text, it is identical with the monochrome display adapter.
  When used for graphics, it has 720×348 pixels (348 lines of 720 pixels each).
- Color graphics adapter, used either for text or for graphics.
  When used for text, it displays 25×80 characters.
  When used for graphics, it has 640×200 pixels (or less if other colors than white and black are used).

These adapters contain memory, which we can address in the usual way. We can

display text or graphics by placing appropriate data in this area of memory. However, the way data is coded for text is essentially different from the way it is coded for graphics. For text, the ASCII value of each character to be displayed is stored in one byte, followed by a so-called attribute byte which contains information about how the character is to be displayed. (For example, there is an attribute to underline characters.) The transformation of these two bytes into a pixel pattern is performed by a special piece of hardware, called a *character generator.* This provides an efficient way of displaying characters. We have $25 \times 80 = 2000$ character positions on the screen, and as each character is coded in two bytes, we need 4000 bytes. This number if very low if we realize that each character is represented on the screen in a box of 9×14 pixels (14 horizontal lines of 9 pixels each). If for each pixel one bit were needed, this would take $2000 \times 14 \times 9/8 = 31500$ bytes. The addresses of the 4000 bytes actually used are B0000, . . . , B0F9F. The important thing to remember is that, when using the *monochrome display adapter,* we cannot suppress the activity of the character generator. This is why the monochrome display adapter cannot reasonably be used for graphics. Note that its name does not include the term *graphics* as with the other two adapters.

The *monochrome graphics adapter* is a most ingenious device. (Incidentally, it should not be confused with the monochrome display adapter mentioned above; the frequently used term 'Hercules card' avoids such confusion, but that term is correct only if the adapter was made by Hercules Computer Technology, which may not be the case.) First of all, the monochrome graphics adapter can be used to generate characters in exactly the same way as the monochrome display adapter. However, with the monochrome graphics adapter we can suppress the activity of the character generator, and switch from 'text mode' to the 'graphics mode' (also called 'bit-mapped mode'). In graphics mode, each pixel corresponds to one bit in memory. If the bit is 1, the corresponding pixel is lit, if it is 0, the pixel is dark. There are 720 pixels on a horizontal line, which corresponds to $720/8 = 90$ bytes. As there are 348 lines, the amount of memory needed is $348 \times 90 = 31320$, which is rounded up to $32K = 32768$ bytes. In hexadecimal notation, the addresses B0000, . . . , B7FFF are used for this purpose. Besides, the monochrome graphics adapter has another 'page' of 32K, with the addresses B8000, . . . , BFFFF. We can use both pages, but at each moment only one of them is displayed. If we use only one we can freely choose either of them. The pages starting at the addresses B0000 and B8000 are numbered 0, 1, respectively. We shall use page 1 for two reasons. First, its begin address B8000 is the same as for the color graphics adapter, as we shall see presently, so we can now always use the same begin address. Second, page 0 overlaps the text display memory, and page 1 does not. So if something valuable is in page 1, it remains unaffected if we switch to text mode. In Chapter 4, we shall use this to produce a 'post-mortem' graphics screen dump.

The *color graphics adapter* is needed if we want pictures in various colors. Like the monochrome graphics adapter, this adapter can operate either in text mode or in a bit-mapped graphics mode. Yet I would recommend it only if colors are really needed, since it has three drawbacks:

1. It generates character patterns in a 8×8 box instead of in a 9×14 box as with the other two adapters. This reduces the readability considerably.