Praktikumsprotokoll – Mikrorechentechnik I
Versuch „C-Programmierung“

Gruppe 63:
Fabian Kurz, Alexander Eder
Stephan Stiebitz, Phillip Burker

16. Dezember 2004

Inhaltsverzeichnis

1 Aufgabenstellung

Zur numerischen Untersuchung von Fraktalen, insbesondere der *Julia-Menge* und der *Mandelbrotmenge*, soll ein Rechenprogramm entworfen und implementiert werden. Für die Iterationsformel

$$z_{i+1} = c + {z_i}^2, \qquad z_i,\ z_{i+1},\ c \in \mathbb{C} \tag{1}$$

mit $z_i = x_i + jy_i$, und $c = a + jb$ soll geprüft werden, nach wie vielen Iterationen i die komplexe Zahl z_i ein vorgegebenes Gebiet G (z.B. einen Kreis) verläßt. Jeder Zahl i, für die z_i das vorgegebene Gebiet verläßt, soll eine Farbe zugeordnet werden. Wenn sich z_i für $i > i_{max}$ immer noch in diesem Gebiet befindet, wird die Iteration abgebrochen und diesem Punkt die Farbe schwarz zugeordnet.
Führt man nun diese Iteration für jeden Punkt auf der komplexen Zahlenebene aus, entstehen fraktale geometrische Figuren, die man bei Variation von

- c als Mandelbrotmenge (nach dem französischen Mathematiker Benoît B. Mandelbrot, diese Menge wird wegen ihres Aussehens auch „Apfelmännchen“ genannt)
- z_i als Julia-Menge (nach dem französischen Mathematiker Gaston Maurice Julia)

bezeichnet.

1.1 Die Mandelbrotmenge

Da bei der Mandelbrotmenge c variiert und die Anfangsbedingung $z_0 = 0$ lautet, ergibt sich (bei ausreichend hohem i_{max} und einem hinreichend kleinem Gebiet G) das typische Erscheinungsbild wie in Abb. 1 dargestellt.

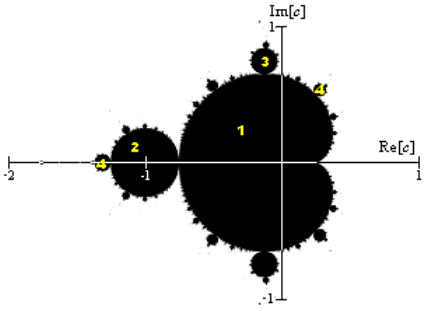

Abbildung 1: Typisches Erscheinungsbild der Mandelbrotmenge in der komplexen Ebene (aus Wikipedia)

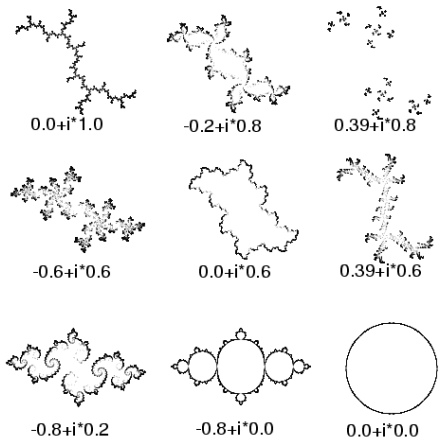

Abbildung 2: Ausgewählte Erscheinungsbilder der Julia-Menge mit entsprechenden Werten von c (aus Wikipedia)

1.2 Die Julia-Menge

Die sich bei der Julia-Menge ergebenen Fraktale kommen in wesentlich vielfältigeren Formen daher, da hier der gewählte Wert für c einen entscheidenen Einfluß auf das Iterationsverhalten der Reihe hat. In Abb. 2 sind exemplarisch einige Formen mit den dazugehörigen komplexen Werten von c dargestellt, im Gegensatz zur Darstellung der Mandelbrotmenge in Abb. 1 wird hier nur der Rand der Menge dargestellt.

2 Entwurf und Implementierung

2.1 Mathematische Grundlagen

Die Berechnung der Iterationsfolgewerte nach (1) erfolgt nach den bekannten Rechenregeln für komplexe Zahlen:

$$\begin{aligned} z_{i+1} &= c + {z_i}^2 = a + ib + (x_i + iy_i)^2 = a + ib + {x_i}^2 - {y_i}^2 \\ &= (a + {x_i}^2 - {y_i}^2) + i(b + 2x_i y_i) \end{aligned}$$

Da die Programmiersprache C keinen Datentyp für komplexe Zahlen kennt und somit auch nicht mit ihnen rechnen kann, wurde ein eigener Datentyp eingeführt, welcher in 2.2.1 näher beschrieben wird. Die Berechnung der Iterationswerte im Programm erfolgt getrennt für den Real- und Imaginärteil:

$$\mathrm{Re}\,(z_{i+1}) = a + {x_i}^2 - {y_i}^2 \tag{2}$$

$$\mathrm{Im}\,(z_{i+1}) = i(b + 2x_i y_i) \tag{3}$$

Um zu ermitteln ob die Reihe sich noch innerhalb des Gebietes G (Kreis mit Radius r) befindet, wird nach dem Satz von Pythagoras die Entfernung vom Anfangspunkt $c = a + ib$ bei der Mandelbrotmenge bzw. $z_0 = x_0 + iy_0$ bei der Julia-Menge ermittelt.

$$d = \sqrt{(a - x_i)^2 + (b - y_i)^2} \qquad \text{bzw.} \qquad d = \sqrt{(x_0 - x_i)^2 + (y_0 - y_i)^2} \tag{4}$$

2.2 Datenstrukturen

2.2.1 Datentyp für komplexe Zahlen: `tComplex`

Der Datentyp `tComplex` ist eine Struktur (`struct`) aus zwei `double`-Werten zur Aufnahme des Real- und Imaginärteils einer komplexen Zahl.

Typendefinition:

```
struct tComplex {          // Struktur zur Beschreibung komplexer Zahlen
  double r;                // Realteil
  double i;                // Imaginärteil
};
```

2.2.2 Datentyp zum Speichern der Fraktalparameter: `tParam`

Die benötigten Parameter zur Berechnung der Fraktale werden in der Struktur `tParam` gespeichert. Sie enthält das Quadrat des Radius `R` des Iterationsgebietes G vom Typ `int` (durch Angabe des Quadrates erspart man das Ziehen der Wurzel bei der Berechnung der Entfernung nach (4)), die maximale Iterationsanzahl `imax` vom Typ `int`, den Fraktaltyp `ftype` vom Typ `tFracttyp` (2.2.3) sowie die Grenzwerte `xmin`, `xmax`, `ymin`, `ymax` des Gebietes der komplexen Ebene auf dem die Berechnungen durchgeführt werden sollen, jeweils vom Typ `float`.

Typendefinition:

```
struct tParam {            // Parameterstruktur
  int         R;           // Quadrat des Radius des Gebietes G
  int         imax;        // maximale Anzahl der Iterationen
  tFracttyp   ftype;       // Fraktaltyp, Apfel/Julia
```

```
  double      xmin;       // Minmaler Realteil des Analysegebietes
  double      xmax;       // Maximaler Realteil des Analysegebietes
  double      ymin;       // Minmaler Im-Teil des Analysegebietes
  double      ymax;       // Maximaler Im-Teil des Analysegebietes
};
```

2.2.3 Datentyp zur Angabe des Fraktaltyps: `tFracttyp`

Die Typendefinition für `tFracttyp` ist eine Enumeration der möglichen Fraktaltypen, also der Julia-Menge und der Mandelbrotmenge (Apfelmännchen). Ein separater Typ ist für diesen Zweck eigentlich nicht nötig (man könnte beispielsweise einen Integerwert mit 1 für die Julia-Menge, 0 für das Apfelmännchen belegen), jedoch gewinnt man dadurch im weiteren Verlauf des Programms an Übersichtlichkeit.

Typendefinition:

```
typedef enum {  Apfel,       // Apfelmännchen
                Julia        // Juliamenge
             } tFracttyp;
```

2.2.4 Datentyp für die Farbpalette: `tColor`

Der Typ `tColor` ist eine Enumeration, in welcher den ersten 16 Farbwerten der VGA Farbpalette entsprechende Bezeichnungen `black`, `blue`, ... zugeordnet werden.

Typendefinition:

```
typedef enum {  black,         /*   0 schwarz         */
                blue,          /*   1 blau            */
                green,         /*   2 grün            */
                cyan,          /*   3 cyan            */
                red,           /*   4 rot             */
                magenta,       /*   5 violett         */
                brown,         /*   6 braun           */
                white,         /*   7 weiß            */
                gray,          /*   8 grau            */
                lightblue,     /*   9 hellblau        */
                lightgreen,    /*  10 hellgrün        */
                lightcyan,     /*  11 hellcyan        */
                lightred,      /*  12 hellrot         */
                lightmagenta,  /*  13 hellviolett     */
                lightyellow,   /*  14 gelb            */
                brightwhite    /*  15 intensivweiß    */
             } tColor;
```

2.3 Programmstruktur

Das Programm wurde in vier Programmmodule geteilt, `V3_MAIN`, `V3_DIALO`, `V3_GRAPH`, `V3_FRAKT`. Diese bestehen jeweils aus einer Headerdatei (`*.h`) als Schnittstelle zum Rest des Programms (außer der Hauptdatei, `V3_MAIN`), welche Definitionen von Datentypen und die Funktionsköpfe enthält und einer Implementierungsdatei (`*.c`), welche den Quelltext der zu implementierenden Funktionen enthält.
Die Trennung der einzenen Programmteile dient dazu den Code übersichtlich zu halten, bestimmte Teile einfacher wiederverwendbar zu machen und die Wartbarkeit des Programms zu erhöhen.
Im Folgenden werden die vier Programmmodule im Detail beschrieben.

2.3.1 Das Hauptmodul: `V3_MAIN`

Im Hauptmodul befindet sich nur die `main`-Funktion mit den eingebundenen Programmmodulen. Hier wird eine Variable vom Typ `tParam` vereinbart und mit bestimmten Werten initialisiert, z_0 und c werden vereinbart und initialisiert und danach eine Schleife ausgeführt, in der der eigentliche Programmablauf stattfindet:

- Abfrage der Fraktal-Parameter (`V3_DIALO`)
- Initialisierung der Grafik (`V3_GRAPH`)
- Berechnung und Ausgabe des Fraktals (`V3_FRAKT`)
- Bildschirm löschen
- Abbruchbedingung abfragen

2.3.2 Parameter-Eingabedialog: `V3_DIALO`

Zum Verändern der am Anfang des Hauptmoduls festgelegten Parameter zur Berechnung des Fraktals wird die Funktion `ParamDialog` im Modul `V3_DIALO` aufgerufen. Der Funktion werden Pointer auf die Struktur zur Speicherung der Fraktalparameter vom Typ `tParam` und auf die komplexe Zahl `c` vom Typ `tComplex` übergeben. Dessen Werte werden in dieser Funktion nacheinander abgefragt, auf Sinnfreiheit überprüft und bei nichtvorhandensein derselben sogar gespeichert.
Die Verwendung von Pointern ist hier die elegantest Methode zur Veränderung der Parameterwerte. Man hätte sie auch als `extern` Variablen definieren können und dann in `ParamDialog` ändern können, jedoch fördert dies nicht gerade die Übersichtlichkeit und Wartbarkeit des Codes.

2.3.3 Grafik-Routinen: `V3_GRAPH`

Die Initialisierung der graphischen Schnittstelle besteht im Wesentlichen aus schnittstellenspezifischen Befehlen, die nichts mit ANSI C zu tun haben.
Der programmtechnisch interessante Teil dieses Moduls ist die Bildung der Parameter zur Umrechnung der Eingabewerte zwischen `ymin` und `ymax` bzw. `xmin` und `xmax` auf die entsprechenden Punkte des Bildschirms. Dabei ist sowohl das gegebene Intervall in x- und y-Richtung auf die Abmessungen des Bildschirms zu strecken/stauchen als auch die y-Koordinate „umzudrehen“, da der Koordinatenursprung (0/0) des Bildschirms in der oberen linken Ecke liegt und die positive y-Achse nach unten zeigt, im Gegensatz zum kartesischen Koordinatensystem bzw. der Gaußschen Zahlenebene, welche es abzubilden gilt.
Letztendlich enthält dieses Modul die Funktion `SetPoint`, der die Variablen `x` und `y` vom Typ `double` und die Farbe `color` vom Typ `tColor`

2.3.4 Fraktalberechnung: `V3_FRAKT`

Die eigentliche Berechnung der Iterationsfolge nach (1) bis (4) findet im Programmodul `V3_FRAKT` statt.
Die Funktion `GetItera` erhält als Eingabewerte zwei komplexe Zahlen z und c sowie eine Parameterstruktur vom Typ `tParam` und gibt als Rückgabewert die Anzahl der Iterationen bis zum Verlassen des in `tParam` festgelegten Gebietes zurück. Dies wird durch eine `while`-Schleife realisiert, die solange ausgeführt wird, wie der Abstand des aktuellen Wertes z_{i+1} noch innerhalb des Radius $r = \sqrt{R}$ liegt *und* die aktuelle Iterationsanzahl noch unter der maximalen Iterationsanzahl `imax` aus der Parameterstruktur liegt.
Die Funktion `GetColorValue` ermittelt aus dem Verhältnis der Iterationen zur maximalen Anzahl von Iterationen einen Farbwert. Für die maximale Anzahl der Iterationen wird die Farbe schwarz festgelegt, die restlichen Werte werden in der Folge der Standard-VGA-Palette belegt. Da im konkreten Fall nur 16 Farben darstellbar sind, wird hier mit Modulo 16 gearbeitet, so daß sich die Farben nachdem die Palette durchlaufen ist wiederholen.
Die genaue Farbe ist im Prinzip auch unwichtig. Wesentlich ist, daß in der ausgegebene Grafik erkennbar ist, daß es in der Fraktalgrafik bestimmte Zonen gibt, in denen das Iterationsverhalten der Reihe gleich ist.
Letztendlich enthält das Programmmodul `V3_FRAKT` noch die Funktion zur eigentlichen Erzeugung der Fraktalgrafik, `Fraktal`.
Als Parameter werden zwei komplexe Zahlen c und z und eine Parameterstruktur vom Typ `tParam` übergeben. In der Funktion wird nun für alle Punkte des in der Parameterstruktur angegebenen Gebietes der komplexen Ebene das Iterationsverhalten untersucht und ein entsprechender Farbpunkt auf dem Display ausgegeben. Dazu kommen zwei `for`-Schleifen zum Einsatz, die den Bereich in x-Richtung in 640, in y-Richtung in 480 Schritten durchlaufen. Die Schrittweite ergibt sich folglich aus

$$s_x = \frac{x_{\max} - x_{\min}}{640} \qquad \text{bzw.} \qquad s_y = \frac{y_{\max} - y_{\min}}{480}. \tag{5}$$

Damit wird erreicht, daß der verfügbare Bildschirmbereich Pixel für Pixel ausgefüllt wird; es sich also keine Lücken oder Überlappungen (je nach Wertebereich von $x_{\min}/x_{\max}$ und $y_{\min}/y_{\max}$) ergeben.
Innerhalb dieser Schleifen wird für jeden Punkt die Anzahl der Iterationen berechnet, danach die passende Farbe dazu berechnet und letztendlich der Farbpunkt auf dem Display dargestellt. Bevor die Berechnung der Iterationsanzahl erfolgen kann, muss jedoch noch abhängig davon, ob eine Julia-Menge oder eine Mandelbrotmenge erzeugt werden soll, der variable Parameter bestimmt werden. Bei der Julia-Menge wird z variiert, bei der Mandelbrotmenge c.

2.4 Programmstruktur und Ablauf

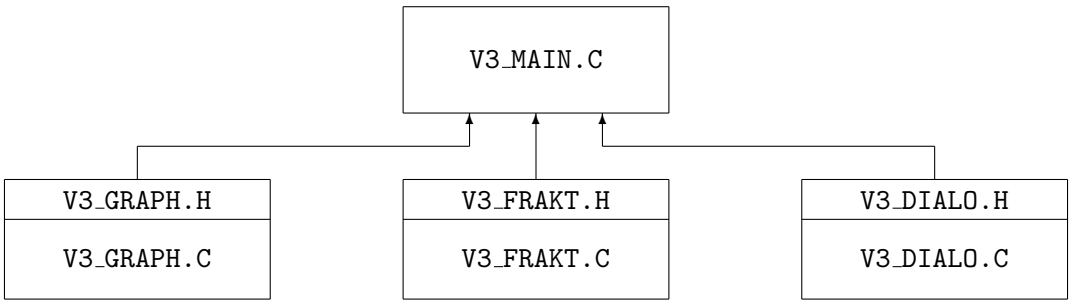

Abbildung 3: Strukturdiagramm

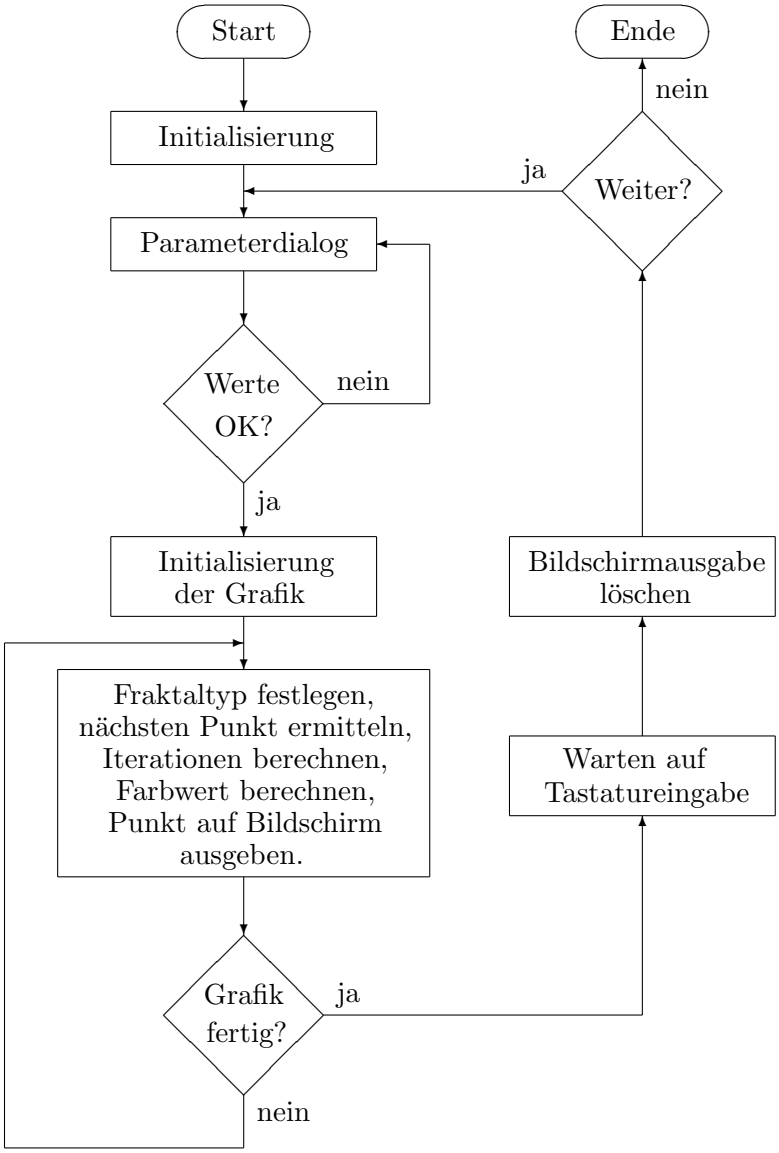

Abbildung 4: Ablaufdiagramm

Die Programmstruktur ist in Abb. 3, der Übersichtlichkeit halber ohne die eingebundenen Standardbibliotheken, dargestellt. Abb. 4 zeigt das Ablaufdiagramm des Programms als Flußdiagramm. Auch hier wurde sich auf das Wesentliche beschränkt um das Diagramm lesbar zu halten. Manchmal wurden mehrere Schritte, auch wenn sie im Programm in unterschiedlichen Funktionen implementiert wurden, in einzelne Boxen zusammengefasst.

3 Probleme

Die Praktikumsaufgabe bereitete keine größeren Schwierigkeiten und konnte dank der guten Vorbereitung durch die Vorlesung sowie die bereitgestellten Unterlagen problemlos gelöst werden. Als zusätzliche Referenz wurde das Buch „Programmieren in C“ von Kernigham und Ritchie benutzt.
Im Praktikum fiel auf, daß die Funktion zum Anfragen neuer Fraktalparameter (`ParamDialog` unabhängig vom Fraktaltyp immer die Werte für eine komplexe Zahl c abfragt, obwohl diese nur bei der Julia-Menge benötigt wird (bei der Mandelbrotmenge wird c variiert).

A Anhang V3_MAIN

A.1 Implementierungsdatei V3_MAIN.C

```
/* Datei v3_main.c                          */
/* Fabian Kurz, 13.12.2004                  */

#include "v3_frakt.h"
#include "v3_graph.h"
#include "v3_dialo.h"
#include <conio.h>       /* getch */
#include <stdio.h>       /* printf */

void main ()
{
  /*--- Variablendeklaration ------------------------------------------------*/
  int a;
  double d;
  int abbruch;

  /*--- Initialwerte für Fraktalberechnung ----------------------------------*/

  tParam p;                  // Variable Parameter von tParam
  parameter.R     =  4;      // initialisieren...
  parameter.imax  = 75;
  parameter.ftype = Julia;
  parameter.xmin  = -2;
  parameter.xmax  =  2;
  parameter.ymin  = -2;
  parameter.ymax  =  2;


  /*--- Initialwerte zur Fraktalanalyse -------------------------------------*/

  tComplex c;                // komplexe Zahlen c und z
  tComplex z;

  c.r = 0.4;                 // Initialisierung derselben
  c.i = 0.4;
  z.r = 0;
  z.i = 0;

  /*--- Parameterdialog und gegf. Fraktalberechnung -------------------------*/

do {
ParamDialog(&p, &c);                  // Parameterdialog

InitGraph(p.xmin,p.xmax,p.ymin,p.ymax);
 Fraktal(c,z,p);                      // Fraktalberechnung
 a = getch();                         // Auf Tastatureigabe warten bevor:
CloseGraph();                         // der Bildschirm gelöscht wird
```

```
do {                                    // Schleife für Abbruchbedingung
    printf("Weiter? (j/n)\n");
    a = getch();                        // Einlesen von a
   } while (!(a == 'j' || a == 'n'));   // nochmal, wenn a nicht j oder n ist

  abbruch = 0;                          // abbruchbedingung auf 0 = falsch
  if (a == 'n') abbruch = 1;            // falls nicht weitergemacht werden
                                        // soll, abbruch auf 1 setzten
} while (abbruch == 0);                 // bei abbruch = 1 ist hier schluß.

} /* v3_main.c */
```

B Anhang V3_DIALO

B.1 Headerdatei V3_DIALO.H

```
/* Datei: v3_dialo.h              */
/* Fabian Kurz, 13.12.2004        */

#ifndef __dialog
#define __dialog

#include "v3_frakt.h" /* tParam */


/*--- Modifikation und Test Analyseparameter -------------------------------*/

int ParamDialog(tParam *p, tComplex *c);

#endif

/* v3_dialo.h */
```

B.2 Implementierungsdatei V3_DIALO.C

```
/* Datei: v3_dialo.c                */
/* Fabian Kurz, 13. Dezember 2004   */

#include "v3_dialo.h"
#include <stdio.h>      /* printf  */
#include <stdlib.h>     /* strtod  */
#include <conio.h>      /* getch   */


/*--- Double-Zahl von der Tastatur lesen -----------------------------------*/

void InputDouble (double *value)
{
  char *endptr;
  char s[80];           /* Charakter-String                          */
                        /* ab [0] Inhalt, wenn <Enter> dann s[0]=0x0 */
  gets(s);              /* String von Tastatur lesen                 */
```

```
  if (s[0]!=0) *value=strtod (s, &endptr);     /* konvertieren       */
                        /* bei Eingabe von <Enter> bleibt der        */
                        /* bisherige Wert "value" unver~Dndert !      */
};


/*--- Int-Zahl von der Tastatur lesen ------------------------------------*/

void InputInt (int *value)
{
  char s[80];           /* Charakter-String                          */
                        /* ab [0] Inhalt, wenn <Enter> dann s[0]=0x0 */
  gets(s);              /* String von Tastatur lesen                 */
  if (s[0]!=0) *value=atoi (s);
                        /* bei Eingabe von <Enter> bleibt der        */
                        /* bisherige Wert "value" unverändert !      */
};

/*--- Parameterdialog und -test ------------------------------------------*/
/* Die benötigten Werte der Struktur p und eine komplexe Zahl werden ein    */
/* gelesen und auf Sinnfreiheit überprüft                                   */

int ParamDialog(tParam *p, tComplex *c)     // Es werden Pointer auf die Struk-
                                            // tur und die komplexe Zahl über-
{                                           // geben..
char s[80];
char *s2;
tParam pa;                  // Hier werden die Originalwerte
tComplex cm;                // zwischengespeichert, bei Korrekturen benötigt
int j=0;
pa = *p;                    // Zwischenspeichern / Sichern
cm = *c;

do {                        // Anfang der Überprüfungsschleife
  printf("Bitte geben Sie das r^2 des Gebietes an: (%d)\n", p->R);
  InputInt(&pa.R);          // Wert einlesen und speichern
} while (pa.R < 1);         // Überprüfe ob r < 1, wenn ja wiederholen
p->R = pa.R;                // Erfolgreich überprüft -> Wert übernehmen

do {
printf("Bitte geben Sie die maximalen Iterationsschritte an: (%d)\n", p->imax);
  InputInt(&pa.imax);
} while (pa.imax < 1);
p->imax = pa.imax;          // Überprüfe ob imax < 1, wenn ja wiederholen

do {
  printf("Bitte geben Sie x_min (%f) ein\n", p->xmin);
  InputDouble(&pa.xmin);
} while (pa.xmin > 100 || pa.xmin < -100);    // prüfe ob xmin zwischen -100
p->xmin = pa.xmin;                            // und 100, wenn nicht wieder
                                              // holen
```

```
do {
  printf("Bitte geben Sie x_max (%f) ein\n", p->xmax);
  InputDouble(&pa.xmax);
} while (pa.xmax <= pa.xmin || pa.xmax > 100); // prüfe ob xmax > xmin und
p->xmax = pa.xmax;                             // xmax < 100

do {
  printf("Bitte geben Sie y_min (%f) ein\n", p->ymin);
  InputDouble(&pa.ymin);                         // dito, siehe xmin
} while (pa.ymin > 100 || pa.ymin < -100);
p->ymin = pa.ymin;

do {
  printf("Bitte geben Sie y_max: (%f) ein\n", p->ymax);
  InputDouble(&pa.ymax);                         // dito, siehe xmax
} while (pa.ymax <= pa.ymin || pa.ymax > 100);
p->ymax = pa.ymax;

do {
  j = (p->ftype == Julia) ? 1 : 0;       // j ist 1 bei Juliamenge, 0 sonst
  printf("Bitte waehlen Sie: Juliamenge (1) oder Apfelmaennchen (0)?
                         (%s)\n", s2=(p->ftype == Julia) ? "Julia" : "Apfel");
 gets(s);                                // Einlesen des neuen Wertes (0/1)
 if (s[0]!=0) j=atoi (s);                // Setzten von j auf neuen Wert
} while (!(j==1 || j==0));               // Überprüfung ob 1 oder 0
p->ftype = (j == 1) ? Julia : Apfel;     // Speichern des neuen ftypes

printf("Bitte geben Sie Re(c) an (%f)\n", cm.r);
InputDouble(&cm.r);                      // Hier keine Überprüfung, strtod
c->r = cm.r;                             // regelt das; alle x \in R erlaubt


printf("Bitte geben Sie Im(c) an (%f)\n", cm.i);
InputDouble(&cm.i);                      // dito
c->i = cm.i;

return 1;                                // 1 zurückgeben, da auf jeden Fall
                                         // erfolgreich, wenn bis hier gekommen
}

/* v3_dialo.c */
```

C Anhang V3_GRAPH

C.1 Headerdatei V3_GRAPH.H

```
/* Datei: v3_graph.h            */
/* Fabian Kurz, 13.12.2004      */

#ifndef __graph
#define __graph
```

```
#include <graphics.h>

/*----- Typdefinitionen ------------------------------------------------*/

typedef enum {  black,         /*   0 schwarz         */
                blue,          /*   1 blau            */
                green,         /*   2 grün            */
                cyan,          /*   3 cyan            */
                red,           /*   4 rot             */
                magenta,       /*   5 violett         */
                brown,         /*   6 braun           */
                white,         /*   7 weiß            */
                gray,          /*   8 grau            */
                lightblue,     /*   9 hellblau        */
                lightgreen,    /*  10 hellgrün        */
                lightcyan,     /*  11 hellcyan        */
                lightred,      /*  12 hellrot         */
                lightmagenta,  /*  13 hellviolett     */
                lightyellow,   /*  14 gelb            */
                brightwhite    /*  15 intensivweiß    */
             } tColor;

/*--- Grafik initialisieren und schließen ----------------------------------*/

void InitGraph (double Xmin,
                double Xmax,
                double Ymin,
                double Ymax);

void CloseGraph (void);


/*--- Grafikpunkt setzen ---------------------------------------------------*/

void SetPoint (double x, double y, tColor color);

#endif

/* v3_graph.h */
```

C.2 Implementierungsdatei V3_GRAPH.C

```
/* Datei: v3_graph.c             */
/* Fabian Kurz, 13. Dec 2004     */

#include <stdio.h>     /* printf */
#include <conio.h>     /* getch  */
#include <stdlib.h>    /* exit   */
#include "v3_graph.h"

/*----- Definitionen ---------------------------------------------------*/
```

```
static int      ixmax, iymax;   /* max. Integerwerte    */
static double   xmax,  ymax;    /* max. Realwerte       */
static double   xmin,  ymin;
static double   ax,bx,ay,by;    /* Umrechungsparameter  */

/*--- Grafik initialisieren ------------------------------------------------*/

void InitGraph (double Xmin,        /* Maximale Realwerte */
                double Xmax,
                double Ymin,
                double Ymax)
{
  int gdriver;
  int gmode;
  int errorcode;

  gdriver   = DETECT;                           /* request autodetection  */
  errorcode = registerbgidriver (EGAVGA_driver); /* register a driver      */
  if (errorcode < 0) {                          /* report any error       */
    printf ("Graphics error: %s\n", grapherrormsg(errorcode));
    printf ("Press any key to halt..\n");
    getch  ();
    exit   (1);                                 /* terminate w/error code */
  }

  initgraph (&gdriver, &gmode, "");             /* initialize graph mode  */
  errorcode = graphresult();                    /* read result of init    */
  if (errorcode != grOk) {                      /* an error occurred      */
    printf ("Graphics error: %s\n", grapherrormsg(errorcode));
    printf ("Press any key to halt...\n");
    getch  ();
    exit   (1);                                 /* return with error code */
  }
  ixmax = getmaxx();                            /* max. Abmaße Bildschirm */
  iymax = getmaxy();
  xmin  = Xmin; xmax  = Xmax;                   /* Zuweisung Init-Param.  */
  ymin  = Ymin; ymax  = Ymax;
  ax    = ixmax/(xmax-xmin);                    /* ix = ax*x + bx         */
  bx    =-ixmax*xmin/(xmax-xmin);
  ay    =-iymax/(ymax-ymin);                    /* iy = ay*y + by         */
  by    = iymax*ymax/(ymax-ymin);

}; /* InitGraph */


/*--- Farbpunkt setzen -----------------------------------------------------*/

void SetPoint (double x, double y, tColor color)
{
  int ix  = x*ax+bx;         // Ermittle Position auf Bildschirm
```

```
  int iy  = y*ay+by;
  int icol= (int) (color);   // Konvertierung von tColor nach integer
  putpixel(ix,iy,icol);      // Setzten des Pixels
};

/*--- Grafik schließen -----------------------------------------------------*/

void CloseGraph() { closegraph(); }; /* der Symmetrie wegen ! */

/* v3_graph.c */
```

D Anhang V3_FRAKT

D.1 Headerdatei V3_FRAKT.H

```
/* Datei: v3_frakt.h            */
/* Fabian Kurz, 13.12.2004       */

#ifndef __frakt
#define __frakt

/*--- Datentypvereinbarungen -----------------------------------------------*/

typedef enum {  Apfel,       // Apfelmännchen
                Julia        // Juliamenge
             } tFracttyp;

struct tParam {           // Parameterstruktur
  int         R;          // Quadrat des Radius des Gebietes G
  int         imax;       // maximale Anzahl der Iterationen
  tFracttyp   ftype;      // Fraktaltyp, Apfel/Julia
  double      xmin;       // Minmaler Realteil des Analysegebietes
  double      xmax;       // Maximaler Realteil des Analysegebietes
  double      ymin;       // Minmaler Im-Teil des Analysegebietes
  double      ymax;       // Maximaler Im-Teil des Analysegebietes
};

struct tComplex {         // Struktur zur Beschreibung komplexer Zahlen
  double r;               // Realteil
  double i;               // Imaginärteil
};

/*--- Fraktal analysieren und grafisch darstellen --------------------------*/

int GetItera (tComplex c, tComplex z, tParam param);

void Fraktal(tComplex c, tComplex z, tParam p);

#endif

/* v3_frakt.h */
```

D.2 Implementierungsdatei V3_FRAKT.C

```
/* Datei: v3_frakt.c                              */
/* Autor Fabian Kurz, Datum 13-Dec-2004           */

#include "v3_frakt.h"
#include "v3_graph.h"
#include <math.h>
#include <stdio.h>

/*--- interne Funktion zur Bildung des Quadrats einer float-Zahl -----------*/

float quad (float a)
{
return a*a;
}

/*--- Interne Funktion: Analyse der Iterationsanzahl -----------------------*/
/*    Gibt die Anzahl der Iterationen zurück, bei der die Folge mit den     */
/*    gegebenen Parametern den vorgegebenen Radius verläßt                  */

int GetItera (tComplex c, tComplex z, tParam param)
{
int i = 0;        // Anzahl der benötigten Iterationen
tComplex alt;     // Ursprünglicher Wert wird hier gespeichert
tComplex z2;      // Arbeitswert

alt.r = z.r;      // Ursprungswert, von dem die Entfernung
alt.i = z.i;      // gemessen wird

while( ((quad(z.r-alt.r)+quad(z.i-alt.i)) < param.R) && (i < param.imax))
{                         // solange der maximale Radius und die maximale
                          // Iterationsanzahl nicht erreicht sind....
 z2  = z;                 // retten, da zwischenzeitlich vermurkst wird
 z.r = c.r + quad(z.r) - quad(z.i);   // Realteil fuer zi+1
 z.i = c.i + 2*z2.r*z2.i;             // Imaginärteil berechnen
 i++;                                 // Iterationsschritt erhoehen
}

 return i;        // Rückgabewert i (Anzahl der Iterationen)
}
/*--- Interne Funktion: Farbwert bestimmen ---------------------------------*/

tColor GetColorValue (int i, int imax)
{
int a;
if (i==imax) a = 0;     // Wenn maximale Iterationszahl erreicht wurde,
                        // also G nicht verlassen wird: schwarz
else {
a = imax/i;             // bei i = imax  1, bei i = 1  imax
a = (a % 16)+1;         // "aufteilen" auf 16 Farben
}
```

```
return (tColor) a;      // Farbe vom Typ tColor zurückgeben
}

/*--- Fraktal-Figur analysieren und zeichnen -------------------------------*/

void Fraktal(tComplex c, tComplex z, tParam p)
{
double x=p.xmin;                // Arbeitswert für Re(z) = x
double y=p.ymin;                // Arbeitswert für Im(z) = y
int it;                         // Iterationszähler
tColor f;                       // Farbwert

for (x=p.xmin;x<p.xmax;x+=(p.xmax-p.xmin)/640) { // xmin->xmax in 640 schritten
  for (y=p.ymin;y < p.ymax;y+=(p.ymax-p.ymin)/480) {  //ymin->ymax " 480 "
   if (p.ftype == Apfel) {         // Überprüfen ob Apfelmännchen oder ..
     c.r = x;       // Apfel: Variation von c
     c.i = y;
    }
    else {                         // etwa doch die Juliamenge
      z.r = x;      // Julia: Variation von z
      z.i = y;
    }

    it = GetItera(c,z,p);          // Finde Iterationsschritte heraus

    f = GetColorValue(it, p.imax); // ermittle Farbe aus it und imax

    SetPoint(x,y,f);               // Setze Farbpunkt an entsprechender Stelle
   }
 }
}

/* v3_frakt.c */
```