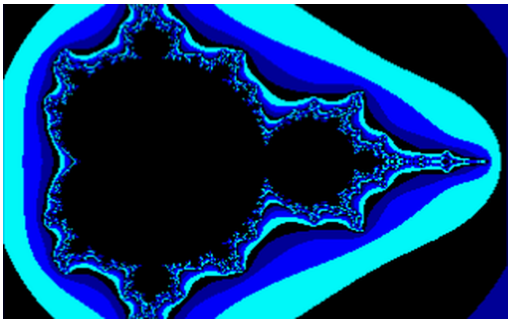
Matthias Uphoff

The Software Experiment

Experimental Mathematics, Simulations
and Artificial Intelligence
on the Schneider CPC



English translation of *Das Software-Experiment*
DMV-Verlag 1988 · HTML edition 2002 · PDF edition 2026

The article series *Das Software-Experiment* appeared in the German magazine *PC International* from the summer of 1986; in 1988 DMV-Verlag published it as a ring binder with a program disk. In 2002 Matthias Uphoff scanned the texts and republished them as web pages (see the Afterword).

This English edition was translated in 2026 from the German HTML edition (`https://www.michael-wessel.info/retro/CPC/SoftEx/Inhalt.htm`) by Claude, an AI model by Anthropic, for Michael Wessel. The translation has not been reviewed by the author. The programs on the disk `SOFTEX.DSK` (also found there) and the screenshots are the German originals: German words on screen and in the word games are kept, with their meaning given in the text. A typeset PDF of the German original is available at the same place.

Typeset with LaTeX by Claude Code (Anthropic) for Michael Wessel.

Contents

Preface (1988) **vi**

Afterword (2002) **viii**

1 Complex Graphics **1**
Waiter, the Numbers, Please! 2
Chaos in the Computer 5
Good Graphics Come to Those Who Wait 8
Operating KOMGRA.BAS 9

2 The Game of Life **13**
In the Beginning Was the Dot... 14
The Cellular Automaton 16
The Game of Life 17
Evolution in the Computer? 21

3 The Ecology Experiment **23**
The Wolf-Sheep-Grass Model 24
Eat and Be Eaten 26
The Eco Program 27
Catastrophes in the Computer 29
The Ecological Balancing Act 30
Software Author Wanted 32

4 The Computer as Strategist **34**
Of Masters and Patterns 35
Let's go bang! 36
Idiotic or Brilliant? 38
Parameter Tuning 42

5 By Brute Computing Force **45**
Kalaha: The Good Ones Go into the Pot... 47
Move Evaluation: A Bit Shortsighted 49

Tree Search with Minimax 50
Recursion: Even in BASIC! 52
The Alpha-Beta Pruning Shears 55

6 The Computer Learns **58**
Mini Chess: Small but Mighty! 59
Slow on the Uptake? 61
Packed Data: Packed Like Sardines 62
The Move Generator 64
Reduced Move Lists 66
The Programmed Little Brother 67

7 The Computer as Scientist **70**
Insight Instead of Programming 71
Model Wanted 73
The Big Bluff: Wordmaster 75
The General Problem Solver 77
Take Away Gerd's Bier, and a Beast Will Appear! 78

8 Word-Chain Problem Solved! **81**
What's the Problem? 83
A Turtle with Reverse Gear 85
The Last Word in Wisdom? 87
All at One Blow! 88

9 Development Aid by Computer **90**
1. Development Aid as a Topic in Geography Lessons 90
2. MORETTI in the Classroom 91
3. Problems of the Sahel Zone 92
4. The Mathematical Model of the MORETTI Simulation 94
5. Operating the Program 94

10 The Readers Have the Floor **99**
The One-Dimensional Cellular Automaton 99
New Sounds with the CPC 101
The Well-Tempered Clavier 103
Pascal Sends His Regards 104

Appendix **108**
1. The Files of the Software Experiment 108
2. The 664/6128 Emulator for the CPC 464 109

Preface (1988)

Starting in the summer of 1986, the magazine PC International ran the article series *The Software Experiment*, which focused on experimental mathematics, simulations and artificial intelligence. The declared goal was not just to supply readers with a batch of programs to type in, but also to encourage them to join in and think along.

The great success of the series was the reason for publishing this extensive material once more in compact form, making it available to all those CPC owners who may have bought their computers only later on. As you will see, the Software Experiment is not only an exceptionally versatile collection of programs; it also packs a hefty load of know-how, which this manual serves up in a clear and entertaining way.

A home computer with graphics and sound like the CPC offers a wealth of creative possibilities, many of which are still unexplored. The most interesting topics of computer graphics and artificial intelligence, however, call for a solid knowledge of data structures and algorithms if you want to take part in this fascinating development not just as a consumer and spectator but as an active researcher. That is why the following chapters explain in detail the methods and techniques used in the various programs of the Software Experiment, to give you ideas for putting your own plans into practice.

Of course, a home computer doesn't have the capacity of the modern mainframes found at universities and research institutes. Even so, with its powerful BASIC, its color graphics and its 64 KB of memory, the CPC can be coaxed into producing astonishing pictures and remarkable feats of intelligence. As you will see, the topics covered in *The Software Experiment* are anything but trivial; you will be spared the bare-bones examples so popular in accounts of AI problems, such as tic-tac-toe, a game that is easy to follow but, to make up for it, pretty boring.

For all the learning we have in mind, fun and games will not be neglected in the Software Experiment. The programs, originally tailored for printing as listings, have been revised once more, so that they now come ready to run on your disk or cassette and work without restrictions on all CPC models (464/664/6128). Only CPC 464 owners need, for one program

(MORETTI.BAS), the 664/6128 emulator (EMU.BAS), which is included as a bonus. You will find the description of the emulator in the appendix.

So most of the programs can simply be used as games, even without any knowledge of the theory behind them. Since they were originally designed as flexible test beds for experiments, however, they generally don't come with elaborate user guidance such as help screens and the like. For this reason it is a good idea to study at least the operating instructions in the corresponding chapters carefully.

In places the texts contain remarks that refer to particular program lines or sections. It helps your understanding if you print out the listings for each chapter by typing LIST #8 after loading. If you don't own a printer, it will do in a pinch to bring the interesting sections up on the screen with LIST.

Afterword (2002)

…and this is where the preface breaks off—the missing page was nowhere to be found. So here, as a replacement, is a closing afterword. That this "new edition" of the article series *The Software Experiment* in the form of HTML files came about at all is the result of a coincidental(?) meeting of several factors, which I would like to list briefly:

It can be interesting to type your own name into Google now and then and see all the places on the Web where you turn up. On one such attempt I came across a thread in a CPC forum in which people were hunting for articles and programs from my article series.

Some time later I discovered that very good CPC emulators are available as freeware. Trying them out sent me first on a retro trip and then down to the basement, where after some rummaging around three CPC disks came to light—one of them held all the files for the article series. It was tucked into an A5 ring-binder edition of *The Software Experiment* that DMV-Verlag published in 1988, unfortunately in extremely poor quality.

Now I wanted to see the old treasures in action again, too. Since I lacked the hardware to convert the 3″ CPC disks into DSK image files an emulator could use, I took a chance and wrote to one of the people in the thread… and found in Michael Wessel a very helpful and interested person who, above all, still owns real CPCs. Battling all sorts of hardware and software problems, he finally managed to convert the disks.

Meanwhile I had become convinced that the accompanying texts were worth rescuing as well. So 180 poorly printed ring-binder pages were scanned and turned back into text files by OCR—a laborious and at the same time fascinating undertaking that called for the mindset of a passionate museum restorer. A few pictures were added too—the screenshots came from the CPC emulator *CaPriCe32* by Ulrich Doewich, on which all the programs of *The Software Experiment* ran without a hitch (at times I was glad that the emulator can optionally run faster than the original CPC—how patient we were back then! Waiting an hour for the *gingerbread man*…). Many thanks at this point for this excellent freeware!

The current version of the emulator can be found here[1]. To run the programs of *The Software Experiment*, first load the accompanying image file **SOFTEX.DSK** via the menu item **Disk/A-Insert Disk**. Typing the command **Cat** followed by RETURN shows the contents of the disk. You then start a program by typing the BASIC command **Run "program name"** and pressing RETURN. It is a good idea, however, to first use the menu item **Tools/Remap Controls** to map the "Y" and "Z" of the PC keyboard to the "Y" and "Z" of the CPC keyboard, adapting the preset English layout to German conventions. Since the CPC's keyboard layout differs considerably from the PC's, especially for symbols and special keys, it is advisable to look at the picture of the CPC keyboard that comes with the emulator as a GIF file and compare it with the entries under Tools/Remap Controls; otherwise it takes a good deal of trial and error to find the characters that are not on their familiar PC keys.

Looked at from today's point of view, the programs of *The Software Experiment* seem rather old-fashioned in their outward appearance—operating them takes some getting used to, to put it mildly. But the results they produce are, for the most part, still interesting and surprising. And anyone interested in experimental mathematics, simulations and artificial intelligence will still get a fair amount out of the accompanying reading today. To everyone who turns to the Software Experiment once more, out of nostalgia or an undiminished delight in tinkering with computers, I wish many entertaining hours and lots of fun!

Matthias Uphoff

[1] `http://www.classicgaming.com/caprice` (as of 2002).

CHAPTER 1

Complex Graphics

It is not only the world of work that modern computer technology is turning upside down. Scientists and artists, too, have more and more tools at their disposal whose possibilities cannot even be fully gauged today.

The CRAY X-MP supercomputer, for example, has made it possible to reconstruct how our universe came into being and to display that history graphically on a screen. Anyone who has ever wrestled with the differential equation describing the motion of just one planet around the sun can appreciate the amount of computation required by the interactions among a truly astronomical number of celestial bodies.

The scientist of earlier times, whose tools amounted to little more than the slide rule and a table of logarithms, would have been hopelessly overwhelmed by such a task. Only microelectronics has made it possible to explore complicated mathematical or physical systems experimentally, by means of simulations. What is particularly remarkable is that the results of these experiments do not merely come in abstract form, as numbers or equations; some of them turn out to be fascinating pictures. Like a telescope or a microscope, the computer offers glimpses of worlds and dimensions that would otherwise remain closed to the human eye. Mathematical and logical structures become visible, appear as colors and shapes on the screen, and can be changed experimentally and playfully—no wonder, then, that mathematics, long decried as a dry subject, has gained enormously in appeal.

It is not only artists who are discovering new motifs and new means of expression here; young people, too, are increasingly taking a lively interest in this science, which only the medium of the computer has made graphic in the truest sense of the word. Our first software experiment, therefore, is a program that uses a relatively simple mathematical formula to produce a wealth of astonishing pictures, which have already caused quite a stir under

the names *chaos graphics* or *gingerbread men*.

The gingerbread-man graphics come from picturing the properties of complex number sequences—but what are complex numbers in the first place? How can a mathematical formula produce structures whose orderly irregularity gives them an almost organic character? To get closer to the answer, let us begin with a little excursion into arithmetic, the theory of numbers:

Waiter, the Numbers, Please!

The concept of number started out with the natural numbers 1, 2, 3,…—called natural, presumably, because they did not grow out of theoretical considerations but came straight from concrete experience. Early man stepped out of his cave and began to count: 1 saber-toothed tiger, 2 saber-toothed tigers…—well, perhaps not exactly like that, but the natural numbers must have come about in some such way. And for many purposes they really are all you need! We can add them or multiply them, and the result will again be a natural number. The set of natural numbers is thus closed under these operations, as the mathematicians say.

Problems crop up, however, when we want to subtract or divide. Sometimes it works and sometimes it doesn't, as the equation $x + 5 = 3$ shows; it has no solution among the natural numbers. To put an end to this unsatisfactory state of affairs, people invented the negative numbers and lumped them together with zero into the set of integers—or, to put it in computerese: the integer was born.

But in doing so, mankind had also left the realm of the tangible. Negative numbers are abstract constructs; they are easy enough to calculate with, but they already put our powers of imagination to a severe test. When I was a student, for example, there was a math lecturer who was thoroughly unpopular because of his miserable lectures. Over the course of the semester his audience visibly dwindled, until one day a measly three students were left—and, to make a long story short, the lecture was once again so incomprehensible that a total of five of them fled the room. 'Right', said the prof with a grim look on his face, 'if two more come in now, there'll be nobody here!'

So negative numbers are kind of dumb, as one of my tutoring pupils put it in a nutshell, but they are also practical, and that is why they have sur-

vived to this day. The same is surely true of the rational numbers, the set of fractions, which let you divide five cows among seven farms without any trouble—in theory, at least—but unfortunately that still doesn't dispose of all our problems, not by a long shot. It can be shown, quite simply and conclusively, that the algebraic equation

$$x^2 = 2$$

cannot be solved by a fraction.

Suppose it could; then x could be written in the form p/q, where we shall assume that this fraction has already been reduced as far as possible. So p and q have no common factor, and that does not change when we square the fraction. On the other hand, substituting it into the equation gives us

$$p^2/q^2 = 2$$

But the result can only be 2 if the fraction can be reduced, which contradicts our assumption, so the assumption must be false. The mathematicians therefore had no choice but to invent new numbers for this and similar cases: the irrational numbers. But what does irrational actually mean? Let's reach for a dictionary of foreign words, and lo and behold, there we find '*beyond the grasp of reason, emotional, driven by instinct*'.

You can tell—now things are getting risky! Personally, I have yet to catch the square root of 2 giving in to its instincts, but the mathematicians must have found these numbers somehow uncanny, for they were promptly married off to the rational numbers to form the set of real numbers, which you will meet on the CPC as the *REAL type*. Real means something like *'realizable, sound, respectable'*, so decency and order would be restored, and we could now calmly turn to other matters, if it weren't for one little problem:

$$x^2 = -1$$

That really is a tough nut to crack: what number, multiplied by itself, gives –1? The real numbers won't get you a single step further, and even our clever CPC is out of its depth here:

```
PRINT SQR(-1)
```

gets nothing out of it but a baffled *Improper Argument*.

But at this all but insurmountable threshold, the mathematicians took a bold step. 'Fine', they said to themselves, 'we don't know the solution, but let's just assume that there is one', and they named this mysterious number i, for which, by definition,

$$i^2 = -1$$

The letter i stands for imaginary (apparent, unreal), a fitting name for an utterly abstract number that no longer has any counterpart in reality. But as experiments show, you can carry out all the familiar arithmetic operations with the imaginary number without running into contradictions. So the extension of the number concept is in order, formally at least. In particular, the number i can be combined with real numbers by addition or multiplication, which gives rise to constructs such as 3+2*i, or in general form

$$c = x + y \cdot i$$

Numbers c like these, made up of a real number and a multiple of the imaginary unit, are called complex numbers. It was the famous mathematician Gauss who proved that this makes our number system complete once and for all; every algebraic equation, in other words, has at least one complex solution.

The only question is why anyone should bother with such figments of the imagination, which at first glance have not the slightest practical value. But much like the negative numbers, complex numbers, abstract as they are, are a very effective tool for certain problems in physics and mathematics. Some differential equations, for example, become much easier to solve if certain variables are treated as complex numbers from the start.

It is not for nothing that high-level programming languages for scientific applications (e.g., FORTRAN) offer the variable type COMPLEX alongside INTEGER and REAL. Although Locomotive BASIC has no statement like DEFCOMPLEX, you can still venture into the realms of higher mathematics with the CPC. Every complex number c=x+y*i can be represented by a perfectly ordinary pair of numbers (x,y), one standing for the real part and the other for the imaginary part. By having the computer work on the two components separately, we spare it the ominous square root of −1, which would of course trigger an error at once.

Such a pair of numbers can, of course, be plotted splendidly in a coordinate system, with the x-axis corresponding to the real number line and the y-axis to the imaginary line. If you draw an arrow from the origin of the coordinate system to the point that represents the complex number, you can even picture what is meant by the absolute value of that number: it is the length of the arrow, which is calculated with the Pythagorean theorem:

$$L^2 = x^2 + y^2$$

In this way it is even possible to assign a size to complex numbers, and this will play an important part in the gingerbread-man graphics.

Chaos in the Computer

Order is half of life, as the old saying goes—and disorder is the other half, adds the seasoned programmer who, after a long and frantic search, finds his system disk in the wastebasket among a few old listings. But all scares aside: in the natural sciences, more and more people are coming round to the conviction that highly complicated structures such as living organisms could only have arisen from the interplay of order and chaos, of law and chance. And this theory is supported in particular by the results of a number of computer simulations!

The exact laws of nature give the world a fixed framework. If all the initial conditions of a system are known, these laws let you calculate, in theory at least, exactly what will happen, down to the last detail. But modern physics has shown that in the realm of elementary particles there are processes that are indeterminate and about which at best statistical statements can be made. So our world has a built-in random number generator that reliably keeps it from stagnating in never-changing patterns.

Every game programmer knows how valuable his computer's RND function is, that preprogrammed uncertainty factor which provides variety and suspense. On request it produces a sequence of numbers in which, try as you might, you can't make out any system. And yet this is not true randomness, only a very good simulation of it! Naturally, a computer can't make up numbers by intuition; it has to calculate them somehow.

A sequence of pseudo-random numbers is usually generated iteratively, i.e., each number is derived from the previous one by a certain rule of for-

mation, which the observer can hardly reconstruct, though, unless he goes poking around in the ROM listing. A simple arithmetic operation such as squaring the number won't do here, of course; the rule would still be too obvious. A lot more artificial chaos results if you also subtract the first number of the sequence as a constant, as the following short program shows:

```
10 xstart=1.5: x=xstart
20 x=x*x-xstart: PRINT x
30 GOTO 20
```

This simple system does not yet meet the rather strict criteria for professional random number generators, but the result is already quite impressive, and things get even more interesting when we apply the same iteration rule to complex numbers. To state the law precisely once more: if the complex number C is the starting value,

$$C_1 = C$$

then we calculate all further numbers by the rule

$$C_{n+1} = C_n^2 - C$$

Simple as this notation looks, in practice it involves a fair amount of arithmetic. The square of a complex number has to be worked out with the binomial formula:

$$C^2 = (x + yi)^2 = x^2 + 2xyi + y^2 i^2$$

Since i * i = −1, we can simplify this further,

$$c^2 = x^2 + 2xyi - y^2$$

and then separate the real and imaginary parts,

$$\mathrm{Re}(c^2) = x^2 - y^2; \qquad \mathrm{Im}(c^2) = 2xy$$

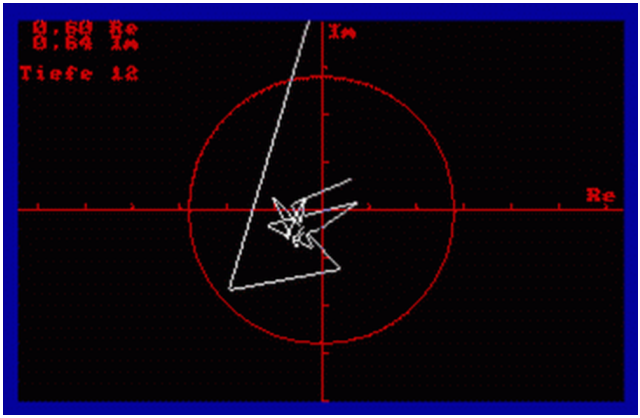
which makes it clear that this operation takes one subtraction and four multiplications in all. Subtracting two complex numbers is fortunately a little less work; you simply subtract the real and the imaginary components of the two numbers separately.

To understand how this arithmetic produces the gingerbread man and his relatives, you should first take a look at the program *Komplexe Folgen* (Complex Sequences, KOMFOL.BAS). Once the program has started, a pair of coordinate axes appears on the screen, formed by the real and the imaginary axis of the number plane. You will also see a circle and a bright dot, which is moved with the **cursor keys** and represents the starting value. The real and imaginary parts of its current position are shown at the top left.

If you now press the SPACE bar several times, the program calculates the next number of the sequence each time by the method described above and draws a line to the new position. Depending on the starting value, this results in a wild zigzag course, which comes to an end, however, as soon as the termination criterion

$$x^2 + y^2 \geq 8$$

is met, that is, as soon as the absolute value of a number in the complex sequence lies outside the circle with a radius of root 8. DEL restarts the program so you can try a different starting value.



The results of these experiments can vary a great deal. Some sequences leave the circle after only a few iterations, others only after a lot of back and forth (try 0.24 / −0.64!), and some, whose starting value lies close enough to 0, get stuck and never reach the termination criterion.

Another remarkable thing is that the basic patterns repeat themselves on a smaller scale at various places in the picture. The great whole is mirrored in the small—a central idea of esoteric teachings and other holistic philosophies, which finds astonishing expression in these pictures.

Good Graphics Come to Those Who Wait

So how do we produce complex graphics on the CPC? Let's start from the technical facts: the graphics screen is a coordinate system as well, which in MODE 1 contains a total of 320 * 200 pixels.

If we pick a region of the complex number plane and project it onto the screen, each point corresponds to a number. That number is the starting value of a complex sequence, which is calculated until the termination criterion

$$x^2 + y^2 \geq 8$$

is reached. So for a complete picture, 64000 sequences have to be generated, each of them, depending on the iteration depth, consisting of several dozen complex numbers—and all that in time-consuming floating-point arithmetic, to boot. It's easy to see, then, that a single picture can take several hours. As experiments with the program KOMFOL.BAS show, however, there are also sequences that never reach the termination criterion. That is why a maximum depth is set from the outset as a fixed limit, so that the program doesn't get caught in an endless loop.

All in all, then, complex sequences behave in a rather chaotic way, and that is exactly what we were after. But now for the punch line: one day the mathematician Benoit Mandelbrot, at the IBM research center in New York, had the brilliant idea of coloring the points of the complex number plane according to the iteration depth reached before termination—and he was more than a little amazed when he saw the result: the first gingerbread man was born! By the way, at first he took the fine structures along the inner edge of the figure for blurring caused by rounding errors. Only when magnified sections were calculated did the astonishing fact come to light that this graphic contains an infinite number of intricate substructures.

Once the number of iteration steps before termination has been found, MODE 1 offers a total of 4 colors (PEN 0 to PEN 3) for coloring the points

with the PLOT command according to the depth reached. PEN 0 can be used for the maximum depth, for example; otherwise the formula color = depth MOD 4 makes the colors repeat cyclically.

The calculation routine of the program *Komplexe Grafik* (Complex Graphics, KOMGRA.BAS) was written in machine language to make it fast. Since the algorithm is hardly recognizable in that form, here is a BASIC listing as well, which essentially corresponds to the machine program. For practical use, though, it would be far too slow.

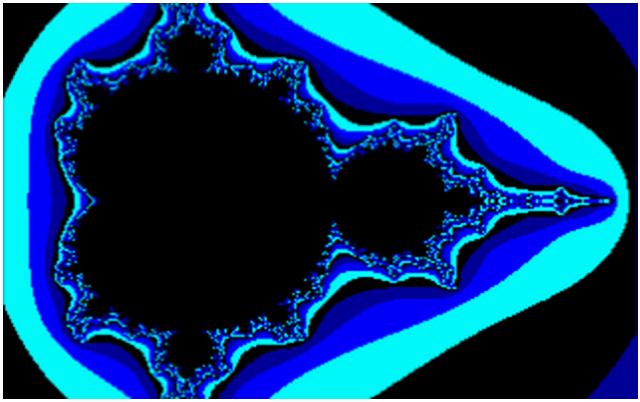
```
10 MODE 1
20 xl = -0.7: xr = 2.1:REM left and right edge, real
30 yu = -1: yo = 1:    REM lower and upper edge, imaginary
40 MaxTiefe = 30:      REM maximum iteration depth
50 dx = (xr - xl)/319: REM x step size
60 dy = (yo - yu)/199: REM y step size
70 y = yu
80 FOR Zeile = 0 TO 199
90    x = xl
100   FOR Spalte = 0 TO 319
110     Tiefe = 0: re2 = x*x: im2 = y*y
120     WHILE (Tiefe < MaxTiefe) AND (re2 + im2 < 8)
130       re = re2 - im2 - x: im = 2*re*im - y
140       re2 = re*re: im2 = im*im: Tiefe = Tiefe + 1
150     WEND
160     IF Tiefe = MaxTiefe THEN f  = 0 ELSE f = Tiefe MOD 4
170     PLOT 2*Spalte,2*Zeile,f
180     x = x + dx
190   NEXT Spalte
200   y = y + dy
210 NEXT Zeile
```

Operating KOMGRA.BAS

And now for how to operate the program *Komplexe Grafik*: once it has started, a small menu window appears on the screen. This is a *pull-down menu*, which you can call up or make disappear at any time with the ENTER key without destroying the graphic. You select the individual menu items

with a pointer, which is moved with the cursor keys up and down. The COPY key activates the function the pointer is pointing to.

To produce the *gingerbread man* as the basis for further pictures, it is best to go straight to the item *Zeichnen* (draw). The program then wants to know the maximum iteration depth; enter the value 30 here and press ENTER. The CPC will then keep itself busy with the picture for about an hour.



When it is finished, the menu window appears again, and with the function ***Speichern*** (save) you can save the graphic to cassette or disk after entering a file name. The program automatically appends the extension .PIC (picture) to the name to mark the binary file as a picture file. With *Laden* (load), too, you must not type the extension, since the program adds it by itself! When saving to disk, be absolutely sure that the name is correct and that there really is a disk in the drive, since otherwise the error message will destroy the picture.

Incidentally, all the important picture data (colors, coordinates, depth) are recorded along with the picture. A trick saves us from needing extra parameter files: the values are kept in an invisible area of the screen memory! After loading, they are read back and set correctly. This even makes it possible to save an unfinished picture and have the program carry on drawing it whenever you like. You interrupt the drawing with ENTER, although the CPC does not react until it has finished the current raster line, which can take a while. The program automatically recognizes an unfinished graphic in memory;

once you select ***Zeichnen*** the CPC carries on with its work without asking silly questions first.

The menu item ***Farben*** (colors) lets you select the four pens (PEN) with the cursor keys up and down, and then change the color assigned to each (INK) with the left and right keys. The number of the color is displayed all the while—see the CPC user manual. ENTER takes you back to the main menu.

The function ***Ausschnitt*** (detail) lets you pick out a particular region of the complex number plane and have it drawn suitably magnified. If there is already a picture in memory, a rectangle appears on the screen, which you can move around with the cursor keys. For fine adjustment, use the cursor keys together with SHIFT or CTRL; they move the upper left or the lower right corner of the region pixel by pixel. In this way you can change the shape and size of the rectangle as you like. It is advisable, however, to choose an aspect ratio roughly matching that of the screen, so that the drawing doesn't come out badly distorted.

If you are interested in the coordinates of the region, you can simply call them up with COPY; any key makes them disappear from the screen again. ENTER completes the choice of the region, and the menu appears. If you select the item ***Ausschnitt*** without a picture in memory, a rectangle makes no sense, of course. In that case the program expects you to define the region by entering four numbers: left edge; right edge; top edge; bottom edge

Here, too, ENTER without an entry takes you back to the main menu. The ***Zeichnen*** part of the program always draws the region set last. If you don't specify a region after starting the program, then, as mentioned before, the *gingerbread man* is drawn automatically with the following coordinates:

$$\text{left} = -0.7; \quad \text{right} = 2.1; \quad \text{bottom} = -1.0; \quad \text{top} = 1.0$$

When the program draws a new picture, it simply draws over the old one. So the old picture stays on the screen unless you first select the item ***Löschen*** (clear). This function erases the picture, but not a region you have set beforehand!

For the next picture, it is best to choose a region that already shows an interesting structure from the start. From the magnifications you can, of course, have further details drawn, and in this way work your way deeper and deeper into the picture, as far as the CPC's numerical precision allows.

The inner edge of the gingerbread man in particular has a lot to offer. If, after selecting the menu item ***Zeichnen***, you enter a greater depth (255 at most), more shapes come to light: the greater the depth, the more structures the program can capture. But the computing time for a complete picture also goes up enormously and can stretch to several hours. So you do need a bit of patience for this adventure in the land of complex numbers!

Interesting coordinates to try out:

left	right	top	bottom	depth
1.778	1.793	0.0069	−0.0069	30
0.038	0.103	0.834	0.845	60
0.090649	0.09416	0.84146	0.84378	60
0.09282	0.094321	−0.84301	−0.84329	90
0.7445	0.7459	−0.1121	−0.1132	200

Further reading:

– Becker/Dörfler: Dynamische Systeme und Fraktale, Vieweg
– Peitgen/Richter: The Beauty of Fractals, Springer-Verlag

CHAPTER 2

The Game of Life

Powerful, graphics-oriented computer systems are becoming more and more important in industry, science and art. In many cases, developing and displaying highly complex structures has only become possible at all through the use of image-processing systems. And, astonishingly enough, even research projects with purely practical aims often turn out pictures you would sooner expect from a brilliant artist. The chaos graphics covered in the last chapter are surely one of the most astonishing examples of this.

But for all our enthusiasm about what the new technology can do, we shouldn't forget that nature has countless creations to show next to which most computer graphics look rather pale. Have you ever looked at snow crystals through a magnifying glass or a microscope? If not, you should make up for it at the next opportunity, or at least read what the novelist Thomas Mann has to say about the experience:

Jewels, stars of knightly orders, diamond brooches such as the most painstaking jeweler could not have fashioned more richly or more minutely... and among the myriads of little magic stars, in their secret, diminutive splendor, invisible, never meant for the human eye, not one was like another.

The inexhaustible inventiveness with which nature produces billions of variations on the theme of sixfold symmetry still seems an incomprehensible miracle. And yet, with the help of cleverly devised computer simulations, it has now become possible to investigate and understand at least some of the basic principles behind these natural processes.

Among other things, it has been shown that a simple set of suitable rules (laws of nature), applied over and over to certain basic elements (molecules), produces structures of any degree of complexity with almost compelling necessity. On the strength of these findings, some researchers speak of a capacity of matter for self-organization, a theory with far-reaching consequences. How, for example, did life on our planet come about—by a unique accident? If matter, under the influence of the laws of nature, acquires the

ability to structure itself, then perhaps the particular forms life takes are a matter of chance, but not the fact of organic life as such. It was bound to arise, with high probability, as soon as the right conditions were in place!

In the Beginning Was the Dot...

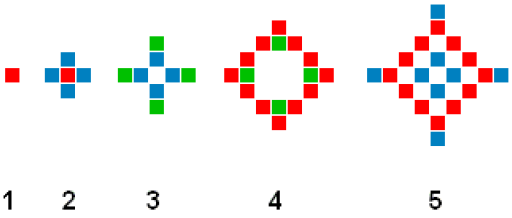
How extraordinarily complex patterns can grow out of the simplest premises will now be demonstrated by a little software experiment on the CPC, a real showpiece of the art of mathematical experiment. You have probably never seen such exquisite filigree graphics on your screen as the ones produced by the program *Reproduktionsmuster* (Reproduction Patterns, REPRO.BAS)! The program works with the simplest basic elements imaginable, namely pixels that are either set (bright) or reset (dark). On top of that come two laws, which are applied iteratively (that is, repeatedly) to the entire screen:

– Simultaneously set all points that have **exactly one** bright point in their neighborhood (left, right, above and below).

All the points newly created in this way during one pass are called a generation. With this definition, the second rule can be stated:

– After generation (n) has been created, erase all points of generation (n-2).

So after a while the points die of old age; after each pass, exactly the two most recent generations are visible on the screen. As long as the rules are applied to an empty screen, of course, nothing at all happens. But a single point, set in line 410 of the program as a seed crystal, starts the avalanche rolling. The following figure illustrates the principle and shows how the first five generations develop:



The machine program called by CALL &A000 carries out the rules and takes care of the screen display. Its programming, incidentally, makes use of special techniques for fast graphics that work directly on the screen memory. The normal system routines or a BASIC program would be hopelessly slow, since up to 384000 points per generation have to be tested for their color!

Showing the generations in different colors is not just a matter of looks. It allows the program to recognize generation (n-2) without storing a huge number of coordinates. The colors also make it possible to apply the laws quasi-synchronously: points already set while the current generation is being created are not yet counted as neighbors. Of course, the computer can't take in all the pixels at the same time, as the rules demand; it can only deal with them one after another—a fact that has to be compensated for by suitable programming tricks to avoid glitches.

If you would like to experiment some more with this system, you can try a different initial situation. The point created in line 410 can be replaced, for example, by a letter or a graphics symbol. Pictures that grow from several seeds scattered across the screen are also very appealing. Although the typical character of the graphics is preserved, each starting figure leads to a different course of events, and the analogy with the formation of snow crystals becomes clear: the particular shape is determined by the crystal nucleus (e.g., a grain of dust), but the characteristic symmetry arises from the laws governing the molecular binding forces.

The Cellular Automaton

Before we turn to another software experiment built on similar foundations, let us first recall how the CPC processes information. As you surely know, your computer contains a microprocessor as its central control unit. It reads binary values from the memory cells, combines them in its internal registers according to certain rules, and puts the result back into other memory cells.

This kind of information processing is called sequential: complicated calculations are carried out by working through the necessary steps one after another under program control. Practically all digital computers work on this principle, and it is only their high processing speed that sometimes creates the impression that the computer can do several things at once.

The computer is thus a faithful image of the language-oriented human intellect. Verbal information processing, too, can only take place sequentially. Just try consciously thinking two different thoughts at the same time: you will see that it is practically impossible.

The intellect, however, is only a very small part of the information-processing system known as *the human being*. The various forms of sensory perception alone require a processing system that works on an entirely different principle. After all, we have no trouble seeing or feeling several things at once—in other words, human beings can take in a large number of different data in parallel.

The intellect is capable of analyzing the information supplied by the sense organs on an abstract level, but it is simply too slow for evaluating it on the spot. Just imagine that, in order to recognize a house, you had to consciously break the image down into a grid of dots, assign each dot a color value and a brightness value, then compare the data with a grid you had stored earlier and from that work out a value telling you how likely it is that you are looking at a house… the long and the short of this long sentence: you would be practically paralyzed!

This example makes it clear why even the best and fastest computers, for all the researchers' efforts, are unable to see or hear in the human sense: the way they work is simply unsuited to it. Experiments with multiprocessor systems are already under way, but they, too, run into considerable practical difficulties. You can't expect a computer with 100 processors to run 100 times as fast as an ordinary one, for example, since part of its capacity is needed to coordinate the data transfer between the various units.

Interestingly enough, though, there is an approach to solving these problems that the mathematician John von Neumann studied theoretically back in the days before computers. The question, raised by Alan M. Turing as early as 1936, of whether a universal thinking machine could exist led him to construct, on paper, a so-called cellular automaton. This is a homogeneous space divided into cells, in which each cell can take on a certain number of states (e.g., different colors). A set of so-called transition rules is applied synchronously to all the cells; exactly what each rule does depends on the state of the cell in question and of its adjacent neighbor cells.

John von Neumann was able to prove theoretically that a configuration of 200,000 cells, each with four neighbors and 29 possible states, is enough to carry out, in principle, any conceivable computation. Such a cellular automaton has no central processor; instead, the memory cells communicate with one another directly, without any detours. With every clock pulse, which brings about transitions throughout the entire space according to the catalog of rules, a large number of operations are carried out at the same time. Information no longer appears as bits and bytes but as spatial or planar patterns that can overlap and influence one another—a principle that nature also applied when it built the human brain.

The Game of Life

Probably the best-known illustration of the cellular automaton principle is the Game of Life, invented in 1970 by the British mathematician Conway. As in our previous experiment, each cell in this game can take on two different states: it is either inhabited or uninhabited. The neighborhood, however, has been extended to include the diagonally adjacent cells—so each cell has eight neighbors—and the transition rules have been formulated so that the course of events depicts the effects of isolation, overpopulation or cooperation in a society of living beings quite realistically.

The rules can be summed up in two simple commands:

- Synchronously fill all empty cells that have exactly three neighbors.
- At the same time (!), erase all cells that have fewer than two or more than three neighbors.

Another way of presenting these laws is a catalog of rules in table form, laid out to suit the computer. It gives the occupancy of a cell in generation (n+1) as a function of its current state and the number of its neighbors. A 1 means that the cell will be inhabited in the next generation, while a 0 stands for an empty cell:

Number of neighbors	0	1	2	3	4	5	6	7	8
0 – cell becomes	0	0	0	1	0	0	0	0	0
1 – cell becomes	0	0	1	1	0	0	0	0	0

At first glance, then, living conditions for the single-celled creatures of Life seem pretty grim; they are very sensitive to loneliness and overcrowding. A few experiments soon show, however, that it is precisely these restrictions that give the system its dynamism and lead to highly interesting developments. No wonder this game caused quite a stir at universities and research institutes after it was invented—and sometimes even trouble, since the terminals of the computer centers were regularly tied up by Life fanatics. After all, there were no home computers for everyone back in the seventies.

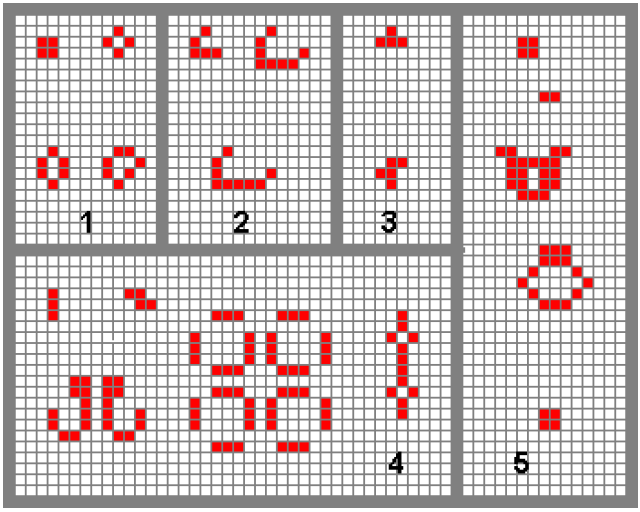
Fortunately, times have changed, and with the program Game of Life (LIFE.BAS) you can experiment on your CPC for hours on end without having to fear that the operating system will kick you off the computer any moment for exceeding your allotted computing time.

In this program the cells are not shown pixel by pixel, as with the reproduction patterns, but somewhat enlarged; altogether there is a field of 50x80 = 4000 cells. The transition rules are again carried out by a small machine program, which manages about two generations per second. Special care was taken to make the screen output extremely fast. And indeed, the transitions appear to take place simultaneously over the entire field.

Once the program has started, you first see the empty playing field, which is marked with a check pattern so that the individual developments can be analyzed accurately. To get an impression of the hustle and bustle of the single-celled Life creatures, you can first press the Z key, whereupon the computer produces a random distribution on the screen. Wait a while until the field is reasonably full, and then stop the process with ENTER. Now you can create each next generation with the SPACE bar, or let the generations

follow one another automatically with A. This function, too, is stopped with ENTER.

To study the properties of particular configurations in a systematic way, you can call up a small editor with E. The cursor is moved with the arrow keys as usual; the 1 key fills the cell at the cursor position, and the 0 key empties it again. To produce whole rows of Life creatures, you can also hold down the 1 key and move the cursor across the screen; it then leaves a trail of filled cells behind it. ENTER leaves the edit mode again; L clears the entire field if you need to.



The figure above shows some typical Life configurations. The group numbered 1 contains four stable figures, which stay unchanged from one generation to the next. Group 2 shows three so-called gliders. These figures are oscillators that periodically regain their original shape but move on by a small step in the process. Stationary oscillators with a period of 2 to 15 generations, on the other hand, are gathered in group 4. A particularly peculiar behavior is shown by the 30-generation oscillator that makes up group 5: without losing any of its substance, it turns out a series of small gliders, which wander off to the right.

Group 3 contains two more figures, which behave expansively. But while the upper figure breaks up into four stationary oscillators after only a few

generations, the lower figure is perfectly capable of filling the whole screen with its offspring. It is considered the smallest figure that keeps on spreading indefinitely; so far no definite final state has been found.

If you want to test all the patterns shown together, it's best to remove the expansive figure and the three gliders from the collection so that they don't disturb the other figures. Apart from that, though, it is very interesting to see what happens when different patterns collide: the result can be total annihilation, or new figures can emerge. Thanks to the tireless efforts of the many Life researchers, people have meanwhile even managed to find patterns that, by influencing one another, simulate logic functions such as NOT, AND, OR or XOR—an important step toward a cellular automaton of practical use.

Unfortunately, one problem that rules out building even a small cellular computer on the CPC is the unavoidable edge of the playing field. Even a 320x200 field would still be much too small. The program treats all cells outside the field as empty, and this causes serious disturbances as soon as a pattern hits the edge. The glider at the top right, however, comes to terms with this restriction in an interesting way: it is reflected and turns into a smaller glider in the process!

As you will find in your own experiments, the limited living space means that every configuration gets stuck after a while; all that remains are stable patterns and oscillators, which can no longer bring about anything fundamentally new. But this phenomenon has yet another cause: the Game of Life makes no room for creative chance. The whole process is deterministic, i.e., once the initial situation is given, everything that will happen is practically settled down to the last detail. But you might find out what the following additional program lines do:

```
981 f=1
982 re= INT (RND*50) + 1
983 sp= INT (RND*80) + 1
984 GOSUB 1060
```

If you now select the automatic change of generations with A, a new creature is born at a random spot on every pass and brings a little variety into the proceedings. As hinted at in Chapter 1, chance plays an important part in the creation of dynamic (living) structures.

And there is one more very interesting way to experiment: DATA lines 530 and 540 hold the Game of Life catalog of rules mentioned earlier. You can, of course, arrange the zeros and ones differently here and so create new transition laws. What happens under a modified catalog of rules is largely unexplored territory and still leaves plenty of room for exciting discoveries.

Evolution in the Computer?

After this rather impressive simulation of a cellular automaton, one question suggests itself: why has no one actually built such a machine yet? The hardware shouldn't be the obstacle; with today's technology the necessary chips could surely be made. No, the problem lies somewhere else entirely: nobody knows how a cellular automaton could be programmed.

As we have already discussed, every language—and so every programming language—is based on sequential logic. We can, of course, also talk about events that happen simultaneously, but only by describing them one after another. It is therefore very doubtful whether a powerful cellular automaton, carrying out hundreds and thousands of operations at once and forming countless patterns that influence one another every which way, can ever be mastered with the means of a sequentially working intellect.

So far only one way out of this dilemma is in sight: the machine has to program itself! When you program your CPC, you use a programming language to formulate an algorithm that specifies, step by step, how result B is produced from situation A. How easy it would be if you could switch on the computer and simply tell it: 'Here, my electronic friend, this is a chessboard, these are the pieces, and these are the rules... and you win if you capture my king before I get yours. Play so that you win if at all possible!'

You see—that is exactly what the cellular automaton would have to be able to do: find ways to solve problems on its own. As we have seen, creating self-organizing systems is no problem. But in this case, a few transition rules that produce some patterns or other won't do the job. To play good chess, the machine would have to produce structures purposefully, geared to the problem, discard unfavorable patterns, gather experience and evaluate it selectively... in other words, go through a learning process, an evolution.

But how to formulate evaluation schemes that steer a self-organizing process toward a goal is a problem that, so far, has only begun to be solved, even

in theory. Some clues, however, come from a field of research that deals with the borderland between biology and computer science. What strategies did nature use in developing life? What statistical laws help certain species to flourish while others die out?

The next chapter of the Software Experiment will provide more material on this subject. Anyone who wants to look into this field more deeply should read the book *Das Spiel—Naturgesetze steuern den Zufall* by Manfred Eigen/Ruthild Winkler (Piper Verlag). It contains numerous simulation games that can easily be implemented on a home computer, too.

CHAPTER 3

The Ecology Experiment

Simulations—replicas of reality—have won themselves a firm place among the standard applications of computer systems. Whether in training pilots for aviation or space flight, in scientific research projects or in economic and military planning: in many fields it is impossible to imagine doing without them.

Why no computer simulation has so far managed to convince the politicians and generals in charge of how dangerous and senseless the arms race is remains an open question. In this installment of the Software Experiment we will deal with another subject that is just as topical. The last installment already introduced the Game of Life, a program that simulates an ecological system in a rudimentary way; and we want to pursue this idea a little further.

So far, human interventions in the extraordinarily complex dynamics of ecological systems have taught us again and again that the results often look quite different from what was actually planned. The long-term consequences of measures that at first seemed very promising, in particular, bring home to us that in the end it is humans themselves who are affected when they destroy the delicate balance of nature through thoughtless interference. Using chemicals improves crop yields—or so it indeed seems at first. But when, after a while, the pests being fought turn out to be resistant, while their natural enemies perish from the poison in droves, and the health consequences of the residues in our food can no longer be justified for humans—what then?

The dynamics of ecological systems are determined by a multitude of interlocking feedback loops and overlapping influences. It is therefore hardly possible to capture the consequences of interventions with simple, causal if-then models. So what could be more natural than to use computer simulations to gain some insight into this so well-ordered chaos? The speed

and storage capacity of modern computers make it possible to run developments that in reality stretch over years and decades in a few seconds, and to explore a whole range of possibilities by deliberately changing the system parameters.

But caution is called for: an image of reality in a computer's memory is always a reduced reality. Just as a map can show various aspects of a landscape but never the landscape as a whole, a computer model can only capture a selection of important factors, and it is the programmer who has to make this selection. That is why one should not overestimate what simulations can tell us. Like maps, they give an overview, but they are no good as prophets. A computer that simulates the functions of a nuclear reactor, for instance, will be able to show in what ways accidents are possible, but not that they are impossible!

The Wolf-Sheep-Grass Model

Many of the games available for the CPC are, in principle, simulations too; sports games or the space adventure Elite would be examples. But while what matters there is essentially the quality of the graphics and sound effects, scientific simulation programs face entirely different demands. They are supposed to reflect reality quantitatively, i.e. in the form of numerical values, and as exactly as possible. This material gives the researcher a suitable basis for gaining new insights or testing certain theories. To the uninitiated, however, long columns of figures or convoluted on-screen diagrams are generally not particularly illuminating.

Working with programs like these requires a fairly solid mathematical grounding (growth functions, for example), and discussing it would go far beyond the scope of this book. So as an introduction to the subject of *ecology simulation* we will deal with a system that makes no claim to scientific exactness but nevertheless allows a whole range of interesting experiments, and offers something for the eye as well: a time-lapse display of a small artificial habitat in which the fate of the individuals is not merely of statistical significance but can also be watched one by one.

Our starting point, then, is the Game of Life discussed in the last chapter, though with a few essential changes. First of all, each cell of the playing field should be able to take on not just two but four different states. To make

things vivid, we will right away give the states, represented by the numbers 0..3, a concrete meaning:

0 = empty 1 = grass 2 = sheep 3 = wolf

What happens to the content of a cell during the game depends, much as in the Game of Life, on its immediate neighborhood; the neighbors are the four directly adjacent cells (above, below, left and right). You can probably already guess what happens to a sheep that finds itself near a hungry wolf—but we'll get to the exact wording of the rules in a moment.



First, though, another important difference from the Game of Life: as already mentioned, Conway's model is strictly deterministic; it leaves no room for chance. About a real ecological system, however, one can only make statistical statements, for example about the frequency or probability of certain events. When and where a wolf will pounce on a sheep cannot be calculated exactly; the precise coordinates of the events in time and space are completely undetermined.

Our ecology experiment takes this fact into account: the events prescribed by the rule catalog are no longer carried out simultaneously over the entire playing field but one after the other, with the computer rolling the dice for the location of each event using the RND function. So it remains clear that a sheep near a wolf can't exactly feel safe; only when it will be eaten—that is written in the stars.

Eat and Be Eaten

Here, then, are the laws that govern neighborly relations in our WSG (wolf-sheep-grass) model:

(1) An empty cell with grass in its neighborhood is occupied by a new tuft of grass.
(2) A cell covered with grass is occupied by a sheep if there is at least one sheep in its surroundings.
(3) A sheep is replaced by a new wolf if there is at least one wolf in its neighborhood.
(4) If a sheep has no grass in its surroundings, it is replaced by an empty cell.
(5) If a wolf has no sheep as a neighbor, it too is replaced by an empty cell.
(6) In all other cases no conversion takes place.

So in rules (2) and (3), eating and reproduction are coupled, which is not necessarily realistic but leads to very dynamic processes. Rules (4) and (5) leave wolves and sheep to starve to death if they get too greedy.

Of course the CPC can't do much with the rules as worded in plain language, which raises the question of how to represent the rule catalog in a computer-friendly form. Programming a case analysis with lots of IF..THEN statements is the obvious approach, but it quickly gets confusing. Changing the catalog later is also very cumbersome that way.

For our purposes the table form shown below is much better suited. In any case, this system is easier to handle than it looks at first glance: the eight columns on the right and the rows marked with question marks need not concern us for now, since they only play a role in an extended system with five cell states; and the meaning of the remaining entries is quickly explained.

Neighborhood	Grass	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0 = absent	Sheep	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1 = present	Wolf	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
	????	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
Conversion	0 = Empty	0	1	0	1	0	1	0	1	0	0	0	0	0	0	0	0
instructions for	1 = Grass	1	1	2	2	1	1	2	2	0	0	0	0	0	0	0	0
the center cell	2 = Sheep	0	2	0	2	3	3	3	3	0	0	0	0	0	0	0	0
	3 = Wolf	0	0	3	3	0	0	3	3	0	0	0	0	0	0	0	0
	4 = ????	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

The upper part of the table shows all possible neighborhood configurations. Anyone with some knowledge of machine language will have noticed at first glance that this is a collection of the binary numbers 0...15, written vertically. This arrangement helps the computer find the relevant column in the rule catalog after it has analyzed the neighborhood. The lower part of the table contains the conversion instructions for the cell picked by the dice. If the computer, for instance, catches a sheep surrounded by grass and a wolf, it has to look in the sixth column from the left. In row 2 of the lower half of the table, the one responsible for sheep, it then finds a 3 at the intersection, so the poor sheep is replaced by a new wolf.

Working through further examples, you can convince yourself that the table entries reproduce exactly the six rules of the WSG model, albeit in a rather abstract form. But with a little practice, coding laws of your own invention is no problem. For ambitious programmers it might be an interesting project to invent a symbolic eco-language and write a kind of compiler that generates the table form from it.

The Eco Program

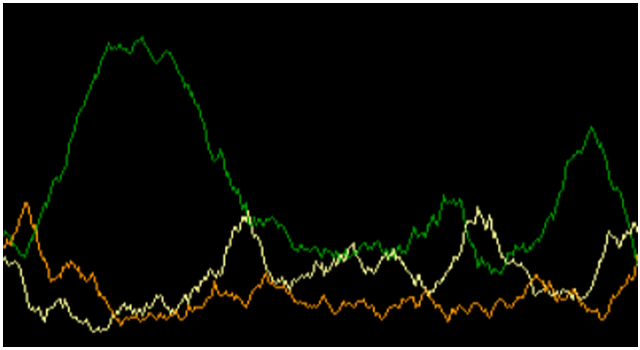
The simulation program **OEKOEXP.BAS** is largely modular in structure and provided with numerous comments, so that changes are easy to make. The main program at the beginning has only one job: to call the various parts of the program in the right order.

In the initialization section starting at line 1000, the arrays for the playing field, the rule catalog, etc. are dimensioned. The actual definition of the WSG model follows from line 2000 on. To display the various inhabitants of our

ecosystem, double-height graphics characters are created, which are assembled from line 2150 on using a few control characters—even the matching color is built right in with CHR$(15). So that printing these super characters at the right edge of the screen causes no trouble, a POKE in the initialization section fools the CPC into believing that MODE 0 has 40 columns.

The rule table is read in this part of the program too; it sits in DATA lines 2300—2340. Here you can make any changes you like to influence the course of the ecology experiments. The subroutine starting at line 3000 prepares the game by generating a random distribution of wolves, sheep and grass as the initial situation. Since the program may end up looking past the edge while exploring a neighborhood, the playing field is internally given a frame filled with the neutral digit 9.

The actual game runs from line 4000 on. First the program waits for a key press as the start signal, and then it sets a time interval of four seconds (200 * 1/50 sec) for the regular census taken by the subroutine at line 6000: it records the current population figures, so that at the end the module at line 5000 can plot them as a function of time.



The subroutine starting at line 10000 is available to the other modules. It not only changes the cell contents but also brings the various counters up to date. The coordinates (X,Y) of a cell and the new content, in the variable neu (German for new), are passed to it as parameters.

Catastrophes in the Computer

Once the program has started, created the initial distribution and begun the game, a lively bustle will unfold on the screen; in addition, three counters at the bottom of the screen report the current population figures. Another counter continuously shows the number of events rolled so far. As becomes clear after only a few tries, the processes are completely different from those in the Game of Life. While the simulation game described in the last chapter tended to stagnate in fixed forms after a while, the chance-driven dynamics of the WSG model easily lead to catastrophes in which one or more species die out completely.

It's true that the WSG laws are formulated so as to counteract extreme developments: a strong increase in sheep, for example, also brings about a corresponding increase in wolves and a reduction of the food supply. But since the spatial distribution of the creatures plays a big part in our game, the control mechanisms often kick in only with a delay, and more or less violent swings in the population figures arise, which can be seen very nicely in the curves drawn after you stop the program with ENTER. Especially on a confined playing field of only 200 cells, fluctuation catastrophes happen

easily. Just how strongly the resilience of an ecological system depends on the size of its habitat is what a further experiment will now show.

For this purpose, extension 1 (OEKOERW1) has to be merged into the main program with the MERGE command. Make sure the main program is in memory, and then type **MERGE "OEKOERW1"**, whereupon the new program parts are loaded in. The extensions still to come can be added the same way. Before that, however, you always have to reload the original program **OEKOEXP.BAS**, since the extensions don't get along with each other!

Extension 1 cuts the playing field into two parts, isolated from each other by a highway. In addition, a new rule comes into force:

(7) Sheep and wolves next to the highway get run over, i.e. replaced by an empty cell.

To represent the highway, the digit 4 is used internally as a fifth cell state, an example showing that this coding can represent more than just living creatures. The new rule catalog for the extended system naturally contains nothing but fours in its last row, since the highway cannot be converted.

The eight columns on the left correspond to the original WSG laws; the eight columns on the right reflect the changes that occur right next to the highway. As a few tries show, the ecological balance in at least one of the two areas is usually destroyed after a short time. The direct effect of the highway doesn't even play the decisive role here. Much worse is the reduced tolerance for the natural fluctuations of the system. A small system is of course far more easily overwhelmed by a swing of +/- 20 animals than a large habitat—a statistical effect that is easily overlooked in land-use planning.

The Ecological Balancing Act

On the other hand, of course, the question arises which measures are suitable for preventing such catastrophes. Two approaches will be examined below:

1. Improving the Energy Supply

Our WSG model is a so-called dissipative system, which has to convert energy constantly just to keep itself going. This energy is represented by the regrowing grass; without growing grass, neither sheep nor wolves stand any chance of survival. Improving the energy supply is very easy: just write ones

all the way through the first DATA line of the rule catalog, after which rule (1) takes the following form:

(1) An empty cell is always occupied by new grass.

So the grass can now grow anywhere (e.g. from wind-blown seeds); having other plants in the neighborhood is no longer necessary. A few tests quickly show that this does indeed give more stability; on average the game runs much longer without a catastrophe. Such ideal conditions are only realistic up to a point, however: plant growth this unchecked will generally take place only part of the time, i.e. in spring or summer.

2. Human Intervention

A critical situation often arises in the WSG model shortly after the start. The thorough mixing of the creatures in the starting distribution leads to a strong increase in the wolf population and a reduction of the vegetation, which promptly plunges the sheep into a crisis. But help is on the way: with extension 2 we will send a few hunters into the field to keep the wolf population down. To do this, proceed exactly as with the first extension: reload the main program **OEKOEXP.BAS** and then type **MERGE "OEKOERW2"**. New DATA lines for the rule catalog introduce the following laws:

(-) An empty cell next to a wolf is occupied by a hunter.

(-) A wolf that has a hunter as a neighbor is replaced by an empty cell.

(-) A hunter with no wolf in his neighborhood disappears again.

The consequences of this addition are quite drastic, however: in most cases the diligent hunters will wipe out the wolves fairly quickly, which does the sheep good at first, but not the grass. The screen gets grazed literally bare. Only the famine that then sets in sometimes prevents a total wasteland.

As you can see, it's not at all easy to design a system that behaves dynamically and yet stably. Anyone ambitious enough to tinker with the rule catalog to develop their ideal ecology should bear in mind that the tendency of a system generally becomes apparent only after several runs. Because of the strong influence of chance, a single run only ever shows what can happen, not what will happen! For this reason our program is quite exciting, but in its present form it is not suitable for developing scientific theories: the experimental results are not reproducible.

Software Author Wanted

On the other hand, our eco-experiment offers something that scientific programs usually don't: it can be expanded into quite an interesting game program. So far all you could do was define the rules and passively watch what happens, but a third extension will show how a way of intervening directly can be programmed.

The program part **OEKOERW3**, which is merged in with **MERGE** just like the previous additions, puts a hunter on the playing field whom you can walk around with the **cursor keys**. Naturally he can't step onto cells already occupied by wolves or sheep, and since he also tramples the grass under his feet, you should move him through the landscape with some care.

On the other hand, he also has a few options for action at his disposal. If you hold down the SPACE bar and press the **cursor keys** as well, then depending on the situation, an empty cell in that direction is planted with grass, or a wolf or a sheep is shot. But nothing is without risk: as you can see from the last DATA line, the hunter is on the wolves' menu as well. If he is next to a wolf and happens to be picked by the dice at that moment, he's done for!

In this way you can step in and regulate the course of your ecology experiments. For a real hit game the program still lacks a gripping motivation, but a few additional ideas could change that. Here are a few suggestions:

Use several rule catalogs that are swapped periodically or at random to simulate seasons or epidemics.

- Give the hunter limited resources (grass plants, ammunition), which have to be used as sensibly as possible to maintain the ecological balance.
- Design a landscape in which the cells offer different qualities of life. For this purpose an additional array lq(x,y) (lq for Lebensqualität, quality of life) could be defined, for example, which assigns each cell of the playing field a value that influences its inhabitants' chances of survival.
- Populate the playing field with further species of plants and animals.

The last point, however, is not easy to get a handle on in terms of programming, since the number M of entries in the rule table grows with the number Z of possible cell states according to the formula

$$M = Z \cdot 2^{(z-1)}$$

– that is, exponentially. So here one would probably have to find a more suitable algorithm.

Further Reading:

Hartmut Bossel: Umweltdynamik, tewi Verlag

Hans Rauch: Modelle der Wirklichkeit, Heise Verlag

CHAPTER 4

The Computer as Strategist

Strategy games are still a popular testing and training ground for the methods of artificial intelligence. A manageable number of pieces, a precisely defined set of rules—the computer likes that sort of thing, because here it can show what it can do, undisturbed by the often incalculable events of the real world.

Even so, the demands made in this kindergarten of artificial intelligence are already quite grown-up. What is called for in particular is

- combinatorial thinking, i.e. the ability to calculate possible sequences of play in advance and to estimate their consequences.
- strategic thinking, i.e. the ability to develop workable plans and carry them out purposefully.

No question, machines with these abilities are of interest not only to the chess-playing computer freak; military men and managers, too, find a welcome tool here for threatening one another, or swindling each other out of money, even more effectively. But the sad fact that machine intelligence paired with human stupidity can make an absolutely catastrophic mixture shall not stop us from exploring, in the next chapters of the Software Experiment, some of the fundamentals that turn computers into wily strategists—trusting that you, dear reader, are too intelligent to put the insights gained to use in some ingeniously devised suicide system.

Each of the game programs discussed in the next chapters demonstrates a particular working principle and, in keeping with the character of the Software Experiment, will not only make a good opponent but also invite you to tinker and experiment; as always, you are called upon to join in. And this much can be promised already: we will not deal with trivial games such as tic-tac-toe, which usually gets trotted out on such occasions, but with strategy games that are sure to give you plenty to think about.

Our first study project will be the Japanese game of Gobang (also known as Go-Moku), one of those charming ancient games that produce very interesting and complex play with simple means and are therefore especially well suited for programming. Before we turn to that subject, however, we will first take a look at how humans actually play strategically.

Of Masters and Patterns

According to popular opinion, chess masters are mental acrobats who can calculate a multitude of possible moves ahead in a flash and are therefore far superior to the average amateur player. But is that really how it is? The answer the American chess master Marshall gave when asked how many moves ahead he generally calculated is surprising, to say the least: 'Two, but two good ones!' At first you feel as if someone is pulling your leg—is that all there is to it? But it has indeed turned out that chess masters are perfectly capable of thinking certain sequences of moves through very far, but in doing so consider only a small selection of the many possible moves—namely, as we said, only the good ones.

For the chess programmer this is very frustrating: how is he supposed to transfer this hard-to-define instinct for the quality of certain moves to his machine? As far as combinatorial ability goes, modern chess machines have certainly reached master level, but they still lack the eye for the position that tells the master at a glance which moves are worth considering at all.

The programmer gets unexpected backup, however, from the psychologists, who are also keenly interested in how such mental feats come about. They have found that the chess master looks less at individual pieces and their possible moves and instead recognizes certain substructures on the chessboard. These patterns are formed, for example, by the pawn chains in a particular opening, by the grouping of the pieces on the king's side, or by the placement of the pieces in the center. The master sees these patterns as a whole, just as we can understand a word immediately without first breaking it down into its letters, and knows from experience which continuations are especially promising in the particular situations.

Fine, says the chess programmer, then we'll do it the same way. First we buy a hard disk drive so we have enough storage space, then the computer gets a list of all relevant substructures that can occur in chess, and last but

not least we hire a specialist to type in the corresponding good moves—and presto, there's our electronic grandmaster! And this concept does indeed look very promising: powerful algorithms for identifying patterns (pattern recognition) have already been researched and are well known, there is no shortage of storage space these days either, and when you add the speed of modern computers and the combinational strength that comes with it… yes, then a computer really would soon be world chess champion, if it weren't for one decisive obstacle.

Let's recall the analogy with language once more. To grasp the sense of a word, it is usually not enough to identify the word; we also have to grasp the particular meaning of the word within a sentence, or even a whole text: the context. In principle it is no problem to cram an entire dictionary into a computer and teach it all the rules of grammar, but that doesn't mean by a long shot that it now understands the language!

While there is always more than enough storage space for the words, declensions, conjugations, etc., the number of possible relations of meaning between different words reaches astronomical dimensions and is very hard to get a handle on in a program. And there's the rub: the chess master not only knows a large number of substructures (chess words), he can also relate them to one another and recognize their meaning within a larger context, and to this day no computer can match him at that.

This fact leads to an essential difference in the way human and electronic chess specialists go about things: the computer has to try out all moves (and usually thousands of follow-up moves as well) before it can judge them by the resulting position; the grandmaster first judges the position and then calculates only a few combinations. It's obvious which method is more effective.

Let's go bang!

Before we get to how such problems look in practice, here are the rules of the game: Gobang is originally played on a 19x19 board. Connoisseurs of this venerable game will forgive us for trimming it down to a computer-friendly 16x16 squares for our program; this made the programming in assembler especially simple.

Two players take turns placing counters of different colors on the board,

the goal being to form an unbroken row of five of their own stones horizontally, vertically or in one of the two diagonal directions. Whoever manages it first wins. So the trick is to find moves that achieve something in several directions at once if possible, e.g. extend one of your own rows and block an opposing row at the same time. Keeping a good overview is essential—it might be a good idea to start the program **GOBANG.BAS** right now and try it out to get a feel for the game; that will make the following explanations easier to follow.

After the program starts, the board is drawn first, and then you can decide who goes first. When it's your move, a cursor appears, which is controlled with the arrow keys as usual. Simply move it to the square where you want to place a stone and press ENTER; a red stone (your color) appears and the program calculates its next move. If there is no winner after 120 moves, the game counts as a draw; once a game is over, pressing any key starts a new one. If, as a Gobang beginner, you have to swallow a few resounding defeats at first, don't let it frustrate you too much: the program can be beaten—but how, you'll have to find out for yourself!

And now to the programming: what alarms the programmer at first is the large number of possible moves in Gobang. While a chess position offers 40 moves to choose from on average, in this game there are more than 200, at least in the opening phase, and in plain language that means the computer cannot bring its real strength to bear here, namely calculating all possible sequences of moves in advance up to a certain depth.

A quick calculation immediately shows how hopeless that undertaking would be: to think just two moves ahead, the program would have to examine a total of 256x255x254x253 positions on the 16x16 board at the start, and that would be too much even for a fast machine-code program. True, methods are known from chess programming for reducing the number of positions to be examined (the alpha/beta algorithm, more on that in the next chapter), but even that would be of limited help in Gobang—who wants to wait an hour or more for the computer's next move?

So, like it or not, we have to fall back on the method of the masters and find a way to determine the value of a move at once, without trying out the possible continuations. At least we have the advantage that there are only two kinds of stones, and fortunately they don't move around the board but stay obediently where they are.

Idiotic or Brilliant?

When humans evaluate a game they think qualitatively, and in this context they speak of idiotic, bad, acceptable, good or even brilliant moves. But a computer can't do much with this verbal classification; its domain is numbers, and so the programmer has to think about how quality can be expressed as numerical values. If the program were able to calculate all the consequences of a move to the end of the game, it could look like this:

$$-1 = \text{lost} \qquad 1 = \text{won} \qquad 0 = \text{draw}$$

and a simple arithmetic comparison then makes clear which moves are better or worse.

But as already mentioned, the final outcome of the game is completely out of sight for the computer because of the multitude of possible moves, and so we have to confine ourselves to evaluating certain partial successes that experience shows to matter for the result. If we look at an empty square that is a candidate for a move—let's call it the target square—it is intuitively clear that the quality of the move has something to do with how the stones already placed are grouped around this target square. They form a pattern with certain features, and these features give us a hint as to whether it's worth putting another stone there.

For the program, therefore, the following procedure suggests itself: the computer examines the surroundings of the target square for the presence of favorable features and awards points for them according to a certain system, depending on whether it finds them or not. These points are then added up, and the best move is ultimately the one whose target square receives the highest rating. The tricky part here is not so much programming the feature recognition but rather setting up an effective evaluation function that determines how many points are given for which features. Here the programmer can only bring his own understanding of the game to bear and find out through many trials which system leads to an acceptable playing strength.

You can enjoy the pleasure of tinkering with an evaluation function yourself right away: CPC Gobang lets you change all the relevant factors as you like. But first, let's explain which features are taken into account in the first place.

To this end, let's consider any incomplete row of five on the board that contains our empty target square. If there are already opposing stones in this row, it is of no further interest to us, since it can no longer serve the ultimate goal, namely forming a row of five of our own stones: for that we categorically award 0 points. Otherwise, though, we will want to know how many of our own stones already lie in this section—the more, the better. But how many points do we give for them? The following formula has proved its worth:

$$P = \text{Count}^2$$

So four of our own stones already in place earn 16 points (and the move wins the game!), three stones 9 points, etc. That is a good start, but it is far from enough. For example, it also matters how our own stones are distributed within the five-square section under consideration. Take a look at the following situations, in which the target square is marked by X, our own stones by 0 and empty squares by a minus sign:

```
a) 0 0 - 0 X
b) 0 0 0 - X
c) - 0 0 0 X
```

All these rows of five already contain three of our own stones, but in row a they are separated, while in rows b and c they are connected. Obviously row c is the most valuable, since the move creates a four that is open at both ends and so leads to winning the game—unless the row runs into the edge or an opposing stone on the right. Rows a and b, on the other hand, can be devalued again at once by a single move of the opponent. This example makes it clear that, besides the number of stones, three additional features have to be taken into account:

1) Are the stones separated or connected? To capture this, the variable ZS (from Zusammenhang, connectedness) is defined:

ZS = 1 if connected, ZS = 0 otherwise

where, looking ahead, the target square already counts as occupied. So ZS = 1 in the following situation as well:

```
- 0 X 0 -
```

2) Is the target square near one of our own stones? The variable NH (from Nähe, nearness) is set accordingly:

NH = 1 if near, NH = 0 otherwise

3) Is the row of five under consideration open at both ends, or blocked on one side? This is where the variable OF (from offen, open) comes into play:

OF = 1 if open, OF = 0 otherwise

Altogether, the evaluation function then looks like this:

$$P = \text{Count} \cdot \text{Count} + \text{ZS} + \text{NH} + \text{OF}$$

As a test, you can work out the score for the following row of five (+ stands for an opposing stone, X is the target square). You should arrive at five points (Count=2, ZS=0, NH=1, OF=0):

```
+ 0 - - 0 X
```

One more thing has to be taken into account, though: a target square belongs to a total of five overlapping rows of five. Look at the following section of the board (+ is again an opposing stone, or a square outside the board):

```
+ + - - 0 X 0 - 0 - -
```

If we evaluate all the rows of five that contain the target square, we get the scores 0, 6, 7, 11 and 6 from left to right. The obvious choice for the resulting value is the maximum score, i.e. the 11 points for the row of five that begins with the fifth square from the left:

$$\text{Value} = \text{MAX}(\text{P1}, \text{P2}, \text{P3}, \text{P4}, \text{P5})$$

With this, the program should already be able to build connected rows with as many of its own stones as possible in a purposeful way. But unfortunately that alone isn't enough; after all, the opponent's promising actions have to be nipped in the bud as well! For this, however, we only need to

repeat the entire evaluation with the roles reversed: now our own stones count as enemies and the opponent's stones as friends. Having looked at the situation through the opponent's eyes in this way, we can tell from the score how badly he would like to put a stone on the target square on his next move. If it turns out that he would like to very much, it is of course advisable to show some wise foresight and place one of our own stones there to prevent it.

So for each target square we get two ratings, a value for the player whose move it is (Value1), and a value for the opponent (Value2). How do we now relate these results to each other? It is certainly better to complete a row of five of our own and win with it than to prevent an opposing row of five—the best defense is a good offense! So we add one or two points to Value1 as an offensive bonus OB and then take the larger result as the rating BW (from Bewertung, rating):

$$\mathrm{BW} = \mathrm{MAX}(\mathrm{Value1} + \mathrm{OB}, \mathrm{Value2})$$

But we haven't reached the end yet: at this point the attentive reader will rightly object that a very important aspect of Gobang has not been considered so far. The appeal of the game comes precisely from the fact that every move has effects in several directions at once. The evaluation function in its present form, however, analyzes only one direction at a time, e.g. a horizontal row.

So we apply the system just described to rows in all directions one after the other—horizontal, vertical and both diagonals—and thus in effect examine a star-shaped pattern with the target square at its center. In this way we get a total of four ratings BW1...BW4, and once again the question arises how these results should be combined. But to make a long story short, here is a system that has proved itself quite well: the results for the different directions are first sorted in descending order, so that:

$$\mathrm{BW1} \geq \mathrm{BW2} \geq \mathrm{BW3} \geq \mathrm{BW4}$$

Then the values are weighted by multiplying each one by a certain factor, and then added up:

$$\mathrm{Total} = \mathrm{F1} \cdot \mathrm{BW1} + \mathrm{F2} \cdot \mathrm{BW2} + \mathrm{F3} \cdot \mathrm{BW3} + \mathrm{F4} \cdot \mathrm{BW4}$$

Choosing suitable factors F1...F4 takes a certain finesse; at any rate, our CPC program already plays quite respectably with the following values:

$$F1 = 64, \quad F2 = 16, \quad F3 = 4, \quad F4 = 1$$

With this method, all the computer has to do is assign a total value to each empty square, and the square with the highest value then wins the bidding—our program moves!

Parameter Tuning

The last section clearly showed how complex an effective position analysis can get even in this relatively simple game. Still, it is astonishing how well the program plays without calculating the possible consequences of its moves in advance. Unfortunately there is as yet no theory that offers guidelines for constructing evaluation functions; the scores used in our case, too, are ultimately based on assumptions tested by experiment. So it is very likely that the playing strength can be raised further by tuning the parameters—but now it's your move!



CPC GOBANG provides a few aids for setting the evaluation parameters: when it's your move, you can switch the display of the move rating on or off with COPY. On the right of the screen, the corresponding rating then appears for whichever target square the cursor is currently on, and reveals

what the computer thinks of your plans. In addition, the program shows the rating for its own moves in the line below.

Pressing TAB, on the other hand, displays all the parameters discussed in the last section on the left side of the screen. To change one or more values on a trial basis, simply move the cursor to the corresponding line, type in the new value and finish with ENTER. You can make these adjustments at any time, even in the middle of a game; ENTER then takes you back to the action.

At the top of the parameter table you will find the values A0...A4, which determine the score for the corresponding number of one's own stones in an incomplete row of five. ZS, NH and OF specify how many points the corresponding structural features receive if they occur in a row, and OB stands for the offensive bonus. Last come the factors F1..F4, which weight the results for the four directions.

So there are 13 values in all for you to play with, which makes a number of interesting experiments possible. For example, it is very instructive to enter a zero for ZS, NH and/or OF, whereupon the computer stops paying attention to these features and plays rather chaotically. In this way you can explore what the various values mean for the program's playing behavior and try to improve it. Of course, it is particularly interesting to analyze the causes after the computer has lost a game and to make suitable changes so that the mistake is avoided in the future.

Finally, a few suggestions and some information for your own projects: not only the playing strength but also the program's ease of use could still be improved. For example, you could build in an array in which the computer remembers all the moves, so that moves can be taken back or the game replayed. Move suggestions can also be implemented with the existing subroutines. So, for all you tinkerers, here is what the three assembler routines do:

CALL &A000 clears the entire board, which is stored row by row in memory starting at address &A300. In the internal representation, a 0 means an empty square, a 1 a stone of the computer and a 2 a stone of the (human) opponent.

CALL &A00D calculates the rating for one square of the board. To do this, the column must be passed in memory location &A1D4 and the row in &A1D5 before the call (both values in the range 0...15); in addition, address &A1CF tells the routine for which player (1 or 2) the rating is to be

done. After the call, &A1D2 holds the low byte and &A1D3 the high byte of the calculated value. Furthermore, memory location &A1D0 shows whether there is a winning move for the computer (bit 1 set) or for the opponent (bit 2 set). The routine does not check whether a square is already occupied!

CALL &A022 finds the move with the highest rating; here too, memory location &A1CF must be set before the call to specify whose move it is. After the call, you will find the column of the square in question in &A1D4 and its row in &A1D5; the rating and the win flags are returned as in the previous routine.

Reference: The algorithm for the evaluation function was taken from the book *Pascal at Work and Play* by Richard S. Forsyth (Chapman and Hall, London / New York), which offers more treats of this kind.

CHAPTER 5

By Brute Computing Force

The declared goal of artificial intelligence is to simulate human thought processes. But whether this goal can ever be reached is a matter of hot debate among the learned. AI research certainly has a few partial successes to show, but to this day the tremendous flexibility and versatility of human thinking remain unmatched. Every supposedly intelligent program in existence masters only a very narrow field of knowledge, and the chess computer you can chat with about everything under the sun while you play will probably keep us waiting for quite some time yet.

But even with a program that matches or even surpasses the abilities of a human expert, there is reason to doubt whether what it does is really an intelligent achievement. As a rule, these doubts arise less in the user, who only gets to see the results, than in the programmer, who has to teach the machine byte by byte how to solve the problem. He knows that the impressive results are often produced by relatively simple procedures that are run through a few hundred times at breakneck speed in some loop or other—so what does that have to do with intelligence?

Even when a human and a machine arrive at the same result on a task, they usually get there by different methods. Here's an example right away: How would you tackle the following problem (without a computer)?

Find two two-digit whole numbers whose product is 999.

Obviously it takes a certain dose of intelligence to solve this problem without a lot of pointless arithmetic. There is in fact a simple method that gets you the answer very quickly… which, of course, we won't give away here! If you are one of the chosen few who spot the trick right away, don't be too depressed when the CPC also comes up with the result of your intellectual feat within seconds, using a pretty primitive four-liner:

```
10 FOR x% = 10 TO 99
20   FOR y% = 10 TO 99
30     IF x%*y% = 999 THEN PRINT x%; y%: END
40 NEXT y%, x%
```

True, the computer spends most of its time here on completely pointless calculations, but its high speed more than makes up for this drawback. The problem is simply solved by brute computing force, and it is not for nothing that the technical literature has dubbed this type of program *brute force*.

In any case, this example clearly shows that the achievements of humans and machines rest on completely different foundations. The human mind is comparatively slow but versatile; computers are tremendously pigheaded, but very fast and precise. An effective method of problem solving naturally has to do justice to these different hardware characteristics. This difference is reflected in the field of strategy games as well. Programs that follow the human method examine only a small selection of sensible moves, but they have to contain a great deal of strategic knowledge and, above all, apply it correctly. Their playing strength depends very heavily on how finely the program discriminates in its analysis: it has to be able to tell apart a large number of different types of positions and features in order to judge their significance in the overall context correctly.

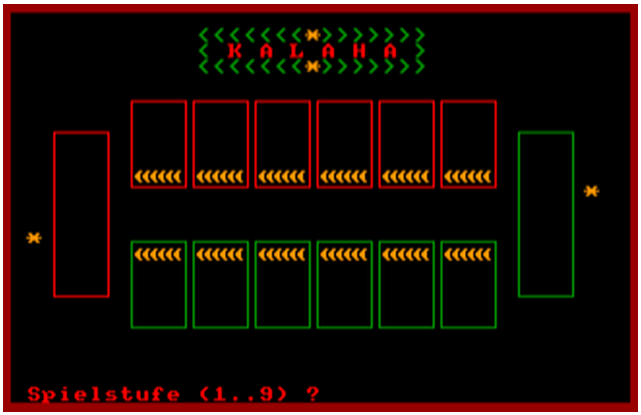
But as already hinted in the last chapter, programming a versatile evaluation function brings considerable problems with it. Anyone who has tried to improve the Gobang program by changing its parameters can surely tell you a thing or two about that: No matter how you twist and turn it, there are always situations in which the move evaluation misses the mark, that is, misjudges important facts or simply overlooks them. A more refined, and therefore more elaborate, evaluation function might help in some cases, but it confronts the programmer with very complicated tasks and, of course, also lengthens the computing time.

Despite intensive efforts, to this day only modest progress has been made in transferring a human approach to a computer with sufficient effectiveness. That is why in this chapter we will deal with a different type of program, whose power rests more on computing speed than on subtle methods of analysis—and it is truly astonishing how much ‘intelligence’ a little Z80 processor clocked at 4 MHz develops when it is allowed to really let rip! This

time our demonstration object will be a game that, for a change, comes not from Asia but from Africa.

Kalaha: The Good Ones Go into the Pot...

For a game of Kalaha you need 72 beans and 14 pots. At the start there are six beans in every pot; only the two stores at the left and right ends stay empty. Each of the two players has access to the six pots on their side of the board. You move by taking the beans out of one of your pots and sowing them one by one into the following pots, going around counterclockwise. If, for example, the computer opens with its fifth pot (top row, counted from the right), its sixth pot then holds seven beans, its store one bean, and the opponent's pots 1..4 (bottom row, counted from the left) seven beans each.



On top of that, there are two special rules.

- (1) If your last bean lands in one of your own empty pots, you capture all the beans in the opponent's pot directly opposite and put them into your store.
- (2) If your last bean drops exactly into your own store, you get to move again.

Otherwise the players take turns. The winner is whoever first gets more than 36 beans into their store. A player who can no longer move because all

their pots are empty has lost. You cannot move out of a store; the beans in it stay put.

After you start the program KALAHA.BAS, the computer first wants to know the Spielstufe (skill level) you'd like—the higher the level, the stronger the program plays, though it also needs more computing time to do so. With a second input you decide who starts the game. When it's your move, an arrow appears on the screen, which you steer with the cursor keys left and right. By pressing ENTER or RETURN you select the pot you want to move from, and the computer then sows the beans around the board. When the program has worked out its own move, an arrow likewise appears first, marking the pot in question. The move is made only after you press ENTER, so you can take your time looking at what your electronic opponent has dreamed up. When the game is over, any key starts a new one.

You'd best make your first attempts at level one or two, so you don't get demoralized right off the bat. As early as level three the program turns downright nasty, and from level 6 on it is very hard to beat. Even if you manage to keep the game even for a while, sooner or later the computer finds sly combinations of moves that give it a clear advantage—and by this point at the latest, every programming freak will be itching to know how this feat of intelligence comes about.

At least the display of the position evaluation in the upper left corner of the screen offers a small clue: During the search, the first number shows the best move found so far and the second number the corresponding expected value. A positive value means the program sees itself ahead, a negative value means it sees itself behind, and the value 0 indicates an even position. If an evaluation of +99 appears, the computer has discovered a sure way to win; at −99, on the other hand, it sees no chance left and resigns.

As you can see from the listing, the move calculation is done by a machine-code program. Of course, the DATA lines give away nothing about how the program goes about it. So we will now replace it with a BASIC routine which, while nowhere near as powerful, demonstrates the principle in a way you can follow. To do this, proceed as follows:

- Delete program lines 90 and 1140 and all lines from 5000 on.
- Add the BASIC move calculation *Zug Computer V1* (computer move) by loading it in with **MERGE "ZUGCOMV1"**.

Line 1140, which we have removed from the main program, normally enforces special rule (2). However, the following discussion becomes much easier to follow if both players move strictly in turn. The price we pay is that Kalaha loses a good deal of its fun without this rule. So if all you want to do is play, you're better off with the original version.

To follow the explanations below, it helps a lot to have a listing of the BASIC move calculation in front of you. You can display it on the screen with LIST 5000- or send it to the printer with LIST 5000-,#8.

Move Evaluation: A Bit Shortsighted

As a first try shows, BASIC version V1 plays decidedly weakly (the level you enter makes no difference here), but in return its approach is very easy to see through: All of the computer's possible moves (six at most) are made on a trial basis, and then the difference between the number of beans in the computer's store and in the opponent's is taken—the bigger the result, the better the move. Since this move evaluation assigns the value 0 to an even score, it is, unlike the Gobang evaluation function in the last chapter, a symmetric function.

Internally, the board is represented by the array in(14,9). The variables in(1,0)…in(14,0) hold the number of beans, with the opponent occupying pots 1…6 and the computer pots 8…13; pots 7 and 14 are the stores. Since we will need a few auxiliary boards, the array was given a second index, building it up in several stories, so to speak. With this information and a few additional comments, the listing should hold no more mysteries.

Line 5020:

The evaluation is initialized to −99, the worst result imaginable. It stays that way if the program finds no legal move, and in that case it signals to the main program that the time has come to resign.

Line 5040:

If the pot to be moved from is empty, the following steps are skipped.

Lines 5050-5080:

To make the move, the board is copied onto an auxiliary board and turned around in the process. Because the sides are swapped, the computer gets pot numbers 1…6, matching the FOR…NEXT loop. A loop vari-

able running from 8…13 would do the job too, but since we will need this technique for the upcoming versions anyway, it is introduced now to keep things simple.

Lines 5090-5140:
This is where the move is made. The beans are taken out of the pot, transferred to the variable anz (Anzahl, number) and distributed over the following pots by the FOR…NEXT loop.

Line 5150:
If the last bean fell into one of the player's own empty pots, special rule (1) is applied.

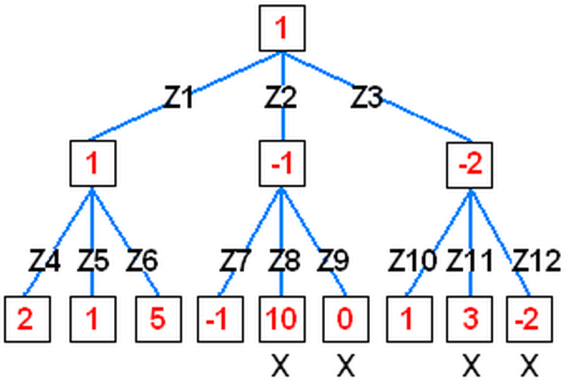
Line 5160:
Moves that lead to a win for the opponent are skipped.

Lines 5170-5180:
The move is evaluated and noted in the variable bestzug if it beats the best result so far.

Such a simple move evaluation naturally has the advantage of being very fast to compute. The trouble is that the opponent's reply is not taken into account in any way, and this makes itself felt again and again, with disastrous results. What good is a proud score of +3 if the countermove pushes the evaluation down to −10? This kind of shortsightedness, also known in the technical literature as the horizon effect, is hard to avoid with a static move evaluation. What we need, therefore, is a dynamic evaluation in which the countermoves are tried out as well.

Tree Search with Minimax

So how is such a dynamic method put into practice? Instead of the move evaluation, line 5170 has to call another subroutine that determines the best countermove. The opponent will of course try to get as few beans as possible into the computer's store while filling his own store as far as he can. If, as before, we use the difference between these two values, the opponent's best move is exactly the one for which the evaluation is at its minimum. The game tree in the following diagram illustrates the procedure.



It assumes that the computer and its opponent each have three possible moves. If the computer makes move Z1 on the left, the replies Z4…Z6 have to be checked, and as the diagram shows, the opponent can push the evaluation down to 1 at most. This minimum is at the same time the value of the computer's move; the evaluation is, as it were, handed up to the level above.

If the computer chooses move Z2, its lead could even shoot up to 10 beans in one of the variations—but the risk is too high: A seasoned opponent will of course reply with Z7 and secure himself a lead of one bean. So the final evaluation of move Z2 is −1, and despite the tempting prospect of +10 it is effectively worse than the move on the left. In the end, then, the program looks for the maximum of the minima handed up to it, which makes it clear why this method has been given the short and snappy name minimax.

No doubt a program using this method will play better than one with the shortsighted static evaluation. But keep in mind that the static evaluation has not been done away with, only put off. So when the program determines its opponent's best move in this way, our experience so far tells us that it is by no means certain this really is the best move. The danger of misjudgment has been reduced, but it is far from eliminated! But what's to stop us from computing the best reply to the opponent's move with the dynamic minimax method as well? All we have to do is add another level to the game tree, in which the computer's possible replies are tried out. For this we need another subroutine level in which, following the tried-and-true scheme, the

computer's optimal move (and the corresponding maximum) is searched for. But that's not all: This move, too, could in turn be determined dynamically…

But to make a long story short: In theory the minimax method does make it possible to examine the complete game tree right to the end of the game, but in practice this plan is thwarted by the explosive growth in the number of branches at the deeper levels. If we assume that a position in Kalaha offers five possible moves on average, a minimax analysis down to the n-th level requires the evaluation of 5^n end positions. Looking 12 moves ahead (six for the computer and six for the opponent) already produces 244 million end positions, and here even very powerful computers run up against insurmountable limits. And Kalaha's branching factor is still comparatively harmless: In chess positions, for example, there are 40 different moves on average.

So no matter how fast the machine is, at some point a limit has to be set and the position has to be assessed statically. What is clear, though, is that the deeper a program pushes its analysis into the game tree, the stronger it plays. Of course, the computing time goes up as well; or, to put it another way, the playing strength within a given time limit depends directly on the speed of the computer.

Recursion: Even in BASIC!

So how do you program a minimax tree search? At first it looks as if you would have to write a separate subroutine for each level, with only the last one performing the static evaluation by taking the difference. But the experienced programmer sees at once that these subroutines all look very much alike, and naturally wonders whether he couldn't get by with a single subroutine that cleverly calls itself.

There is indeed a technique known as recursion that solves this problem very elegantly. Although BASIC does meet the basic requirement for recursive program structures—namely, that GOSUBs can be nested—a few difficulties crop up here. First take a look at the following heavily simplified structure:

```
100 st=st+1
110 FOR zug=1 TO 6
120   'copy board'
```

```
130   'swap sides'
140   'make move'
150   IF st=stmax THEN 'static evaluation' ELSE GOSUB 100:
      ↪  'dynamic evaluation'
160   IF 'new best value' THEN 'remember move and best value'
170 NEXT zug
180 st=st-1: RETURN
```

A termination condition has been built in here as a precaution: As soon as the search depth st reaches a preset limit stmax, the program performs the final evaluation and ends the recursion. Of course, the variable st has to be increased by 1 with every call and decreased by 1 with every RETURN, so that the program always knows which level it is on. This is also where the advantage of the program section *swap sides* becomes clear: It automatically makes sure that the computer's moves and the opponent's moves always alternate, so one subroutine is enough for both sides.

We will run into trouble, though, with the loop variable zug, since it is reinitialized after every GOSUB 100 and so scrambles the count on the previous level. Here it becomes painfully obvious that BASIC, unlike Pascal or LOGO, has no local variables, which, even when they share a name, belong unambiguously to a single subroutine or subroutine level. The same goes, by the way, for the variables bestwert and bestzug: Here, too, each level needs a drawer of its own.

So what to do? The first idea, naturally, is to use an array zug(st),

```
110 FOR zug(st) = 1 TO 6
```

and while that looks very pretty, it unfortunately doesn't work, because Locomotive BASIC does not allow subscripted variables in FOR…NEXT loops: Syntax Error! Luckily, WHILE…WEND gives us another loop construct, which doesn't solve the problem quite as elegantly but does accept subscripted variables:

```
105 zug(st) = 6
110 WHILE zug(st) > 0
:
:
:
165   zug(st) = zug(st) - 1
170 WEND
```

But enough gray theory! The subroutine *Zug Computer V2* puts all the ideas explained so far into practice; you can merge it into the program in place of the old V1 version with **MERGE "ZUGCOMV2"**. Here, too, you should make a printout with LIST 5000-,#8 so that the explanations below make sense.

The maximum search depth stmax is set by the level you enter after starting the program; at first, though, you shouldn't enter a value higher than 3, or the computing time gets uncomfortably long. At level 1 the program plays just as weakly as version V1, but at the second level there is already a marked improvement: The computer no longer stumbles into every trap, because looking ahead at the countermove usually tips it off to what you're up to. Finally, at the third level it proves to be a pretty tough opponent, and you really have to work at it to still win.

The structure of the new program module largely matches version V1. A few additional notes:

Lines 5020-5050:
This preamble initializes the search depth st to 0, calls the actual recursive subroutine and handles passing the variables back to the main program.

Lines 5110-5140:
The board is copied and turned around. Naturally, each level has its own auxiliary board in(topf,st).

Lines 5150-5210:
The move is made.

Line 5230:
After the recursive call, the evaluation of the best move is taken over from the next level down. Since turning the board around also swaps the two stores, the evaluation comes back with the wrong sign, which has to be corrected.

Line 5240:
What is confusing at first here is that, contrary to the minimax principle, the program seems to look for a maximum even for the opponent's moves. But since, as mentioned, the sign of the evaluation difference in(7,st) – in(14,st) is flipped with every recursive call, the maximum on every second search level is really a minimum… as if it weren't compli-

cated enough already, some readers may grumble—but the elegant solution with just one subroutine has its price. In any case, the display of the current search depth in the upper left corner of the screen helps you make sense of the whole business: There you can watch the program clambering around in the game tree.

The Alpha-Beta Pruning Shears

Unfortunately, a big drawback of the minimax method is the steep rise in computing time with every additional search level. At level 4 it already becomes unbearably long. So we will look at one more, complementary technique that, in a surprisingly simple way, makes it possible to ignore large parts of the game tree and still get the same result as with a complete search.

To see how, let's take another look at the picture of the game tree. Once the program has calculated move Z1 through with all its consequences, it can count on a value of at least +1 and notes the result in the variable bestwert(1). Next, Z2 is examined, and looking at just the first countermove, Z7, makes it clear that the opponent can reach a result of −1 in any case. Since we hand the minimum of the evaluations up from this level, examining the remaining countermoves Z8 and Z9 might make the evaluation even worse, but it certainly cannot make it better.

So the expected value from examining Z1 cannot be beaten by Z2, and since this is already settled after evaluating the countermove Z7, moves Z8 and Z9 can be ignored along with all their further branches, which the diagram indicates with an X. Similar considerations apply to move Z3: The expected value is matched by Z10, but not improved on, so here, too, further variations no longer matter—unless you want the program to choose among equally good moves at random.

The expected value for the computer's moves is usually denoted by the Greek letter alpha; it corresponds to our bestwert(st) for odd st. A move counts as refuted when the evaluation of a countermove falls below alpha; cutting off the remaining countermoves is called an *alpha cutoff*. In the same way, the technique can be applied to the dynamic evaluation of the countermoves: The expected value for the opponent is denoted by beta, and the cutoff happens when the computer's reply goes above this value (the opponent is looking for a minimum, after all!).

Despite all these complicated considerations, it takes surprisingly little effort to build the alpha-beta algorithm into our program. Add the following line to the subroutine *Zug Computer V2*:

```
5245 IF -wert <= bestwert(st-1) THEN zug(st) = 0
```

Setting the move counter down to 0 makes the program leave the WHILE…WEND loop at the next opportunity; all remaining moves are disregarded. Here, too, of course, we have to allow for the change of sign caused by turning the board around. There is also the danger that on the first level (st=1) the program will access the undefined bestwert(0). Naturally, no moves may be cut off on the first level, and the following addition reliably prevents that:

```
5020 st = 0: bestwert(0)= -99
```

When you start the program again, the depth display lets you watch large parts of the game tree being skipped, which makes the program considerably faster. Even level 4 now offers bearable computing times. It must be mentioned, though, that the effectiveness of the alpha-beta algorithm depends heavily on the order in which the moves are examined. The sooner the best move turns up, the more the game tree can be pruned. If, for example, the search in the game tree shown in the diagram ran from right to left, we wouldn't get a single cutoff! That's why many programs first estimate the moves with a relatively shallow search and presort them before the actual analysis begins—one of the many refinements of the minimax method that could easily fill a whole book.

In this chapter, however, we have to confine ourselves to the essential basics, which play a key role in practically every strategy game program. At least the BASIC examples presented here give a clear picture of how the Kalaha machine-code program works, even though special rule 2 calls for a few extra tricks that can unfortunately only be hinted at here: When a player gets to move again, the board is of course not turned around when the next level is called, and the change of sign in the evaluation function is dropped as well. Furthermore, no alpha-beta cutoff may take place in this case… all a bit complicated, but doable! In addition, the machine-code program was given a random number generator that makes the decision between equally good moves and provides some variety.

The BASIC versions of the move calculation were written to put material in your hands that you can experiment with to your heart's content. So don't be shy about making changes and trying out ideas of your own: Only practice gives you the experience you need to write 'intelligent' programs yourself.

CHAPTER 6

The Computer Learns

As the last installment made clear, it is hard to credit a program with intelligence once you have seen through exactly how it works. The Kalaha program may play cleverly enough to outclass most people at the higher levels, but the computer's role as an electronic drudge that merely carries out a sequence of given instructions is all too obvious. After all, any pocket calculator handles numbers better than a human being without being considered intelligent for it.

What our program examples so far have lacked is the ability to achieve truly independent mental feats that lead to surprising results even for the programmer. Granted, some programs, especially while they are being developed, do give the impression of having a hefty dose of life of their own, but as a rule that is due more to the programmer's imperfection than to machine intelligence. A thorough debugging session usually puts a quick end to these first signs of a robot rebellion.

What is anathema to the ordinary programmer—namely, a program that changes the contents of memory, or even itself, on its own initiative—is, on the contrary, considered a goal worth striving for in the field of artificial intelligence. The topic is learning ability, which still poses a major challenge to AI researchers, and not just out of scientific ambition. Learning programs could, for example, help considerably to cut the enormous costs of developing and installing software by adapting themselves to the differing requirements of their users. The programmer would then no longer face the dilemma of writing either a highly specialized program that may be good for only a single application, or a versatile one that becomes very complicated and unwieldy.

Despite these attractive prospects, such systems have not come very far yet. Even though some quite convincing demonstrations of learning programs already exist (electronic rats, for example, that find their own way

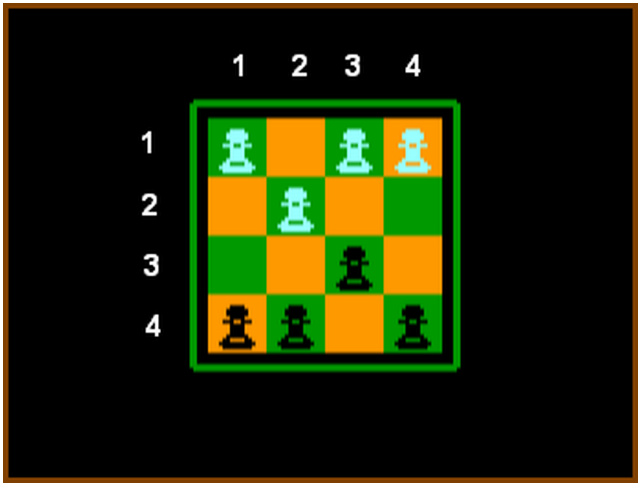
through a maze), their fields of application are mostly still too academic to gain any practical importance. Moreover, many of these experimental programs need expensive mainframes to work effectively, and there is also a certain psychological barrier that shouldn't be underestimated: Machines that develop initiative beyond the instructions they have been given do have something uncanny about them. Countless chilling visions in science fiction novels are a clear indication of the fears many people harbor.

But if you examine the various requirements a program has to meet to achieve even a rough approximation of learning ability in the human sense, disillusionment quickly sets in. The range of human perception and processing abilities is too complicated to simply be transferred to a machine, and given today's state of the art, independently thinking, power-hungry robots still belong in the realm of fiction. Much more real, by contrast, is the threat that can come from a simple program bug in a military defense system!

Mini Chess: Small but Mighty!

To investigate the possibilities and problems of machine learning, this installment uses a mini chess game that was invented especially for the purpose. The game is played on a 4x4 board on which, as the picture shows, eight pawns are set up at the start.

As in the full-size original, they move one square straight ahead; they can capture another piece only diagonally. As soon as a pawn reaches the last row, it turns into a queen, which can move any distance in any direction. However, to keep the number of possible moves, and with it the memory needs of our learning system, from growing unnecessarily, compulsory capture was introduced for the queens: They must capture another piece if they can. A player loses when he can no longer move, either because he has no pieces left or because all his pawns are blocked. The game is a draw if for four moves (two on each side) no pawn has moved and no piece has been captured.



At the start, the program asks whether a file with the results of previous games already exists on tape or disk. If so, it is loaded; otherwise the computer makes its first move right away. There's a good reason, by the way, why it always gets to go first: In an earlier test version, the program moved second, playing only the black pieces, and—as befits a learning program—it promptly sprang a big surprise. After taking a considerable number of sound beatings, it suddenly declared 'Ich habe verloren' (I have lost) right after my first move and refused to play on at any price. Not that my CPC was sulking, mind you. It had merely discovered that with optimal strategy the player who moves first always wins! So the tables were turned, and in the present version the program has to prove that it can find this winning line on its own.

Now for the rest of the controls: You enter a move by first steering the cursor to the piece you want and pressing ENTER or COPY. Then move the cursor to the destination square; a second press of the ENTER or COPY key makes the move. Illegal moves are not accepted; in that case the program just responds with a warning beep. Note that no pawn move is possible if your queen can capture an enemy piece! If, after the game is over, you answer the question 'Noch ein Spiel' (another game?) with N, the program writes everything it has learned so far to tape or disk under the name **MSCHACH.DAT** and then quits.

Slow on the Uptake?

After the first few games against the mini chess program, this question really does seem to have some justification. Faced with determined play, it cheerfully loses one game after another without making any notable progress, and the expert can already guess what's going on: The moves are chosen purely at random. If you watch closely, though, you'll notice that the program never loses the same way twice. Sooner or later there will also be situations in which the program resigns before it is actually checkmated; it sees the inevitable loss coming in advance.

These learning effects become all the more obvious the more consistently you favor certain variations (as far as the random number generator allows). It should also be pointed out that the program can apply its experience to mirror-image situations (that is, positions reflected about the vertical center line) as well. Still, there's no hiding the fact that, compared to a human, it learns maddeningly slowly. That it never repeats a lost game does give some hope, of course: Sooner or later its playing strength has to improve; the only question is after how many games.

Later in this chapter we will discuss a change to the program that takes a large part of the teaching off your hands. Before we get to that, though, we'll explain in detail how the program works in the first place, what lies behind its mental sluggishness and what is different in humans. To this end, we'll start with the internal representation of the board, which is very simple: It is an array called b(sp,re), which can hold the following values:

−1	border square off the board
0	empty
1	white pawn
2	white queen
3	black pawn
4	black queen

So instead of the 4x4 array that is really all we need, a 6x6 field with a border is used, which, as we'll see, comes in very handy when examining the possible moves. Let's now take a look at the mini chess listing, which you should print out or display on the screen so you can follow the explanations

below. The initialization section essentially just dimensions various arrays and prepares the graphics elements. Once the game starts at line 450, the move counter zz and the draw counter rz (it keeps an eye on the draw rule!) are first set to 0, and the board b(x,y) is filled with the starting setup.

GOSUB 1250 takes care of the screen output, and the assignment spieler=1 means that the computer always starts. If you want to go first, this has to read spieler=2. The files that result from this version of the game should be kept separately, though, or in the long run you'll be facing a *Memory full*. The actual course of the game is controlled in lines 600-670 by calling the subroutines *Zug Computer* (computer move) and *Zug Gegner* (opponent move) in turn. After the game is over, the variable erg holds the result; depending on this value, an evaluation follows, which we'll come to later. First, let's dig a little deeper into the program's data structures.

Packed Data: Packed Like Sardines

One thing is clear: Anyone who wants to learn needs a good memory, and as we know, humans actually have two of them, a short-term and a long-term memory. This separation makes it possible to learn selectively, that is, only the really important data is kept on hand in the long term after it has been evaluated. On a microcomputer with limited memory in particular, it is extremely advisable to copy this approach; but first we have to clarify which data is worth storing at all.

The first candidates are, for example, the various game situations that come up over the course of the games. After all, the program is supposed to remember in which positions it won or lost with which moves. But this already hints at the first memory worries: A 4x4 integer array to represent the board needs a total of 16x2=32 bytes, which comes to almost 32 KB for 1000 different positions—and mini chess racks those up in no time! As you can see, with memory-hungry problems it pays to do this kind of thinking before you write the program, so as to be safe from nasty surprises.

So what to do? One helpful technique is so-called data packing, which looks for an encoding that takes up as little memory as possible. Intuitively it's clear that an integer variable, which can represent values from –32768 to 32767, is grossly underemployed with a range of 0 to 3: It offers a total of 16 bits (including the sign), which can hold a great deal more information.

Since our board has exactly 16 squares, the obvious first step is to encode, bit by bit, which of them are occupied (bit set) and which are empty (bit cleared). Read row by row, the starting position then looks like this:

```
psn1 = 1111 0000 0000 1111
```

But that alone is not enough, of course. We also have to record which pieces they are. On the board, the different kinds of pieces are assigned the digits 1 to 4; if you subtract 1, you get a range of 0 to 3, which takes only two bits to represent:

Piece	dec.	bin.
1 (white pawn)	0	00
2 (white queen)	1	01
3 (black pawn)	2	10
4 (black queen)	3	11

This lets us simply list the pieces (skipping the empty squares) in the order in which they appear from top left to bottom right. Since there will certainly never be more than eight pieces romping around on the board, 16 bits are enough here as well; for the starting position they look like this:

```
psn2 = 00 00 00 00 10 10 10 10
```

As you can convince yourself with further examples, these two 16-bit values capture the entire game situation unambiguously. Although we now need only 4 bytes for the representation instead of 32 (that is, 2 integer variables), no information is lost! The only drawback is that packed data first has to be decoded again before it can be worked on, which of course takes a little time.

Program lines 1400-1520, within the subroutine *Zug Computer*, show how the packing is done in practice. To avoid trouble with the sign bit, the packed data is first collected in real variables (p1! and p2!). After that, the UNT function converts them into signed integer values (psn1 and psn2). As you can see from the listing, the whole procedure is then repeated, this time with the

column index running backward, which produces the encoded form of the mirrored board (psx1 and psx2).

Next, a small machine-code program goes into action. Besides the variables just discussed, it is handed the start addresses of two arrays that form an important part of the long-term memory: In l.psn1(i) and l.psn2(i) (the small l stands for long-term), the program remembers all the decisive situations from previous games in encoded form. The assembler routine practically just checks whether the current board position is already recorded there. If it is, the variable flag is set to −1 in line 1570; i then holds the number under which the position is recorded. The variable xflag is set if the mirror-image position was found.

With that, our program already meets one essential requirement for learning ability: It can remember. But knowing that a game situation has come up before is, of course, no help in choosing a move. What's missing is some indication of which moves have already proved good or bad at this point.

The Move Generator

Many programs for strategic board games contain a so-called move generator. The job of this part of the program is to draw up a list of all legal moves, from which the main program then makes its choice. Move lists can also be used to check input: If a move is not in the list, it is illegal and is rejected.

Our mini chess program contains such a rules expert, too, which we'll discuss in more detail shortly. But first, once again, the question arises of a suitable data structure for representing the move lists. If you take another look at how the chessboard is represented, you'll see right away that a move is completely described by four numbers from 1 to 4: from (column,row) to (column,row). Incidentally, throughout the program the start coordinates are held in the variables sp,re and the destination coordinates in x,y.

Anyone who has followed the discussion of data packing above carefully can easily work out that 4x2=8 bits (that is, 1 byte) are perfectly sufficient to store these four numbers; so each packed move gives us a value from 0 to 255. And now for a special trick: This value can easily be turned into an ASCII character with the CHR$ function, so we can represent the move list as a string! The subroutine starting at line 2570 shows what this looks like in practice.

Handling the move list as a string offers some solid advantages. For example, an empty string immediately tells you that a player has lost, since he can no longer make a move. Moreover, the INSTR function makes it extremely easy to find out whether a move is in the list and therefore legal. Manipulations with MID$ and similar functions are possible as well, for example to strike moves recognized as bad from a list.

The only drawback of this method is something CPC 464 owners in particular will now and then experience when memory is full and the computer performs a garbage collection to tidy up its strings, our move lists among them: To do so, it suspends everything it is doing and doesn't respond to anything in the meantime. The string management of the 664/6128 was improved in this respect, but in return it needs two extra bytes of memory per string variable.

But now, as promised, a closer look at the move generator starting at line 2060. At the beginning, a variable seite (side) is defined here, which makes it possible to build the move lists for either the white or the black pieces, depending on whose move it is. Then the board is scanned for queens, and if there are any, a subroutine is called that writes any possible captures into the move list—they're compulsory, after all! That's also why it returns right away (RETURN) if the move list is no longer empty after this procedure. Otherwise the board is checked through once more to register pawn moves and non-capturing queen moves.

The three subroutines starting at line 2220 explore the various ways the pieces can move. For working out the queen moves, two arrays dx(ri) and dy(ri), already created in the initialization section, are especially useful; for each of the eight directions of movement they contain a so-called direction offset, which is added to the coordinates of the starting square again and again until an obstacle (for example, the border filled with –1) puts an end to the queen's path. If you'd like to try mini chess with different rules, you can make the following changes here:

```
FOR ri = 0 TO 3: 'turns the queens into rooks
```

or

```
FOR ri = 4 TO 7: 'turns the queens into bishops.
```

Reduced Move Lists

Let's now return to the *Zug Computer* module. What happens here from line 1590 on, after our program has tried to remember with the help of the machine-code program? If the current position is not yet in the long-term memory, the move generator we just discussed is called to work out the corresponding move list. Otherwise a ready-made move list l.zliste$(i) is used, which is also part of the long-term memory and—as you have probably guessed by now—has been stripped of all the moves that turned out to be losers in this particular situation in previous games. So it contains only good moves, or moves about which nothing is known yet. If an empty list should turn up in one case or another, the program knows that it has lost (erg=2), and it goes back to the main program.

Otherwise, though, the game goes on with a few entries in the short-term memory, which serves to record the course of a game in every detail. The variable arrays all begin with k (for Kurzzeit, short-term), use the move counter zz as their index and have the following meanings:

k.psn1(zz), k.psn2(zz): The board position before a move is made, in packed form.

k.zliste$(zz): The move list used to choose the move.

k.index(zz): This variable is set to −1 if the current position is not yet recorded in the long-term memory. Otherwise it gives the index under which the entry was found.

k.znr(zz): Gives the position of the chosen move in the move list; it is determined at random in line 1670. If you want the program to learn systematically, you can simply change this line to k.znr(zz)=1; the program will then stubbornly always make the first move in the list.

What happens after the entry in the short-term memory is quickly explained: The move picked by the dice is taken from the move list with MID$, converted into a number and then broken down bit by bit into four coordinates, with which the subroutine *Zug ausführen* (make move) does the rest. After that, all that's left is to increment the move counter zz, and it's back to the main program: It's the opponent's turn.

And that brings us to the evaluation after the end of a game. Let's first look at what happens when the computer has won (from line 810 on). It's immediately clear that the move list belonging to the position before the winning move can be reduced to exactly that one move; the program cer-

tainly won't find a better one. The only remaining question is whether the position is already recorded in the long-term memory or not. In the first case, the move list is manipulated right there; in the second case, a new entry has to be made first, and the variable mp (memory pointer), which always points to the next free slot in the long-term memory, has to be increased by 1. So the only positions that make it from short-term into long-term memory are those for which there is a concrete reason to reduce the move list. For all other positions, the move generator can produce the complete lists at any time, after all.

By the way, the memory pointer, that is, the number of positions and move lists stored so far, is shown continuously in the upper left corner of the screen. Should it creep alarmingly close to 2000, you'll need to increase the dimensions of the long-term memory in line 150 as a precaution.

The evaluation is a bit more complicated if the computer loses a game (from line 870 on). It's immediately clear that the last move has to be eliminated from the move list, but a special case can come up here. If this move is the only one in the list, that is, if there are no alternative moves left, that automatically means that the computer's previous move must also be regarded as a losing move—after all, it leads, if the opponent so chooses, to a position with an empty move list! If the previous move likewise has no alternative, the evaluation has to be extended to the move before that, and so on. In this way, a single lost game can eliminate entire sequences of moves all the way back to the start of the game.

Finally, that leaves the question of what kind of evaluation is appropriate for a draw. What matters here is whether the program is supposed to learn to win or not to lose—in the first case, the same procedure as for a lost game has to be applied; in the second, on the other hand, the evaluation for won games has to be used. The program strikes a wise compromise: No evaluation takes place at all! If you like, though, you can put a suitable GOTO in line 940 to route the draw routine through one evaluation or the other.

The Programmed Little Brother

You have now met all the parts of the program that matter for the computer's learning behavior, and we'll now turn to the question of why the learning process is comparatively slow, even though the data structure does have

parallels with human memory. But storing positions and eliminating losing moves is obviously not enough. In particular, humans also have the ability to derive general rules from their experience, such as

It is bad when the opponent gets a queen,

and can apply those rules back to concrete game situations. It's obvious that this ability speeds up learning enormously, since the experience already gained can also be applied to situations that have never come up before.

This takes us into an area that electronic brains have trouble with. Rules like the example above are usually captured by evaluation functions (see chapter 4). In practice, then, what we would need is a program that changes the evaluation function itself in the course of learning, or at least the weighting factors of a given formula. But so far there are hardly any results worth showing in this respect, and the few experimental programs mostly suffer from the same problem as our example: They learn too slowly. True, our program does master one kind of generalization, namely applying its knowledge to mirror-image situations, but that is very little.

The following little scene shows how absurdly this lack of abstraction can make itself felt: Through days of selfless teaching you have brought the mini chess program to the point where it can cope with all your dirty tricks and tactical ploys. Even with your strongest play, you can hardly win a game against the computer anymore. Proud of your pupil's success, you are just about to quit the program and have it write everything it has learned to tape or disk when in bursts your little brother. Just for fun, you explain the rules to him and invite him to a game. And lo and behold: Although the little guy often makes the dumbest moves imaginable, the oh-so-studious program suddenly starts losing one game after another again—what happened?

It's quite simple: You taught the program how to win against a good player, but not how to win against a bad one! Since the program doesn't generalize, the experience from its previous games is no use to it whatsoever when it runs into completely unfamiliar positions. So you urge the little squirt to play some more games, to make the program weatherproof in this respect as well, but unfortunately after 10 minutes he's had enough and takes off. Now what?

But before you're tempted to play the dummy yourself, here's a better solution: Why shouldn't the computer take over this role? That's no problem

at all. All you have to do is load the *kleiner Bruder* (little brother) extension with **MERGE "BRUDER"**, and from then on the computer learns by playing against itself, one game after another! When you press DEL, the carnage stops at the next opportunity, and the program writes its long-term memory to tape or disk as usual.

At first glance, an old AI dream (or nightmare?) really seems to be coming true here: Without any human help, the machine is acquiring a qualification that will one day enable it to prove itself in the real world (that is, against human opponents). But if you look at what's going on on the screen, it's enough to make your hair stand on end: Since both opponents choose their moves at random, hardly any sensible sequences of moves come about. That stays the same for whoever is playing the black pieces, while the one playing white keeps learning; so after enough games, white ought to win the majority of them. Go ahead and give your CPC two or three nights, and see what comes out of this long-term experiment.

The author's program now works with a file that takes up about 30 KB on disk and holds 1950 stored positions. With it, the program does indeed win most of its games against the little brother, and even experienced opponents sometimes get soundly trounced. The file was built up through a number of hard lessons and nights of nonstop computing; combining the two methods has proved quite sensible.

All the same, it shouldn't be concealed that the program in its present form is hardly satisfying in terms of performance. It meets only some of the expectations placed in the project when it was planned. Even so, there were good reasons to present it in the Software Experiment. On the one hand, it demonstrates interesting programming techniques (packed data and the move generator, for example) that can come in handy on other occasions, too; on the other hand, it's quite revealing to watch the program in its efforts to learn. And maybe one reader or another has an idea for improving the program?!

CHAPTER 7

The Computer as Scientist

To get us in the mood, let's start with a quotation from the Frankfurter Rundschau of November 3, 1986:

"While the Vatican's relationship with the natural sciences seems to have been settled—after all, Galileo Galilei has been rehabilitated by John Paul II, 350 years after his trial—its relationship with the human sciences is in many respects still a Galilean situation. Witness the Vatican's latest statement on homosexuality in men and women. 'It is only in the marital relationship that the use of the sexual faculty can be morally good. A person engaging in homosexual behavior therefore acts immorally', read the two key sentences of this document."

The development of the modern scientific way of thinking began in the 17th century, and Galileo Galilei is generally regarded as its intellectual father. As everyone knows, his claim that the sun, and not the earth, was the center of our planetary system got him into a fair amount of trouble at the time. At any rate, the champions of the prevailing doctrine had enough means of pressure at their disposal to make him recant. Galileo's famous words on his deathbed, '…and yet it moves', may well have been invented after the fact, but they stand as a symbol of the fact that the insights, revolutionary in his day, could no longer be stopped in spite of it all.

Legend has it that Galileo invited his adversaries to look through the telescope he had developed, so that they could see the laws of planetary motion for themselves. But the Pharisees and scribes declined: 'Why should we look through your tube, when we know perfectly well that what you say cannot be so…' As children of the space age, we look back on this mixture of stupidity and arrogance with bafflement today and wonder how such a thing could ever have happened—but wait! Are you, dear reader, prepared to take a self-critical stance for a moment? Then think about the following questions: How do you actually know that the earth revolves around the sun?

Did you learn it at school, read it in a book, or hear it on television? Be honest now: couldn't the opposite view have been drummed into you in just the same way, turning you into a convinced advocate of a geocentric worldview without your ever having questioned it?

In this context, you have to bear in mind that in Galileo's day it was common practice to derive all scientific knowledge from the Bible or from the writings of classical antiquity. Galileo's faux pas lay not so much in contradicting the prevailing doctrines; what really mattered was that he rejected the scientific methodology of his time root and branch. As a counterweight to deduction, the application of given laws to individual cases (natural phenomena), he introduced the scientific experiment as a question put to nature: are the claims contained in the theories actually true at all? The theologians of his age saw this as an attempt to subordinate theology to the natural sciences and set the Inquisition on Galileo, since, understandably enough, they had no wish to give up their dominant position.

The scientific method in Galileo's sense thus involves a decidedly critical way of thinking that calls for great intellectual agility. For it, there are no God-given laws standing around like sacred cows, mooing away to themselves in the marble temple of science. Every claim (whether from the Bible or from television) is merely an occasion to ask further questions, which may of course end up throwing the original approach out the window altogether. To sum up the process once more: from the observation of individual natural phenomena, certain laws are inferred (induction). From these laws, in turn, one can derive how nature will behave in other cases (deduction). Experiments establish whether these predictions agree with reality. If they don't, the theory has to be modified accordingly and used as the starting point for further experiments. Scientific knowledge arises from this constant interplay of induction and deduction; theory and practice work hand in hand as equal partners.

Insight Instead of Programming

Galileo gained his insights through a constant confrontation with reality; he refused to twist reality by force into an interpretation that would make it compatible with preprogrammed knowledge. His method amounts, in practice, to a systematized form of learning, and this is where the AI programmer

pricks up his ears: Method? System? Wherever these words crop up, an algorithm is often not far away—a procedure that can be broken down into individual steps and can therefore also be carried out by a machine.

And that brings us back to the question that already occupied us in the last chapter: How can a computer be guided toward learning on its own? We did have some partial successes to show with the mini-chess program, after all, but on the whole they were not particularly satisfying. The learning was achieved practically only through well-organized data storage, and at a snail's pace at that, since our program could not derive any generally valid rules from its experience.

Before we turn to the problem of giving a computer at least a glimmer of genuine insight, though, let's take a closer look at the concept of 'learning'. In a certain sense, even programming a computer, that is, storing a sequence of instructions, can be regarded as a learning process. But this kind of learning, which requires practically nothing but a good memory, is not particularly interesting for artificial intelligence. What is called for instead are systems that change their behavior of their own accord on the basis of their experience, and that requires not only enough memory but also highly developed powers of comprehension, so that the consequences of different decisions can be perceived and processed.

The difference shows in human beings, too. It certainly takes more intelligence to learn a language than to learn vocabulary: in the first case, complex relationships of meaning have to be grasped; in the second, a good memory will do. It is always fascinating to watch the natural intelligence with which a small child acquires language; frightening, on the other hand, is the realization that in many respects people can be programmed just like computers.

Advertising is a good example: through endless repetition and subliminal influence, it is meant to steer our behavior as consumers. That this kind of programming evidently works well can be seen from the mere fact that, year after year, industry undauntedly invests billions to convince, for example, good little housewives that without RENOL they ought to have a guilty conscience toward their families.

Let's recall once more the awkward questions at the beginning of this chapter. You have certainly learned a great deal in your life. But tell me: is what you have learned based on insights gained through experience and well-aimed questions, or on programming delivered by your upbringing or the media? Whatever the outcome of your investigation, this comparison

gives you an idea of what matters in an 'artificially intelligent' ability to learn.

Model Wanted

Reality, with all its aspects, is still far too complex to serve as the object space for programming artificial intelligence. All attempts therefore deal with a precisely defined (and usually very small) slice of reality, or with a greatly simplified model of real processes. As the last chapters have already made clear, games are superbly suited to this purpose. On the one hand, they are limited to a manageable set of objects (the pieces) and operations (the possible moves); on the other hand, they contain enough uncertainty factors (the opponent's moves) to which the program has to react with a certain degree of independence.

The application of scientific methods can also be demonstrated very nicely with a game that came onto the market a few years ago under the names 'Superhirn' (super brain) and 'Mastermind' and became very popular. Basically, the idea is to crack a secret code consisting of a hidden row of colored pegs. For every code you try, your opponent gives you a score that tells you the number of correct colors and correct positions. Through logical reasoning and well-designed experiments, you are supposed to get to the bottom of the code as quickly as possible.

In this installment we will take a look at a particularly appealing variant of this game idea, in which the same procedure is used to find a secret word—the ideal game for hackers, in other words! Of course, only words in common usage are allowed as codes or guesses, and before the game begins you have to agree on a particular word length. Here's an example right away: the secret word is HAUS (house); the experiment DACH (roof) then scores two correct letters (A and H) and one correct position (the A is in the right place). From now on we will simply note down the results like this,

```
DACH # +
```

with two marks for two correct letters, the # sign standing for the correct position. Note that a letter that occurs twice only scores two hits if it also occurs twice in the secret word. With the secret code HEXE (witch), for

example, the guesses HUHN (chicken) and ESEL (donkey) can produce the following results:

```
HUHN #    ESEL + +
```

To close in on the secret word, there is no getting around evaluating the scores, and logical reasoning usually lets you rule out a number of cases. Sometimes this is very easy: if a word doesn't get a single hit, its letters do not occur in the secret code. Things get a little trickier, though, when the first two attempts, EHRE (honor) and WORT (word), produce the following result:

```
EHRE #    WORT +
```

This proves beyond doubt that the secret word contains no R—can you figure out why? Conclusions drawn from looking at three or more attempts together get a great deal more complicated still, and often you even have to rely on guesswork. Take a look at the following results for BERG (mountain) and WERT (value):

```
BERG # #    WERT # #
```

You can't help suspecting that the combination ER in the middle is responsible for the two correct positions; how could it be otherwise? So with our third attempt, HERZ (heart), we hammer away at the same spot,

```
HERZ #
```

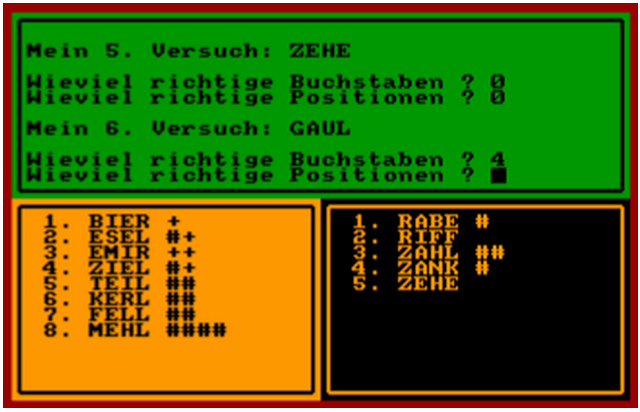
and get only one correct position! This gives you a faint idea of how a scientist feels when the result of an experiment sends a theory he thought was rock-solid down the drain. By the way, the secret word in this case was BART (beard)…

To get a complete picture of what this game is like, it's best to find a partner and play through a few rounds. It quickly becomes clear that blind trial and error gets you absolutely nowhere. A good method, however, is to set up a theory about the makeup of the secret word on the basis of the previous results, sometimes deliberately accepting that it might be wrong. The next attempt is then, as in our last example,

chosen to match this working hypothesis; the score tells you whether the assumptions were justified. With this scientific strategy you have a clear line in view that leads to the goal with relative certainty. All we need now is an intelligent program that can crack the secret code as well.

The Big Bluff: Wordmaster

First, though, the question arises whether we haven't bitten off a bit more than we can chew. Obviously the program has to be able to cope with highly complicated logical conclusions, set up assumptions and discard them again… don't we need at least a mainframe and specialized programming languages like LISP or PROLOG for this? Far from it—WORDMASTER impressively demonstrates that even a microcomputer can solve this problem in pure BASIC!



After you start the program with **RUN "WMASTER"**, the main menu awaits you, offering among other things the option *Spielen* (Play). Anyone who jumps right in, however, is in for a disappointment: WORDMASTER first needs a word list; without a certain basic vocabulary, the program can't do a thing. On the program disk or tape you'll find two ready-made word lists **(LISTE4.DAT, LISTE5.DAT)**, which you can load with menu option 3. One contains four-letter words and the other five-letter words. Only nouns in common German usage without umlauts or ß were included, and no proper

names or geographical names either. These restrictions have proved sensible in practice; as you will see, the game is difficult enough even so!

Menu option 2 lets you enter, change or display a word list; the screen gives you all the necessary instructions. Note that when you enter a new list, the first word determines the length of all further words; entries of a different length are rejected from then on. WORDMASTER automatically sorts new words into alphabetical order, so with long lists there is a slight delay after each entry. A list can hold a maximum of 1000 words. If you want to exceed this limit, you have to increase the dimension of the array w$() in lines 100 and 980. Words should not be longer than six letters, since otherwise the display in the main program runs into problems.

Back in the main menu, WORDMASTER is hopefully well fed and content, and the fun can begin. The whole game is played in dialogue with the program, so lengthy instructions are unnecessary. If you decide to start a game, you first have to guess a word that WORDMASTER has thought up. Besides entering your guesses, you can browse through the word list at any time with the cursor keys up and down if you run out of ideas. In an emergency you can even have the secret word revealed by entering a question mark, but then you are automatically charged 10 attempts, and in all likelihood you have lost the round.

If you enter a word that WORDMASTER doesn't know yet, the program wants to know whether the word should be added to the list; only then is it accepted as a guess. Beware of using nonsensical combinations of letters, because WORDMASTER adapts to your choice of words and may well use your entry as a secret code at the next opportunity! If the newly acquired knowledge is to be kept, the expanded word list must of course be saved again with option 4 of the main menu after the game. In this way, the program enlarges its knowledge base with every round it plays.

Your guesses and scores are logged in a separate window on the screen. Whatever you do, try to get by with as few attempts as possible! Once you have found the word, it's WORDMASTER's turn, and you get to think up a secret word. As you will see, the program usually finds the code very quickly, and if it needs fewer attempts than you did, you have lost the round. On the other hand, it can of course happen that WORDMASTER doesn't know the word yet, and then, as a matter of poetic justice, the program gets saddled with 10 attempts. But even then it learns something new!

The results you give WORDMASTER consist of two numbers (correct letters/correct positions). If you press DEL after the first number, you can correct it; DEL before the first number starts the guessing process over again. Aborting like this only makes sense if you discover a typing mistake in a previous result, which of course keeps WORDMASTER from cracking the code. Typing mistakes that go unnoticed are generally found and pointed out by the program once the (unsuccessful) search has ended.

So WORDMASTER always keeps track of things, and when you watch how single-mindedly it homes in on the secret word, the program does indeed seem to master the scientific strategy described above with ease. But honesty is the best policy: appearances notwithstanding, it can neither draw logical conclusions nor set up and test theories—the only thing WORDMASTER really achieves is a convincing simulation of these abilities!

Anyone who remembers Chapter 5 (By Brute Computing Force) and takes a critical look at the thinking process displayed on the screen will soon realize that nothing more than a fast search algorithm is at work here… but hold it! So that the game keeps its appeal for a while longer, this time, for once, we'll leave it to you to work out the exact procedure. Program lines 2880–3090 should offer a few clues for a thorough analysis; we can't go into them here, though, since we still need room for another hot topic:

The General Problem Solver

Before this heading confirms your suspicion that the author of these lines has finally succumbed to galloping megalomania, here is a brief look back at the history of artificial intelligence that explains everything: at the dawn of the computer era, optimism was unbroken despite hardware that was pitiful by today's standards, and in the fifties people worked in all seriousness on programs that were supposed to solve not just specific tasks but any problem whatsoever. And believe it or not, these efforts actually succeeded! 1957 saw the birth of the *General Problem Solver* (GPS) by Newell, Shaw and Simon, famous among experts. Naturally, this begs the question of why on earth the program was never put to work to finally solve the great problems of humanity—but we'll come to that in a moment. First, let's outline the approach this all-round software genius works with.

In principle, every problem arises from a difference between a given ac-

tual state and a desired target state, and can therefore be reduced to the question 'How do I get from A to B?' You are at home but would love to go to Honolulu, or you are single and would like to get married, or you are broke but need at least two grand for the PC of your dreams… just a moment, we'll have that sorted out in no time: the world the problem refers to generally contains useful objects (means of transportation, magazines, savings banks) and matching operators (call a taxi, place a marriage ad, rob a bank) that can be used to transform state A, that is, to turn it into another state.

Of course, you can't expect a single operator to produce the desired state B right away; often whole chains of operators have to be applied in the right order. But don't worry, the GPS will figure those out all right. We just have to explain to the program beforehand how which operators act with which objects on which states…

You see—and that's exactly why the GPS never caught on. Encoding and preparing all the relevant data is so time-consuming that the program's contribution to solving the problem looks very modest by comparison; in other words, problems can usually be solved faster without a computer. The GPS would only make sense if it could gather the data on its own, but in this respect the experts today are very pessimistic. All in all, the project must therefore be called a failure, although it gave important impetus to AI research as a whole. Besides, the idea is simply too interesting to be swept under the rug here. A demonstration of how the GPS works on a CPC would be aiming a bit too high—that would finally be the moment to switch to PROLOG—but what about a special-purpose problem solver that puts at least some of the ideas into practice, if only in rudimentary form? For this purpose, the WORDMASTER word lists turn up just when we need them:

Take Away Gerd's Bier, and a Beast Will Appear!

You have probably come across the following type of puzzle before, in the puzzle corner of some magazine:

```
GERD .... .... .... WOLF
```

Every night when the moon is full, Gerd (a German first name) inevitably turns into a wolf—how does that happen? Each time, one letter is exchanged so that a new valid word (from the list) is formed, and with that the operators

for our problem are already defined. The difference between the actual state and the target state can be taken to be the number of letters that differ. Since the difference is four letters at the start and the puzzle is to be solved in four steps, it is clear that each transformation must reduce the difference by 1; that's also why this puzzle is not particularly difficult.

A much tougher problem, however, comes with another transformation that occurs quite frequently in real life, namely turning GELD (money) into BIER (beer). A little trial and error will convince you that the difference cannot be reduced right away, since none of the letters of BIER fits into GELD, that is, leads to an existing word. We will certainly need more than four steps and will have to accept some detours. So how many steps will be needed at the very least? This question is not easy to answer. So far, one solution is known that requires six intermediate stations. It begins by turning GELD into HELD (hero)—see if you can find the rest! The author was greatly helped in this by a small program that searches one of the WORDMASTER lists for four-letter words that differ from the word entered in exactly one letter.

```
10 OPENOUT "X": MEMORY HIMEM-1: CLOSEOUT
20 OPENIN "LISTE4.DAT"
30 a=0: DIM w$(1000)
40 WHILE NOT EOF: a=a+1: INPUT#9,w$(a): WEND
50 CLOSEIN
60 INPUT "Wort: ",a$: a$=UPPER$(a$)
70 FOR i=1 TO a
80    dif=0
90    FOR j=1 TO 4
100     IF MID$(a$,j,1) <> MID$(w$(i),j,1) THEN dif=dif+1
110   NEXT j
120   IF dif=1 THEN PRINT w$(i);"
130 NEXT i
140 PRINT: PRINT: GOTO 60
```

Working with this program is very revealing, since it tells you a lot about the structure of the word list. There are islands, that is, single words (e.g. AUTO, car) and small groups of words that allow no further transformation, and there are also dead ends, words that have only a single transformation partner and do not allow chaining in several directions. Seen from

this angle, the list looks like a kind of word maze through which the beer-thirsty researcher painstakingly gropes his way. A new and interesting game idea could certainly be developed from this, but let's stick with our problem solver: what we are looking for is a program that a) solves puzzles of this kind, and b) constructs puzzles of this kind!

If we stick to simple puzzles with four steps (see Gerd the werewolf), such a program seems, if not exactly trivial, at least feasible, since reducing the letter difference provides a clear condition for every step. Things get hairy, though, with our GELD-BIER problem: what is supposed to give the search a direction here? How can a program recognize islands and dead ends, and avoid endless loops?

These open questions were passed on to the readers in issue 1/87 of PC International, in the hope that one or another skilled data tinkerer would find at least a partial answer. And the response did indeed exceed all expectations: read in the next chapter what happened…

CHAPTER 8

Word-Chain Problem Solved!

In the January 1987 issue of PC International, readers were set the task of developing a program that connects any two words by a word chain, exchanging only one letter at each step. As a particularly hard nut to crack, we had presented the problem of turning GELD into BIER in this way, and we had also raised the question of whether it is possible at all to solve puzzles of this kind in general on a microcomputer like the CPC. Our readers, however, did not keep us waiting long for an answer: shortly after issue 1/87 came out, the first programs on the subject arrived at the editorial office, and after a few tests and experiments with the material, only one conclusion remained: the problem has indeed been solved!



One really should not underestimate the creative potential of the microcomputer scene: here, untroubled by economic constraints or scientific objectives, people playfully solve problems that sometimes draw nothing more from the professionals than a weary "For that I'd need a bigger computer". Mr. Raab of Nimtofte, Denmark, who also tackled the word-chain problem, wrote: "Daily work with the computer leaves hardly any time for things like this, and most computer systems nowadays are so complex that you can't afford to experiment. The computer at home is much better for that. It is a bit slow, but I can do whatever I like with it, as long as the machine hasn't been commandeered by my sons."

And anyone who carelessly challenges the ambition of the microcomputer freaks had better know what he is doing: Commodore, of all companies, which contributed a good deal to this development with its C 64, was on the receiving end some time ago, when its wonder machine, the Amiga, was presented to the public. Who doesn't know the famous "Bouncing Ball", which set new standards in animated graphics as a screen-filling macro sprite? Naturally, it didn't take long before pesky rival Atari came up with a faithful copy of the checkered super ball on its ST. All right, Commodore hastened to announce, but please note that this already kept the Atari 90% busy, whereas

the Amiga, thanks to its custom chips, could easily play a Bach prelude and a game of chess at the same time.

The argument had something to it. But when, once again only a short time later, the bouncing ball turned up on all kinds of micros, right down to the Sinclair Spectrum, Commodore preferred to meet further developments with wise restraint. Incidentally, there is also an astonishing "Bouncing Ball" program for the CPC, in which a whole legion of balls bounces around the screen in front of a moving background (!). This graphics demo, published in PC International 7/88, simply has to be seen to be believed!

None of this takes anything away from the Amiga's power as a graphics machine, but it does show that the 8-bit dwarfs still haven't been pushed to their limits. It is precisely the hardware limitations that seem to spur programmers on to ever new feats, and so we really should have seen it coming: the doubts hinted at in Chapter 7 as to whether the word-chain problem could be solved on a CPC at all were a real challenge for the readers of PC International.

What's the Problem?

First of all, let's reveal the solution to the GELD/BIER puzzle (money to beer, by way of hero, stove, gentleman, army, tar and animal):

```
GELD - HELD - HERD - HERR - HEER - TEER - TIER - BIER
```

That's how it's done—and if you allow the word HIER (here), you can even manage with one step less. To solve a puzzle like this, the computer, just like a human being, needs a certain vocabulary. Mr. Mühlenweg of Krefeld describes how the size and composition of the word list affect things: "Whether a transformation is easy (e.g. GERD-WOLF) or difficult depends only on the computer's vocabulary. If, for example, the word GOLD is missing, the easy problem becomes a difficult or even unsolvable one".

A very large vocabulary, however, is no guarantee of outstanding skill at finding word chains. After all, the computer has to filter out suitable words. The GELD/BIER problem in particular puts up quite a bit of resistance here. By what system, for example, can one tell that turning GELD into HELD is a suitable first step? Mr. Raab describes the basic difficulties very vividly: "To solve the problem, you need a measure of how far you are at any given

moment from the goal you have set, that is, the word BIER. The number of letters that differ between the target word and a comparison word is a natural measure of this distance. If you picture the target word as a mountain peak, you can think of the difference as the distance as the crow flies. Unfortunately, we have no measure for the route we have to cover if we cannot fly but have to keep to a path, that is, to neighboring words".

The computer as a pathfinder in rough terrain—clearly, there is hardly any way around an effective search algorithm that finds the path by trial and error. The first program to seek a solution in this way arrived as early as the beginning of January and comes from Mr. Klaas Wedemeyer of Hamburg. It uses a subroutine that calls itself (a recursive one): for the start word it looks up a neighboring word (one letter different) from the list, then a neighboring word for that word, and so on. If the program runs into a dead end, it automatically goes back to the previous step and tries a different continuation. Endless loops are prevented by checking every candidate to see whether it already occurs in the word chain.

AI expert Martin Schlöter of PASCAL International immediately had the right technical term at hand when I showed him the program. "This method, in which the computer retraces its own steps to find another branch, is called backtracking, and it's built right into the PROLOG language", he explained, and then muttered something that sounded like "you don't do that kind of thing in BASIC anyway". Before he withdrew once more into the loftier realms of the programmer's art, however, I managed to cadge a little LOGO program from him that illustrates recursion very graphically:

```
to Baum :Länge :Winkel :Tiefe
if :Tiefe 0 then [stop]
fd :Länge
rt :Winkel / 2
Baum :Länge * 0.9 :Winkel :Tiefe - 1
lt :Winkel
Baum :Länge * 0.9 :Winkel :Tiefe - 1
rt :Winkel / 2
bk :Länge
end
```

A Turtle with Reverse Gear

So the LOGO program draws a tree (Baum), exploiting the fact that every branch, together with its forks, can itself be regarded as a tree; all the routine has to do is call itself. If you have a LOGO interpreter, start it up, type in the listing and then enter, for example:

```
Baum 50 60 4
```

The first parameter (Länge, length) determines the length of the branches, the second (Winkel, angle) the opening angle at a fork, and the third (Tiefe, depth) the recursion depth. If you watch the turtle's path on the screen, you can easily picture how the search through the word list proceeds: whenever the graphically gifted reptile has reached an end point in the tree, it climbs back down to carry on drawing at another spot where a branch is still missing.

At first glance, the principle looks promising; the first experiments with a large word list, however, were disappointing: since all conceivable sequences are examined without restriction, the computer quite quickly loses its way in endless word chains. True, the method guarantees that all existing solutions are found (no matter how long), but it takes a great deal of time to do so. Besides, a solution in 20 steps is hardly of interest when there is also one in 6 steps. So what to do? The question was soon answered, though: only a short time later, Mr. Mühlenweg's listing arrived, together with a few further ideas.

In his program, the user is asked at the start for the maximum number of steps the chain may contain; this makes a longer solution disappear beyond the search horizon, but in return the computing time becomes controllable. The author also sees another way of cutting the computing time: "If there is no longer any prospect that all the differing letters can still be exchanged in the remaining levels of the search tree (down to the given search horizon), the word in question is out of the running. This surely reduces the search effort without changing the result".

If, for example, a word still has three wrong letters two steps before the end of the chain, it isn't even considered. This goal-directed search does indeed avoid a lot of wrong turns. The only annoying thing is that the program keeps falling for detours such as

```
HASE - NASE - VASE
```

From HASE (hare) to VASE (vase), a single step would have done, as you can see at once. But there is a remedy for this, too: you only have to check whether a word differs in just one letter from the word two places before it in the chain. If it does, there is obviously a detour, and the word is rejected. This check doesn't catch all detours (after all, they can stretch over more than three steps), but it effectively eliminates the most common case. In principle, this test could be extended to further words in the chain; the only question is whether in the long run that wouldn't use up more time than it saves.

So far, so good—if you combine these ideas, you already get a pretty usable program. The BASIC program **WKETTE1.BAS** shows how the various elements are implemented. It works from a list of four-letter words (LISTE4.DAT), which WORDMASTER already used in the last chapter and which the word-chain program loads into the array wort$ in lines 190–250, before the user enters the puzzle.



After you start it with **RUN "WKETTE1"**, the program first expects three entries from you, each confirmed with RETURN: the four-letter start word, the end word, and the maximum number of steps allowed for forming the word chain.

The main work is done by the recursive subroutine starting at line 450. The current search level is kept in the variable schritt (step); it is increased

by 1 with every call. In addition, there is a pointer for each level that marks the word currently being examined in the list (that is, in the array wort$). Before a candidate is accepted as the next link in the chain, it has to undergo a few strict tests. If it fails, the program jumps straight to line 890, where the pointer is moved on to the next word. When the end of the word list is reached, the variable schritt is decreased by 1, and the program returns to the previous level.

If the candidate meets all the criteria, however, it is entered into the array kette$ (chain) in line 840. If at this point the difference from the target word (zdif) is only 1, a solution has been found, which is displayed with GOSUB 960; otherwise the routine calls itself with GOSUB 450 to examine the candidates for the next step. On the screen you can watch very nicely how the program builds up the word chains and, if need be, takes them apart again when they don't lead to the desired goal.

The Last Word in Wisdom?

No, that our program most certainly is not. To keep it easy to read, we deliberately refrained from using every trick in the book to squeeze out maximum speed. Ambitious tinkerers can surely get quite a bit more out of it. But programming questions aside: is this algorithm really the only way to solve the word-chain problem? Further letters from readers prove that there are indeed alternatives. They are based on the idea of structuring the word list in memory from the very start in such a way that a particularly efficient search becomes possible.

Mr. Maleschka of Hagen, for example, sent us a Pascal program built on the following idea: "The data are stored in memory in so-called islands. What is special about the words of an island is that from every word of an island you are sure to find a way to every other word of that island." As soon as a new word is entered, the program checks whether it has a neighbor on one of the existing islands. If so, the word is filed there. If it has neighbors on several islands, the word forms a bridge, as it were, between the groups of words, and they are merged. The advantage is obvious: the program can determine very quickly whether there is a solution at all (start and target words must, after all, be on the same island!) and can completely ignore the overseas territories during the search.

BASIC, however, has a hard time when it comes to structuring data. Apart from its inflexible arrays, it offers no tools for the job, unlike Pascal, where records, pointers and linked lists make complex data structures no problem at all. But you can also go the other way and descend into the depths of assembly language programming, where data can be managed very flexibly, too—not as comfortably, and without seat belts, but very fast. Mr. Gerald Steffens of Harsewinkel took this route and sent in a program whose performance is truly impressive. It arranges the word list in memory in the form of a graph: for each word there is a group of pointers giving the memory addresses of the neighboring words. If you write all the words on a large sheet of paper and connect neighboring words with lines, you have a picture of the structure right before your eyes.

The computer only has to build the graph once at the start (unless new words are added), and after that, things really take off: since the program no longer has to search laboriously for the neighboring words but finds a pointer to them right there in memory, it can swing its way through the graph at lightning speed. A sound theoretical background helps with the programming, too: "The pure mathematical idea that remains when you analyze the problem turns up again in graph theory and combinatorics", writes Mr. Steffens, citing in this connection Dijkstra's algorithm, which he also used as the basis of his program.

All at One Blow!

Unfortunately, there isn't room to discuss this method in detail, so just this much: it is basically a maze algorithm that finds the shortest path in the graph with the unerring certainty of a sleepwalker, and not just to a given target word but to all reachable points at once. It is quite impressive to see how, once the start word has been entered, the program spits out a list of every possible target word, together with the minimum number of steps in which each can be reached. Enter one of these words, and the program promptly delivers the corresponding chain. Of course, this also gives you a large number of potential puzzles to choose from right away—the ideal program for a puzzle magazine, in other words!

The program consists of two parts. First, the loader for the machine code is started with **RUN "WKETTE2"**; it then automatically loads the main pro-

gram KETTEV2. Here are a few notes on the items in the main menu:

Laden (Load): A word list can be loaded in either binary or ASCII format. **LISTE4.DAT** on your disk/tape is an ASCII file.

Speichern (Save): The word list is saved in binary form after you specify a file number.

Ausgabe (Output): The list is displayed on the screen.

Wort einfügen (Insert word): A new word can be entered.

Korrektur (Correction): A wrong word is replaced by the correct one.

Einstieg ins Suchprogramm (Enter search program): Used to enter the start and target words.

You can get back to the main menu at any time by pressing ENTER without typing anything first.

So now we have a nearly perfect word-chain program, and all that remains is to thank all the readers who—named or unnamed—examined the problem from every angle in their contributions, analyzed it and finally solved it. It was only through their active help that this chapter of the Software Experiment became possible.

CHAPTER 9

Development Aid by Computer

by H. Brendjes / J-D. Kaufmann

Should computers be used in school lessons? On this question educators are divided. In a world where children are already at the mercy of ubiquitous electronic media, they should not have to sit in front of a screen at school as well, the critics warn. Children and teenagers will have to come to grips with computer technology sooner or later anyway, the advocates counter, so why shouldn't that happen in a sensible way in the classroom? But whichever way you turn it: in the end the question will be settled not by theoretical considerations but by practical experience in everyday school life.

But this development is still in its infancy; suitable software is hardly available so far. Teachers who like to experiment therefore usually have no choice but to write the programs themselves. To provide material and to pass on experience, we are publishing the following report, which describes working with a computer simulation in geography lessons.

1. Development Aid as a Topic in Geography Lessons

The classroom topic of development aid is itself in need of some aid to its development. Teachers and students alike are increasingly overwhelmed when it comes to reaching the goals set for this field of learning. Both find themselves caught in the contradiction between the theoretical treatment of development efforts and reality, namely the frequent portrayal of impoverishment in the Third World by the media.

The difficulty with the topic of the *Third World* lies in its complexity. It is reflected in the mere inability to define central concepts such as under-

development or development precisely and universally. One-sided attempts to explain underdevelopment, and equally one-sidedly justified strategies of development aid, produce half-truths at best. Underdevelopment and development, however, are invariably highly ramified problems that call for a multi-layered approach and that, of necessity, cannot be adequately illustrated by media such as text and graphics.

Anyone who wants to make development problems understandable in the classroom faces a twofold difficulty: students do not get to know developing countries and their problems through their own observation and experience, but only indirectly and in fragments, through the media. What hunger means in a Sahel country is something a student cannot feel even if he voluntarily goes without any food for a few days. As a rule, help for these problems boils down, for the student, to food aid and the classic drilling of wells.

Judging the effects of these measures on their surroundings requires an understanding of the network-like sociological and ecological structures, which cannot be conveyed to a student in conventional ways. This is where using computers in the classroom practically suggests itself: with the help of a suitable mathematical model, the situation can be simulated on the basis of certain simplifications and assumptions.

2. MORETTI in the Classroom

In the summer of 1985, the German regional public television channels broadcast a program in which six participants in an adult-education course were to carry out a development-aid project in a computer simulation. Communication with the computer took place exclusively through printed tables and a presenter who operated the keyboard. We found the idea of playing through such a complex web of relationships with the help of a computer thoroughly convincing; putting it into practice in the classroom, however, called for a few changes.

Using the program/computer had to be made so simple that no presenter/teacher is needed to communicate with the computer; instead, every student, alone or in a small group (up to 3 students), can easily manage it. Likewise, the data for analyzing the situation (the actual state) and for planning the measures for the coming years (the target state) had to be presented in a

form that students in grades 9 to 11 find understandable and appealing. Last but not least, some of the relationships built into the program seemed to us still too simplified or incomplete.

During the cold northern German winter months of 1985/86, we (two secondary-school teachers and hobby computer scientists) knitted together a program for the Schneider CPC in about 100 man-hours. It met our requirements and was then tried out in a 10th-grade class (a geography elective) by the colleague who taught it. This colleague initially had a pronounced computer phobia, but we pushed him into it for his own good: he used the program for about 10 class periods as part of his teaching unit and afterwards was just as enthusiastic as his students.

From now on, the program MORETTI.BAS is to be used regularly for practicing what has been worked out in theory on the subject of development aid. In our view, using the computer at school is highly questionable if a topic can be brought home to students better in some other way. In this case, however, working with the computer opens up entirely new areas; only in this way is it possible to study the development of mathematically describable systems in a vivid way. The decisive advantage is that students can now experience and apply theoretical relationships in a practical and playful way; some relationships even reveal themselves only through hands-on work with the simulation.

3. Problems of the Sahel Zone

The Sahel zone of West Africa, the transitional area between the Sahara Desert and the tropical savanna, has long been a contact zone between nomadic and settled populations. On the one hand, this makes it possible to exchange crops and livestock products; on the other, the twofold claim to the land leads to serious problems in agricultural development. This is especially true when droughts restrict the ways in which the habitat can be used. The relationship between the ecological potential and the actual use of natural resources proves to be a vital question. It has been shown that agricultural overuse, both in crop farming and in livestock farming, causes serious damage and encourages the spread of areas with desert-like conditions. (1)

3.1. Rainfall

Rainfall fluctuates very widely; in drought years the deviation sometimes exceeds 50%. When assessing the rainfall to be expected, one must therefore not go by the absolute annual mean. Dry years clearly outnumber wetter years. The decisive feature of the Sahel zone is the short summer rainy season with highly variable amounts of rain, and the danger that several dry years in a row will lower the water table and so trigger a drought disaster.

3.2. Livestock Farming

In good years the herds multiply rapidly. This leads to overgrazing: the pasture is so badly damaged by grazing that the plant cover fails to regenerate naturally. The area thus turns into desert, which reduces the food supply for the cattle. In drought years, large parts of the herds that grew in better times then starve, because the animals are not sold in time and so no capital gain is made. The reason for this is a historically rooted notion of ownership: the owner cannot dispose of the cattle as he pleases; rather, he is responsible for them on behalf of his ancestors, and that of course means preserving his holdings as proof of special power and standing. In northern Ghana, for example, an attempt was made to build up a meat industry, but despite the existing herds the farmers sold only a few animals, which was not enough for the factory.

3.3 Groundwater and Wells

The water supply in the Sahel is threatened by desertification wherever too many wells make the groundwater reserves dwindle rapidly. Ring-shaped areas form around the wells in which vegetation is no longer possible: the water table sinks, and in addition the plants are eaten and trampled by the herds that crowd around the wells. In this way, rings of bare ground up to 32 km in diameter develop around deep wells. Even farther out, part of the vegetation dies from lack of water.

3.4 Population Growth

The rapid growth of the population in the developing countries is one of the greatest obstacles to breaking the cycle of poverty. According to current estimates, the world population will grow from about 4.7 billion people at present to about 6.1 billion by the year 2000. Since about 80% of this growth takes place in the developing countries, 78% of the world's population will be living in developing countries at the turn of the millennium, instead of just over 70% today. The causes of this population explosion lie both in rising birth rates and in falling death rates. Through the spread of the achievements of modern medicine, the fight against epidemics with mass

vaccinations, and improved hygienic conditions, the average life expectancy of the population in the developing countries has been raised to 55 years (for comparison: about 70 years in Central Europe). Family planning is a crucial contribution to getting the population explosion under control. (2)

3.5 Trade at Markets

Markets are held every few days in central towns and larger villages. That does not mean, however, that lively trading goes on there according to the market-economy principles we are used to. Cattle, poultry or grain (millet) are sold only in order to use the proceeds to buy other essential goods that people cannot produce themselves. Supply and demand on the market have hardly any influence on the production of agricultural goods.

4. The Mathematical Model of the MORETTI Simulation

We have described all the aspects listed in the previous section, together with their mutual dependencies and feedback loops, by mathematical equations. They form the heart of the simulation.

In constructing these equations, we were advised on the one hand by an expert on agriculture in the Third World; on the other hand, we tested the behavior of the model through many calculations and trial runs of parts of the program. It turned out that the linear relationships used very often in other programs do not lead to realistic results. Increasing the number of wells from 2 to 4, for instance, will have different effects than increasing it from 4 to 6 or from 7 to 9. Furthermore, the amount of vegetation also depends on other parameters: the number of cattle, rain, the water table, the vegetation area and evaporation.

5. Operating the Program

First of all, an important note for CPC 464 owners: **MORETTI.BAS** uses the extended BASIC command set of the CPC 664/6128. To be able to run it on your computer anyway, you have to install the 664/6128 emulation, which is on your disk / cassette and makes the extended BASIC command set available on the CPC 464 as well. You do this by typing **RUN "EMU"** and then waiting until the message *Basic 1.1 installiert* (Basic 1.1 installed) appears on the screen. After that you can start the Moretti program as usual

with **RUN "MORETTI"**. You will find more information on the emulator in the appendix.

The simulation program MORETTI is about leading a group of people in the African savanna, namely the Moretti, to greater stability in their living conditions through suitable measures, that is, making sure that there is enough food and enough water and that people and animals become sufficiently resistant to epidemics. For this purpose, a capital of 500,000 DM is available, which the development worker has to manage carefully.

By using these funds wisely, he is to enable the people to get by without subsidies from then on and to raise the money they need by selling cattle. If this fails and the population or the number of cattle drops below a certain limit, the project has failed, and the program says so.

The parameters are linked to one another by the mathematical model mentioned above and offer every opportunity to carry out a concept of your own, which may of course contain all the mistakes of past development-aid policy: exhausting the groundwater with deep wells, restricting oneself to food aid, medical care without birth control, and so on. A project is successful if it manages to keep the number of Moretti roughly constant for 30 years, preferably with improved living conditions (cattle per inhabitant = standard of living) and a relatively high resistance to natural influences (epidemics, dry spells).

To explain further how to proceed in the program, the various screens are shown and commented on below; from the main screen (Chronologie, i.e. chronology) you can always switch to any of the other screens. When working with this simulation, the lesson is often over before the project is; for this reason the current state can be saved to disk at any time with the S key (speichern, save), so that in another lesson you can load it again via L (laden, load) and carry on where you left off.

5.1 The Main Screen, Called Chronologie (Key C)

The upper part contains the essential information for the last three years, so that trends can be recognized and assessed. For the water table, +0.0 stands for the mean value. Deviations from it result from rain, drought and the operation of wells. The lower part gives a brief overview of the measures taken for the current year, so that, together with the other information, it provides a basis for planning the following year. The TAB key brings up the key assignments for selecting the other screens. Upper and lower case do

not matter here; keys that have no function are ignored by the program and do not cause any malfunctions.



5.2 Entwicklungshilfemaßnahmen (Development Aid Measures), Key M

On this screen you can plan various measures for the following year, which can be applied at different intensities with correspondingly different effects. This screen serves both as an overview and explanation of the measures and for changing them (key V, for verändern, change). When fighting cattle epidemics, using contraceptives (means of preventing conception) and providing medical care, bear in mind that every time you raise the intensity level, the basic costs per level are incurred, so frequent changes of level use up a correspondingly large amount of capital. To buy or sell cattle, or to buy millet, the program switches to another screen that shows the market situation. Supply and demand, as well as the prices, change from year to year. None of the decisions made here take effect until you move on to the following year on the main screen (Chronologie).



5.3 Rindermarkt von Dorso Duro (Cattle Market of Dorso Duro)

The cattle market is used to enlarge the herds after epidemics or, when the herds are large enough, to raise funds for necessary measures. As with the next item, you are taken to this screen automatically when you make changes.



5.4 Hirsemarkt von Grana Nera (Millet Market of Grana Nera)

Here you can buy millet, which serves as food aid. If millet has been bought but was not needed, part of it is eaten and fed to the animals, or it is stored, and part of the stored millet spoils, so that the stock slowly shrinks.

5.5 Darstellung der kritischen Werte (Display of the Critical Values), Key K

This screen helps you assess the situation of the Moretti. Two bar charts show how good the food base is for the people and for the cattle. At values below the subsistence minimum, some of the animals or of the Moretti die. The cattle's food supply is an important aid in deciding whether to buy or sell animals; buying cattle makes no sense, for instance, if even the animals you already have do not get enough fodder.

The graph for the people's food supply takes only the number of cattle into account as a food base, so with existing millet stocks or a timely purchase of millet, values below the subsistence minimum can be bridged without anyone starving to death.

5.6 Graph of the Population Figures (Key G for Grafik)

The main purpose of this graph is to show how the numbers of people and cattle develop, and so both to document the course of the project over the entire period of 30 years and to make long-term developments and their causes visible at a glance. Particularly useful for this are the colored rectangles shown on the time axis, which mark when epidemics broke out among the people (below the axis) and the animals (above the axis) and thus explain the shape of the two curves. By comparing several graphs after the 30 years are over, the effects of different development-aid strategies can be shown very clearly and their effectiveness discussed.

References:

(1) Der Bundesminister für wirtschaftliche Zusammenarbeit: Leben am Rande der Sahara, Rautenstrauch-Joest-Museum, Köln 1981

(2) Der Bundesminister für wirtschaftliche Zusammenarbeit: Politik der Partner, 6th edition 1983

Der Bundesminister für wirtschaftliche Zusammenarbeit: Zusammenarbeit mit Entwicklungsländern: So sieht die Praxis aus, 4th edition 1984

B. Rinder: Unterrichtsskizze Dürrekatastrophe im Sahel. In: Geographie im Unterricht, 6/80; Friedrich Verlag, Seelze

CHAPTER 10

The Readers Have the Floor

In this chapter you will find two readers' contributions to the Software Experiment, standing in for the many other letters this series prompted, and, as a bonus, the CPC version of an article that was originally published in the magazine PASCAL International.

The One-Dimensional Cellular Automaton

Self-organizing systems and cellular automata were already covered in the second chapter. We owe a particularly interesting and graphically very appealing variant of this theme to Mr. Brinkmann of Ankum.

The program *Zellautomat* (cellular automaton; **ZAUTOMAT.BAS**) is a tool for the graphic display of one-dimensional cellular automata. Unlike the Game of Life, which spreads out over a surface, the one-dimensional cellular automaton develops along a line. The fate of a cell in the following generation is determined by the state of its left and right neighbors and of the cell itself. By arranging one generation below the other, it is possible to make the whole development visible in a single picture. The logic, or rules, of each cellular automaton can be freely defined by the user.

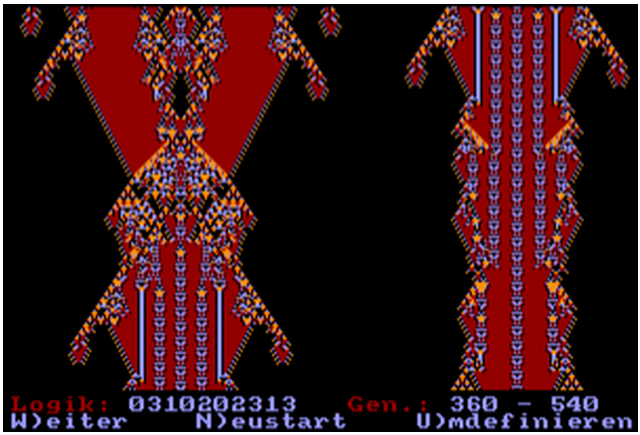
In theory, 4^{10} (more than 1 million) different assignments are conceivable. Most of them, however, are uninteresting. Such cellular automata can not only produce pretty graphics but also serve as models for physical processes such as turbulent flows. It is also suspected that some of the one-dimensional cellular automata are universal computers, just like the Game of Life.

The cells (here 320 per line) can take the values 0, 1, 2 and 3 and are shown on the screen as a dot in the corresponding PEN color. To calculate the next generation, three adjacent values are added up each time. An assignment

table maps the result to a number that determines the state of the middle cell in the next generation. Here is an example:

Sum of the three cell values:	0	1	2	3	4	5	6	7	8	9
New value of the middle cell:	0	3	1	0	2	0	2	3	1	3

The following picture shows the graphic effect of this assignment table. The example given is the cellular automaton that is available after the program starts. Since the upper row of the table always stays the same, giving the lower row defines the cellular automaton completely.



And now a few notes on operation: after you start the program ZAUTOMAT.BAS, an arrow appears at the top edge of the screen and the X position of this arrow at the bottom edge, and the computer waits for you to enter the starting points. If you press the COPY key, a dot in the color of PEN 1 appears in the top screen line directly above the arrow. Pressing it again turns it to color 2 or 3, or erases it. The arrow itself is moved with the cursor keys left and right. Once you have entered the starting points, you can start the cellular automaton with the RETURN key. You can interrupt the drawing of the picture at any time by pressing any key, which takes you to a small menu. It offers the following three items:

W)eiter (continue): The output of a picture already begun is continued, or the output of the next 10 generations is started.

N)eustart (restart): This takes you back to the input mode for the starting points.

U)mdefinieren (redefine): This item lets you define a new logic pattern by entering, as explained above, the lower row of the assignment table without commas or separators. The following table shows a few more interesting logic patterns to try out. With an assignment table of your own and the starting conditions, the possibilities for experimenting are practically inexhaustible.

Sum of the three cell values:	0	1	2	3	4	5	6	7	8	9
New value of	0	2	3	0	1	0	3	2	3	1
the middle cell:	0	2	3	0	2	1	3	2	3	1
	0	2	1	1	3	3	2	2	0	3
	0	2	1	1	0	3	2	0	1	3
	0	1	2	1	3	0	2	3	2	2
	0	1	2	1	3	1	0	0	0	0

New Sounds with the CPC

Unfortunately, the response to the suggestion made in the Software Experiment, to invent methods and programs that make mathematical structures on the CPC not only visible but also audible, was not as great as we had hoped. That is actually surprising, since even the basics of music theory are full of mathematics. In the following contribution, Mr. Winkelmann of Hanover shows how the computer can be used to break new ground in sound generation in a mathematical way, ground on which conventional musical instruments cannot be used at all, or only to a limited extent:

Familiar (European) music divides an octave into twelve steps, the so-called semitone steps. This raises the question of whether an octave could not also be divided into a different number of steps. To realize this idea, however, you first have to know the mathematical law by which the familiar steps can be calculated. With the octave this is still relatively simple:

it corresponds to double the frequency, so the tone period in the SOUND command (the second parameter) has to be halved. Here is a short example program:

```
100 INPUT "Grundperiode:",grund
110 oktave=grund/2
120 REM Play the base period
130 SOUND 7,grund,200,7
140 REM Play the octave
150 SOUND 7,oktave,200,7
160 REM Both at the same time
170 SOUND 1,grund,300,7
180 SOUND 2,oktave,300,7
```

For the semitone steps the calculation is not much more difficult either, once you know that in equal temperament there is a simple mathematical relationship between the frequencies:

Base tone	$440 \cdot 2^{0/12}$ Hertz
Minor second	$440 \cdot 2^{1/12}$ Hertz
Major second	$440 \cdot 2^{2/12}$ Hertz
Minor third	$440 \cdot 2^{3/12}$ Hertz

…and so on. The numerator in the exponent thus corresponds to the number of semitone steps. That would solve the problem, and you can write a short program that makes the semitone steps audible. Note, however, that the tone periods in the CPC's SOUND command are reciprocal to the tone frequencies. So you must not multiply by the powers of two but must divide by them to get an ascending sequence of tones. These considerations lead to the following program:

```
100 grundton = 284:REM concert pitch A
110 FOR halb = 0 TO 12
120   leiterton= grundton/2^(halb/12)
130   Sound 7,leiterton,100,7
140 NEXT halb
```

But to divide the octave into some number of tones other than the usual twelve, you simply replace the 12 in lines 110 and 120 with the number you want. This leads to the following program:

```
100 INPUT "Unterteilung:",anz.ton
110 grundton = 284
120 FOR schritt = 0 TO anz.ton
130   leiterton = grundton /2^(schritt/anz.ton)
140   SOUND 7,leiterton,100,7
150 NEXT schritt
```

With the help of this suggestion, anyone should be able to calculate a table of unusual tone steps and experiment with it. Dividing the octave into 24 steps is particularly interesting, because on the one hand you still have the familiar semitones at your disposal, and on the other hand also the quarter tones in between.

The Well-Tempered Clavier

Besides equal temperament, which is customary today for instruments such as the piano or the guitar and divides an octave into twelve equal semitone steps, some instruments, such as the violin, still use what is called just intonation, in which the frequencies of the tones form certain whole-number ratios. One consequence of this kind of tuning is that a violinist fingers the note *E-flat* differently from the note *D-sharp*, although in equal temperament they are the same note. The equal temperament customary today is really only a good and practical approximation to the original just intonation; it was what made it possible in the first place, for example, to build an instrument like the piano, on which all keys can be used on an equal footing.

But just intonation also lends itself to experiments. As a starting point, there is the program *Frequenzverhältnisse* (frequency ratios; **SOUND1.BAS**), which makes the ratios between frequencies audible. After you enter a base frequency and a ratio, the second frequency is calculated. The two tones are played first one after the other and then together.

Experimenting with this short program, you quickly find that simple numerical ratios produce consonances (tones that sound harmonious together); the simpler the ratio, the more beautiful the sound. And indeed, in just intonation the common consonances have the following frequency ratios:

Unison	1:1	Major third	5:4
Octave	2:1	Minor third	6:5
Fifth	3:2	Major sixth	5:3
Fourth	4:3	Minor sixth	8:5

The degree of consonance decreases here from top to bottom. In the Middle Ages, only those intervals whose ratios had a numerator and denominator between 1 and 4 were perceived as consonant, that is, the unison, the fourth, the fifth and the octave. Since then, the other four intervals listed have also come to be heard as consonances. With the frequent use of seventh chords and similar dissonances in today's music, our sense of consonance will probably go on dulling, and the difference between consonance and dissonance will become more and more blurred in the future.

Here, too, the computer in principle lets you break new ground. For example, by dividing the octave into 24 tones you could build a triad that lies between major and minor. In a major triad, a major third and a perfect fifth sit on top of a root (e.g. C-E-G); in a minor triad, by contrast, a minor third and a perfect fifth sit on top of the root (e.g. D-F-A). If you now build a triad from a root, a perfect fifth and the tone between the minor and the major third, you get a chord that lies between major and minor. The program **SOUND2.BAS** makes it audible. There is certainly still a wide field here for experiments of your own!

Pascal Sends His Regards

After this excursion into the world of sound, back to graphics: the following piece, from the author's own pen, would have fit perfectly into the series *The Software Experiment*, but unfortunately it came too late. Instead, it was published in issue 10/87 of the magazine PASCAL International, and it demonstrates how hidden structures in Pascal's triangle can be made visible with amazingly simple means. Here we make up for the omission with the corresponding BASIC version (**TRIANGLE.BAS**), which takes us back to the topic of the first chapter and brings us full circle.

In the beginning there was nothingness: an infinite field of numbers made up entirely of zeros, perfectly homogeneous, symmetric in every respect.

```
0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0
```

But behind this monotonous uniformity there lay hidden a law that related all the numbers to one another, an invisible force field:

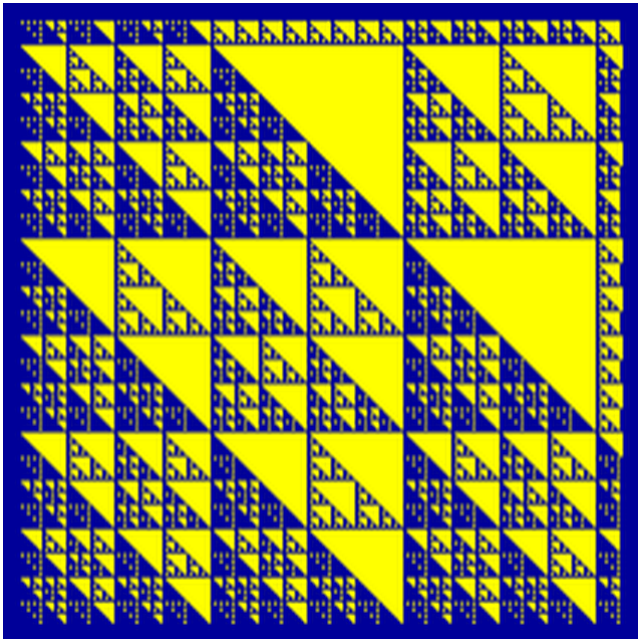
Every number is equal to the sum of the numbers above it to the left and to the right,

and the zeros obeyed the law in an utterly meaningless way, without being able to give it any expression. But just as our physical space reveals its properties only when it contains matter, the number field showed its true character one day, when out of the blue a small, inconspicuous ONE strayed into it, shattered at one stroke the boundless harmony of the self-identical, and, like a condensation nucleus, set off an avalanche of numbers that poured unstoppably across the plane into all infinity:

```
0   0   0   0     1     0   0   0   0
  0   0   0     1     1     0   0   0
0   0   0   1     2     1   0   0   0
  0   0   1     3     3     1   0   0
0   0   1   4     6     4   1   0   0
  0   1   5    10    10     5   1   0
0   1   6   15    20    15   6   1   0
```

All of a sudden there was greater and smaller, even and odd, right and left, direction and opposite direction… in short, the plane had acquired a concrete structure. And this structure is well known to mathematicians: it is Pascal's triangle, which proves particularly useful for calculating the coefficients of a binomial expression of the form $(a+b)^n$. But tables of numbers and formulas can hardly convey what has really come into being in this mathematical act of creation, any more than a sheet of music can make the music it describes

ring out. Although the visual presentation of mathematics is quite an old discipline, it has really only received its musical instrument with the computer, an instrument that makes mathematical structures visible and audible and, like a microscope, gives us glimpses into worlds that would otherwise remain closed to us.



Even so, the programming effort needed to create such pictures is often amazingly small. In principle, the program **TRIANGLE.BAS** merely determines which numbers of Pascal's triangle are divisible by a given value and marks them on the screen with a dot—and that is enough to make the order and harmony hidden in the structure visible. To make good use of the screen, the number field is tilted by 45 degrees and limited to a square section. Since the numbers of the triangle grow fairly quickly and overtax the range and precision of the CPC's internal number representation, the program carries on calculating only with the remainder obtained with MOD, which does not change the graphical result, however.

After starting the program, all you need to do is enter an integer greater than 1 and press RETURN, and the CPC then calculates the corresponding picture. When it is finished, the computer waits for any key to be pressed and then accepts a new divisor. If you want to experiment a little with the program, you can try assigning different colors to the remainder in the variable r2(spalte). Line 220 could then look like this, for example (MOD 4 merely limits the range of numbers to valid PEN numbers):

```
220 PLOT spalte*2+120,Zeile*2,r2(spalte) MOD 4
```

The algorithm comes from the book *Kunst! Computer?* by KarlHeinz Koch (tewi-Verlag), which, besides many philosophical references and critical remarks on computer art, contains numerous other experiments of this kind. Admittedly, most of the programs are written in Atari BASIC (800 XL) and for output on a dot-matrix printer. The detailed documentation, however, makes it easy to transfer the methods used to the CPC.

Appendix

1. The Files of the Software Experiment

The following list tells you in which chapter each file on the Software Experiment disk is described and what it is for:

File name	Chapter	Description
KOMFOL.BAS	1	Demonstrates complex number sequences
KOMGRA.BAS		Generates complex graphics (fractals)
REPRO.BAS	2	Generates multicolored reproduction patterns
LIFE.BAS		Game of Life, an example of cellular automata
OEKOEXP.BAS	3	Main program for the ecology experiment
OEKOERW1 (*)		Extensions to the eco experiment (MERGE)
OEKOERW2 (*)		
OEKOERW3 (*)		
GOBANG.BAS	4	Program for the board game Gobang
KALAHA.BAS	5	Program for the bean game Kalaha
ZUGCOMV1 (*)		Experimental versions of the move calculation in BASIC (MERGE)
ZUGCOMV2 (*)		
MSCHACH.BAS	6	Mini chess program that learns
BRUDER (*)		Modification for Mini Chess (MERGE)
WMASTER.BAS	7	Wordmaster word-guessing program
LISTE4.DAT		Word lists for Wordmaster (4- and 5-letter nouns)
LISTE5.DAT		
WKETTE1.BAS	8	Word chain program, version 1
WKETTE2.BAS		Loader for the word chain program, version 2
KETTEV2		Machine code, loaded by WKETTE2.BAS.

File name	Chapter	Description
MORETTI.BAS (**)	9	Development aid simulation
ZAUTOMAT.BAS	10	Demonstration of a one-dimensional cellular automaton
SOUND1.BAS		Demo programs for new ways of generating sound
SOUND2.BAS		
TRIANGLE.BAS		Graphic structure of Pascal's triangle
EMU.BAS	App.	Loader for the 664/6128 emulator
EMU.BIN		Machine code for the emulator

(*) To get around a bug in the MERGE function of the CPC 464, all BASIC program parts that are loaded in later are stored in ASCII format. If you want to copy these files to another disk on the CPC 464, you must therefore save them again with SAVE "filename",A. On the CPC 664/6128 the storage format makes no difference.

(**) On the CPC 464, the program MORETTI.BAS runs only if the 664/6128 emulator has first been installed with RUN "EMU". To make sure it works smoothly with the emulator, the program was saved in ASCII format, which makes for a relatively long loading time. You can, however, save the program again normally (i.e., without the A option); it will then load faster, but in this form it will run only on your type of computer.

2. The 664/6128 Emulator for the CPC 464

The emulator lets you use the extended command set of Locomotive BASIC 1.1 on the CPC 464 as well. The program consists of two parts: running EMU.BAS hooks the machine-code program EMU.BIN into the BASIC interpreter of the CPC 464, where it takes care of handling the new commands. **Before you install the emulator with RUN "EMU", you should reset the computer to its power-on state with CTRL-SHIFT-ESC!** After a short loading time the message *BASIC 1.1 installiert* appears on the screen. The emulator remains active until the computer is next reset or switched off.

With the help of the emulator, almost all BASIC programs written for the CPC 664/6128 run on the CPC 464. Problems can arise, however, if the programs contain a machine-code part that uses the same memory area as the emulator (from &A200 up). Also, in a program written on the 664/6128, the CLEAR INPUT command still causes a Syntax error on the CPC 464 despite the emulator, and as usual the offending line is shown on the screen. In that case, press ENTER to enter the line again unchanged, and restart the program. The emulator takes care of the adjustment automatically. This problem does not occur if the 664/6128 program was saved in ASCII format with SAVE "name",A.

And now the list of the new commands:

CLEAR INPUT

Clears the keyboard buffer and prevents characters typed while a program is running from being picked up unintentionally by INPUT or INKEY$.

COPYCHR$(#⟨*window*⟩)

This function returns the character at the current cursor position in the specified text window.

CURSOR ⟨*switch1*⟩**,** ⟨*switch2*⟩

Switches the cursor off or on while a program is running. The effect of the switches is shown in the following table:

`CURSOR 0,0`	off/off during INPUT
`CURSOR 0,1`	off/on during INPUT
`CURSOR 1,0`	off/off during INPUT
`CURSOR 1,1`	on/on during INPUT

DEC$ ⟨*numeric expression*⟩**,** ⟨*format string*⟩

Returns the value of the given numeric expression as a formatted string. The format is specified as with PRINT USING (see the CPC user's manual). Example:

```
a$ = DEC$(PI,"#.###"): PRINT a$
```

The function actually already exists on the CPC 464, but because of a bug in the interpreter it always demands two opening parentheses after DEC$.

FILL ⟨*pen*⟩

Fills an area of any shape with the specified color. Only lines drawn in the current graphics color (GRAPHICS PEN) or in the fill color count as its outline or boundary. Filling starts at the current position of the graphics cursor, which must therefore first be placed inside the area with MOVE.

FRAME

Waits for the monitor's vertical retrace, which makes it easier to program flicker-free animated graphics.

GRAPHICS PAPER ⟨*pen*⟩

Sets the color of the graphics background for text output at the graphics cursor position (after TAG) and when the MASK command is used.

GRAPHICS PEN ⟨*pen*⟩**,** ⟨*background*⟩

Selects the drawing color and the background mode for graphics. The second, optional parameter has the following meaning:

0 = normal (overwrite), 1 = transparent.

MASK ⟨*bit mask*⟩

A bit mask is a value from 0…255 that determines the state of each group of eight adjacent pixels when a line is drawn. This lets you produce lines that are broken up or dashed in any pattern you like.

PEN ⟨*#window*⟩**,** ⟨*pen*⟩**,** ⟨*background*⟩

As on the CPC 464, except that the last parameter now lets you set the background mode for text output (0= normal, 1= transparent). Either of the last two values may be left out.

MERGE and CHAIN MERGE "filename"

These commands now work properly with disk drives, too.

Passing parameters with CALL and RSX commands:

With the CALL command and with RSX commands, strings can be passed directly, without the detour via the at sign (@). Example: |ERA, "TESTPROG.BAK"

Additional graphics parameters:

– All graphics commands are extended to the format **COMMAND x, y, pen, write mode**. This applies to the commands MOVE, MOVER, PLOT, PLOTR, DRAW and DRAWR. The optional parameter *write mode* specifies how the new pixels are combined with the background:

0 = overwrite, 1 = XOR mode, 2 = AND mode, 3 = OR mode