Kapitel 21

Graphikprogrammierung unter Linux

Eine redselige Dame konsultierte einst Doktor Heim: „Herr Professor“, klagte sie ganz bewegt, „ich glaube, ich habe mich überanstrengt!“. „Na“, lächelte Heim leicht verschmitzt, „dann zeigen Sie mal Ihre Zunge.“

Im Rahmen dieses Buches wurden eigene Graphikroutinen entworfen, mit denen eine einfache Graphikprogrammierung möglich ist. Die hier entworfenen Graphikroutinen ähneln denen von BGI (*Borland Graphics Interface*) des früher unter MS-DOS sehr beliebten Turbo-C und Turbo-Pascal. Diese Graphik-Implementierung hat den Namen LCGI (*Linux C Graphics Interface*). Natürlich sind Programme, welche die in diesem Kapitel vorgestellten Graphikroutinen verwenden, nicht mehr C89- bzw. C99-konform, sondern sind nur unter dem X-Windows-System von Linux/Unix ablauffähig, wenn die Graphikbibliothek *Qt* der Firma *Trolltech* installiert ist, was wohl meistens der Fall sein wird; andernfalls müssen Sie *Qt* erst installieren.

21.1 Bezug und Installation von LCGI

21.1.1 Bezug von LCGI

Die Graphikbibliothek LCGI befindet sich in dem gezippten tar-Archiv

```
lcgi-1.0.tgz
```

und kann von der Webseite

```
http://www.susepress.de/de/download
```

heruntergeladen werden.

21.1.2 Installation von LCGI

Um die Graphikbibliothek LCGI zu installieren, müssen Sie sich als `root` anmelden und anschließend in das Directory wechseln, in dem sich das heruntergeladene Archiv *lcgi-1.0.tgz* befindet. Nun müssen Sie die folgenden Schritte durchführen:

1. Entpacken des Archivs `lcgi-1.0.tgz` mit
   ```
   linux:~ #  tar xzvf lcgi-1.0.tgz ⏎
   ```
2. Wechseln in das Directory `lcgi-1.0`
   ```
   linux:~ #  cd lcgi-1.0 ⏎
   ```
3. Lesen Sie die Datei *README*, die die weiteren Installationsschritte beschreibt.

21.2 Benutzung von LCGI

21.2.1 Inkludieren von <**graphics.h**>

Da die hier vorgestellten Funktionen in der Headerdatei `graphics.h` deklariert sind, sollte immer angegeben werden, wenn man von diesen Funktionen in seinem Programm Gebrauch macht.

```
#include <graphics.h>
```

21.2.2 Angabe von `main()` mit Parametern

Wenn Sie in Ihrem Programm Routinen aus diesem Kapitel aufrufen, müssen Sie statt

```
int main(void)
```

immer folgendes angeben:

```
int main( int argc, char *argv[] )
```

21.2.3 Kompilieren und Linken von Graphikprogrammen

Wenn Sie ein Programm erstellt haben, das Routinen aus diesem Kapitel aufruft, müssen Sie dieses immer mit dem mitgelieferten Kommando `lcc` statt `gcc` bzw. `cc` kompilieren und linken.
Wenn Sie z. B. ein Programm `polygon.c` entworfen haben und dieses nun kompilieren wollen, müssen Sie z. B. folgendes aufrufen:

```
user@linux:~ >   lcc -o polygon polygon.c ⏎
```

`lcc` bietet die gleichen Optionen an wie das Kommando `gcc` bzw. `cc`.
Geben Sie danach zum Testen die folgende Kommandozeile ein:

```
user@linux:~ >  polygon ⏎
```

bzw. die folgende, wenn sie als `root` angemeldet sind:

```
linux:~ #  ./polygon ⏎
```

Wenn Sie mit der integrierten Environment `xwpe` arbeiten, müssen Sie unter dem Menüpunkt *Options* → *Compiler* → *C* die folgenden Einträge vornehmen:

Compiler:	`g++`
Compiler-Options:	`-g -I/usr/include/lcgi`
Loader-Options:	`-L/usr/lib/qt2/lib`[1] `-L/usr/lib -L/usr/X11R6/lib -lXext -lm -llcgi -lqt -lXext -lX11`
LAnguage:	`C`
File-Postfix:	`.c`

Die so eingestellten Optionen müssen Sie dann über den Menüeintrag *Options* → *Save Options* sichern.

21.3 Einige für Graphik benötigte C-Konstrukte

In diesem Kapitel werden einige C-Konstrukte benötigt, die erst in späteren Kapiteln behandelt werden. Die hier vorgeschobenen kurzen Erklärungen sollten jedoch ausreichen, diese Konstrukte zu verstehen und dann auch richtig einsetzen zu können.

21.3.1 Dynamisches Erstellen von Zeichenketten

Manchmal möchte man sich dynamisch Zeichenketten „zusammenbauen". Dazu steht die Funktion `sprintf()` zur Verfügung:

```
sprintf( string, format, ... );
```

Diese Funktion `sprintf()` kann genauso wie die Funktion `printf()` verwendet werden. Anders als `printf()` schreibt diese Funktion die formatierte Zeichenkette nicht auf den Bildschirm, sondern in die Variable `string`. Die Variable `string` selbst muss jedoch zuvor z. B. mit

```
char string[100];
```

definiert werden. Statt 100 kann auch eine größere oder eine kleinere Zahl angegeben werden. Die hier angegebene Zahl legt jedenfalls die maximale Anzahl von Zeichen fest, die ein so hergestellter String haben kann. Hat er mehr Zeichen führt dies zu einer Speicherüberschreibung, was katastrophale Folgen für das jeweilige Programm haben kann. Da C das Ende einer Zeichenkette mit einem 0-Byte kennzeichnet, muss man sogar Platz für ein Zeichen mehr reservieren. Wenn man also z. B. maximal 10 Zeichen (wie z. B. `bild10.gif`) nach `dateiname` schreiben möchte, muss man mindestens Speicherplatz für 11 Zeichen reservieren:

```
char dateiname[11];
```

Möchte man z. B. unter 10 Graphikdateien, die die Namen `bild1.gif` bis `bild10.gif` haben, eine zufällige auswählen, so wäre dies mit dem folgenden C-Code möglich:

[1] Sollten Sie für `QTDIR` einen anderen Pfad als `/usr/lib/qt2` eingestellt haben, müssen Sie diesen hier anstelle von `/usr/lib/qt2` angeben.

```
char dateiname[20];
int  z;
.....

z = rand()%10+1;
sprintf(dateiname, "bild%d.gif", z);
printf("%s\n", dateiname); /* gibt bild1.gif, bild2.gif oder ... oder bild10.gif
                              aus */
```

21.3.2 Kurze Beschreibung von Arrays

Die beiden später vorgestellten Graphikfunktionen `drawpoly()` (zeichnet den Umriss eines Polygons) und `fillpoly()` (zeichnet ein ausgefülltes Polygon) machen von so genannten *Arrays* Gebrauch. Ein Array ist eine Zusammenfassung von mehreren hintereinanderliegenden Speicherplätzen unter einem Namen.
Mit folgender Deklaration wird z. B. ein Array mit dem Namen `a` deklariert:

```
int a[10];
```

Dieses Array kann nun 10 `int`-Werte aufnehmen, wie z. B.

```
   a[0]=125; /* Im 1.Speicherplatz des Arrays 125 ablegen */
   a[1]=345; /* Im 2.Speicherplatz des Arrays 345 ablegen */
   a[2]=72;  /* Im 3.Speicherplatz des Arrays 72 ablegen  */
   ........
   a[9]=737; /* Im 10.Speicherplatz des Arrays 737 ablegen */
```

Anstelle von 10 `int`-Variablen wurde also ein Array `a` deklariert, das 10 `int`-Variablen unter einem Namen zusammenfasst:

```
      a
       +---------+
   [0] |   125   |
       +---------+
   [1] |   345   |
       +---------+
   [2] |    72   |
       +---------+
   [3] |   ...   |
   ... |   ...   |
   ... |   ...   |
   [8] |   ...   |
       +---------+
   [9] |   737   |
       +---------+
```

Um die einzelnen Speicherplätze zu unterscheiden, sind diese von `0` bis `9` (nicht bis `10`) durchnumeriert, wobei die entsprechende Nummer in `[...]` direkt nach dem Array-Namen (hier `a`) anzugeben ist. Arrays werden ausführlich in Kapitel 25 auf Seite 517 beschrieben.

21.4 Graphikmodus ein- und ausschalten

`initgraph(int breite, int hoehe)`

schaltet den Graphikmodus ein, indem ein Fenster eingeblendet wird, das `breite` Pixel breit und `hoehe` Pixel hoch ist.

`closegraph()`

beendet den Graphikmodus. Da in diesem Fall auch das Graphikfenster gelöscht wird, wird meist vor `closegraph()` noch ein `getch()` angegeben, so dass erst auf einen Tastendruck des Benutzers hin der Graphikmodus verlassen wird.

Typisch für die Graphikprogrammierung ist deshalb z. B. folgender Programmausschnitt:

```
#include <graphics.h>
.......
int  main( int argc, char *argv[] )
{
    initgraph( 640, 480 );
       ........
         Graphik-Programmteil
       ........
    getch();  /* auf einen Tastendruck warten */
    closegraph();
}
```

21.5 Eingaben im Graphikmodus

Während man im Graphikmodus arbeitet, kann man nicht mehr die Standardroutinen für die Ein- und Ausgabe (`printf()`, `scanf()`, `getchar()` und `putchar()`) verwenden, sondern muss die eigens dafür angebotenen Routinen verwenden, welche nachfolgend vorgestellt sind:

`getcharacter(char *text, ...)`

fordert den Benutzer durch Ausgabe des Textes `text` zur Eingabe eines Zeichens auf und liefert das vom Benutzer eingegebene Zeichen zurück.

`gettext(char *text, ...)`

fordert den Benutzer durch Ausgabe des Textes `text` zur Eingabe eines Textes auf und liefert den vom Benutzer eingegebenen Text als Rückgabewert.

`getint(char *text, ...)`

fordert den Benutzer durch Ausgabe des Textes `text` zur Eingabe einer ganzen Zahl auf und liefert die vom Benutzer eingegebene Zahl als `int`-Rückgabewert.

`getdouble(char *text, ...)`

fordert den Benutzer durch Ausgabe des Textes `text` zur Eingabe einer Gleitpunktzahl auf und liefert die vom Benutzer eingegebene Zahl als `double`-Rückgabewert.

`kbhit()`

prüft, ob eine Taste gedrückt wurde. Falls eine Taste gedrückt wurde, liefert `kbhit()` einen `int`-Wert ungleich 0 (TRUE), falls nicht, liefert diese Funktion 0 (FALSE) zurück. Das dabei eingegebene Zeichen kann mit der nachfolgend vorgestellten Routine `getch()` nachträglich erfragt werden.

`getch()`

liest ein Zeichen ein.

Anders als bei `getchar()` findet hier jedoch keine Zwischenpufferung statt, sondern das Zeichen wird direkt von der Tastatur gelesen. Das eingegebene Zeichen wird dabei nicht am Bildschirm angezeigt. In jedem Fall liefert diese Routine den Tastencode des eingegebenen Zeichens zurück. Mit `getch()` lassen sich nicht nur ASCII-Zeichen, sondern auch beliebige andere Steuerzeichen, wie z. B. Funktionstasten, ↑, Bild↓ usw. einlesen. `getch()` liefert einen eigenen Code für das eingegebene Zeichen. Für jede einzelne Taste steht im Graphikmodus eine eigene Konstante zur Verfügung:

```
Verschiedene Tasten
     Key_Escape, Key_Tab,    Key_Backtab, Key_Backspace, Key_Return, Key_Enter,
     Key_Insert, Key_Delete, Key_Pause,   Key_Print,     Key_Sysreq,
Tasten zur Cursor-Steuerung
     Key_Home,   Key_End,      Key_Left, Key_Up, Key_Right, Key_Down,
     Key_Pageup, Key_Pagedown,
Umschalt-Tasten
     Key_Shift,    Key_Control, Key_Meta,       Key_Alt,
     Key_Capslock, Key_Numlock, Key_Scrolllock,
Funktions-Tasten
     Key_F1,  Key_F2,  Key_F3,  Key_F4,  Key_F5,  Key_F6,  Key_F7,  Key_F8,
     Key_F9,  Key_F10, Key_F11, Key_F12, Key_F13, Key_F14, Key_F15, Key_F16,
     Key_F17, Key_F18, Key_F19, Key_F20, Key_F21, Key_F22, Key_F23, Key_F24,
     Key_F25, Key_F26, Key_F27, Key_F28, Key_F29, Key_F30, Key_F31, Key_F32,
     Key_F33, Key_F34, Key_F35,
Sonder-Tasten
     Key_Super_L, Key_Super_R, Key_Menu, Key_Hyper_L, Key_Hyper_R,
7-Bit druckbare ASCII-Tasten
     Key_Space,     Key_Exclam,     Key_Quotedbl,  Key_Numbersign,
     Key_Dollar,    Key_Percent,    Key_Ampersand, Key_Apostrophe,
     Key_Parenleft, Key_Parenright, Key_Asterisk,  Key_Plus,
     Key_Comma,     Key_Minus,      Key_Period,    Key_Slash,
     Key_0, Key_1, Key_2, Key_3, Key_4, Key_5, Key_6, Key_7, Key_8, Key_9,
     Key_Colon, Key_Semicolon,
     Key_Less,  Key_Equal,     Key_Greater, Key_Question, Key_At,
     Key_A, Key_B, Key_C, Key_D, Key_E, Key_F, Key_G, Key_H, Key_I, Key_J,
```

```
Key_K, Key_L, Key_M, Key_N, Key_O, Key_P, Key_Q, Key_R, Key_S, Key_T,
Key_U, Key_V, Key_W, Key_X, Key_Y, Key_Z,
Key_Bracketleft, Key_Backslash, Key_Bracketright, Key_Asciicircum,
Key_Underscore,  Key_Quoteleft, Key_Braceleft,    Key_Bar,
Key_Braceright,  Key_Asciitilde
```

`outtextxy(int left, int top, int right, int bottom, char *text, ...)`

gibt den Text `text` in dem Rechteck aus, dessen linke obere Ecke durch `(left, top)` und dessen rechte untere Ecke durch `(right,bottom)` festgelegt wird. `outtextxy()` benutzt die Schriftart und Formatierung, die mit `settextstyle()` und `settextjustify()`[2] festgelegt werden kann.

Bei den Graphikfunktionen `getcharacter()`, `gettext()`, `getint()`, `getdouble()` und `outtextxy()` bedeuten die drei Punkte, dass `text` die Formatierungszeichen von `printf()` enthalten kann, und dass dann eventuell angegebene weitere Argumente entsprechend formatiert im Text eingebettet werden, wie z. B.:

```
int  ganz, unten, oben;
....
unten = 5;
oben = 107;

ganz = getint( "Gib eine Zahl zwischen %d und %d ein", unten, oben );
```

Dieser Programmausschnitt blendet ein Fenster ein, in dem der Benutzer mit dem Satz „`Gib eine Zahl zwischen 5 und 107 ein`" zur Eingabe einer ganzen Zahl aufgefordert wird.
Eine weitere in der Graphikbibliothek angebotene Routine ist:

`delay(int millisekunden)`

hält die Programmausführung für `millisekunden` an.

Beispiel:
Das folgende Programm `einaus.c` demonstriert die Verwendung der Routinen zum Einlesen von Zeichen, Zahlen und Text.

```
#include <graphics.h>

int main( int argc, char *argv[] ) {
   char      zeich;
   int       iZahl;
   double    dZahl;
   const char *string;

   initgraph( 400, 250 );
   cleardevice( WHITE ); /* löscht den Inhalt des Graphikfensters
                            und setzt seinen Hintergrund auf weiss */
   zeich = getcharacter( "Gib ein Zeichen ein!" );
```

[2] siehe Kapitel 21.9 auf Seite 324

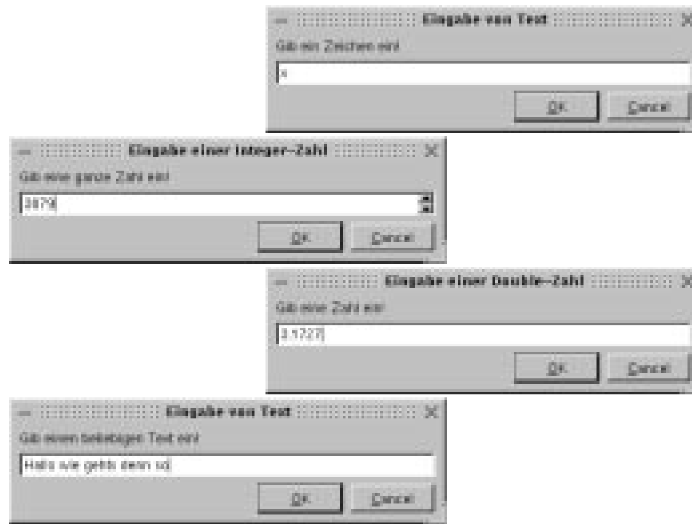


Abbildung 21.1: Fenster zur Eingabe von einzelnen Zeichen, Zahlen und Text

```
    iZahl = getint( "Gib eine ganze Zahl ein!" );
    dZahl = getdouble( "Gib eine Zahl ein!" );
    string = gettext( "Gib einen beliebigen Text ein!" );
    outtextxy( 100,  50, 400, 250, "Das Zeichen ist: %c", zeich );
    outtextxy( 100, 100, 400, 250, "Die ganze Zahl ist: %d", iZahl  );
    outtextxy( 100, 150, 400, 250, "Die Gleitpunktzahl ist: %g", dZahl  );
    outtextxy( 100, 200, 400, 250, "Der Text ist: %s", string  );
    getch();
    closegraph();
    return(0);
}
```

Abbildung 21.2: Anzeige der Eingaben im Hauptfenster

Das Programm `einaus.c` blendet die in Abbildung 21.1 gezeigten Fenster für die jeweiligen Eingaben ein. Nachdem der Benutzer alle geforderten Eingaben getätigt hat, werden ihm im Hauptfenster nochmals seine Eingaben angezeigt; siehe auch Abbildung 21.2.

Beispiel:

Mit dem folgenden C-Programm `reaktion.c` können Sie Ihre Reaktionszeit testen:

```
#include <graphics.h>
#include <stdlib.h>
#include <time.h>

int main( int argc, char *argv[] ) {
   double   zeit, min=1000000000;
   clock_t  start, ende;

   srand( time(NULL ) );
   initgraph( 640, 480 );

   do {
      cleardevice( WHITE ); /* loescht Graphikfenster und fuellt es mit weiss */
      outtextxy( 10, 10, 400, 400,
                "Reaktionstest\n"
                "============\n\n\n\n"
                "Gleich wird dieses Fenster gelb\n\n"
                "Dann musst du so schnell wie möglich eine Taste drücken....");
      delay( 2000+ rand()%4000 );
      if (kbhit()) {
         cleardevice( LIGHTRED ); /* loescht Graphikfenster u. fuellt's hellrot*/
         outtextxy(100, 200, 400, 400,
                   "Du hast versucht zu schummeln. Das ist nicht erlaubt!\n");
         while (kbhit())
            getch();
      } else {
         cleardevice( YELLOW ); /* loescht Graphikfenster und fuellt es gelb */
         start=clock(); /* Stopp-Uhr beginnt zu ticken */
         while (!kbhit())
            ;
         ende=clock(); /* Stopp-Uhr wird angehalten */
         getch(); /* Ueberlesen des eingegebenen Zeichens */
         zeit = (ende-start)/(double)CLOCKS_PER_SEC;
         if (zeit<min)
            min=zeit;
         outtextxy( 100, 300, 500, 400,
                    "Du hast %g Sek. gebraucht. (Bisheriger Rekord: %g Sek.)\n",
                     zeit, min);
      }
      outtextxy( 100, 350, 400, 400,
                 ".....Willst du es noch einmal probieren (j/n) ? ");
   } while (getch()==Key_J);
```

```
   closegraph();
   return(0);
}
```

21.6 Bildschirm-, Farben- und Pixel-Operationen

Tabelle 21.1 zeigt die im Graphikmodus möglichen Farben.
Der Bildschirm entspricht bei dieser Graphik einem x-y-Koordinatensystem, dessen Nullpunkt die linke obere Ecke ist (x=0,y=0). Der Graphikcursor kann unter Angabe eines (x,y)-Werts positioniert werden:

```
+------------------> x
|
|
|
v y
```

Folgende Graphikroutinen können dabei verwendet werden:

`cleardevice(int farbe)`

löscht den ganzen Inhalt des Graphikfensters und füllt es mit Farbe `farbe`.

`getmaxx()` bzw. `getmaxy()`

liefern die größte x- bzw. y-Koordinate (als `int`-Wert) des Graphikfensters.

`getx()` bzw. `gety()`

liefern die aktuelle x- bzw. y-Koordinate (als `int`-Wert) des Graphikcursors.

Tabelle 21.1: Die im Graphikmodus möglichen Farbekonstanten bzw. -nummern

Name	Wert	auf deutsch
BLACK	0	schwarz
BLUE	1	blau
GREEN	2	grün
CYAN	3	türkis
RED	4	rot
MAGENTA	5	violett
BROWN	6	braun
LIGHTGRAY	7	hellgrau
DARKGRAY	8	dunkelgrau
LIGHTBLUE	9	hellblau
LIGHTGREEN	10	hellgrün
LIGHTCYAN	11	helles Türkis
LIGHTRED	12	hellrot
LIGHTMAGENTA	13	helles Violett
YELLOW	14	gelb
WHITE	15	weiss

`getmaxcolor()`

liefert die höchste Farbnummer (als `int`-Wert), die im Graphikmodus verwendet werden kann.

`putpixel(int x, int y, int farbe)`

zeichnet einen Punkt (Pixel) mit der Farbe `farbe` an der Position (`x,y`).

`getpixel(int x, int y)`

liefert die Farbe (als `int`-Wert) des Pixels an der Position (`x,y`).

Beispiel:
Dieses Beispiel stammt aus dem 11. Informatik-Wettbewerb. Das hier verwendete Modell gibt stark vereinfacht eine Vorstellung davon, wie Öl in das Erdreich einsickert. Dazu stellt man sich einen vertikalen Schnitt durch den Erdboden vor und verwendet dafür ein rechteckiges Feld mit ganzzahligen Koordinaten. Die y-Achse des Koordinatensystems sei nach unten orientiert, die x-Achse nach rechts.
Über die anfängliche Verteilung des Öls nimmt man folgendes an: Das Öl befindet sich in den mittleren beiden Vierteln der oberen Erdschicht, d. h. an Positionen mit den Koordinaten (x,y), für die folgendes gilt:
$\frac{1}{4} \cdot Feldbreite <= x <= \frac{3}{4} \cdot Feldbreite$ und $y = 1$
Zusätzlich sei `x` ungerade, also nur jedes zweite Pixel.
Über das Eindringen des Öls in den Erdboden machen wir folgende Annahmen: Befindet sich Flüssigkeit an Position (x,y), so dringt sie mit der Wahrscheinlichkeit p (0 < p <= 1) in die nächsttiefere Schicht zur Position (x-1,y+1) und unabhängig davon auch mit der Wahrscheinlichkeit p zur Position (x+1,y+1) vor. Dabei kann p als Maß für die Bodenbeschaffenheit gedeutet werden. Befindet sich an Position (x,y) keine Flüssigkeit, gelangt von dieser Stelle her auch keine Flüssigkeit mehr in die nächsttiefere Erdschicht. Folgendes C-Programm `oel.c` löst diese Aufgabenstellung:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <graphics.h>

int  main( int argc, char *argv[] ) {
   float  p;  /* Einzulesende Wahrscheinlichkeit */
   int grenze, x, y, x_max, y_max, farbe,
       noch_oel=1; /* Boole'sche Var.: 1, solange Oel da, sonst dann 0 */
    srand(time(NULL)); /* Zufallszahlen-Generator initialisieren */
      /*--- Graphik einschalten */
    initgraph( 640, 480 );
      /*--- Wahrscheinlichkeit fuer Oel-Eindringen einlesen */
    p = getdouble("Wahrscheinlichkeit fuer Eindringen des Oels ? " );
    grenze = (int)(RAND_MAX*p);
      /*--- Graphikbildschirm loeschen und maximal moegl. Koordinaten ermitteln */
    cleardevice( LIGHTGRAY );
    x_max = getmaxx(); y_max = getmaxy();
```

Abbildung 21.3: Wahrscheinlichkeit von 0.62 (links) und 0.7 (rechts)

```
      /*--- Eindringen des Oels am Bildschirm simulieren */
    for (x=x_max/4 ; x<=x_max*3/4 ; x++)
       if ((x+1)%2==0)
          putpixel(x,1,BLUE);
    for (y=1 ; y<y_max-1 && noch_oel ; y++) {
       noch_oel=0;
       for (x=0 ; x<x_max-1 ; x++) {
          farbe = getpixel(x,y);
          if (farbe==BLUE || farbe==RED) {
             noch_oel=1;
             if (x-1>=0 && rand()<=grenze)
                putpixel(x-1,y+1,BLUE);
             if (x+1<=x_max && rand()<=grenze)
                putpixel(x+1,y+1,RED);
          }
       }
       if ( kbhit() )
          break;
    }
    getch();
    closegraph();
    return(0);
}
```

Abbildung 21.3[3] zeigt das Einsickern von Öl für die Wahrscheinlichkeiten von `0.62` und `0.7`.

Beispiel:

Das folgende Programm `pixel.c` wählt zufällig eine Hintergrundfarbe und malt dann 1 000 kleine Vierecke an zufällige Bildschirmpositionen mit zufällig gewählten Farben. Ein Viereck sind dabei 9 Pixel:

```
xxx
xox
xxx
```

[3]Beim Zeigen von Graphik-Bildschirmausgaben sind die Farben, wie sie wirklich am Bildschirm erscheinen, hier nicht erkennbar

wobei nur der Mittelpunkt (o) zufällig gewählt ist. Danach wird der Inhalt dieses Fensters gelöscht, mit einer zufälligen Hintergrundfarbe gefüllt und wieder mit neuen zufälligen kleinen Rechtecken bemalt. Mit einem Tastendruck kann dieses Programm beendet werden.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <graphics.h>

#define PIXEL_ZAHL    1000

int  main( int argc, char *argv[] )
{
   int i,
       maxx, maxy,
       x, y;
   int farbe,
       max_farbe;

    srand(time(NULL));

    initgraph( 640, 480 );
       /* Maximal moegliche Koordinaten */
    maxx=getmaxx();
    maxy=getmaxy();
       /* Maximale Farbennummer */
    max_farbe = getmaxcolor();
       /* Zufaellige Hintergrund-Farben und Setzen von farbigen Pixeln */
    while ( !kbhit()) {
       cleardevice( rand()%(max_farbe+1) );
       for (i=1 ; i<=PIXEL_ZAHL ; i++) {
          x = rand()%maxx;
          y = rand()%maxy;
          farbe = rand()%(max_farbe+1);
             /* Aus Pixel ein kleines Viereck malen */
          putpixel(x, y, farbe);
          putpixel(x-1,y-1,farbe);
          putpixel(x,y-1,farbe);
          putpixel(x+1,y-1,farbe);
          putpixel(x+1,y,farbe);
          putpixel(x+1,y+1,farbe);
          putpixel(x,y+1,farbe);
          putpixel(x-1,y+1,farbe);
          putpixel(x-1,y,farbe);
       }
    }
    closegraph();
    return 0;
}
```

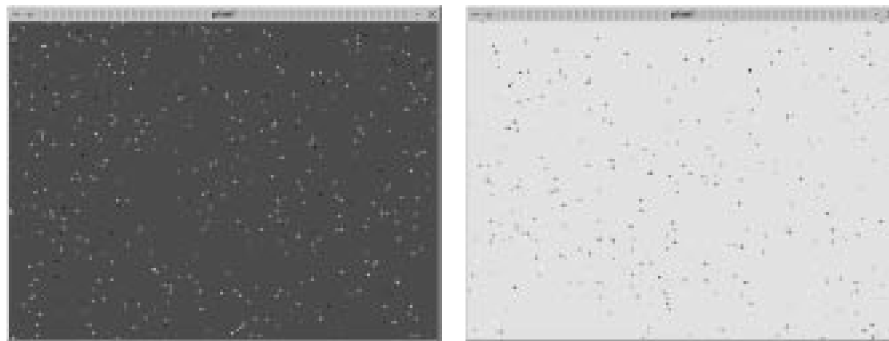


Abbildung 21.4: Ständige Ausgabe von kleinen Vierecken auf wechselnden Hintergrund

Abbildung 21.4 zeigt die Inhalte des von Programm `pixel.c` eingeblendeten Fensters zu gewissen Zeitpunkten.

21.7 Positionieren, Linien zeichnen und Farbe einstellen

`setcolor(int farbe)`

legt die Zeichenfarbe für Textausgaben und das Malen von Linien, Kreisen, Vierecken usw. auf die Farbe `farbe` fest.

`getcolor(int farbe)`

liefert die aktuell eingestellte Zeichenfarbe für Textausgaben und das Malen von Linien, Kreisen, Vierecken usw. (als `int`-Wert).

`moverel(int dx, int dy)`

bewegt den nicht sichtbaren Graphikcursor um eine Distanz (`dx,dy`) relativ zu seiner momentanen Position, ohne dabei zu zeichnen.

`moveto(int x, int y)`

positioniert den Graphikcursor auf den Punkt (`x,y`).

`line(int x1, int y1, int x2, int y2)`

zeichnet eine Linie von (`x1,y1`) nach (`x2,y2`). Dabei werden die aktuelle Farbe, Linienart und Linienbreite verwendet.

`linerel(int dx, int dy)`

zeichnet eine Linie relativ (`dx,dy`) zur momentanen Position des Graphikcursors. Es werden die aktuelle Farbe, Linienart und Linienbreite verwendet.

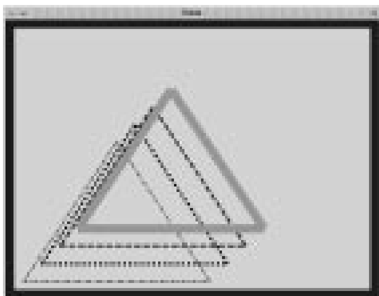


Abbildung 21.5: Anzeige des Programms `linie.c` nach vier Tastendrücken

`lineto(int x, int y)`

zeichnet eine Linie von der momentanen Position des Graphikcursors zu dem angegebenen absoluten Punkt (`x,y`). Es werden die aktuelle Farbe, Linienart und Linienbreite verwendet.

`setlinestyle(int linestyle, int dicke)`

setzt die Linienart und -dicke für folgende Zeichenaktionen. Die hier gesetzte Linienart wird von linienzeichnenden Graphikfunktionen, wie z. B. `line()`, `linerel()`, `lineto()`, `rectangle()`, `drawpoly()`, `arc()`, `circle()`, `ellipse()` oder `pieslice()` verwendet.

Für `linestyle` ist einer der folgenden Namen anzugeben:

Name	Wert	Art der Linie
NO_LINE	0	keine
SOLID_LINE	1	durchgezogen
DASHED_LINE	2	gestrichelt
DOTTED_LINE	3	gepunktet
DASH_DOT_LINE	4	Strich Punkt Strich Punkt...
DASH_DOT_DOT_LINE	5	Strich Punkt Punkt Strich Punkt Punkt Strich...
MAXLINESTYLE	5	

Beispiel:

Das folgende Programm `linie.c` zeichnet Dreiecke in die linke untere Ecke des Bildschirms. Die einzelnen Dreiecke sind dabei immer um etwas nach rechts oben versetzt. Abbildung 21.5 zeigt den Inhalt des von Programm `linie.c` eingeblendeten Fensters, nachdem viermal eine beliebige Taste gedrückt wurde.

Das Programm `linie.c`:

```
#include <graphics.h>

#define OFFSET   30

int  main( int argc, char *argv[] ) {
```

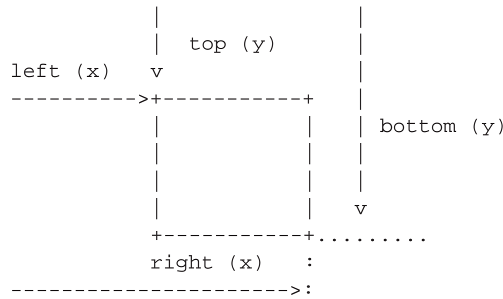
```
int maxx, maxy;
 initgraph( 640, 480 );
    /* Maximal moegliche Koordinaten */
 maxx = getmaxx();
 maxy = getmaxy();
    /*--- Bildschirm loeschen ---*/
 cleardevice( LIGHTGRAY );
    /*--- Blauen dicken Rahmen um Bildschirm zeichnen ---*/
 setcolor( BLUE );
 setlinestyle( SOLID_LINE, 30 );
 line( 0, 0, maxx, 0 );
 line( maxx, 0, maxx, maxy );
 line( maxx, maxy, 0, maxy );
 line( 0, maxy, 0, 0 );
 getch();
    /*--- Rotes Dreieck an linken unteren Bildschirm ---*/
 setcolor( RED );
 setlinestyle( DASH_DOT_LINE, 3 );
 moveto( OFFSET, maxy-OFFSET );
 lineto( maxx/2+OFFSET, maxy-OFFSET );
 linerel( -maxx/4,-maxy/2 );
 linerel( -maxx/4,maxy/2 );
 getch();
    /*--- Schwarzes Dreieck mit dicker gepunkteter Linie etwas nach ---*/
    /*--- rechts oben versetzt                                       ---*/
 setcolor( BLACK );
 setlinestyle( DOTTED_LINE, 5 );
 moverel( +OFFSET, -OFFSET );
 linerel( maxx/2, 0 );
 linerel( -maxx/4, -maxy/2 );
 linerel( -maxx/4, maxy/2 );
 getch();
    /*--- Blaues Dreieck mit gestrichelter Linie nach rechts oben versetzt */
 setcolor( BLUE );
 setlinestyle( DASHED_LINE, 4 );
 moverel( +OFFSET, -OFFSET );
 linerel( maxx/2, 0 );
 linerel( -maxx/4, -maxy/2 );
 linerel( -maxx/4, maxy/2 );
 getch();
    /*--- Gruenes Dreieck mit durchgezogener Linie nach rechts oben -*/
 setcolor( GREEN );
 setlinestyle( SOLID_LINE, 15 );
 moverel( +OFFSET, -OFFSET );
 linerel( maxx/2, 0 );
 linerel( -maxx/4, -maxy/2 );
 linerel( -maxx/4, maxy/2 );
 getch(); getch();
 closegraph();
```

```
    return(0);
}
```

21.8 Figuren zeichnen und ausfüllen

`rectangle(int left, int top, int right, int bottom)`

zeichnet ein Rechteck, wobei die aktuellen Einstellungen für Linienart, -stärke und -farbe benutzt werden; siehe auch `setcolor()` und `setlinestyle()`. Die linke obere Ecke ist dabei durch die Koordinaten (`left,top`) und die rechte untere Ecke durch (`right,bottom`) in Pixeln festgelegt:

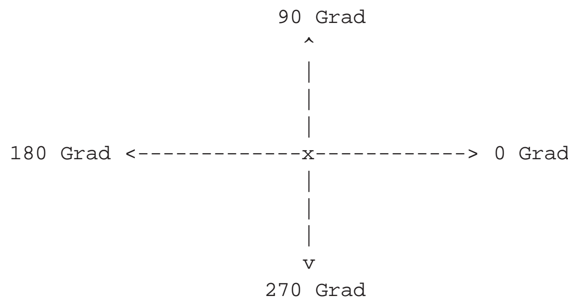


`circle(int x, int y, int radius)`

zeichnet in der aktuellen Zeichenfarbe (siehe `setcolor()`) einen Kreis um den Mittelpunkt (`x,y`) mit dem Radius `radius`.

`arc(int x, int y, int start, int end, int radius)`

zeichnet in der aktuellen Zeichenfarbe (siehe `setcolor()`) einen Kreisbogen. (`x,y`) ist der Mittelpunkt. Die Winkel `start` und `end` legen Start- und Endpunkt des mit Radius `radius` gezogenen Kreisbogens fest. Die Angabe beider Winkel erfolgt in Grad, wobei entgegen dem Uhrzeigersinn gezählt wird:



`ellipse(int x, int y, int start, int end, int xradius, int yradius)`

zeichnet einen elliptischen Kreisausschnitt um den Mittelpunkt (`x,y`) mit den Radien `xradius` (horizontal) und `yradius` (vertikal). `start` und `end` legen dabei den Start- und Endpunkt des elliptischen Bogens fest. Hat `start` den Wert 0 und `end` den Wert 360, wird eine vollständige Ellipse gezeichnet. Die Angabe beider Winkel erfolgt in Grad, wobei entgegen dem Uhrzeigersinn gezählt wird; siehe auch `arc()`.

`setfillstyle(int pattern, int farbe)`

setzt das Muster (`pattern`) und die dabei zu verwendende `farbe` für Flächenfüllungen. Für `pattern` ist einer der folgenden Namen anzugeben:

Name	Wert	Füllmuster
EMPTY_FILL	0	Hintergrundfarbe
SOLID_FILL	1	angegebene `farbe`
DENSE1_FILL	2	sehr stark ausgefüllt
DENSE2_FILL	3	‖
DENSE3_FILL	4	‖
DENSE4_FILL	5	‖
DENSE5_FILL	6	‖
DENSE6_FILL	7	⇓
DENSE7_FILL	8	sehr schwach ausgefüllt
HORLINE_FILL	9	horizontale Linien
VERLINE_FILL	10	vertikale Linien
CROSS_FILL	11	horizontale und vertikale Linien
LDIAG_FILL	12	//////////
RDIAG_FILL	13	\\\\\\\\\\\\
CROSSDIAG_FILL	14	beides (`LDIAG_FILL` und `RDIAG_FILL`)
MAXFILLSTYLE	14	

`fillellipse(int x, int y, int xradius, int yradius)`

zeichnet eine ausgefüllte Ellipse mit (`x,y`) als Mittelpunkt; `xradius` wird dabei als horizontaler und `yradius` als vertikaler Radius verwendet. Die Ellipse selbst wird mit dem aktuellen Füllmuster und der aktuellen Füllfarbe gefüllt; siehe auch `setfillstyle()`.

`bar(int left, int top, int right, int bottom)`

zeichnet einen zweidimensionalen Balken. Wie bei `rectangle()` wird die linke obere Ecke durch die Koordinaten (`left,top`) und die rechte untere Ecke durch (`right,bottom`) in Pixeln festgelegt. Es wird dabei die aktuelle Füllfarbe und das aktuelle Füllmuster (siehe `setfillstyle()`) verwendet. Einen Umriss zeichnet `bar()` nicht; dazu müßte `bar3d()` mit `depth=0` verwendet werden.

`bar3d(int left, int top, int right, int bottom, int depth)`

zeichnet einen dreidimensionalen Balken. Ein dreidimensionaler Balken besteht aus einem gefüllten Rechteck, bei dem – wie bei `rectangle()` – die linke obere Ecke durch die Koordinaten (`left,top`) und die rechte untere Ecke

durch (`right,bottom`) in Pixeln festgelegt wird. `bar3d()` zeichnet zuerst die Umrisslinien in der aktuellen Zeichenfarbe (siehe `setcolor()`) und der aktuellen Linienart (siehe `setlinestyle()`). Danach füllt es die umschlossene Fläche mit dem aktuellen Füllmuster; siehe auch `setfillstyle()`). Über `depth` (zu deutsch: Tiefe) läßt sich die räumliche Tiefe des dreidimensionalen Balkens (in Pixel) festlegen. Eine Faustregel besagt dabei, dass die Tiefe ungefähr 25% der Breite betragen sollte. Bei `depth=0` wird nur ein zweidimensionaler Balken mit einer Umrisslinie gezeichnet.

`drawpoly(int numpoints, int *polypoints)`

zeichnet den Umriss eines Polygons mit `numpoints` Eckpunkten in der aktuellen Linienart und -farbe. `numpoints` gibt die Anzahl der Eckpunkte an. `polypoints` muss der Name des Arrays sein, das fortlaufende Koordinatenpaare enthalten muss. Zum Zeichnen einer geschlossenen Figur mit *n* Eckpunkten muss `polypoints` *n+1* Koordinatenpaare enthalten, wobei das letzte Paar dieselben Werte wie das erste Paar hat; siehe auch nachfolgende Beispiele.

`fillpoly(int numpoints, int *polypoints)`

zeichnet ein ausgefülltes Polygon mit `numpoints` Eckpunkten in der aktuellen Linienart und -farbe. Danach wird dieses Polygon mit der aktuellen Füllfarbe und dem aktuellen Füllmuster (siehe `setfillstyle()`) gefüllt. Wie bei `drawpoly()` gilt: `numpoints` gibt die Anzahl der Eckpunkte an. `polypoints` muss der Name des Arrays sein, das fortlaufende Koordinatenpaare enthalten muss. Zum Zeichnen einer geschlossenen Figur mit *n* Eckpunkten muss `polypoints` *n+1* Koordinatenpaare enthalten, wobei das letzte Paar dieselben Werte wie das erste Paar hat; siehe auch nachfolgende Beispiele.

Beispiele:
Das folgende Programm `figur.c` malt eine Person mit Tisch auf den Bildschirm. Die Farben und Muster in den jeweiligen Symbolen sind dabei zufällig ausgewählt. Bei einem Tastendruck wird immer ein neues Bild gemalt. Das Programm kann mit Drücken der ESC-Taste verlassen werden. Mögliche durch `figur.c` gezeigte Bilder könnten z. B. aussehen, wie sie in Abbildung 21.6 gezeigt sind.
Das Programm `figur.c`:

```
#include <graphics.h>
#include <time.h>
#include <stdlib.h>

#define ZUFALLSFARBE \
       farbe = rand()%(max_farbe+1);   /* Zufaellige Zeichenfarbe */ \
       setcolor(farbe);                /* setzen                  */

#define ZUFALLS_FUELLMUSTER \
       muster = rand()%MAXLINESTYLE;   /* Zufaelliges Fuellmuster */ \
       hfarbe = rand()%(max_farbe+1);  /* Zufaellige Fuellfarbe   */ \
```

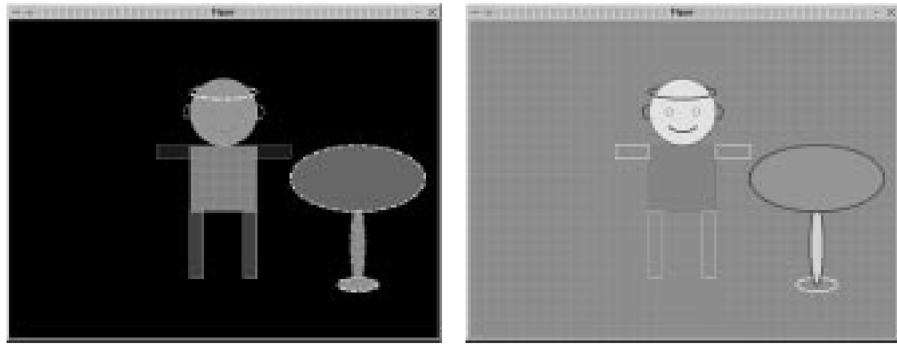


Abbildung 21.6: Unterschiedliche Anzeigen des Programms `figur.c` nach einem Tastendruck

```
      setfillstyle(muster,hfarbe);    /* setzen */

int  main( int argc, char *argv[] ) {
   int maxx, maxy,
       muster, lmuster,
       farbe, hfarbe,
       mittex, mittey,
       max_farbe, poly[10];

    srand(time(NULL));  /* Zufallsgenerator initialisieren */

    initgraph( 640, 480 );

    maxx = getmaxx();  /* Maximal moegliche Koordinaten */
    maxy = getmaxy();
    mittex = maxx/2;   /* Bildschirm-Mittelpunkt */
    mittey = maxy/2;
    max_farbe = getmaxcolor();      /* Maximale Farbennummer */

    while (1) {  /* Figur malen */
          /*------ Bildschirm-Rahmen und -Hintergrund -------------------*/
       cleardevice( BLACK );
       setlinestyle( SOLID_LINE, 1 );
       ZUFALLSFARBE; ZUFALLS_FUELLMUSTER;
       poly[0] = poly[1] = poly[3] = poly[6] = poly[8] = poly[9] = 0;
       poly[2] = poly[4] = maxx;
       poly[5] = poly[7] = maxy;
       fillpoly( 5, poly );
          /*------ Bauch -----------------------------------------------*/
       ZUFALLSFARBE; ZUFALLS_FUELLMUSTER;
       bar3d( mittex-50, mittey-50, mittex+50, mittey+50, 0 ); /* Bauch malen */
          /*------ Kopf ------------------------------------------------*/
       ZUFALLSFARBE;
```

```
        ZUFALLS_FUELLMUSTER;
        fillellipse( mittex, mittey-100, 50, 50 );   /* Kopf malen */
           /*------ Beine -------------------------------------------------*/
        ZUFALLSFARBE; ZUFALLS_FUELLMUSTER;
        bar3d( mittex-50, mittey+50, mittex-30, mittey+150, 0 ); /* Linkes Bein */
        bar3d( mittex+30, mittey+50, mittex+50, mittey+150, 0 ); /* Rechtes Bein */
           /*------ Arme --------------------------------------------------*/
        ZUFALLSFARBE; ZUFALLS_FUELLMUSTER;
        bar3d( mittex-100, mittey-50, mittex-50,  mittey-30, 0 ); /* Linker Arm */
        bar3d( mittex+50,  mittey-50, mittex+100, mittey-30, 0 ); /* Rechter Arm */
           /*------ Augen -------------------------------------------------*/
        ZUFALLSFARBE; ZUFALLS_FUELLMUSTER;
        fillellipse( mittex-20, mittey-100, 5, 5 ); /* Linkes Auge */
        fillellipse( mittex+20, mittey-100, 5, 5); /* Rechtes Auge */
           /*------ Ohren ------------------------------------------------*/
        lmuster = rand()%MAXLINESTYLE;   /* Zufaelliges Linienmuster */
        setlinestyle( lmuster, 3 );      /* und dicke Linie setzen */
        arc( mittex-50, mittey-100,  90, 270, 10 ); /* Linkes Ohr */
        arc( mittex+50, mittey-100, 270,  90, 10 ); /* Rechtes Ohr */
          /*------- Mund -------------------------------------------------*/
        arc( mittex, mittey-100, 225, 315, 30 );
           /*------ Heiligenschein ---------------------------------------*/
        ZUFALLSFARBE;
        ellipse( mittex, mittey-130, 135,45, 50,10 );
           /*------ Tisch ------------------------------------------------*/
        ZUFALLSFARBE; ZUFALLS_FUELLMUSTER;
        fillellipse( mittex+200,mittey+160, 30,10 );
        ZUFALLSFARBE; ZUFALLS_FUELLMUSTER;
        fillellipse( mittex+200,mittey+100, 10,60 );
        ZUFALLSFARBE;
        setfillstyle( SOLID_FILL, rand()%getmaxcolor() );
        fillellipse(mittex+200,mittey, 100,50 );

        if ( getch() == Key_Escape ) /* Abbruch bei ESC */
           break;
     }
     closegraph();
     return(0);
}
```

Das folgende Programm `fmuster.c` gibt nacheinander alle Füllmuster mit `bar()` und `bar3d()` am Bildschirm aus, wie es in der Abbildung 21.7 gezeigt ist.

```
#include <graphics.h>

int  main( int argc, char *argv[]) {
   int links, oben, i;
    initgraph( 640, 500 );
    setcolor( RED );
       /* Alle Fuellmuster fuer bar durchlaufen */
    for (i=SOLID_FILL ; i<=MAXFILLSTYLE ; i++) {
```

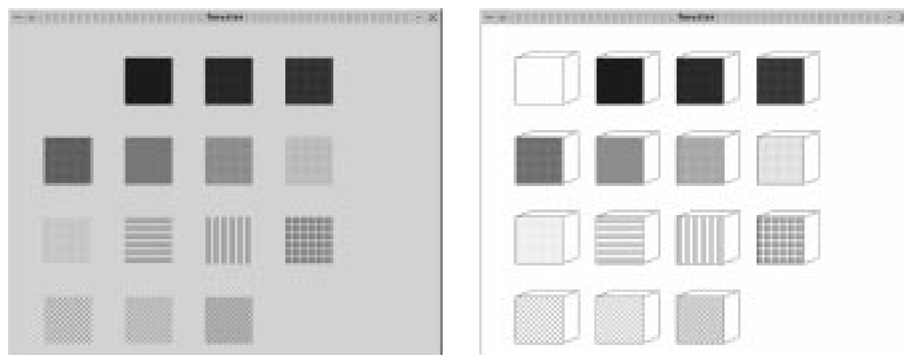


Abbildung 21.7: Unterschiedliche Anzeigen des Programms `fmuster.c` nach einem Tastendruck

```
        links=i%4;
        oben=i/4;
        setfillstyle( i, BLUE );
        bar(50+links*120, 50+oben*120, 120+links*120, 120+oben*120);
     }
     getch();
     cleardevice( WHITE );
        /* Alle Fuellmuster fuer bar3d durchlaufen */
     for (i=EMPTY_FILL ; i<=MAXFILLSTYLE ; i++) {
        links=i%4;
        oben=i/4;
        setfillstyle( i, BLUE );
        bar3d(50+links*120, 50+oben*120, 120+links*120, 120+oben*120, 25 );
     }
     getch();
     closegraph();
     return 0;
}
```

Das folgende Programm `polygon.c` gibt „Stern-Polygone“ nacheinander am Bildschirm aus. Die einzelnen Polygone, deren Eckpunkte in einem gewissen Bereich zufällig gewählt werden, haben immer den Fenster-Mittelpunkt als Mittelpunkt und werden übereinander gezeichnet. Der Benutzer kann dabei am Anfang des Programms angeben, ob die Polygone ausgefüllt sein sollen oder nicht.

```
#include <ctype.h>
#include <stdlib.h>
#include <time.h>
#include <graphics.h>

int main( int argc, char* argv[] ) {
   int ausgefuellt;  /* 1, wenn ausgefuellte Polygone gewuenscht, sonst 0 */
   int mx, my;
   int poly[18]; /*----- immer eins mehr hier angeben als Elemente
```

```
                 ----- wirklich benoetigt werden. */

    printf("Ausgefuellte Polygone (j/n): ");
    ausgefuellt = toupper(getchar())=='J' ? 1 : 0;
    srand(time(NULL)); /* Zufallszahlen-Generator initialisieren */

    initgraph( 640, 480 );
    mx = getmaxx()/2; my = getmaxy()/2;
    setcolor( BLUE );
    while (1) {
       poly[0] = mx+rand()%mx;    poly[1] = my;              /* 1.Eckpunkt */
       poly[2] = mx+20+rand()%50; poly[3] = my-20-rand()%50; /* 2.Eckpunkt */
       poly[4] = mx;              poly[5] = my-rand()%my;    /* 3.Eckpunkt */
       poly[6] = mx-20-rand()%50; poly[7] = my-20-rand()%50; /* 4.Eckpunkt */
       poly[8] = mx-rand()%mx;    poly[9] = my;              /* 5.Eckpunkt */
       poly[10] = mx-20-rand()%50;poly[11] = my+20+rand()%50;/* 6.Eckpunkt */
       poly[12] = mx;             poly[13] = my+rand()%my;   /* 7.Eckpunkt */
       poly[14] = mx+20+rand()%50;poly[15] = my+20+rand()%50;/* 8.Eckpunkt */
          /* Da drawpoly das Polygon nicht automatisch schließt, muss als */
          /* Endpunkt wieder der Anfangspunkt angegeben werden.           */
       poly[16] = poly[0]; poly[17] = poly[1];
          /* Polygon zeichnen */
       if (ausgefuellt) {
          setfillstyle(SOLID_FILL, rand()%getmaxcolor());
          fillpoly(9, poly);
       } else
          drawpoly(9, poly);
       if (getch()==Key_Escape) /* Abbruch mit ESC */
          break;
    }
    closegraph();
    return 0;
}
```

Mögliche durch `polygon.c` ausgegebene Bilder nach x-maligen Tastendruck zeigt Abbildung 21.8.

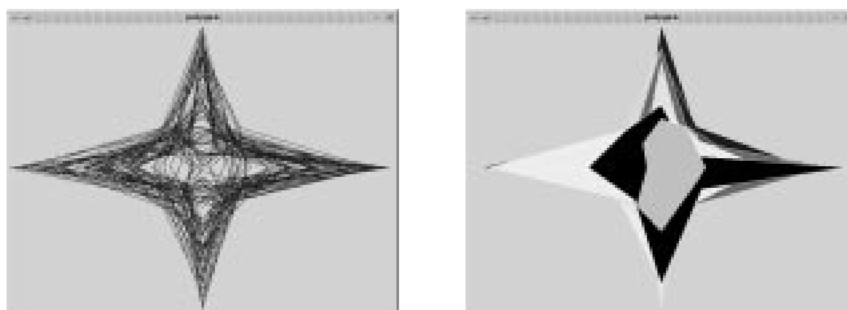


Abbildung 21.8: Anzeige des Programms `polygon.c` nach x-maligen Tastendruck (links: nicht ausgefüllt, rechts: ausgefüllt)

21.9 Einstellungen für Textausgaben

`settextjustify(int horiz,int vert)`

legt die Ausrichtung (Justierung) nachfolgender Textausgaben mit `outtextxy()` fest. Der Text kann horizontal und vertikal justiert werden. Für `horiz` und `vert` sind folgende Werte möglich:

Parameter	Name	Wert	Justierung
horiz	`LEFT_TEXT`	0	linksbündig
	`HCENTER_TEXT`	1	horizontal zentriert
	`RIGHT_TEXT`	2	rechtsbündig
vert	`BOTTOM_TEXT`	0	an Grundlinie ausgerichtet
	`VCENTER_TEXT`	1	vertikal zentriert
	`TOP_TEXT`	2	oben ausgerichtet

`settextstyle(char *name, int groesse)`

`name` legt den Zeichensatz (`"Times", "Helvetica"` usw.) und `groesse` die Größe der Zeichen für folgende Textausgaben fest.

`textheight()`

liefert die Höhe des aktuell eingestellten Zeichensatzes in Pixeln (als `int`-Wert) zurück.

`textwidth(char *text)`

liefert die Breite, die der Text `text` benötigt, in Pixeln (als `int`-Wert) zurück.

Das folgende Programm `groesse.c` zeigt alle Schriftgrößen des Font *Times* ab der Größe 10 in 10er-Schritten an; siehe auch Abbildung 21.9

```
#include <stdio.h>
#include <stdlib.h>
#include <graphics.h>
```

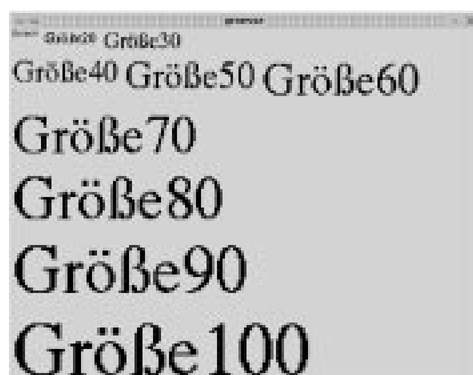


Abbildung 21.9: Anzeige des Programms `groesse.c`

```
int  main( int argc, char *argv[] ) {
   int i, x=1, y=0;
   char string[80];  /* Auszugebende Zeichenkette wird hier mit sprintf
                        (aus stdlib.h) vor der Ausgabe hinterlegt        */
    initgraph( 640, 480 );

    for (i=10 ; i<=100 ; i+=10) {
       /* Textstil, Richtung und Groesse festlegen */
       settextstyle( "Times", i );

       /* Text formatiert mit sprintf (aus stdlib.h) in string ablegen */
       sprintf(string, "Größe%d", i );

       /* Ausgabe des Textes aus string */
       outtextxy( x, y, getmaxx(), getmaxy(), string );

       /* Naechste Textzeile (y-Koordinate) berechnen */
       if (i<=60) {
          x += textwidth(string)+10;
          if (i==30 || i==60) {
             x = 1;
             y += textheight();
          }
       } else
          y += textheight();
    }

    getch();
    closegraph();
    return 0;
}
```

Das folgende Programm `just.c` demonstriert alle Justierungs-Kombinationen in Tabellenform; siehe auch Abbildung 21.10.

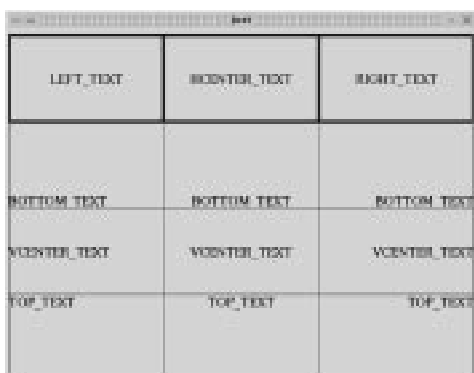


Abbildung 21.10: Anzeige des Programms `just.c`

```
#include <graphics.h>

int  main( int argc, char *argv[] ) {
   int x, y=0, maxx, maxy, yd, h, v;
    initgraph( 640, 480 );
    maxx = getmaxx(); maxy = getmaxy();
    x    = maxx/3;
    yd   = maxy/4;
    settextstyle( "Times", 18 );
       /* Alle Justierungs-Kombinationen als Tabelle anzeigen */
    for ( h=LEFT_TEXT; h<=RIGHT_TEXT; h++ ) {
       y = 10;
       setlinestyle( SOLID_LINE, 5 );
       settextjustify( HCENTER_TEXT, VCENTER_TEXT );
       setcolor( BLUE );
       rectangle( h*x, y, (h+1)*x, y+yd );
       setlinestyle( SOLID_LINE, 1 );
       setcolor( BLACK );
       switch (h) {
         case LEFT_TEXT:
             outtextxy( h*x, y, (h+1)*x, y+yd, "LEFT_TEXT" );    break;
         case HCENTER_TEXT:
             outtextxy( h*x, y, (h+1)*x, y+yd, "HCENTER_TEXT" ); break;
         case RIGHT_TEXT:
             outtextxy( h*x, y, (h+1)*x, y+yd, "RIGHT_TEXT" );   break;
       }
       for ( v=BOTTOM_TEXT; v<=TOP_TEXT; v++ ) {
          y += yd;
          settextjustify( h, v );   /* Text-Ausrichtung festlegen */
          setcolor( BLUE );
          rectangle( h*x, y, (h+1)*x, y+yd );
          setcolor( BLACK );
          switch (v) {
            case BOTTOM_TEXT:
                outtextxy( h*x, y, (h+1)*x, y+yd, "BOTTOM_TEXT" );  break;
            case VCENTER_TEXT:
                outtextxy( h*x, y, (h+1)*x, y+yd, "VCENTER_TEXT" ); break;
            case TOP_TEXT:
                outtextxy( h*x, y, (h+1)*x, y+yd, "TOP_TEXT" ) ;    break;
          }
       }
    }
    getch(); closegraph(); return 0;
}
```

21.10 Externe Bilder laden, Bildteile speichern und einblenden

`loadimage(char *filename, void **imagebuffer)`

lädt ein externes Bild, das den Dateinamen `filename` hat, in den Puffer `imagebuffer`. Um z. B. die beiden Dateien `lausbub.gif` und `urlaub.jpg` zu laden, wäre der folgende Code denkbar:

```
void *bild1, *bild2;
....
loadimage( "lausbub.gif", &bild1);
loadimage( "urlaub.jpg", &bild2);
```

Das Anzeigen eines geladenen Bildes ist mit der weiter unten vorgestellten Funktion `putimage()` möglich. Es werden alle wichtigen Graphikformate unterstützt, wie z. B. `.gif`, `.jpg`, `.bmp`, `.png`, `.xpm` usw.

`getimage(int left, int top, int right, int bottom, void **imagebuffer)`

kopiert einen rechteckigen Bildausschnitt, dessen linke obere Ecke durch (`left, top`) und dessen rechte untere Ecke durch (`right,bottom`) festgelegt ist, in den Puffer `imagebuffer`.

`putimage(int left, int top, void *imagebuffer, int op)`

kopiert den Inhalt des Puffers mit der Kennung `imagebuffer`, der zuvor mit `loadimage()` bzw. `getimage()` gefüllt wurde, bitweise in einen rechteckigen Ausschnitt des Graphikbildschirms. Die linke obere Ecke dieses Zielbereichs wird dabei mit (`left,top`) festgelegt. `op` legt dabei fest, wie die Farbe jedes Pixels im Zielbereich basierend auf der Farbe der bereits dort vorhandenen Pixels und der jeweiligen Pixel aus dem zum kopierenden Bild zusammenzufügen sind. Für `op` kann einer der folgenden Namen angegeben werden:

Name	Wert	Bedeutung
COPY_PUT	0	Kopieren (Zielbereich einfach überschreiben)
XOR_PUT	1	XOR-Operation mit Pixeln des Zielbereichs
OR_PUT	2	OR-Operation mit Pixeln des Zielbereichs
AND_PUT	3	AND-Operation mit Pixeln des Zielbereichs
NOT_PUT	4	Pixel des zu kopierenden Bildes invertieren

`freeimage(void **imagebuffer)`

gibt einen zuvor mit `loadimage()` bzw. `putimage()` angelegten Puffer wieder frei.

Beispiel:
Im folgenden C-Programm `sandmann.c` läuft ein Männchen über den Bildschirm und gibt oben bzw. unten die Meldung „Ich bin Kaffeetrinken" aus. Das Männchen läuft dabei vom linken zum rechten Bildschirmrand und streut Sand. Beim Laufen bewegt es sich zufällig einige Pixel nach unten oder oben. Hat es den rechten Rand

erreicht, wird das Bild gelöscht und das Männchen beginnt erneut von links nach rechts zu laufen usw., bis ein Tastendruck des Benutzers das Programm beendet.

```
#include <stdlib.h>
#include <time.h>
#include <graphics.h>

#define EINH  5

int  main( int argc, char *argv[] ) {
   int x, y, alty, distanz;
    srand(time(NULL));  /* Zufallsgenerator initialisieren */
    initgraph( 640, 480 );
    cleardevice( BLACK );
    x = getmaxx()/2; y = getmaxy()/2;
    setcolor(getmaxcolor());
    settextstyle( "Times", 48 );   /* Textstil festlegen */
        /* Sandmaennchen in die Mitte malen */
    moveto(x,y); linerel( 2*EINH, 2*EINH); linerel(-EINH, EINH);  /* Bein vorne */
    moveto(x,y); linerel(-2*EINH, 2*EINH); linerel(-EINH, -EINH); /* Bein hinten*/
    moveto(x,y); linerel(0, -2*EINH);                             /* Koerper    */
    moveto(x,y-EINH); linerel(-EINH,0); linerel(-EINH,EINH/2);    /* Arm hinten */
    moveto(x,y-EINH); linerel(EINH,EINH); linerel(EINH,-EINH);    /* Arm vorne  */
    circle(x,y-2*EINH-EINH/2,EINH/2);                             /* Kopf       */
       /* Bild vom Sandmaennchen in internen Puffer hinterlegen */
    getimage( x-3*EINH, y-3*EINH, x+3*EINH, y+3*EINH, 1 );
       /* Sandmaennchen wandert quer über den Bildschirm */
    while (!kbhit()) {
       cleardevice( BLACK );
       y=rand()%getmaxy();
       outtextxy( 0, (y>200)?10:350, getmaxx(), getmaxy(), "Ich bin Kaffeetrinken");
       putimage(0, y-3*EINH, 1, XOR_PUT);
       distanz=EINH/9+1;
       for (x=3*EINH+distanz ; x<=getmaxx() ; x+=distanz ) {
          alty=y;
          y += rand()%7-3;
            /* Neues Sandmaennchen malen und .... */
          putimage(x-3*EINH, y-3*EINH, 1, XOR_PUT);
            /* altes Sandmaennchen loeschen */
          putimage(x-3*EINH-distanz, alty-3*EINH, 1, XOR_PUT);
            /* Sand streuen */
          putpixel(x-3*EINH-distanz, alty+3*EINH, YELLOW);
          if ( kbhit() )
             break;
          delay( 1 );
       }
    }
    closegraph();
    return(0);
}
```

Abbildung 21.11: Anzeige des Programms `sandmann.c`

Abbildung 21.11 zeigt das von Programm `sandmann.c` angezeigte Fenster zu einem bestimmten Zeitpunkt. Dieses Programm könnte z. B. gestartet werden, wenn man seinen Bildschirmplatz längere Zeit verlässt, und man Information am Bildschirm darüber ausgeben lassen möchte, wo man sich gerade befindet und wann man wieder zurückkommt. Hierzu müsste man jedoch dann die auszugebende Meldung einlesen lassen. Eine entsprechende Anpassung dieses Programms an die speziellen eigenen Bedürfnisse sollte aber nicht allzu schwierig sein.

Beispiel:

Das folgende Programm `karten.c` blendet bei jedem Tastendruck immer sieben zufällig ausgewählte Karten ein, die es übereinander legt (siehe auch Abbildung 21.12). Mit der `ESC`-Taste kann dieses Programm beendet werden.

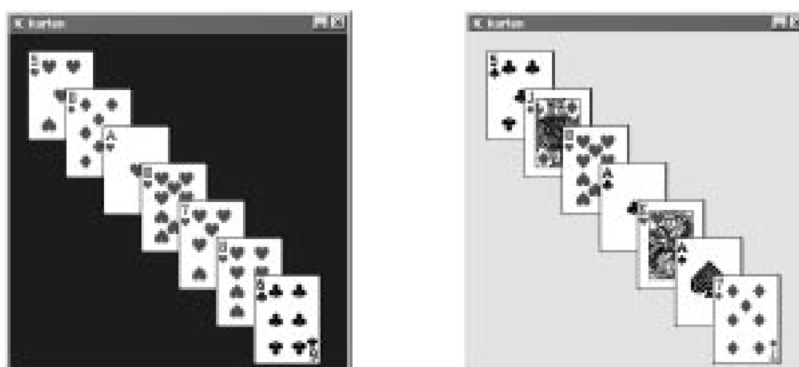


Abbildung 21.12: Unterschiedliche Anzeigen des Programms `karten.c` nach einem Tastendruck

```
#include <stdlib.h>
#include <time.h>
#include <graphics.h>

#define EINH  5

int  main( int argc, char *argv[] ) {
   int   i;
   void *karte;
   char  name[30];

   srand(time(NULL));  /* Zufallsgenerator initialisieren */

   initgraph( 360, 360 );
   setcolor(getmaxcolor());
   settextstyle( "Times", 48 );  /* Textstil festlegen */
   while (1) {
      cleardevice( rand()%getmaxcolor() );
      for ( i=0; i<7; i++ ) {
          sprintf( name, "%d.bmp", rand()%52+1 ); /* aus 52 Karten eine
                                                     zufaellig auswaehlen */
          loadimage( name, &karte );
          putimage( 20+i*40, 20+i*40, karte, COPY_PUT );
          freeimage( &karte );
      }
      if ( getch() == Key_Escape ) /* Abbruch mit Escape */
         break;
   }
   getch();
   closegraph();
   return(0);
}
```

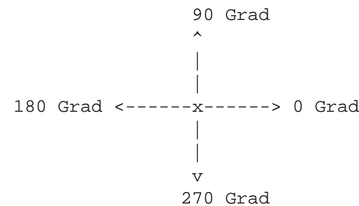
21.11 Kuchenstücke malen

`pieslice(int x, int y, int start, int end, int radius)`

`sector(int x, int y, int start, int end, int xradius, int yradius)`

`pieslice()` zeichnet ein kreisförmiges ausgefülltes „Kuchenstück“ und `sector()` zeichnet ein elliptisches ausgefülltes „Kuchenstück“. (`x,y`) ist in beiden Fällen der Mittelpunkt. Die Winkel `start` und `end` legen in beiden Fällen den Start- und Endpunkt des Kreisbogens fest. `pieslice()` zeichnet einen Kreisbogen mit Radius `radius` und `sector()` einen Ellipsenausschnitt mit `xradius` bzw. `yradius` als horizontaler bzw. vertikaler Radius. Die Endpunkte beider Bögen werden mit dem Kreismittelpunkt verbunden und bei beiden Funktionen werden die aktuelle Farbe, Füllfarbe und Füll-

muster verwendet. Die Angabe der Winkel `start` und `end` erfolgt in Grad, wobei entgegen dem Uhrzeigersinn gezählt wird:



Beispiel:
Das folgende Programm `piesarc.c` ist nicht nur ein Demonstrationsprogramm zu den Graphikroutinen `pieslice()` und `sector()`, sondern auch ein weiteres Beispiel zur Routine `arc()`. Dieses Programm `piesarc.c` blendet die in Abbildung 21.13 gezeigten Bilder ein.

```
#include <graphics.h>

int  main( int argc, char *argv[] )
{
   int links, oben, i,
       awink1=45, ewink1=135, awink2=250, ewink2=290,
       xrad=50, yrad=70, radius=50;

   initgraph( 640, 640 );

   /* Alle Fuellmuster durchlaufen */
   for (i=EMPTY_FILL ; i<=MAXFILLSTYLE ; i++) {
      links=i%4;
      oben=i/4;
      setfillstyle( i, BLACK );   /* Fuellstil festlegen */
      pieslice( links*150+100, oben*150+100, awink1, ewink1, radius );
      sector( links*150+100, oben*150+100, awink2, ewink2, xrad, yrad );
   }
   getch();

   /* Bogen malen */
   cleardevice( WHITE );
   setlinestyle( SOLID_LINE, 80 );
   setcolor( BLUE );
   arc( getmaxx()/2, getmaxy()/2, 90, 330, 150 );
   setcolor( GREEN );
   arc( getmaxx()/2, getmaxy()/2, -20, 45, 150 );
   getch();

   closegraph();
   return 0;
}
```

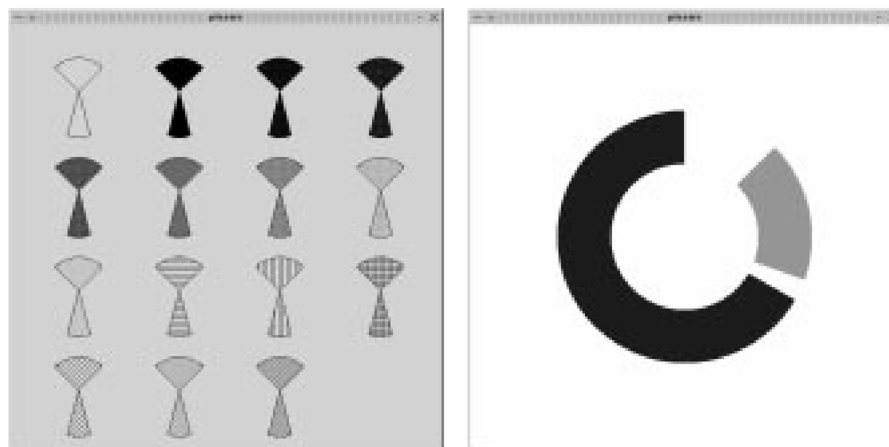


Abbildung 21.13: Anzeige des Programms `piesarc.c` (rechts: nach Tastendruck)

21.12 Graphikpaket neu bzw. anders einrichten

`setpalette(int farbe, int neufarbe)`

legt für die Farbe `farbe` eine neue Farbe (`neufarbe`) fest. Beispielsweise ändert `setpalette(RED,BLUE)` die Farbe `RED` so, dass ab nun bei Angabe von `RED` blau angezeigt wird.

`setrgbpalette(int farbe, int red, int green, int blue)`

legt für die Farbe `farbe` eine neue Farbe fest, welche sich aus den RGB-Komponenten `red`, `green` und `blue` zusammensetzt. Da für jede dieser drei Komponenten ein Werte zwischen `0` und `255` angegeben werden kann, hat man also `255 · 255 · 255 = 16581375` Farben zur Verfügung.

`setwritemode(int modus)`

legt fest, ob beim Zeichnen mit Routinen wie `line()`, `linerel()`, `lineto()`, `rectangle()`, `drawpoly()` usw. der vorherige Bildschirminhalt direkt überschrieben werden soll (`modus=COPY_PUT`) oder ob dabei eine XOR-Operation für die einzelnen Pixeln verwendet werden soll (`modus=XOR_PUT`). Mit `XOR_PUT` läßt sich die gezeichnete Linie durch einen weiteren Zeichenbefehl im selben Modus ohne Zerstörung bei einem schwarzen Hintergrund aus dem Fenster wieder entfernen.

Beispiel:
Das folgende Programm `farbstuf.c` malt in das eingeblendete Fenster eine große gefüllte Ellipse, deren Inhaltsfarbe langsam Farbstufen von blau, rot und grün durchläuft.

```
#include <graphics.h>

int  main( int argc, char *argv[] )
{
   int   r=0, g=0, b=0, z=0,
         mr=1, mg=1, mb=1;

   initgraph( 640, 480 );

   setcolor( LIGHTGRAY );
   setfillstyle( SOLID_FILL, LIGHTGRAY );

   do {
      setrgbpalette( LIGHTGRAY, r, g, b );
      fillellipse( getmaxx()/2, getmaxy()/2, 250, 150 );
      if ( z%2==0 ) {
         b += mb;
         if (b==0 || b==255)
            mb*=-1;
      }
      if ( z%3==0 ) {
         g += mg;
         if ( g==0 || g==255 )
            mg *= -1;
      }
      if ( z%5==0 ) {
         r += mr;
         if ( r==0 || r==255 )
            mr *= -1;
      }
      z = ++z%10;
   } while (!kbhit());
   closegraph();
   return 0;
}
```

21.13 Arbeiten mit mehreren Zeichenfenstern

`setviewport(int left, int top, int right, int bottom)`

richtet im aktuellen Graphikfenster ein neues Zeichenfenster ein: `setviewport()` legt mit (`left,top`) die linke obere Ecke und mit (`right,bottom`) die rechte untere Ecke eines neuen Zeichenfensters fest. `setviewport()` erwartet immer absolute Koordinaten. Z. B. erzeugt `setviewport(0,0,100,100)` ein neues Zeichenfenster mit 100 Pixeln Breite und Länge in der linken oberen Bildschirmecke, unabhängig davon, ob zuvor ein anderes Zeichenfenster gesetzt war oder nicht. Unabhängig vom aktuell gesetzten Zeichenfenster müssen bei allen Graphikroutinen weiterhin absolute Koordinaten für das ganze Graphikfenster angegeben werden.

`setviewport()` wird nur zum Löschen eines Teilfensters mit der nachfolgenden Routine benötigt.

`clearviewport(int farbe)`

Mit dieser Routine kann der Inhalt des momentan aktiven Graphik-Zeichenfensters gelöscht werden, wobei als Füllfarbe bei diesem Löschen die Farbe `farbe` verwendet wird.

`setviewport()` kann verwendet werden, um im Graphikmodus einen bereits früher an einer bestimmten Stelle ausgegebenen Text wieder zu überschreiben. Dazu muss zunächst `setviewport()` mit den Koordinaten des entsprechenden Bildausschnitts aufgerufen werden. Ein folgender Aufruf von `clearviewport()` löscht dann den dort stehenden Text. Nun kann dort neuer Text ausgegeben werden. Mit einem neuen `setviewport()`-Aufruf kann dann in das vorherige Fenster zurückgeschaltet werden.

21.14 Programmierung der Maus

Mit den folgenden Routinen ist eine Mausprogrammierung möglich:

`mouse_getpos(int *x, int *y)`

liefert die aktuelle Mausposition in (`*x,*y`).

`mouse_setpos(int x, int y)`

positioniert die Maus an die Position (`x,y`).

`mouse_left()`

liefert 1, wenn die linke Maustaste gerade gedrückt ist, und ansonsten 0.

`mouse_mid()`

liefert 1, wenn die mittlere Maustaste gerade gedrückt ist, und ansonsten 0.

`mouse_right()`

liefert 1, wenn die rechte Maustaste gerade gedrückt ist, und ansonsten 0.

`mouse_button()`

liefert 1, wenn eine beliebige Maustaste gerade gedrückt ist, und ansonsten 0.

`mouse_show()`

bewirkt, dass die Maus sichtbar wird.

`mouse_hide()`

bewirkt, dass die Maus nicht mehr sichtbar ist.

`mouse_visible()`

liefert 1, wenn die Maus gerade sichtbar ist, und ansonsten 0.

`mouse_setcursor(int mouseCursor)`

legt eine Mausform fest. Für `mouseCursor` kann dabei eine der in der folgenden Tabellen gezeigten Konstanten bzw. Werte angegeben werden:

Name	Wert	Form des Mauscursors
`arrowCursor`	0	Standard Pfeil-Cursor
`upArrowCursor`	1	Aufwärts-Pfeil
`crossCursor`	2	Kreuzform
`waitCursor`	3	Uhrform
`ibeamCursor`	4	Text-Cursor
`sizeVerCursor`	5	Vertikaler Strich mit Pfeilen an beiden Enden
`sizeHorCursor`	6	Horizontaler Strich mit Pfeilen an beiden Enden
`sizeBDiagCursor`	7	Diagonaler Strich (/) mit Pfeilen an beiden Enden
`sizeFDiagCursor`	8	Diagonaler Strich (\) mit Pfeilen an beiden Enden
`sizeAllCursor`	9	Kreuz mit Pfeilen an allen vier Enden
`blankCursor`	10	Unsichtbarer Cursor
`splitVCursor`	11	<-||->
`splitHCursor`	12	Horizontale Anzeige von `splitVCursor`
`pointingHandCursor`	13	Zeigende Hand

`mouse_setwindow(int left, int top, int right, int bottom)`

legt fest, dass die Maus nicht aus dem Teilfenster bewegt werden kann, dessen linke obere Ecke mit (`left,top`) und dessen rechte untere Ecke mit (`right, bottom`) festgelegt ist.

`mouse_inwindow(int left, int top, int right, int bottom)`

liefert 1, wenn sich der Mauscursor gerade im Fensterausschnitt befindet, dessen linke obere Ecke mit (`left,top`) und dessen rechte untere Ecke mit (`right, bottom`) festgelegt ist; ansonsten liefert diese Routine den Wert 0 zurück.

Beispiel:
Das folgende Programm `mausdemo.c` demonstriert einige dieser Routinen:

```
#include <graphics.h>

int  main( int argc, char *argv[] ) {
   int   x, y, c=0;
   char  string[100];
   initgraph( 640, 480 );
   do {
      mouse_getpos(&x,&y);
      setviewport( 0, 0, getmaxx(), 20 );
      clearviewport( LIGHTBLUE );
      sprintf( string, "x: %3d  y: %3d  Left: %d  Mid: %d  Right: %d ",
                        x, y, mouse_left(), mouse_mid(), mouse_right());
      outtextxy( 10, 5, getmaxx(), getmaxy(), string );
      setviewport( 0, 0, getmaxx(), getmaxy() );
      if ( mouse_mid() ) {
         switch(c) {
            case 0: c=1; mouse_setcursor(sizeAllCursor);       break;
            case 1: c=2; mouse_setcursor(pointingHandCursor);  break;
            case 2: c=0; mouse_setcursor(arrowCursor);         break;
```

```
        }
        while (mouse_left() || mouse_mid() || mouse_right())
            ;
        delay( 10 );
      }
   } while (!kbhit());
   getch();
   closegraph(); return 0;
}
```

Startet man dieses Programm, zeigt es oben immer die aktuelle Mausposition (x- und y-Koordinate) und zusätzlich zeigt es an, ob die linke, mittlere oder die rechte Maustaste momentan gedrückt wird. Drückt man die mittlere Maustasten, wird der Mauscursor in ein Kreuz mit Pfeilen an allen vier Enden verändert. Drückt man nochmals die mittlere Maustaste, nimmt der Mauscursor die Form einer zeigenden Hand an. Das Programm kann durch einen beliebigen Tastendruck beendet werden.

Beispiel:

Das folgende Programm `mauscursor.c` blendet ein Fenster ein, wie es in Abbildung 21.14 gezeigt ist. Bewegt der Benutzer den Mauscursor in die angezeigten Felder, wird ihm der zur jeweiligen Beschriftung gehörige Mauscursor angezeigt.

```
#include <graphics.h>

int main( int argc, char *argv[] ) {
  int  i, left, top, right, bottom, x, y;
  char name[100];
   initgraph( 800, 400 );
   settextstyle( "Times", 24 );
   settextjustify( HCENTER_TEXT, VCENTER_TEXT );
   setfillstyle( SOLID_FILL, WHITE );
   for ( i =0; i<= pointingHandCursor; i++ ) {
      if      ( i == arrowCursor)        sprintf( name, "arrowCursor" );
      else if ( i == upArrowCursor)      sprintf( name, "upArrowCursor" );
```

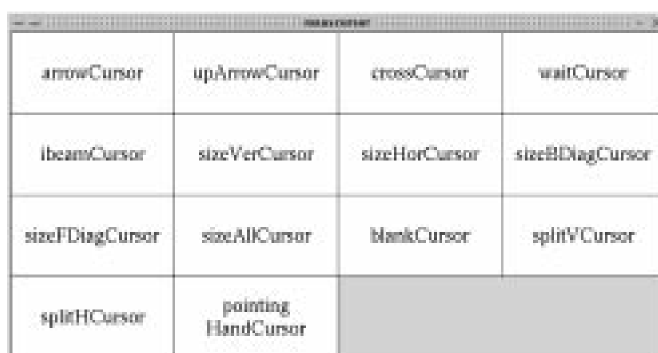


Abbildung 21.14: Interaktive Anzeige der verschiedenen Mauscursor

```
      else if ( i == crossCursor)        sprintf( name, "crossCursor" );
      else if ( i == waitCursor)         sprintf( name, "waitCursor" );
      else if ( i == ibeamCursor)        sprintf( name, "ibeamCursor" );
      else if ( i == sizeVerCursor)      sprintf( name, "sizeVerCursor" );
      else if ( i == sizeHorCursor)      sprintf( name, "sizeHorCursor" );
      else if ( i == sizeBDiagCursor)    sprintf( name, "sizeBDiagCursor" );
      else if ( i == sizeFDiagCursor)    sprintf( name, "sizeFDiagCursor" );
      else if ( i == sizeAllCursor)      sprintf( name, "sizeAllCursor" );
      else if ( i == blankCursor)        sprintf( name, "blankCursor" );
      else if ( i == splitVCursor)       sprintf( name, "splitVCursor" );
      else if ( i == splitHCursor)       sprintf( name, "splitHCursor" );
      else if ( i == pointingHandCursor) sprintf( name, "pointing\nHandCursor" );
      left   = i%4*200;
      top    = i/4*100;
      right  = (i%4+1)*200;
      bottom = (i/4+1)*100;
      bar3d( left, top, right, bottom, 0 );
      outtextxy( left, top, right, bottom, name );
   }
   while ( !kbhit() ) {
      mouse_getpos( &x, &y );
      mouse_setcursor( y/100*4 + x/200 );
   }
   closegraph();
   return 0;
}
```

21.15 Transformation mathematischer Koordinaten in das Graphikfenster

Eine mathematische Funktion kann oft Koordinaten besitzen, die außerhalb des Bildschirms liegen. In diesem Fall müssen die mathematischen Koordinaten in Bildschirmkoordinaten transformiert werden; siehe dazu auch Abbildung 21.15.
Wenn wir folgende Bezeichnungen einführen:

xb = x-Koordinate auf Bildschirm
yb = y-Koordinate auf Bildschirm
x = wirkliche x-Koordinate (in Funktion)
y = wirkliche y-Koordinate (in Funktion)

dann lassen sich aus obiger Abbildung folgende Beziehungen herleiten:

(1a) $\frac{xb}{xbmax} = \frac{x - xfmin}{xfmax - xfmin}$

(1b) $\frac{yb}{ybmax} = \frac{yfmax - y}{ymax - ymin}$

Die Maßstabsfaktoren ergeben sich dann wie folgt:

(2a) $xfaktor = \frac{xbmax}{xfmax - xfmin}$

(2b) $yfaktor = \frac{ybmax}{yfmax - yfmin}$

Diese beiden Faktoren `xfaktor` und `yfaktor` geben die Anzahl der Bildpunkte (Pixel) an, die einer Längeneinheit in x-Richtung bzw. in y-Richtung entspricht.

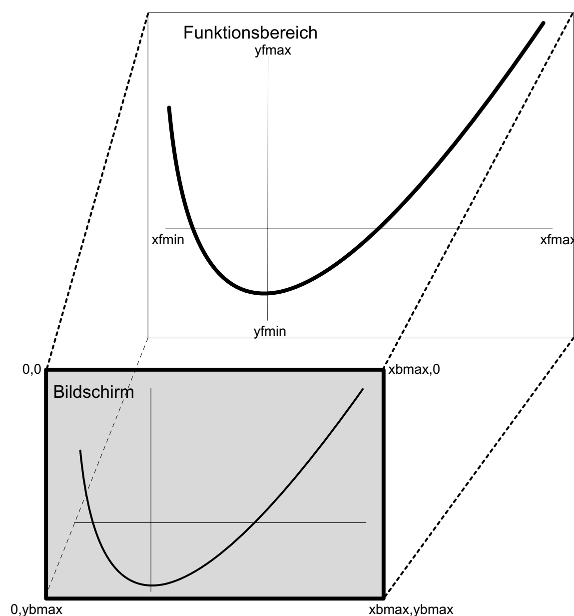


Abbildung 21.15: Transformation mathematischer Koordinaten in das Graphikfenster

Soll nun ein Punkt (x,y) einer Funktion auf den Bildschirm transformiert werden, so müssen die beiden folgenden Formeln angewendet werden:

(3a) $xb = (x - xfmin) \cdot xfactor$

(3b) $yb = (yfmax - y) \cdot yfaktor$

Dem Nullpunkt (0,0) im mathematischen Koordinatensystem entspricht der Bildschirmpunkt:

(4a) $xb_0 = -xfmin \cdot xfaktor$

(4b) $yb_0 = yfmax \cdot yfaktor$

Bei solchen Projektionen ist es notwendig, zunächst ohne Zeichnen alle Funktionswerte zu berechnen, um so `yfmax` und `yfmin` zu ermitteln. Erst nach dieser Ermittlung kann in einem erneuten Durchgang die Funktion auf den Bildschirm projeziert werden.

Beispiel:

Das folgende Programm `funkproj.c` berechnet die Funktion

$y = ax^3 + bx^2 + cx + d$

Die Koeffizienten `a, b, c` und `d` kann dabei der Benutzer ebenso eingeben, wie den kleinsten und den größten x-Wert.

```
#include <stdio.h>
#include <graphics.h>
#include <stdlib.h>
#include <time.h>

#define SW   0.01
```

```
int main( int argc, char *argv[] ) {
   double   a, b, c, d, x, xfmin, xfmax, y, yfmin, yfmax,
                        xb, yb, xfaktor, yfaktor, xbnull, ybnull;
    srand(time(NULL));
       /*------- Eingaben des Benutzers --------------------------------------*/
    printf("Graphische Darstellung der Funktion\n"
           "          3     2\n"
           "   y =  ax  + bx  + cx + d\n\n");
    printf("a? ");  scanf("%lf", &a);
    printf("b? ");  scanf("%lf", &b);
    printf("c? ");  scanf("%lf", &c);
    printf("d? ");  scanf("%lf", &d);
    printf("\n\nxmin? ");  scanf("%lf", &xfmin);
    printf("xmax? ");      scanf("%lf", &xfmax);
       /*------- Funktion berechnen, um yfmin und yfmax zu ermitteln ----------*/
    yfmin = yfmax = a*xfmin*xfmin*xfmin + b*xfmin*xfmin + c*xfmin + d;
    for (x=xfmin+SW ; x<=xfmax ; x+=SW) {
       y = a*x*x*x + b*x*x + c*x + d;
       if (y>yfmax)
          yfmax=y;
       if (y<yfmin)
          yfmin=y;
    }
       /*------- Massstab-Faktoren berechnen ----------------------------------*/
    initgraph( 640, 480 );
    xfaktor = getmaxx() / (xfmax-xfmin);
    yfaktor = getmaxy() / (yfmax-yfmin);
       /*------- Ausgabe der Koordinaten-Achse --------------------------------*/
    xbnull = -xfmin * xfaktor;
    ybnull =  yfmax * yfaktor;
    line(0, (int)ybnull, (int)xbnull, (int)ybnull);
    line((int)xbnull, 0, (int)xbnull, (int)ybnull);
    line(getmaxx(), (int)ybnull, (int)xbnull, (int)ybnull);
    line((int)xbnull, getmaxy(), (int)xbnull, (int)ybnull);
       /*------- Koordinaten-Beschriftung -------------------------------------*/
    outtextxy(0, (int)(ybnull-10), getmaxx(), getmaxy(), "|%-10.1lf", xfmin );
    outtextxy(getmaxx()-90, (int)(ybnull-10), getmaxx(),getmaxy(), "%10.1lf|",xfmax);
    outtextxy((int)(xbnull-3), 0, getmaxx(), getmaxy(),"-%-10.1lf", yfmax );
    outtextxy((int)(xbnull-3), getmaxy()-10, getmaxx(),getmaxy(), "-%-10.1lf",yfmin);
       /*------- Ausgabe der Funktion -----------------------------------------*/
    for (x=xfmin ; x<=xfmax ; x+=SW) {
       y = a*x*x*x + b*x*x + c*x + d;
       xb = (x - xfmin) * xfaktor;
       yb = (yfmax -y) * yfaktor;
       putpixel((int)xb, (int)yb, BLACK );
    }
    getch();
    closegraph();
    return(0);
}
```

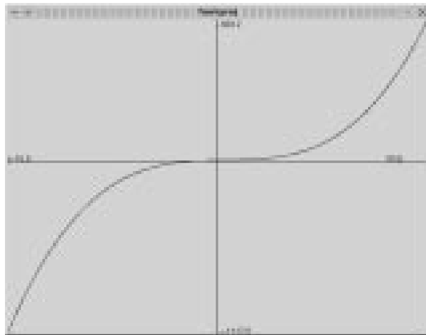


Abbildung 21.16: Plot der Funktion zum ersten Ablaufbeispiel

1. Ablaufbeispiel:

```
..........
a? 1 ↵
b? -1 ↵
c? 2 ↵
d? 3 ↵
xmin? -10 ↵
xmax? 10 ↵
```

   Abbildung 21.16 zeigt das Graphikfenster für dieses Ablaufbeispiel.

2. Ablaufbeispiel:

```
..........
a? 0 ↵
b? 0 ↵
c? 3 ↵
d? -5 ↵
xmin? -10 ↵
xmax? 20 ↵
```

   Abbildung 21.17 zeigt das Graphikfenster für dieses Ablaufbeispiel.

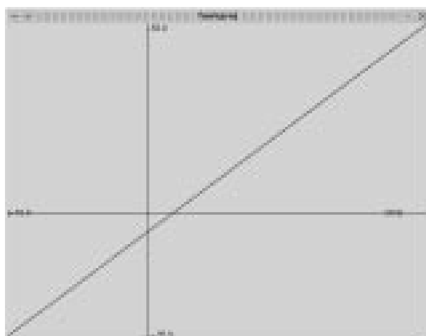


Abbildung 21.17: Plot der Funktion zum zweiten Ablaufbeispiel

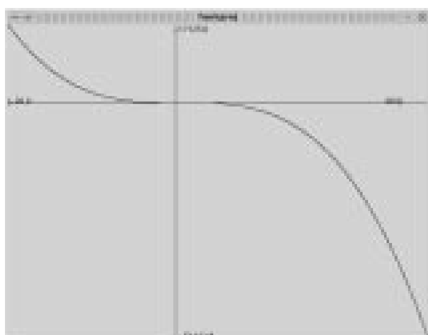


Abbildung 21.18: Plot der Funktion zum dritten Ablaufbeispiel

3. Ablaufbeispiel:

```
..........
a? -2 ⮐
b? 3 ⮐
c? 4 ⮐
d? 5 ⮐
xmin? -20 ⮐
xmax? 30 ⮐
```

Abbildung 21.18 zeigt das Graphikfenster für dieses Ablaufbeispiel.

21.16 Kurzer Einblick in Fraktale und Chaos

Hier wird ein kurzer Einblick in die neue, faszinierende und fächerübergreifende Wissenschaft *Fraktale und Chaostheorie* gegeben, da es sich hierbei um ein Gebiet handelt, in dem man sich sehr interessante Graphikbilder erzeugen lassen kann.
Wohl niemals zuvor hat ein Forschungsgebiet der Mathematik soviel Aufmerksamkeit in der Öffentlichkeit erregt wie die Fraktale. Ein Fraktal ist eine geometrische Figur, in der sich das Motiv in stets kleinerem Massstab wiederholt. Früher galten diese Gebilde eher als mathematische Spielereien, aber mit dem Aufkommen von schneller Computer-Graphik stieg das Interesse der Wissenschaftler und der Öffentlichkeit an den faszinierenden Bilder, die Fraktale liefern. Heute weiss man, dass Fraktale in vielen aktuellen Bereichen der Wissenschaften zu wichtigen neuen Erkenntnissen geführt haben. Bei einigen Übungen wird deshalb hier nicht nur die Ästhetik des Fraktals in den Vordergrund gestellt, sondern auch auf ihre praktische Verwendung eingegangen.
Der „Vater“ der Fraktalen ist der französisch-amerikanische Mathematiker *Benoit. B. Mandelbrot*, der durch die Veröffentlichung seines Buchs *“The Fractal Geometry of Nature“* die Begeisterung für Fraktale auslöste. Fraktale sind durch eine Art „Selbstähnlichkeit“ gekennzeichnet, wobei ein Bild, das Motiv, sich in immer kleinerem Massstab fortlaufend wiederholt. Ein Fraktal bietet dem geistigen Auge eine Möglichkeit, die Unendlichkeit zu schauen.

Abbildung 21.19: Kochsche Schneeflocke zu Beginn (links) und nach einer Transformation (rechts)

Ein gutes Beispiel ist die *Kochsche Schneeflocke*, die nach dem schwedischen Mathematiker *Helge Koch* benannt wurde, der diese Form 1904 als erster beschrieb. In einer der nachfolgenden Übungen sollten Sie diese Schneeflocke auch selbst programmieren. Hier wird sie zunächst zur Klärung des Begriffs *„Selbstähnlichkeit"* herangezogen.
Um eine Kochsche Kurve zu konstruieren, stelle man sich ein Dreieck vor, dessen Seiten jeweils einen Meter lang sind, wie dies links in Abbildung 21.19 gezeigt ist.
Nun denke man sich eine bestimmte Transformation: eine bestimmte, wohl definierte, leicht zu wiederholende Reihe von Regeln. Bei der Kochschen Flocke sind die Transformationsregeln wirklich sehr einfach:

> *Man füge in der Mitte einer jeden Seite ein neues Dreieck hinzu, das in der Form identisch ist, aber nur ein Drittel so groß. Das Ergebnis ist ein Davidstern. Statt aus drei Segmenten von einem Meter besteht der Umriss der Form nun aus 12 Segmenten von 1/3 Meter (33,333... cm). Statt drei Ecken gibt es nun 6; siehe auch rechts in Abbildung 21.19.*

Nun nehme man jede der zwölf Seiten und wiederhole die Transformation, indem man ein wiederum kleineres Dreieck im mittleren Drittel einer jeder Kante anfügt; siehe auch links in Abbildung 21.20. Führt man eine weitere Tarnsformation durch, erhält man ein Bild wie es rechts in Abbildung 21.20 gezeigt ist.
Diese Transformationen kann man nun unendlich fortsetzen, wobei der Umriss immer detaillierter wird. Die Gestalt ähnelt schliesslich einer Art idealer Schneeflocke, wie sie in Abbildung 21.21 gezeigt ist.

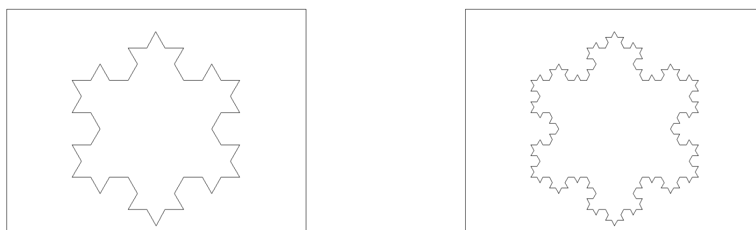


Abbildung 21.20: Kochsche Schneeflocke nach zwei (links) und drei (rechts) Transformationen

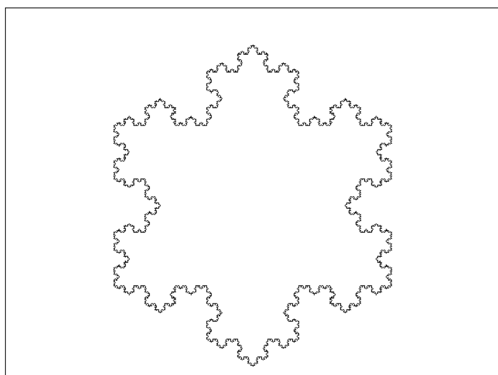


Abbildung 21.21: Kochsche Schneeflocke nach *n* Transformationen

Diese Kochsche Schneeflocke weist einige interessante Eigenschaften auf. Zunächst einmal ist sie eine fortlaufende Schleife, die sich nie überschneidet, da die Seitenlänge der neu hinzukommenden Dreiecke stets so kurz ist, dass ein Zusammenstoß vermieden wird. Ferner fügt jedes neu hinzukommende Dreieck der Kurve ein kleines Areal hinzu, obwohl der Gesamtflächeninhalt begrenzt bleibt und den des ursprünglichen Dreiecks nicht wesentlich übersteigt. Dennoch ist die Kurve (Länge des Umfangs) unendlich lang:

Umfang des ersten Dreiecks: 3
Umfang des Davidsterns (1. Transformation): $\frac{4}{3}+\frac{4}{3}+\frac{4}{3}$
Umfang der 2. Transformation: $(\frac{4}{3})^2+(\frac{4}{3})^2+(\frac{4}{3})^2=\frac{16}{9}+\frac{16}{9}+\frac{16}{9}$
Umfang der 3. Transformation: $(\frac{4}{3})^3+(\frac{4}{3})^3+(\frac{4}{3})^3$
....
Umfang der n. Transformation: $(\frac{4}{3})^n+(\frac{4}{3})^n+(\frac{4}{3})^n=3\cdot(\frac{4}{3})^n$

Dieses paradoxe Ergebnis – eine unendliche Strecke innerhalb eines begrenzten Raums – verwirrte so manchen Mathematiker, der um die Jahrhundertwende darüber nachdachte. Die Kochsche Kurve half im übrigen die Antwort auf die Frage zu finden:
Wie lange ist die englische Küste?
Diese Frage hatte sich der englische Meteorologe *Lewis Fry Richardson* gestellt. Mandelbrot griff auf die Veröffentlichungen von Richardson zurück und ihm fiel auf, dass die Länge der englischen Küste vom Massstab der verwendeten Landkarte abhängig ist. Eine Karte, auf der 1 cm hundert Kilometer in der Wirklichkeit entspricht (Maßstab 1:10 000 000) ist nun einmal ungenauer als eine Wanderkarte, auf der 1 cm einem Kilometer in der Wirklichkeit entspricht (Maßstab 1:100 000). Je mehr Details sichtbar werden, desto länger wird die Küstenlinie.
Dieser kurze Einblick in die Fraktale soll hier genügen. Bei den Übungen in begleitenden Übungsbuch wird immer, wenn notwendig, noch einige Theorie zu den entsprechenden fraktalen Strukturen gegeben.

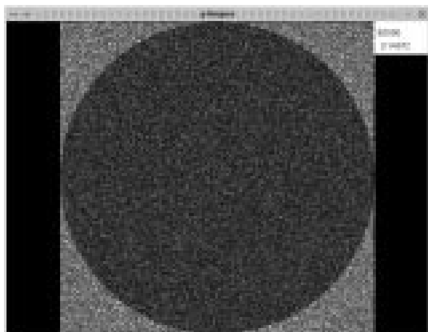


Abbildung 21.22: Ermitteln der Zahl π durch das Fallen von Regentropfen

21.17 Übungen

21.17.1 Ermitteln der Zahl PI mit Regentropfen

Erstellen Sie ein C-Programm `piregen.c`, das zunächst ein rotes Quadrat im Graphikfenster malt. In dieses Quadrat soll es dann einen genau passenden Kreis malen. Dieser Kreis soll z. B. blau ausgemalt sein. Nun lassen Sie zufällig Regentropfen in das Quadrat fallen. Das Fallen eines Regentropfens kann dadurch angezeigt werden, dass an der entsprechenden (zufällig ermittelten Position) ein Pixel mit einer bestimmten Farbe gezeichnet wird. Dabei sollte für außerhalb des Kreises gefallene Regentropfen eine andere Farbe verwendet werden, wie für Tropfen im Kreis. Zählt man nun immer die Regentropfen, die in den Kreis fielen und teilt diese Zahl durch die Gesamtzahl der bisher gefallenen Regentropfen, so ergibt sich eine Näherung für $\frac{\pi}{4}$. Nimmt man also das Ergebnis mal 4, so hat man eine Näherung für die Zahl π. Es wäre schön, wenn Ihr Programm alle 1 000 Tropfen die bisher ermittelte π-Näherung als Zwischeninformation am oberen rechten Bildschirmrand anzeigen würde; siehe auch Abbildung 21.22. Wie viele Regentropen fallen sollen, muss der Benutzer vor dem Beginn einer Simulation eingeben. Nach der Beendigung einer Simulation wird dem Benutzer die ermittelte Zahl π angezeigt, wobei er auch gefragt wird, ob er eine weitere Simulation wünscht; siehe auch Abbildung 21.23.

Abbildung 21.23: Anzeigen der ermittelten Zahl π mit Frage, ob weitere Simulation gewünscht

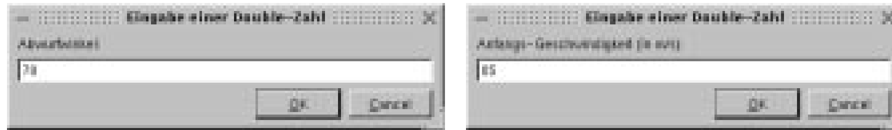


Abbildung 21.24: Eingabe des Abwurfwinkels und der Abwurfgeschwindigkeit

21.17.2 Basketball spielen

Erstellen Sie ein C-Programm `basket.c`, das zunächst einen Basketballkorb an einer zufällig ermittelten Position in der rechten Hälfte des Graphikfensters zeichnet und am linken unteren Rand des Graphikfensters einen Ball hüpfen läßt. Dieser Ball soll am Boden die Startposition einnehmen, wenn der Benutzer mit der Eingabe des Abwurfwinkels (siehe links Abbildung 21.24) beginnt. Nach der Eingabe dieses Winkels muss der Benutzer noch die Anfangsgeschwindigkeit, mit welcher der Ball abgeworfen werden soll, in Meter pro Sekunde eingeben (siehe rechts in Abbildung 21.24).
Nach diesen Eingaben soll die Flugbahn des Basketballs simuliert werden; siehe auch Abbildung 21.25. Wird der Korb getroffen, so soll dies dem Benutzer gemeldet

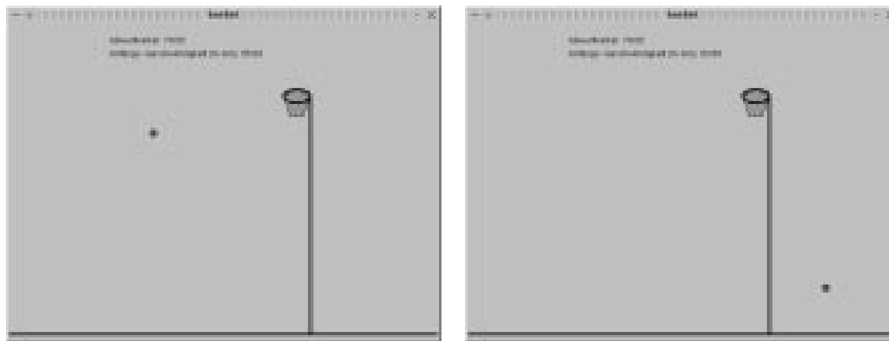


Abbildung 21.25: Simulation der Flugbahn des Basketballs

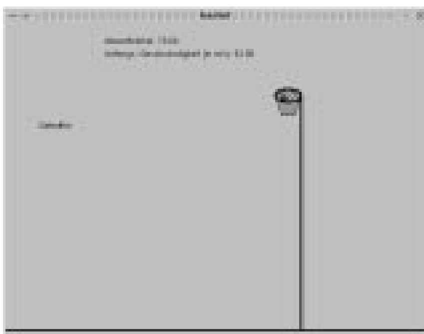


Abbildung 21.26: Anzeige bei einem Treffer im Basketballspiel

werden und der Ball dann senkrecht zu Boden fallen; siehe auch Abbildung 21.26. Grundsätzlich gilt, dass beim Auftrefen des Balls auf dem Boden, dieser mit dem gleichen Winkel wieder abspringt, mit dem er aufsprang, allerdings nur mit halber Geschwindigkeit. Während des Flugs soll keinerlei Reibung (wie z. B. Luftwiderstand) berücksichtigt werden.

21.17.3 Die Kochsche Schneeflocke

Erstellen Sie ein C-Programm `koch.c`, das die Kochsche Schneeflocke auf den Bildschirm malt. Der Benutzer soll dabei vor dem Malen eingeben können, ob

- Farbe zu verwenden ist,
- bei den Transformationen das vorherige Bild zu löschen ist und
- die Kurve ausgefüllt oder nicht ausgefüllt zu zeichnen ist.

Die Abbildungen 21.27, 21.28 und 21.29 zeigen die Folge von eingeblendeten Kochkurven, wobei die nächste Transformation immer durch einen Tastendruck ausgelöst wird. Der Schwierigkeitsgrad dieses Programms ist sehr hoch, weshalb Ihnen vielleicht der folgende kleine Tip weiterhilft: ***Das Zeichnen der Linien kann unter Zuhilfenahme des 4er-Systems erfolgen.***

Abbildung 21.27: Ausgangspunkt und erste Transformation

Abbildung 21.28: Zweite und dritte Transformation

Abbildung 21.29: Vierte und letzte Transformation

21.17.4 Zeichnen und Füllen von Quadraten mit der Maus

Erstellen Sie ein C-Programm `quadzeic.c`, das Ihnen erlaubt, Quadrate am Bildschirm zeichnen und mit Farbe füllen zu lassen. Das Zeichnen eines Quadrates beginnt dabei mit dem Drücken der linken Maustaste. Solange die linke Maustaste nicht losgelassen wird, wird der Umriß des bisherigen Quadrates, das sich von der

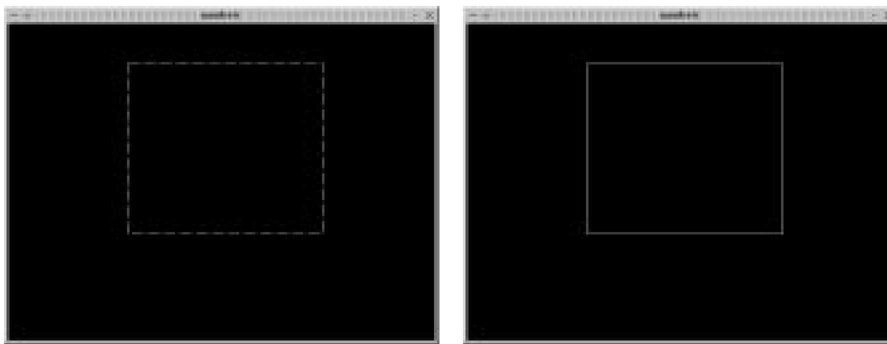


Abbildung 21.30: Zeichnen von Quadraten mit der linken Maustaste

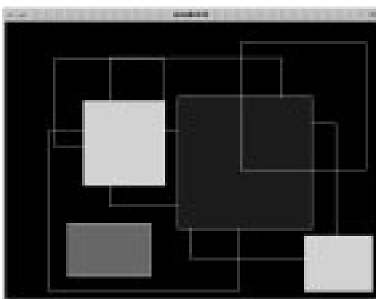


Abbildung 21.31: Füllen des zuletzt gezeichneten Quadrats mit zufälliger Farbe bei Klick auf rechte Maustaste

Startposition bis zur momentanen Position erstreckt, immer gestrichelt angezeigt; siehe auch links in Abbildung 21.30. Erst nachdem der Benutzer die linke Maustaste losläßt, wird das Quadrat wirklich auf den Bildschirm gezeichnet, und zwar mit einer durchgehenden Linie; siehe auch rechts in Abbildung 21.30. Der Benutzer soll beliebig viele Quadrate zeichnen können. Drückt der Benutzer die rechte Maustaste, dann soll das zuletzt gezeichnete Quadrat mit einer zufälligen Farbe gefüllt werden; siehe auch Abbildung 21.31. Erst wenn der Benutzer eine Tastatureingabe durchführt, soll sich das Programm beenden.