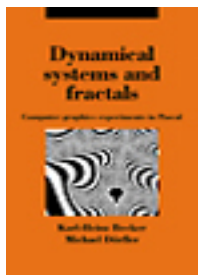


Dynamical Systems and Fractals

Computer Graphics Experiments with Pascal

Karl-Heinz Becker, Michael Dörfler, Translated by I. Stewart

Book DOI: http://dx.doi.org/10.1017/CBO9780511663031

Online ISBN: 9780511663031

Hardback ISBN: 9780521360258

Paperback ISBN: 9780521369107

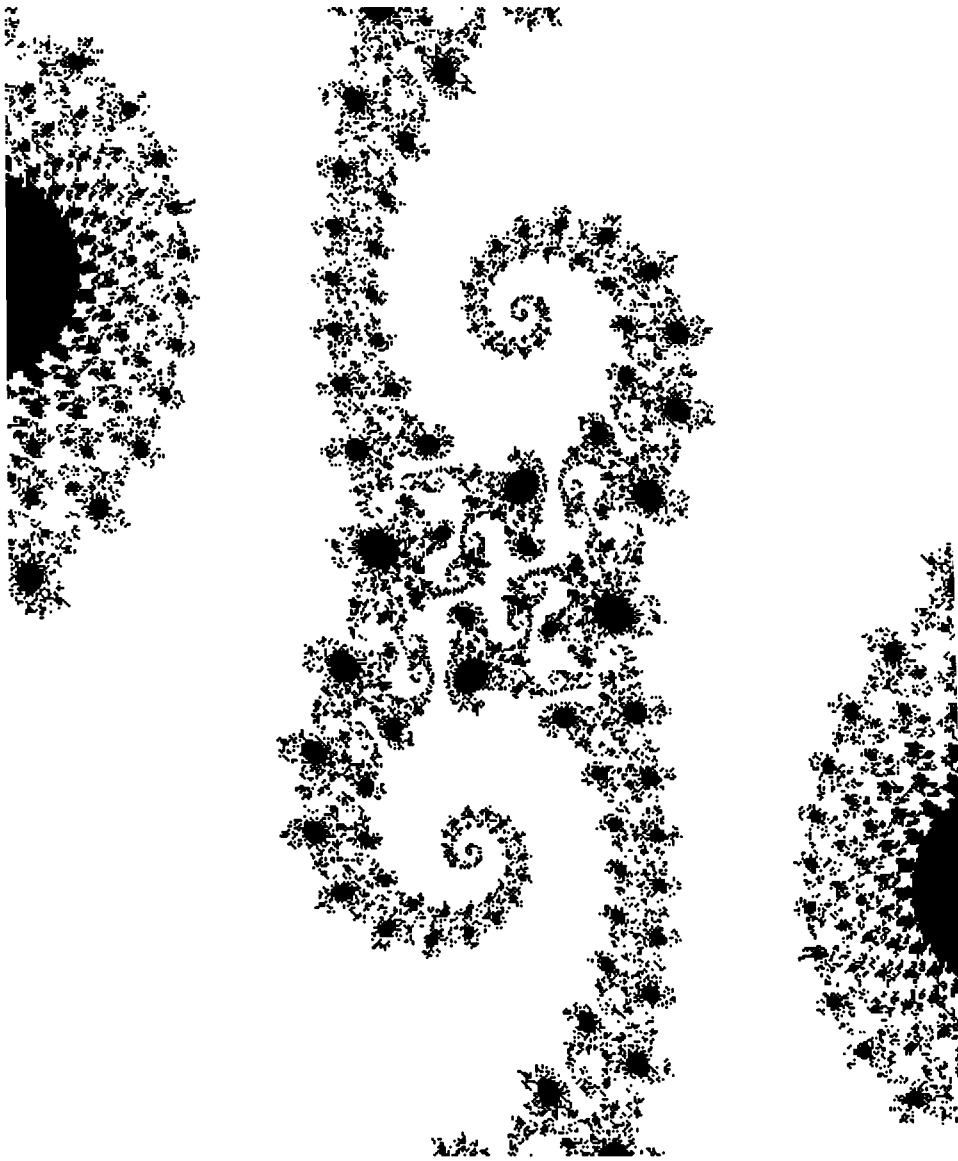
Chapter

12 - Pascal and the Fig-trees pp. 327-378

Chapter DOI: http://dx.doi.org/10.1017/CBO9780511663031.014

Cambridge University Press

12 Pascal and the Fig-trees



12.1 Some Are More Equal Than Others - Graphics on Other Systems

In this chapter we show you how to generate the same graphic of a Feigenbaum diagram on different computer systems. We have chosen the Feigenbaum diagram because it does not take as long to produce as, say, the Gingerbread Man. We will give a 'Feigenbaum reference program' for a series of types of computer, operating systems, and programming languages. Using it you can see how to embed your algorithms in the appropriate program.

Our reference picture is shown in Figure 12.1-1.

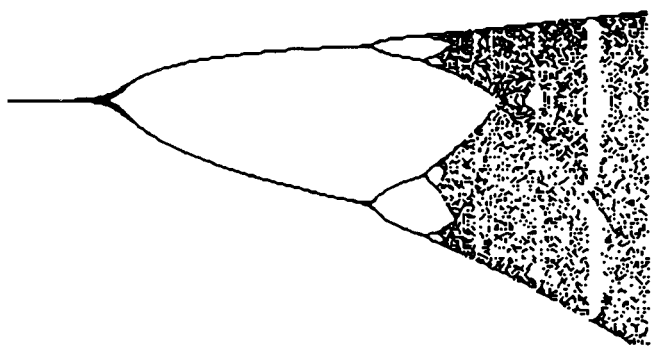


Figure 12.1-1 Feigenbaum reference picture.

12.2 MS-DOS and PS/2 Systems

With IBM's change of direction in 1987, the world of IBM-compatibles and MS-DOS machines changed too. In future as well as MS-DOS there will be a new IBM standard: the OS/2 operating system. The programmer who wishes to develop his Gingerbread Man program on these computers can do so on both families. Turbo Pascal from the Borland company, version 3.0 or higher, is the system of choice. The only difficulties involve different graphics standards and different disk formats, but these should not be too great a problem for the experienced MS-DOS user. The new graphics standard for IBM, like that for the Macintosh II, is a screen of 640 × 480 pixels. Our reference picture has 320 × 200 pixels, corresponding to one of the old standards.

The experienced MS-DOS user will already be aware of the different graphics standards in the world of MS-DOS. It will be harder for beginners, because a program that uses graphics may run on computer X but not on computer Y, even though both are MS-DOS machines. For this reason we have collected here a brief summary of the important graphics standards. In each case check further in the appropriate manual.

MDA (monochrome display adapter)

- 720 × 348 points
- only text, 9 × 4 pixels per symbol, only for TTL-monitor

CGA (Colour Graphics Adapter with different modes)

- 00: Text 40 × 25 monochrome
- 01: Text 40 × 25 colour
- 02: Text 80 × 25 monochrome
- 03: Text 80 × 25 colour
- 04: Graphics 320 × 200 colour
- 05: Graphics 320 × 200 monochrome
- 06: Graphics 640 × 200 monochrome

HGA (Hercules Graphics Adapter)

- 720 × 348 pixels graphics

EGA (Enhanced Graphics Adapter)

- 640 × 350 pixels graphics in 16 colours, fore- and background

AGA (Advanced Graphics Adapter)

- Combines the modes of MDA, CGA, HGA

For our reference program we aim at the lowest common denominator: the CGA standard with 320 × 200 pixels. If you possess a colour screen, you can in this case represent each point in one of four colours. To use the colour graphics commands, see the handbook.

Many IBM-compatible computers use the Hercules graphics card. Unfortunately with this card the incorporation of graphics commands can vary from computer to computer. We restrict ourselves here to the graphics standard defined in the Turbo Pascal handbook from the Borland company for IBM and IBM-compatible computers.

The top left corner of the screen is the coordinate (0,0); x is drawn to the right and y downwards. Anything that lies outside the screen boundaries is ignored. The graphics procedures switch the screen to graphics mode, and the procedure `TextMode` must be called at the end of a graphics program to return the system to text mode.

The standard IBM Graphics card of the old MS-DOS machines up to the Model AT includes three graphics modes. We give here the Turbo Pascal commands and the colour codes:

`GraphMode;`	`GraphColorMode;`	`HiRes;`
320 × 200 pixels	320 × 200 pixels	640 × 200pixels
$0 \le x \le 319$	$0 \le x \le 319$	$0 \le x \le 639$
$0 \le y \le 200$	$0 \le y \le 200$	$0 \le y \le 199$
black/white	colour	black + one colour

Table 12.2-1 Turbo Pascal commands

Dark	Colours	Light	Colours
0	black	08	dark grey
1	blue	09	light blue
2	green	10	light green
3	cyan	11	light cyan
4	red	12	light red
5	magenta	13	light magenta
6	brown	14	yellow
7	light grey	15	white

Table 12.2-2 Colour codes for high-resolution graphics: Heimsoeth (1985), p. 165.

Table 12.2-2 shows the colour codes needed if you want to use colour graphics. But we recommend you to draw your pictures in black and white. To see that this can produce interesting effects, look at the pictures in this book. The old IBM standard, in our opinion, produces unsatisfying colour pictures: the resolution is too coarse. With the new AGA standard, colour becomes interesting for the user. This of course is also true of colour graphics screens, which can represent 1000 × 1000 pixels with 256 colours. But such screens are rather expensive.

In each of the three graphics modes, Turbo Pascal provides two standard procedures, to draw points or lines:

`Plot (x, y, colour);` draws in point in the given colour.

`Draw (x1, y1, x2, y2, colour)`: draws a line of the given colour between the specified points.

We will not use any procedures apart from these two.

The graphics routines select colours from a palette. They are called with a parameter between 0 and 3. The active palette contains the currently used colours. That means, for example, that `Plot (x,y, colour)` with `colour = 2` produces red on `Palette(0);`, with `colour = 3` the point is yellow with `Palette(2);`. `Plot (x,y,0)` draws a point in the active background colour. Such a point is invisible. Read

the relevant information in the manual.

The following Turbo Pascal reference program is very short, because it works with 'Include-Files'. In translating the main program, two types of data are 'compiled together'. The data `UtilGraph.Pas` contains the useful subroutines and the graphics routines. The data `feigb.Pas` contain the problem-specific part and the input procedure that sets up the data for the Feigenbaum program. We recommend you to construct all of your programs in this way, so that only one command

(*§I feigb.pas *)

relative to the above data names is required.

The Turbo Pascal reference program follows.

Program 12.2-1 (Turbo Pascal reference program for MS-DOS)

```
PROGRAM EmptyApplicationShell;
                    (* only TurboPascal on MS-DOS *)
  CONST
     Xscreen = 320; (* e.g. 320 points in x-direction  *)
     Yscreen = 200; (* e.g. 200 points in y-direction  *)
     palcolour = 1;(*TurboPascal MS-DOS: for palette *)
     dcolour = 15;(*TurboPascal MS-DOS: for draw, plot *)
  TYPE Tstring = string[80];(* only TurboPascal *)
  VAR
     PictureName : Tstring;
     Penx, Peny : integer;
  Left, Right, Top, Bottom : real;
(* Insert further global variables here *)
  Population, Feedback : real;
  Visible, Invisible : integer;

(*$I a:UtilGraph.Pas *)

PROCEDURE Hello;
BEGIN
  ClrScr;
  TextMode;
  InfoOutput ('Representation of                    ');
  InfoOutput ('------------------------------');
  Newlines (2);
  CarryOn ('Start :');
  Newlines (2);
END;
PROCEDURE Goodbye;
```

```
    BEGIN
       CarryOn ('To finish :');
    END;
(* ------------------------------------------------------  *)
(* Here Include-File with problem-specific procedures      *)
    (*$I a:feigb.Pas *)
(* ------------------------- MAIN ---------------------  *)
    BEGIN
       Hello;
       Initialise;
       ComputeAndDisplay;
       Goodbye;
    END.
```

Two useful hints: In Turbo Pascal there is no predefined data type 'string', having a length of 80. Unfortunately the string length must always be specified in square brackets. Our string variables are thus all of type `Tstring`. In the Include-Files `a:` specifies the drive. This is just an example to show what to do when the Turbo Pascal system is on a different drive from the Include-Files.

The Include-File follows: **Util.Graph.Pas**.

```
(* ----------------------UTILITY-------------------------  *)
(* BEGIN: Useful subroutines *)
    PROCEDURE ReadReal (information : Tstring; VAR value
                        : real);
    BEGIN
       Write (information);
       ReadLn (value);
    END;

    PROCEDURE ReadInteger (information : Tstring; VAR value
                        : integer);
    BEGIN
       Write (information);
       ReadLn (value);
    END;

    PROCEDURE ReadString (information : Tstring; VAR value :
Tstring);
    BEGIN
       Write (information);
       ReadLn (value);
```

```
    END;

    PROCEDURE InfoOutput (information : Tstring);
    BEGIN
       WriteLn (information);
       WriteLn;
    END;

    PROCEDURE CarryOn (INFORMATION : TSTRING);
    BEGIN
       Write (information, 'Hit <RETURN>');
       ReadLn;
    END;

    PROCEDURE CarryOnIfKey;
    BEGIN
       REPEAT UNTIL KeyPressed;
    END;

    PROCEDURE NewLines (n : integer);
       VAR
          i : integer;
    BEGIN
       FOR i := 0 TO n DO WriteLn;
    END;
(* END: Useful subroutines *)
(* ----------------------UTILITY-------------------------  *)

(* ------------------------GRAPHICS-------------------------*)
(* BEGIN:  Graphics Procedures *)

    PROCEDURE SetPoint (xs, ys : integer);
    BEGIN
       (*  Insert machine-specific graphics commands here   *)
       Plot (xs, Yscreen-ys, dcolour);
       Penx := xs; Peny := ys
    END;

    PROCEDURE SetUniversalPoint (xu, yu: real);
       VAR xs, ys : real;
    BEGIN
```

```
    xs := (xu - Left) * Xscreen / (Right - Left);
    ys := (yu - Bottom) * Yscreen / (Top - Bottom);
    SetPoint (round(xs), round(ys));
  END;

  PROCEDURE GoToPoint (xs, ys : integer);
  BEGIN
    Plot (xs, Yscreen - ys, 0);
    Penx := xs; Peny := ys;
  END;

  PROCEDURE DrawLine (xs, ys : integer);
  BEGIN
    Draw (Penx, Yscreen - Peny, xs, Yscreen-ys,
                dcolour);
    Penx := xs; Peny := ys;
  END;

  PROCEDURE DrawUniversalLine (xu, yu : real);
    VAR xs, ys : real;
  BEGIN
    xs := (xu - Left) * Xscreen/(Right - Left);
    ys := (yu - Bottom) * Yscreen / (Top - Bottom);
    DrawLine (round(xs), round(ys));
  END;

(*  PROCEDURE TextMode; implemented in Turbo Pascal *)
(*  already exists                                  *)


  PROCEDURE GraphicsMode;
(*  DANGER!     DO NOT CONFUSE WITH GraphMode!!!!!   *)
  BEGIN
    ClrScr;
    GraphColorMode;
    Palette (palcolour);
  END;

  PROCEDURE EnterGraphics;
  BEGIN
    Penx := 0;
    Peny := 0;
```

```
      writeln ('To end drawing hit <RETURN> ');
      write ('now hit <RETURN> '); readln;
      Gotoxy(1,23);
      writlen (' ------------ Graphics Mode --------------');
      CarryOn ('Begin :');
      GraphicsMode;
   END;

   PROCEDURE ExitGraphics
(* Machine-specific actions to exit from Graphics Mode      *)
   BEGIN
      ReadLn;
      ClrScr;
      TextMode;
      Gotoxy(1,23);
      Writeln (' -------------- Text Mode ---------------');
   END;

(* END: Graphics Procedures *)
(* ----------------------GRAPHICS--------------------------*)
```

In the implementation of Turbo Pascal for MS-DOS and CP/M computers we must introduce two special global variables `Penx` and `Peny`, to store the current screen coordinates.

```
(* ---------------------APPLICATION-------------------------*)
(* BEGIN: Problem-specific procedures *)
    FUNCTION f(p, k : real) : real;
    BEGIN f := p + k * p * (1-p);
    END;

    PROCEDURE FeigenbaumIteration;
       VAR
          range, i: integer;
          population, deltaxPerPixel : real;
       BEGIN
          deltaxPerPixel := (Right - Left) / Xscreen;
          FOR range := 0 TO Xscreen DO
             BEGIN
                Feedback := Left + range * deltaxPerPixel;
                population := 0.3;
```

```
            FOR i := 0 TO invisible DO
               population := f(population, Feedback);
            FOR i := 0 TO visible DO
               BEGIN
                  SetUniversalPoint (Feedback, population);
                  population := f(population, feedback);
               END;
         END;
   END;

(* END: Problem-specific procedures                          *)

(*-----------------------APPLICATION-----------------------*)

   PROCEDURE Initialise;
   BEGIN
      ReadReal ('Left                >', Left);
      ReadReal ('Right               >', Right);
      ReadReal ('Top                 >', Top);
      ReadReal ('Bottom              >', Bottom);
      ReadInteger ('Invisible        >', invisible);
      ReadInteger ('Visible          >', visible);
         (* possibly further inputs *)
      ReadString ('Name of Picture   >', PictureName);
   END;

   PROCEDURE ComputeAndDisplay;
   BEGIN
      EnterGraphics;
      FeigenbaumIteration;
      ExitGraphics;
   END;
```

In our reference program in §11.2 we gave a very clear sequence of procedures. All of these should be retained if you do not work with Include-Files. In the example shown here we have changed these procedures slightly. The procedures also are not grouped according to their logical membership of the class 'Utility', 'Graphics', or 'Main'. Instead, `Initialise` and `ComputeAndDisplay` are included among the problem-specific procedures. In this way we can rapidly locate the procedures that must be modified, if another program fragment is used. You need only include new global variables in the main program, and change the initialisation procedure and the procedure calls that lie between `EnterGraphics` and `ExitGraphics`.

12.3 UNIX Systems

The word has surely got around by now that UNIX is not just an exotic operating system found only in universities. UNIX is on the march. And so we will give all of you who are in the grip of such an efficient operating system as MS-DOS a few hints on how to dig out the secrets of the Gingerbread Man on UNIX computers.

Let us begin with the most comfortable possibilities. The UNIX system on which you can most easily compute is a SUN or a VAX with a graphics system. UNIX is a very old operating system, and previously people did not think much about graphics. Usually UNIX systems are not equipped with graphics terminals.

If this is the case at your institution, get hold of a compatible standard Pascal compiler (Berkeley Pascal, OMSI Pascal, Oregon Pascal, etc.). Unfortunately these Pascal systems do not include graphics commands as standard. You must supply the appropropriate commands to set a point on the screen with the help of external C routines. Get together with an experienced UNIX expert, who will quickly be able to write out these C routines, such as `setpoint` and `line`. Of course they may already exist.

A typical Pascal program, to represent a picture on a graphics terminal, has the following structure:

```
PROGRAM EmptyApplicationShell;
                    (* Pascal on UNIX systems *)
  CONST
     Xscreen= 320;   (* e.g. 320 points in x-direction *)
     Yscreen= 200;   (* e.g. 200 points in y-direction *)
  VAR
     Left, Right, Top, Bottom : real;
(* Insert other global variables here *)
  Feedback : real;
  Visible, Invisible : integer;
PROCEDURE setpoint (x, y, colour : integer);
                    external
PROCEDURE line (x1, y1, x2, y2, colour : integer);
                    external;

(* ----------------------UTILITY---------------------------- *)
  ...
BEGIN (* Main Program *)
  Hello;
  Initialise;
  ComputeAndDisplay;
  Goodbye;
END
```

To incorporate the graphics you can without difficulty use the program fragments from this book. One point that causes problems is the data type 'string', which in standard Pascal is not implemented. Replace the use of this type by procedures you write yourself, using the type `packed array[..] of char`.

Another important hint: in many compilers the declaration of external procedures must occur before all other procedure declarations. In Berkeley Pascal everything must be written in lower case.

UNIX is a multi-user system. On a UNIX system the methods shown here, to calculate the picture data in the computer and display them on the screen, are not entirely fair to other system users. With these methods (see below) you can hang up a terminal for hours.

We recommend that you use the methods of §11.5, generating only the picture data on the UNIX computer, and only after finishing this time-consuming task should you display the results on a graphics screen with the aid of other programs.

This has in particular the consequence that your program to generate the picture data can run as a process with negligible priority in the background. It also allows the possibility of starting such a process in the evening around 10 o'clock, leaving it to run until 7 in the morning, and then letting it 'sleep' until evening, etc. Of course you can also start several jobs one after the other. But remember that all these jobs take up CPU time. Start your jobs at night or at weekends, to avoid arguments with other users or the system manager.

Many UNIX systems do not possess a graphics terminal, but these days they often have intelligent terminals at their disposal. That is, IBM-ATs, Macintoshes, Ataris, or other free-standing personal computers are connected to the UNIX system over a V24 interface at 9600 Baud serial. In this case you should generate your picture data on the UNIX system in one of the three formats suggested (see §11.5) and transfer the resulting data on to floppy disk with a file transfer program. Then you can display the picture calculated on the UNIX system on the screen of your computer.

To make everything concrete, we have collected it together for you here.

The Pascal program for generating the picture data has already been listed in §11.5. After typing Program 11.5-1 into the UNIX system you must compile it and run it. The data can be input interactively. Do not worry if your terminal shows no activity beyond that. Doubtless your program is running and generating picture data. However, it can sometimes happen that your terminal is hung up while the program runs. That can take hours.

Press the two keys <CTRL> <Z>. This breaks into the program, and the promptline[1] of the UNIX system appears:

```
%
```

Type the command `jobs`. The following dialogue ensues:

[1] In this case the prompt symbol is the % sign. Which symbol is used differs from shell to shell.

```
% jobs
[1] + Stopped SUNcreate
% bg % 1
[1] SUNcreate
%
```

What does that mean? The square brackets give the operating system reference number of the program that is running. For instance, you can use the `kill` command to cut off this process. The name of the translator program is e.g. `SUNcreate`. By giving the command `bg` and the process number the program is activated and sent into background. The prompt symbol appears and the terminal is free for further work; your job continues to run in the background.[2]

The Pascal program to reconstruct a picture from a text file and diplay it on your computer is also given in §11.5. We recommend you to use the UNIX system as a calculating aid, even if there is no graphics terminal connected to it. You can transfer the resulting picture data to your PC using 'Kermit' and convert them into a picture.

Many users of UNIX systems prefer the programming language C. For this reason we have converted programs 11.5-1, 11.5-3, and 11.5-6 'one to one' into C. Here too the programs should all remain in the background. The resulting data can be read just as well from C as from Pascal.

C Program Examples[3]

Program 12.3-1 (Integer encoded; see 11.5-1)

```
/* EmptyApplicationShell, program 11.5-1 in C */

#include <stdio.h>

#define Xscreen 320 /* e.g. 320 pixels in x-direction */
#define Yscreen 200 /* e.g. 200 pixels in y-direction */
#define Bound 100.0
#define True 1
#define False 0
#define Stringlength 8
typedef char string8 [Stringlength];
typedef FILE *IntFile;
typedef int bool;
   IntFile F;
```

[2]This kind of background operation works only in BSD Unix or in Unix versions with a C shell. Ask your system manager, if you want to work in this kind of fashion.

[3]The C programs were written by Roland Meier from the Research Group in Dynamical systems at the University of Bremen.

```
double Left, Right, Top, Bottom;
int MaximalIteration;
/* include further global variables here */
/* ------------------------ file ---------------------- */
/* begin: file procedures */
void Store (F, number)
IntFile F;
int number;
{
   fwrite (&number, sizeof(int), 1, F);
}

void EnterWriteFile(F, Filename)
IntFile *F;
String8 Filename;
{
   *F = fopen(Filename, ''w'');
}

void ExitWriteFile (F)
IntFile *F;
{
   fclose(F);
}

/* end: file procedures */
/* ---------------------- file ----------------------- */
/* --------------------- application ----------------- */
/* begin: problem-specific procedures */

double x0, y0;

int MandelbrotComputeAndTest (cReal, cImaginary)
double cReal, cImaginary;
{
#define Sqr(X) (X)*(X)

      int iterationNo;
      double x, y, xSq, ySq, distanceSq;
      bool finished;
```

```
   /* StartVariableInitialisation */
   finished = False;
   iterationNo = 0;
   x = x0;
   y = y0;
   xSq = sqr(x);
   ySq = sqr(y);
   distanceSq = xsq + ysq;
   do { /* compute */
      iterationNo++;
      y = x*y;
      y = y+y-cImaginary;
      x = xSq - ySq -cReal;
      xSq = sqr(x);
      ySq = sqr(y);
      distanceSq = xsq + ysq;
      /* test */
      finished = (distanceSq > Bound);
   } while (iterationNo != MaximalIteration &&
            !finished);
   /* distinguish, see also Program 11.5-1 */
   return iterationNo;
#undef sqr
}

void Mapping ()
{
   int xRange, yRange;
   double x, y, deltaxPerPixel, deltayPerPixel;

   deltaxPerPixel = (Right - Left) / Xscreen;
   deltay  PerPixel = (Top - Bottom) / Yscreen;
   x0 = 0.0;
   y0 = 0.0;
   y = Bottom;
   for (yRange = 0; yRange < Yscreen; yRange++){
      x= Left;
      for (xRange = 0; xRange < Xscreen; xRange++){
         store (F, MandelbrotComputeAndTest (x, y));
            x += deltaxPerPixel;
         }
```

```
            store (F, 0);    /* at the end of each row */
        y += deltayPerPixel;
    }
}

/* end: problem-specific procedures */
/* -------------------- application ------------------  */

/* ----------------------- main ----------------------- */
/* begin: procedures of main program */
void hello()
{
   printf(''\nComputation of picture data '');
   printf (''\n------------------------ '');
}

void Initialise()
{
   printf (''Left           > ''); scanf(''%lf'', &Left);
   printf (''Right          > ''); scanf(''%lf'', &Right);
   printf ('Bottom         > ''); scanf (''%lf'', &Bottom);
   printf (''Top            > ''); scanf (''%lf'', &Top);
   printf (''Maximal Iteration   > ''); scanf (''%lf'',
                  &MaximalIteration );
/* insert further inputs here */
}

void ComputeAndStore;
{
     Enterwritefile (&F, ''IntCoded'');
     Mapping();
     ExitWriteFile (F);
}

/* end: procedures of main program */
/* ----------------------- main ----------------------  */

main()  /* main program */
   Hello();
   Initialise();
   ComputeAndDisplay();
}
```

Program 12.3.2 (Compress to int; see 11.5-3)

```
/* CompressToInt, Program 11.5-3 in C */
#include <stdio.h>
#define Stringlength  8
typedef FILE *IntFile;
typedef char String8 [Stringlength];

String8 Dataname;
IntFilein, out;
int   quantity, colour, done ;

void EnterReadFile (F, Filename)
IntFile *F;
string8 FileName;
{
   *F = fopen(Filename, ''r'');
}

void EnterWriteFile (F, FileName)
IntFile *F;
string8 FileName;
{
   *F = fopen(FileName, ''w'');
}

void ExitReadFile(F);
IntFile F;
{
   fclose(F);
}

void ExitWritefile(F);
IntFile F;
{
   fclose(F);
}

void ReadIn(F, number)
IntFile F;
int *number;
```

```
{
   fread(number, sizeof(int), 1, F);
}

void store (F, number);
IntFile F;
int number;
{
   fwrite(&b=number, sizeof(int), 1, F);
}

main()
{
   EnterReadFile (&in, ''IntCoded''); /* Integer encoded */
   EnterWriteFile (&out, ''RLIntDat'');
                  /* RL encloded Integer */
   while (!feof(in)) {
      quantity = 1;
      ReadIn (in, &colour);
      if (!feof(in)) {
         do{
            ReadIn (in, &done);
            if (done != 0)
               if (done == colour)
                  quantity++;
               else {
                  store (out, quantity);
                  store (out, colour);
                  colour = done;
                  quantity = 1;
               }
         } while (done != 0 && !feof(in));
         store (out, quantity);
         store (out, number);
         store (out, 0);
      }
   }
   ExitReadFile (in);

   ExitWriteFile (out);
}
```

Program 12.3-3 (Transfer int to char, see 11.5-5)

```
/* TransferIntToChar, Program 11.5-5 in C */

#include <stdio.h>

#define Stringlength 8

typedef FILE*IntFile;
typedef FILE*CharFile;
typedef charString8[Stringlength];
IntFile in;
CharFile outText;
int quantity, colour;
char CharTable[64];
String8 DataName;

void EnterReadFile (F, FileName)
IntFile *F;
String8 FileName;
{
   *F = fopen(Filename, ''r'');
}

void EnterWriteFile (F, FileName)
CharFile *F;
String8 FileName;
{
   *F = fopen(Filename, ''w'');
}

void ExitReadFile (F)
IntFile F;
{
   fclose (F);
}

void Exitwritefile (F)
CharFile F;
{
   fclose (F);
}
```

```
void ReadIn (F, number)
IntFile F;
int *number;
{
   fread(number, sizeof(int), 1, F);
}

void store (outText, number)
CharFile outText;
int number;
{
   if (number == 0)
      fputc('\n', outText);
   else {
      fputc(CharTable[number/64], outText);
      fputc (CharTable[number % 64], outText);
   }
}

void InitTable()
{
   int i;
      for (i = 0; i < 64; i++) {
         if (i < 10)
            CharTable[i] = '0' + i;
         else if (i < 36)
            CharTable[i] = '0' + i + 7;
         else if (i < 62)
            CharTable[i] = '0' + i + 13;
         else if (i == 62)
            CharTable[i] = '>'
         else if (i == 63)
            CharTable[i] = '?'
   }
}

main()
{
   InitTable();
   EnterReadFile (&in, ''RLIntDat'');
   EnterWriteFile (&outText, ''RLCharDat');
   while (!feof(in)){
```

```
      ReadIn (in, &quantity);
      if (quantity == 0)
         store (outText, 0);
      else {
         store (outText, quantity);
         readin (in, &colour);
         store (outText, colour);
      }
   }
   ExitReadFile (in);
   ExitWriteFfile (outText);
}
```

Those were the hints for UNIX. UNIX is in particular available on the Macintosh II, which brings us to another operating system and another computer family: the Macintosh.

12.4 Macintosh Systems

There is an enormous range of Pascal implementations for the Macintosh family of computers, all very suitable for computer graphics experiments. They include Turbo Pascal (Borland), Lightspeed Pascal (Think Technologies), TML-Pascal (TML Systems), and MPW (Apple), the Apple development system on the Macintosh. Of course the programs can also be written in other programming languages such as C or Modula II. We now give the corresponding Reference Program for one of the cited implementations.

Program 12.4-1 (Turbo Pascal reference program for Macintosh)

```
PROGRAM EmptyApplicationShell;
                    (* Turbo Pascal on Macintosh *)
USES MemTypes, QuickDraw;
   CONST
      Xscreen = 320;  (*e.g. 320 points in x-direction  *)
      Yscreen = 200;  (*e.g. 200 points in y-direction  *)
   VAR
      PictureName : string;
      Left, Right, Top, Bottom : real;
      (* include additional global variables here *)
      Feedback : real;
      Visible, Invisible : integer;
```

```
(* ----------------------UTILITY----------------------------*)
(* BEGIN:  Useful Subroutines *)
    PROCEDURE ReadReal (information : STRING; VAR value
                        : real);
    BEGIN
       write (information);
       readln (value);
    END;

    PROCEDURE ReadInteger (information : STRING; VAR value
                        : integer);
    BEGIN
       write (information);
       readln (value);
    END;

    PROCEDURE ReadString (information : STRING; VAR value
                        : string);
    BEGIN
       write (information);
       readln (value);
    END;

    PROCEDURE InfoOutput (information : STRING);
    BEGIN
       writeln (information);
       writeln;
    END;

    PROCEDURE CarryOn (information : STRING);
    BEGIN
       write (information, ' hit <RETURN>');
       readln
    END;

    PROCEDURE CarryOnIfKey;
    BEGIN
       REPEAT
       UNTIL KeyPressed
    END;
```

```
    PROCEDURE Newlines (n : integer);
       VAR
          i : integer;
    BEGIN
       FOR i := 1 TO n DO
          writeln;
    END;

(* END: Useful Subroutines *)
(* ------------------------UTILITY--------------------------*)

(* ------------------------GRAPHICS-------------------------*)
(* BEGIN:  Graphics Procedures *)

    PROCEDURE SetPoint (xs, ys : integer);
    BEGIN
       (*  Insert machine-specific graphics commands here   *)
       moveto (xs, Yscreen - ys);
       line (0,0)
    END;

    PROCEDURE SetUniversalPoint (xu, yu: real);
       VAR xs, ys : real;
    BEGIN
       xs := (xu - Left) * Xscreen / (Right - Left);
       ys := (yu - Bottom) * Yscreen / (Top - Bottom);
       SetPoint (round(xs), round(ys));
    END;

    PROCEDURE GoToPoint (xs, ys : integer);
    BEGIN
       (*  Insert machine-specific graphics commands here   *)
       moveto (xs, Yscreen - ys);
    END;

    PROCEDURE DrawLine (xs, ys : integer);
    BEGIN
       (*  Insert machine-specific graphics commands here   *)
       lineto (xs, Yscreen - ys);
    END;
```

```
PROCEDURE DrawUniversalLine (xu, yu : real);
  VAR xs, ys : real;
BEGIN
  xs := (xu - Left) * Xscreen/(Right - Left);
  ys := (yu- Bottom) * Yscreen / (Top - Bottom);
  DrawLine (round(xs), round(ys));
END;

PROCEDURE TextMode;
(*  Insert machine-specific graphics commands here      *)
BEGIN
  GotoXY(1,23);
  writlen (' ------------- Text Mode -----------------');
END;

PROCEDURE GraphicsMode;
(*  Insert machine-specific graphics commands here      *)
BEGIN
  ClearScreen;
END;

PROCEDURE EnterGraphics;
BEGIN
  writeln ('To end drawing hit <RETURN> ');
  write ('now hit <RETURN> '); readln;
  GotoXY(1,23);
  writeln('-------------- Graphics Mode ---------------');
  CarryOn('BEGIN :');
  GraphicsMode;
END;

PROCEDURE ExitGraphics
BEGIN
   (*machine-specific actions to exit from Graphics Mode*)
  readln;
  ClearScreen;
  TextMode;
END;

(END: Graphics Procedures *)
(-------------------------GRAPHICS--------------------------*)
```

```
(-----------------------APPLICATION-------------------------*)
(BEGIN: Problem-specific procedures *)
(* useful functions for the given application problem       *)

    FUNCTION f(p, k : real) : real;
    BEGIN f := p + k * p * (1-p);
    END;

    PROCEDURE FeigenbaumIteration;
       VAR
          range, i: integer;
          population, deltaxPerPixel : real;
       BEGIN
          deltaxPerPixel := (Right - Left) / Xscreen;
          FOR range := 0 TO Xscreen DO
             BEGIN
                Feedback := Left + range * deltaxPerPixel;
                population := 0.3;
                FOR i := 0 TO Invisible DO
                   population := f(population, Feedback);
                FOR i := 0 TO Visible DO
                   BEGIN
                      SetUniversalPoint (Feedback, population);
                      population := f(population, Feedback);
                   END;
             END;
       END;

(* END: Problem-specific procedures                          *)
(*-----------------------APPLICATION---------------------- *)

    PROCEDURE Initialise;
    BEGIN
       ReadReal ('Left              >', Left);
       ReadReal ('Right             >', Right);
       ReadReal ('Top               >', Top);
       ReadReal ('Bottom            >', Bottom);
       ReadInteger ('Invisible      >', invisible);
       ReadInteger ('Visible        >', visible);
          (* possibly further inputs *)
       ReadString ('Name of Picture >', PictureName);
    END;
```

```
    PROCEDURE ComputeAndDisplay;
    BEGIN
       EnterGraphics;
       FeigenbaumIteration;
       ExitGraphics;
    END;

(*---------------------------MAIN-------------------------- *)
(* BEGIN: Procedures of Main Program *)

    PROCEDURE Hello;
    BEGIN
       TextMode;
       InfoOutput ('Calculation of                          ');
       InfoOutput ('--------------------------');
       Newlines (2);
       CarryOn ('Start :');
       Newlines (2);
    END;

    PROCEDURE Goodbye;
    BEGIN
       CarryOn ('To stop : ');
    END;

(* END: Procedures of Main program *)

(*---------------------------MAIN-------------------------   *)

BEGIN (* Main Program *)
       Hello;
       Initialise;
       ComputeAndDisplay;
       Goodbye;
    END.
```

Of course all of this also works with Include-Files. Compare this program with the reference program from §12.2.

After the start of the program a window appears with the name of the main program. This window is simultaneously a text and graphics window. That is, not only characters but also the usual graphics commands in Turbo Pascal such as `ClearScreen`

and `GotoXY` apply. It can now be given the appropriate inputs.

On logical grounds we still distinguish here between `TextMode` and `GraphicsMode`. In practice we have implemented this in such a way that text called from the procedure `TextMode` is written in the 22nd row and rolls upwards.

When the drawing is finished, you should see the following picture:

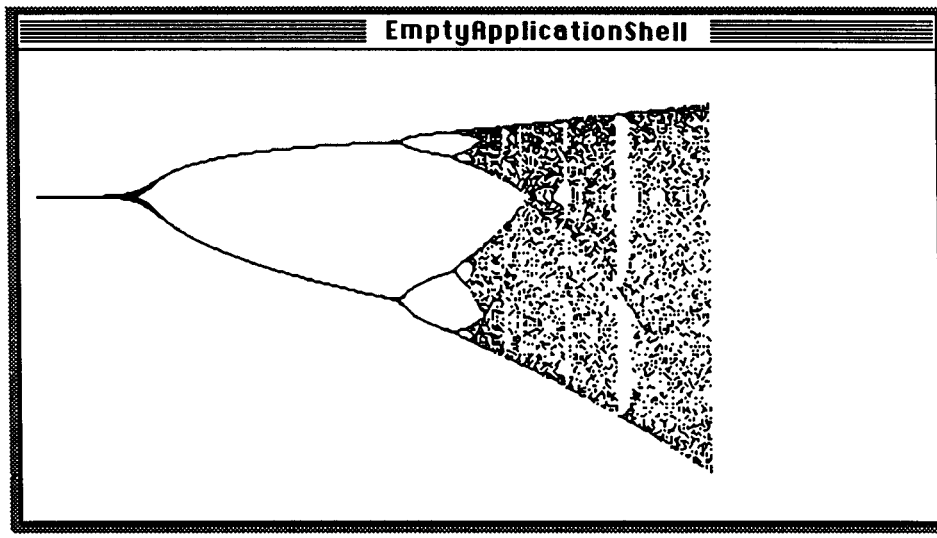


Figure 12.4-1 Turbo Pascal reference picture.

After the drawing is finished the blinking text cursor appears in the left upper corner. You can print out the picture with the key combination

<Shift> <Command> <4>

or make it into a MacPaint document with

<Shift> <Command> <3>.

If you then press the <RETURN> key further commands can be given.

Next, some hints on turtle graphics. We described this type of computer graphics experiment in Chapter 8. You can implement your own turtle graphics as in the solutions in §11.3, or rely on the system's own procedures.

In Turbo Pascal on the Macinstosh, as in the MS-DOS version, a turtle graphics library is implemented. Read the description in Appendix F, *Turtle Graphics: Mac graphics made easier* in the *Turbo Pascal Handbook*, editions after 1986.

In addition to Turbo Pascal there is another interesting Pascal implementation. In Lightspeed Pascal it is not possible to modularise into pieces programs that can be compiled with the main program in the form of Include-Files. In this case we recommend that you use the so-called *unit* concept. You can also use units in the Turbo Pascal version. In the main program only the most important parts are given here.

Program 12.4-2 (Lightspeed Pascal reference program for Macintosh)

```
PROGRAM EmptyApplicationShell;
                    (* Lightspeed Pascal Macintosh *)
(* possible graphics library declarations here *)
USES UtilGraph;
(* include further global variables here *)
VAR
   Feedback: real;
   Visible, Invisible : integer;

PROCEDURE Hello;
BEGIN (* as usual *)
...
END;

PROCEDURE Goodbye;
BEGIN
   CarryOn ('To stop : ');
END;
(* ----------------------APPLICATION-------------------------*)
(* BEGIN: Problem-specific procedures *)
(* useful functions for the given application problem        *)

FUNCTION f(p, k : real) : real;
BEGIN
   f := p + k * p * (1-p);
END;

PROCEDURE FeigenbaumIteration;
   VAR
      range, i: integer;
      population, deltaxPerPixel : real;
   BEGIN  (* as usual *)
   ...
   END;

   PROCEDURE Initialise;
   BEGIN  (* as usual *)
   ...
   END;
```

```
      PROCEDURE ComputeAndDisplay;
      BEGIN
         EnterGraphics;
         FeigenbaumIteration;
         ExitGraphics;
      END;
(* END: Problem-specific procedures            *)
(*-------------------------APPLICATION----------------------
    *)

    BEGIN (* Main Program *)
       Hello;
       Initialise;
       ComputeAndDisplay;
       Goodbye;
    END.
```

In the unit `UtilGraph` we put the data structures and procedures that always remain the same. You can comfortably include all data structures that are global for all programs in such a unit.

```
    UNIT UtilGraph;
    INTERFACE
       CONST
          Xscreen = 320; (* e.g. 320 points in x-direction *)
          Yscreen = 200; (* e.g. 200 points in y-direction *)
       VAR
          CursorShape : CursHandle;
          PictureName : STRING;
          Left, Right, Top. Bottom : real;

(* ------------------------UTILITY ------------------------  *)
    PROCEDURE ReadReal (information : STRING; VAR value
                        : real);
    PROCEDURE ReadInteger (information : STRING; VAR value
                        : integer);
    PROCEDURE ReadString (information : STRING; VAR value
                        : STRING);
    PROCEDURE InfoOutput (information : STRING);
    PROCEDURE CarryOn (information : STRING);
    PROCEDURE CarryOnIfKey;
```

```
    PROCEDURE Newlines (n : integer);
(* ------------------------UTILITY ------------------------  *)
(* ----------------------GRAPHICS ------------------------  *)
    PROCEDURE InitMyCursor;
    PROCEDURE SetPoint (xs, ys : integer);
    PROCEDURE SetUniversalPoint (xw, yw : real);
    PROCEDURE DrawLine (xs, ys : integer);
    PROCEDURE DrawUniversalLine (xw, yw : real);
    PROCEDURE TextMode;
    PROCEDURE GraphicsMode;
    PROCEDURE EnterGraphics;
    PROCEDURE  ExitGraphics;
(* ----------------------GRAPHICS ------------------------  *)
    IMPLEMENTATION
(* ----------------------UTILITY----------------------------*)
(* BEGIN:  Useful Subroutines *)
    PROCEDURE ReadReal;
    BEGIN
       write (information);
       readln (value);
    END;

    PROCEDURE ReadInteger;
    BEGIN
       write (information);
       readln (value);
    END;

    PROCEDURE ReadString;
    BEGIN
       write (information);
       readln (value);
    END;

    PROCEDURE InfoOutput;
    BEGIN
       writeln (information);
       writeln;
    END;
    PROCEDURE CarryOn;
    BEGIN
       write (information, ' hit <RETURN>');
```

```
      readln;
   END;

   PROCEDURE CarryOnIfKey;
   BEGIN
      REPEAT UNTIL button;  (* Lightspeed Pascal *)
   END;

   PROCEDURE Newlines;
      VAR
         i : integer;
   BEGIN
      FOR i := 1 TO n DO
         writeln;
   END;

(* END: Useful Subroutines *)

(* -------------------------UTILITY--------------------------*)

(* -------------------------GRAPHICS-------------------------*)
(* BEGIN:  Graphics Procedures *)

   PROCEDURE InitMyCursor;
   BEGIN
      CursorShape := GetCursor(WatchCursor);
   END;

   PROCEDURE SetPoint (xs, ys : integer);
   BEGIN
      (*  Insert machine-specific graphics commands here   *)
      moveto (xs, Yscreen - ys);
      line (0,0)
   END;

   PROCEDURE SetUniversalPoint (xu, yu: real);
      VAR xs, ys : real;
   BEGIN
      xs := (xu - Left) * Xscreen / (Right - Left);
      ys := (yu - Bottom) * Yscreen / (Top - Bottom);
      SetPoint (round(xs), round(ys));
```

```
END;

PROCEDURE DrawLine (xs, ys : integer);
BEGIN
   lineto (xs, Yscreen - ys);
END;

PROCEDURE DrawUniversalLine (xu, yu : real);
   VAR xs, ys : real;
BEGIN
   xs := (xu - Left) * Xscreen/(Right - Left);
   ys := (yu - Bottom) * Yscreen / (Top - Bottom);
   DrawLine (round(xs), round(ys));
END;

PROCEDURE TextMode;
(*  Insert machine-specific graphics commands here      *)
CONST delta = 50;
VAR window : Rect;
BEGIN
   SetRect (window, delta, delta, Xscreen+delta,
                  Yscreen+delta);
   SetTextRect (window);
   ShowText;  (* Lightspeed Pascal *)
END;

PROCEDURE GraphicsMode;
(*  Insert machine-specific graphics commands here      *)
CONST delta = 50;
VAR window : Rect;
BEGIN
   SetRect (window, delta, delta, Xscreen+delta,
                  Yscreen+delta);
   SetDrawingRect (window);
   ShowDrawing;
   InitMyCursor; (* initialise WatchCursorForm *)
   SetCursor(CursorShape^^); (* set up WatchCursorForm *)
END;

PROCEDURE EnterGraphics;
BEGIN
   writeln ('To end drawing hit <RETURN> ');
```

```
      write ('now hit <RETURN> '); readln;
      writeln('--------------- Graphics Mode --------------');
      CarryOn('BEGIN :');
      GraphicsMode;
   END;

   PROCEDURE ExitGraphics;
   BEGIN
      (*machine-specific actions to exit from Graphics Mode*)
      InitCursor (* call the standard cursor *)
      readln;  (* graphics window no more frozen *)
      SaveDrawing(PictureName);
                     (* store pic as MacPaint document *)
      TextMode;  (* text window appears *)
      writeln('-----------TextMode--------------');
   END;
(* END: Graphics Procedures *)
(* ----------------------GRAPHICS--------------------------*)
   END
```

If you run this program then the following two windows (Figure 12.4-2) appear one after the other.

After inputting the numerical value and pressing <RETURN>, the 'drawing window' of Lightspeed Pascal appears, and the Feigenbaum diagram is drawn. While it is being drawn the 'watch cursor' - the cursor that resembles a wristwatch - is visible. When the drawing is finished the normal cursor appears. Press <RETURN> to give further commands.

As well as Turbo Pascal and Lightspeed Pascal there is a whole series of other Pascal versions or programming languages that will run on the Macintosh. We will make a few remarks about some of these. The graphics procedures are naturally the same in all Pascal versions.

TML Pascal

In contrast to Turbo Pascal and Lightspeed Pascal, TML Pascal generates the most efficient code. Nevertheless we recommend Lightspeed Pascal. In our opinion it is the most elegant development system as regards the simplicity of giving commands to the Macintosh. The graphics procedures are the same, and we do not give a sample program.

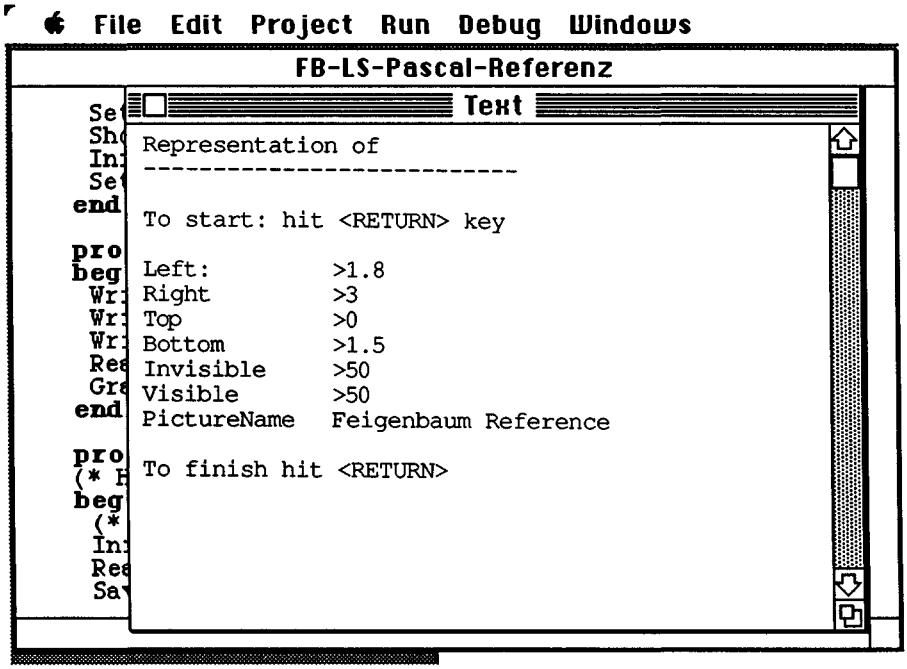


Figure 12.4-2 Screen dialogue.

MPW Pascal

MPW Pascal is a component of the software development environment 'Macintosh Programmer's Workshop', and was developed by Apple. It is based on ANSI Standard Pascal. It contains numerous Apple-specific developments such as the SANE library. SANE complies with the IEEE standard 754 for floating-point arithmetic. In addition MPW Pascal, like TML Pascal, contains facilities for object-oriented programming. MPW Pascal, or Object Pascal, was developed by Apple in conjunction with Niklaus Wirth, founder of the Pascal language.

MacApp

MacApp consists of a collection of object-oriented libraries for implementing the standard Macinstosh user interface. With MacApp you can considerably simplify the development of the standard Macintosh user interface, so that the main parts of Mac programs are at your disposition for use as building blocks. MacApp is a functional (Pascal) program, which can be applied to an individual program for particular extensions and modifications.

Modula II

Another interesting development environment for computer graphics experiments is Modula II. Modula II was developed by Niklaus Wirth as a successor to Pascal. On the Macintosh there are at present a few Modula II versions: TDI-Modula and MacMETH. MacMETH is the Modula II system developed by ETH at Zurich. We choose TDI-Modula here, because the same firm has developed a Modula compiler for the Atari.

Lightspeed C

C holds a special place for all those who wish to write code that resembles machine language. Many large applications have been developed using C, to exploit the portability of assembly programming. Three examples of C programs have been given already in §12.3 (UNIX systems). They were originally programmed in Lightspeed C on the Macintosh and thus run without change on all C compilers.

12.5 Atari Systems

The Atari range has become extremely popular among home computer fans in the last few years. This is doubtless due to the price/power ratio of the machine. Of course, the Atari 1024 is no Macintosh or IBM system 2, but it can produce pretty good Gingerbread Men - in colour.

Among the available programming languages are GFA Basic, ST Pascal Plus, and C. We give our Reference Program here for ST Pascal Plus.

Program 12.5-1 (ST Pascal Plus reference rrogram for Atari)

```
PROGRAM EmptyApplicationShell;
                    (* ST Pascal Plus on Atari *)
  CONST
     Xscreen = 320;   (* e.g. 320 points in x-direction *)
     Yscreen = 200;   (* e.g. 200 points in y-direction *)
(*$I GEMCONST *)
  TYPE Tstring = string[80];
(*$I GEMTYPE *)
  VAR
     PictureName : Tstring;
     Left, Right, Top, Bottom : real;
     (* include additional global variables here *)
     Feedback : real;
     Visible, Invisible : integer;
(*$I GEMSUBS *)
(*$I D:UtilGraph.Pas *)
```

```
    PROCEDURE Hello;
    BEGIN
       Clear_Screen;
       TextMode;
       InfoOutput ('Calculation of                          ');
       InfoOutput ('-------------------------');
       Newlines (2);
       CarryOn ('Start :');
       Newlines (2);
    END;

    PROCEDURE Goodbye;
    BEGIN
       CarryOn ('To stop : ');
    END;

(* ------------------------------------------------------- *)
(* include file of problem-specific procedures here :------ *)
(*$I  D:feigb.Pas  *)
(* ------------------------MAIN------------------------- *)
    BEGIN
       Hello;
       Initialise;
       ComputeAndDisplay;
       Goodbye;
    END.
```

A hint here too for Include-Files: in Include-Files the drive is specified as, for instance, `D:`. That is just an example to show what to do when the Pascal system is on a different drive from the Include-Files.

The Include-File follows; **UtilGraph.Pas**.

```
(*-------------------------UTILITY---------------------- *)
(* BEGIN:  Useful Subroutines *)
    PROCEDURE ReadReal (information : Tstring; VAR value
                      : real);
    BEGIN
       write (information);
       readln (value);
    END;
```

```
    PROCEDURE ReadInteger (information : Tstring; VAR value
                        : integer);
    BEGIN
       write (information);
       readln (value);
    END;

    PROCEDURE ReadString (information : Tstring; VAR value
                        : string);
    BEGIN
       write (information);
       readln (value);
    END;

    PROCEDURE InfoOutput (information : Tstring);
    BEGIN
       writeln (information);
       writeln;
    END;

    PROCEDURE CarryOn (information : Tstring);
    BEGIN
       write (information, ' hit <RETURN>');
       readln
    END;

    PROCEDURE CarryOnIfKey;
    BEGIN
       REPEAT
       UNTIL KeyPress;    (* NOT as for Turbo! *)
    END;

    PROCEDURE Newlines (n : integer);
       VAR
          i : integer;
    BEGIN
       FOR i := 1 TO n DO
          writeln;
    END;
(* END: Useful Subroutines *)
(* -------------------------UTILITY--------------------------*)
```

```
(* ------------------------GRAPHICS-------------------------*)
(* BEGIN:  Graphics Procedures *)

    PROCEDURE SetPoint (xs, ys : integer);
    BEGIN
       (*  Insert machine-specific graphics commands here   *)
      move_to (xs, Yscreen - ys);
      line_to (xs, Yscreen - ys);
    END;

    PROCEDURE SetUniversalPoint (xu, yu: real);
       VAR xs, ys : real;
    BEGIN
       xs := (xu - Left) * Xscreen / (Right - Left);
       ys := (yu - Bottom) * Yscreen / (Top - Bottom);
       SetPoint (round(xs), round(ys));
    END;

    PROCEDURE GoToPoint (xs, ys : integer);
    BEGIN
       (*  Insert machine-specific graphics commands here   *)
      move_to (xs, Yscreen - ys);
    END;

    PROCEDURE DrawLine (xs, ys : integer);
    BEGIN
       (*  Insert machine-specific graphics commands here   *)
      lineto (xs, Yscreen - ys);
    END;

    PROCEDURE GoToUniversalPoint (xu, yu : real);
    BEGIN
       VAR xs, ys : real;
    BEGIN
       xs := (xu - Left) * Xscreen / (Right - Left);
       ys := (yu - Bottom) * Yscreen / (Top - Bottom);
       GotoPoint (round(xs), round(ys));
    END;

    PROCEDURE DrawUniversalLine (xu, yu : real);
       VAR xs, ys : real;
```

```
    BEGIN
       xs := (xu - Left) * Xscreen/(Right - Left);
       ys := (yu - Bottom) * Yscreen / (Top - Bottom);
       DrawLine (round(xs), round(ys));
    END;

    PROCEDURE TextMode;
    BEGIN
       Exit_Gem;
    END;

    PROCEDURE GraphicsMode;
       VAR i : integer
    BEGIN (* machine-specific graphics commands *)
       i := INIT_Gem;
       Clear_Screen;
    END;

    PROCEDURE EnterGraphics;
    BEGIN
       writeln ('To end drawing hit <RETURN> ');
       writeln('---------------- Graphics Mode --------------');
       CarryOn('BEGIN :');
       GraphicsMode;
    END;

    PROCEDURE ExitGraphics
    BEGIN
       (*Machine-specific actions to exit from Graphics Mode*)
       readln;
       Clear_Screen;
       TextMode;
    writeln ('------------------Text Mode ----------------');
    END;

(* END: Graphics Procedures *)
(* ------------------------GRAPHICS--------------------------*)
```

The problem-specific part does not change (see Turbo Pascal, §12.2).

12.6 Apple II Systems

It is certainly possible to take the view that nowadays the Apple IIe is 'getting on a bit'. It is undisputedly slow, but nevertheless everything explained in this book can be achieved on it. The new Apple IIGS has more speed compared with the Apple II, and better colours, so that devotees of the Apple are likely to stay with this range of machines, rather than try to cope with the snags of, e.g., MS-DOS.

Turbo Pascal

Turbo Pascal 3.00A can only run on the Apple II under CP/M. It functions pretty much as in MS-DOS. Specific problems are the graphics routines, which have to be modified for the graphics system of the Apple II. Recently several technical journals (such as *MC* and *c't*) have given hints for this.

TML Pascal/ORCA Pascal

Compared with the Apple IIe the Apple IIGS is a well-tried machine in new clothes. In particular, the colour range is improved. We recommend the use of TML Pascal (see Program 12.6-3) or ORCA Pascal.

UCSD Pascal

Many Pascal devotees know the UCSD system. At the moment there is version 1.3, which runs on the Apple IIe and also the Apple IIGS. Unfortunately until now the UCSD system recognises only 128K of memory, so that extra user memory (on the GS up to 4 MB) can be used only as a RAMdisk.

Basically there are few major changes to our previous reference program. The use of Include-Files or units is possible.[4]

Program 12.6-1 (Reference program for Apple II, UCSD Pascal)

```
PROGRAM EmptyApplicationShell;
                    (* UCSD Pascal on Apple II *)
  USES applestuff, turtlegraphics;
  CONST
    Xscreen = 280;   (* e.g. 280 points in x-direction *)
    Yscreen = 192;   (* e.g. 192 points in y-direction *)
  VAR
    PictureName : string;
    Left, Right, Top, Bottom : real;
    (* include additional global variables here *)
    Feedback : real;
    Visible, Invisible : integer;
```

[4]Examples for Include-Files are given in §12.2, for units in §12.4.

```
    (*$I  Utiltext  *)

    PROCEDURE Hello;
    BEGIN
       TextMode;
       InfoOutput ('Calculation of                          ');
       InfoOutput ('-------------------------');
       Newlines (2);
       CarryOn ('Start :');
       Newlines (2);
    END;

    PROCEDURE Goodbye;
    BEGIN
       CarryOn ('To stop : ');
    END;

(* ------------------------------------------------------------ *)
(* include file of problem-specific procedures here :------ *)

(*$I  feigb.Pas  *)

(* ------------------------MAIN------------------------  *)
    BEGIN
       Hello;
       Initialise;
       ComputeAndDisplay;
       Goodbye;
    END.
```

Most changes occur in the graphics procedures. UCSD Pascal implements a kind of turtle graphics, which does not distinguish between `line` and `move`. Instead the colour of the pen is changed using the procedure `pencolor`. On leaving a procedure the colour should always be set to `pencolor(none)`. Then a program error, leading to unexpected movements on the graphics screen, cannot harm the picture.

Program 12.6-2 (Include-File of useful subroutines)

```
(*-------------------------UTILITY-----------------------  *)
(* BEGIN:  Useful Subroutines *)
    PROCEDURE ReadReal (information : STRING; VAR value
                        : real);
    BEGIN
```

```
   write (information);
   readln (value);
END;

PROCEDURE ReadInteger (information : STRING; VAR value
                  : integer);
BEGIN
   write (information);
   readln (value);
END;

PROCEDURE ReadString (information : STRING; VAR value
                  : string);
BEGIN
   write (information);
   readln (value);
END;

PROCEDURE InfoOutput (information : STRING );
BEGIN
   writeln (information);
   writeln;
END;

PROCEDURE CarryOn (information : STRING );
BEGIN
   write (information, ' hit <RETURN>');
   readln;
END;

PROCEDURE CarryOnIfKey;
BEGIN
   REPEAT
   UNTIL KeyPressed;
END;

PROCEDURE Newlines (n : integer);
   VAR
      i : integer;
BEGIN
   FOR i := 1 TO n DO
```

```
        writeln;
    END;

(* END: Useful Subroutines *)
(* -------------------------UTILITY--------------------------*)
(* ------------------------GRAPHICS--------------------------*)
(* BEGIN:  Graphics Procedures *)

    PROCEDURE SetPoint (xs, ys : integer);
    BEGIN
       (*  Insert machine-specific graphics commands here   *)
       moveto (xs, ys);
       pencolor(white); move(0);
       pencolor(none);
    END;

    PROCEDURE SetUniversalPoint (xu, yu: real);
       VAR xs, ys : real;
    BEGIN
       xs := (xu - Left) * Xscreen / (Right - Left);
       ys := (yu - Bottom) * Yscreen / (Top - Bottom);
       SetPoint (round(xs), round(ys));
    END;

    PROCEDURE GoToPoint (xs, ys : integer);
    BEGIN
       (*  Insert machine-specific graphics commands here   *)
       moveto (xs, ys);
    END;

    PROCEDURE DrawLine (xs, ys : integer);
    BEGIN
       (*  Insert machine-specific graphics commands here   *)
       pencolor(white);
       moveto (xs, ys);
       pencolor(none);
    END;

    PROCEDURE GoToUniversalPoint (xu, yu : real);
    BEGIN
       VAR xs, ys : real;
    BEGIN
```

```
      xs := (xu - Left) * Xscreen / (Right - Left);
      ys := (yu - Bottom) * Yscreen / (Top - Bottom);
      GotoPoint (round(xs), round(ys));
   END;

   PROCEDURE DrawUniversalLine (xu, yu : real);
      VAR xs, ys : real;
   BEGIN
      xs := (xu - Left) * Xscreen/(Right - Left);
      ys := (yu- Bottom) * Yscreen / (Top - Bottom);
      DrawLine (round(xs), round(ys));
   END;

(* PROCEDURE TextMode is defined in turtle graphics          *)
(* PROCEDURE GraphMode is defined in turtle graphics         *)

   PROCEDURE EnterGraphics;
   BEGIN
      writeln ('To end drawing hit <RETURN> ');
      Page(Output);
      GotoXY(0,0);
      writeln('-------------- Graphics Mode --------------');
      CarryOn('BEGIN :');
      InitTurtle;
   END;

   PROCEDURE ExitGraphics
   BEGIN
      (*Machine-specific actions to exit from Graphics Mode*)
      readln;
      TextMode;
      Page(oputput);
      GotoXY(0,0);
   END;

(* END: Graphics Procedures *)
(* ------------------------GRAPHICS--------------------------*)
```

The problem-specific part does not change (see Turbo Pascal, §12.2).

As we have already explained at the start of this chapter, we recommend you to use TML Pascal on the Apple II. There are two versions:

- TML Pascal (APW)
- TML Pascal (Multi-Window)

The first version is best considered as a version for the software developer who wishes to use Apple Programmer's Workshop, the program development environment from Apple Computer Inc. The second version is easier to use. The multi-window version has a mouse-controlled editor and is similar to TML Pascal on the Macintosh.

Below we once again give our reference program, this time for TML Pascal on the Apple IIGS. For a change, here we draw a Gingerbread Man. We confine ourselves to the most important procedures; you have already seen in previous sections that the surrounding program scarcely changes.

Program 12.6-3 (TML Pascal for Apple IIGS)

```
PROGRAM EmptyApplicationShell(input, output);
   USES
      ConsoleIO,   (* Library for plain vanilla I/O     *)
      QDIntf;      (* Library for Quick-draw calls      *)
   CONST
      Xscreen = 640.0;
(* SuperHIRES screen 640x400  points*)
      Yscreen = 200;       (* NOTE: real constants!     *)
   VAR
      Left, Right, Top, Bottom : real;
      I, MaximalIteration, Bound : integer;
      R: Rect;

PROCEDURE ReadReal (s : string; VAR number : real);
BEGIN
   writeln;
   write (s);
   readln (number);
END;

PROCEDURE ReadInteger (s : string; VAR number : integer);
BEGIN
   writeln;
   write (s);
   readln (number);
END;

PROCEDURE SetPoint (x, y, colour : integer);
BEGIN
   SetDithColor(colour);   (* 16 possible colours *)
```

```
   Moveto (x, y);
   Line (0, 0);
END;

PROCEDURE sqr(x :real) : real;
BEGIN
   sqr (x) := x * x;
END;

PROCEDURE Mapping;
   VAR
      xRange, yRange : intger;
      x, y, x0, y0, deltaxPerPixel, deltayPerPixel : real;

   FUNCTION MandelbrotComputeAndTest (cReal, cImaginary
                  : real)
             : integer;
   VAR
      iterationNo : integer;
      x, y, xSq, ySq, distanceSq : real;
      finished: boolean;
   BEGIN
(*    StartVariableInitialisation *)
      finished := false;
      iterationNo := 0;
      x := x0;
      y := y0;
      xSq := sqr(x);
      ySq := sqr(y);
      distanceSq := xSq + ySq;
(*  StartVariableInitialisation *)
   REPEAT
(*  Compute *)
      iterationNo := iterationNo + 1;
      y := x*y;
      y := y+y-cImaginary;
      x := xSq - ySq -cReal;
      xSq := sqr(x); ySq := sqr(y);
      distanceSq := xSq + ySq;
(*  Compute *)
(*  Test *)
```

```
      finished := distanceSq > 100.0;
(*  Test  *)
   UNTIL (iterationNo - MaximalIteration) OR finished;
(*  distinguish  *)
      IF iterationNo = MaximalIteration THEN
         MandelbrotComputeAndTest : = 15
         ELSE BEGIN
            IF iterationNo > Bound THEN
               MandelbrotComputeAndTest : = 15
            ELSE MandelbrotComputeAndTest : =
                  iterationNo MOD 14;

            END;
(*  distinguish  *)
      END;
   BEGIN (* Mapping *)
      SetPenSize(1, 1);
      deltaxPerPixel := (Right - Left) / Xscreen;
      deltayPerPixel := (Top - Bottom) / Yscreen;
      x0 := 0.0; y0 := 0.0;
      y := Bottom;
      FOR yRange := 0 TO trunc (yScreen) DO BEGIN
         x:= Left;
         FOR xRange := 0 TO trunc (xScreen) DO BEGIN
            SetPoint (xRange, yRange,
               MandelbrotComputeAndTest (x, y));
            x := x + deltaxPerPixel;
            IF KeyPressed THEN exit  (Mapping);
         END;
         y := y + deltayPerPixel;
      END;
   END;
   PROCEDURE Initialise;
   BEGIN
      ReadReal ('Left                > ', Left);
      ReadReal ('Right               > ', Right);
      ReadReal ('Top                 > ', Top);
      ReadReal ('Bottom              > ', Bottom);
      ReadInteger ('Max. Iteration   > ',MaximalIteration);
      ReadInteger ('Bound            > ', Bound);
   END;
```

```
BEGIN (* main *)
   Initialise;
   Mapping;
   REPEAT UNTIL KeyPressed;
END.
```

Observe that:

- We have collected together all local procedures inside the functional procedure `MandelbrotComputeAndTest`.
- The graphics procedure calls are similar to those for the Macintosh. Therefore it is not necessary for us to list all of them. They are identical.
- The Library `ConsoleIO` contains the following useful subroutines:

```
Function KeyPressed : boolean
Procedure EraseScreen;
Procedure SetDithColor (color : integer);.
```

The value for the variable `color` must lie between 0 and 14, corresponding to the following colours:

Black (0), Dark Grey (1), Brown (2), Purple (3), Blue (4), Dark Green (5), Orange (6), Red (7), Flesh (8), Yellow (9), Green (10), Light Blue (11), Lilac (12), Light Grey (13), White (14).

The way to use colour is easy to see in the program listing. In the algorithmic part `distinguish` (inside `MandelbrotComputeAndTest`) the value of `IterationNo` determines the colour value.

12.7 'Kermit Here' - Communications

In the course of time we become ever more specialised. Now the discussion will be extremely special. The difficult probem of computer-computer communication is the final topic. What we now discuss will be most likely to appeal to 'freaks' - assuming they do not already know it.

The problem is well known: how can we get data and text files from computer X to computer Y? We have already given the main answer in §11.6, 'A Picture Takes a Trip'. But the gap between direct computer-computer connection and e-mail is vast. To bridge it you need to read manuals, think about cables and connectors - all of which takes time. We cannot make the process effortless, but we can give a little help.

First you must buy or assemble a cable, to connect your computer to a modem or another computer via the V24 interface. This hardware problem is the most disagreeable part, because the computer will not do anything if you use the wrong pin-connections. The best solution is the help of a knowledgeable friend, or the purchase of a ready-made cable. Then the next problem raises its head: software. We recommend 'Kermit'. This communications package exists for virtually every computer in the world: try to get this 'public domain' software from usergroups or friends. Of course you can also get other

software, provided both computers use the same communications protocol, for instance XModem. Kermit is the most widely available and its documentation is clearly written.

Without going too much into fine detail, we will first describe how file transfer works[5] between an IBM-PC running under MS-DOS and a UNIX system, such as a VAX or SUN.

We assume that your IBM-compatible has two drives or a hard disk, and that the Kermit program is on drive `b:` or `c:`. Then you must enter into the following dialogue with the host computer, for example the VAX.

(... means that the screen printout is not given in full.)

Kermit Dialogue: MS-DOS <--> UNIX

```
b>kermit
IBM-PC Kermit-MS V2.26
Type ? for help
Kermit-MS>?
BYE    CLOSE    CONNECT    DEFINE
...
STATUS      TAKE

Kermit-MS>status
Heath-19 emuilation ON          Local Echo Off
...
Communication port: 1           Debug Mode Off
Kermit-MS>set baud 9600
...
```

Now you must set the other parameters such as XON/XOFF, Parity, Stop-bits, 8-bit, etc. as applicable. Once both computers are similarly configured, the connection can be made.

```
Kermit-MS>connect
[connecting to host, type control-] C to return to PC]
```

Hit <RETURN> and the UNIX system reports...

```
login : kalle
password:
...
```

[5]Under Kermit, the procedure is much the same for other machines.

Assume that you want to transfer a file named **rlecdat**, containing your picture data, from the VAX to the PC. Then you do this:

```
VAX>kermit s rlecdat
```

The UNIX computer waits until the PC is ready for the transfer. You must return to the Kermit environment of your running MS-DOS program. Type:

```
<CTRL-]><?>
```

A command line appears. Give it the letter `c`, as an abbreviation for close connection.

```
COMMAND>c
Kermit-MSreceive picturedata
```

The data `rlecdat` (on a UNIX system) become `picturedata` on the MS-DOS system. The transfer then starts, and its successful completion is reported.

```
Kermit-MS>dir
```

The transferred file is written into the directory of your MS-DOS machine. If you already have a file of the same name on your PC, things do not work out very well.

The converse procedure is also simple:

```
VAX>Kermit r
```

You return to the Kermit environment and give the command:

```
Kermit-MS>send example.p
```

The transfer then begins in the opposite direction...

Everything works much the same on other computers, including the Macintosh. But here there is an elegant variant. In the USA there are two programs named MacPut and MacGet, which can transfer Macintosh text and binary files under the MacTerminal 1.1 protocol to and from a UNIX system.

Get hold of the C source files, transfer them to your VAX, compile them into the program - and you are free of all Kermit problems.

```
login: kalle
password:
4.3 BSD UNIX °4: Thu Feb 19 16:00:24 MET 1987
Mon Jul 27 18:28:20 MET DST 1987
SUN>macget msdosref.pas -u
```

```
SUN>ls -l
total 777
-rwxrwxr-x    1 kalle     16384 Jan 16 1987 macget
-rw-r--r--    1 kermit    9193 Jan 16 1987 macget.c.source
-rwxrwxr-x    1 kalle     19456 Jan 16 1987 macget
-rw-r--r--    1 kermit    9577 Jan 16 1987 macget.c.source
-rw-rw-r--    1 kalle     5584 Jul 27 18:33 msdosref.pas
```