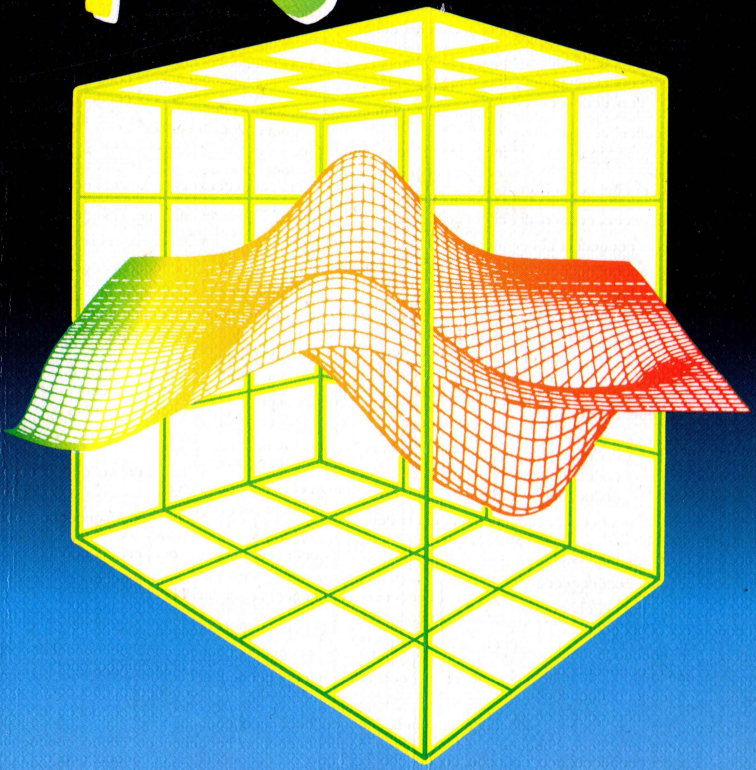
Das neue Supergrafikbuch

Trapp
Weltner



AMIGA

DATA BECKER

Trapp
Weltner

Das neue Supergrafikbuch zum Amiga

DATA BECKER

1. Auflage 1989

ISBN 3-89011-345-1

DATA BECKER GmbH
Merowingerstr. 30
4000 Düsseldorf

Text verarbeitet mit Word 4.0, Microsoft
Ausgedruckt mit Hewlett Packard LaserJet II
Druck und Verarbeitung Graf und Pflügge, Düsseldorf

Wichtiger Hinweis:
Die in diesem Buch wiedergegebenen Verfahren und Programme werden ohne Rücksicht auf die Patentlage mitgeteilt. Sie sind ausschließlich für Amateur- und Lehrzwecke bestimmt und dürfen nicht gewerblich genutzt werden.

Alle technischen Angaben und Programme in diesem Buch wurden von den Autoren mit größter Sorgfalt erarbeitet bzw. zusammengestellt und unter Einschaltung wirksamer Kontrollmaßnahmen reproduziert. Trotzdem sind Fehler nicht ganz auszuschließen. DATA BECKER sieht sich deshalb gezwungen, darauf hinzuweisen, daß weder eine Garantie noch die juristische Verantwortung oder irgendeine Haftung für Folgen, die auf fehlerhafte Angaben zurückgehen, übernommen werden kann. Für die Mitteilung eventueller Fehler sind die Autoren jederzeit dankbar.

Vorwort

Die Grafikfähigkeiten des Amiga sind phantastisch. Das ist unbestritten, und Sie haben Lobesbekenntnisse dieser Art sicherlich zur Genüge vernommen. Wie aber macht man sich diese Wundermaschine gefügig? Wie programmiert man eigene Grafikprogramme, und was geht eigentlich im Inneren des Amiga vor sich?

Die Antworten auf diese und viele hundert weitere Probleme finden sich in diesem Buch. Und zwar eingebettet in ein Konzept, das für den Einsteiger genauso interessant ist wie für den erfahrenen Profi.

In den ersten Kapiteln wird der Einsteiger behutsam in die faszinierende Grafikwelt des Amiga eingeführt. Hier lernen Sie anhand zahlloser dokumentierter AmigaBASIC- und GFA- BASIC-Programme, wie sich Grafiken erstellen lassen, wie die Spezialbefehle des BASIC effizient genutzt werden, auf welche Weise bewegliche Objekte geschaffen werden etc. etc.

Fortgeschrittene kommen im zweiten Teil auf ihre Kosten. Sie erforschen Stück für Stück das Multi-Tasking-Grafikbetriebssystem des Amiga und lernen die vielfältigen Grafik-Routinen des Betriebssystems kennen, die normalerweise im Kickstart-ROM verborgen sind. Besonderer Knüller: Alle wichtigen Datenstrukturen (wie Window, Screen, Viewport und andere) werden detailgenau erklärt und zum leichten Nachschlagen in Tabellenform aufgelistet. Am Ende dieses Teils werden auch BASIC-Programmierer ihre Grafiken auf den Drucker ausgeben, Grafiken im 64-farbigen Halfbrite- oder 4096-farbigen HAM-Modus erstellen und den Coprozessor programmieren können. Dinge also, die dem BASIC-Programmierer vorher unmöglich waren!

Ob Sie das "Supergrafik"-Buch zur behutsamen Einführung, als potenten Problemlöser oder als kompetentes Nachschlagewerk nutzen - die vor Ihnen liegenden Seiten gehören in unmittelbare Reichweite!

Die Autoren

Inhaltsverzeichnis

1. Grafik auf dem Amiga

Der Amiga ist in vielerlei Hinsicht ein besonderer Computer. Seine tollen Eigenschaften haben schon viele überzeugt. Dabei sind die besten Argumente für den Amiga wohl seine Grafikfähigkeiten.

Beim Amiga wird man - besonders als ehemaliger Homecomputer-Besitzer - immer wieder durch die große Anzahl der Befehle und wegen der hohen Geschwindigkeit der Grafikbefehle überrascht und fasziniert sein. Während einige Befehle ziemlich komplex oder nur im Verbund zu gebrauchen sind, gibt es auch ein paar ganz einfache Befehle, mit denen Punkte, Linien und Kreise auf den Bildschirm gezeichnet werden.

Im Gegensatz zu vielen anderen Computern werden Textgrafik und Hi-Res-Grafik auf den gleichen Bildschirm ausgegeben. Man kann Grafik und Text also ohne Probleme miteinander verbinden. Deshalb brauchen wir auch keinen extra Grafikbildschirm zu öffnen, sondern können gleich ans Eingemachte gehen und die Grafikbefehle ausprobieren.

1.1 Erste Gehversuche: Mit PSET Punkte setzen

Die kleinste Einheit einer Grafik ist der Punkt. Denn jede Computergrafik, von einem ganzen Bild bis zu vereinzelten Linien und Kreisen, setzt sich ja nur aus vielen kleinen Bildschirmpunkten zusammen.

Der Befehl, mit dem man Punkte setzt, ist einfach und kurz:

```
PSET (10,20)
```

setzt beispielsweise einen Punkt in einer Entfernung von elf (10+1) Bildschirmpunkten vom rechten und einundzwanzig

(20+1) Bildpunkten vom oberen Fensterrahmen. (Beachten Sie bitte, daß die Adressierung der Bildschirmzeilen und -spalten mit Null beginnt.)

1.1.1 Mit Maus und PSET ein einfaches Zeichenbrett

Mit dem Befehl PSET kann man jeden beliebigen Punkt innerhalb des Ausgabefensters setzen. Das folgende Programm ist dafür ein recht gutes Beispiel. Immer wenn Sie die linke Maustaste drücken, wird dort, wo sich der empfindliche Punkt des Mauszeigers (der Punkt an der Spitze des Mauszeigers) befindet, ein Punkt gesetzt. Damit hat man ein primitives Malprogramm. Aber auch die ganz großen Grafikprogramme werden schließlich nach diesem Prinzip bedient.

```
REM Zeichnen mit der Maus

PRINT "Jetzt koennen Sie mit der Maus zeichnen"
WHILE INKEY$=""

IF MOUSE(0)<>0 THEN
  x=MOUSE(1)
  y=MOUSE(2)
  PSET (x,y)
END IF

WEND
```

Wie man sieht, reichen wenige Programmzeilen für dieses kleine Malprogramm aus. Die MOUSE-Funktionen dienen natürlich zur Kontrolle der Maus. Wenn die linke Maustaste gedrückt wurde, ist MOUSE(0) ungleich null. Nun können wir die Koordinaten des Mauszeigers mit MOUSE(1) (das ist der x-Wert) und MOUSE(2) (für den y-Wert) lesen.

1.1.2 Punkte löschen

Wenn man Punkte setzen kann, muß man sie ja sinnvollerweise auch wieder entfernen können. Beim AmigaBASIC geschieht dies

ähnlich wie das Setzen von Punkten. Es gibt einen Befehl, der ähnlich klingt und die gleiche Schreibweise hat wie unser PSET-Befehl. Der Befehl lautet:

```
PRESET (x,y)
```

Wie Sie sehen, unterscheiden sich beide Befehle in ihrer Schreibweise nur um zwei Buchstaben. Im Programm könnte das ganze dann folgendermaßen aussehen:

```
REM Demo fuer PRESET

a=200
b=400
c=1

zurueck:
  FOR x=a TO b STEP c
    PSET (x,100)
    PRESET (x-40*c,100)
  NEXT x
  SWAP a,b
  c=-c
GOTO zurueck
```

Bei diesem Programm wird eine Linie von 40 Pixeln auf den Bildschirm gezeichnet. Es scheint, als würde sich die ganze Linie bewegen. Dabei wird lediglich an die Linie vorne ein neuer Punkt dazu gesetzt und hinten einer gelöscht.

1.1.3 So wird die Farbenpracht auf den Bildschirm gebracht

Alle bisherigen Programme haben den Amiga noch nicht sehr gefordert. Sie ließen sich auch ohne weiteres auf andere Computer übertragen. Doch das wird nicht bei allen Programmen so einfach sein: z.B., wenn es um die Benutzung von Farben geht, denn eine Besonderheit dieses Computers ist seine Farbdarstellung. Damit ist nicht das wunderschöne Grau der Tastatur, sondern sind die 4096 verschiedenen Farben gemeint, die man auf dem Bildschirm sichtbar machen kann. Es verfügen wohl nur wenige Computer über eine ähnlich große Farbpalette. Zwar werden wir Ihnen an späterer Stelle zeigen, wie sich bis zu

64 oder sogar alle 4096 Farben gleichzeitig von BASIC aus nutzen lassen, doch wollen wir uns hier mit den normalerweise maximal vorhandenen 32 Farben begnügen. Die farbigen Punkte setzt man fast genauso, wie wir es schon in den vorherigen Programmen mit einfarbigen Punkten gemacht haben. Dabei wird nur der Wert eines Farbregisters, das die gewünschte Farbe enthält, hinter den Befehl gehängt:

```
PSET (10,20),2
```

Die Farbe des Punktes soll aus dem zweiten Farbregister genommen werden. Anfangs befindet sich dort immer die Farbe Schwarz, also wird der Punkt schwarz.

Wenn Sie jetzt mit dem PSET-Befehl auch die restlichen versprochenen 31 Farben auf den Bildschirm bringen wollen, werden Sie recht schnell auf Schwierigkeiten stoßen. Spätestens, wenn Sie das fünfte oder ein höheres Register ausprobieren, gibt der Computer eine Fehlermeldung aus. Woran liegt das? Nun, da man nicht immer alle Farben benötigt, spart der Amiga lieber Speicherplatz, indem man nicht auf jedem Screen 32 Farben benutzen kann. Denn je mehr Farben man benutzt, desto mehr Speicher benötigt man (Näheres dazu später).

Also müssen wir erst einmal einen neuen Screen öffnen (ein vom Computer erzeugter Bildschirm, der dann gleichzeitig mit dem Workbench-Screen besteht und hinter diesem versteckt ist oder diesen seinerseits bedeckt). Wenn man einen neuen Screen öffnet, kann man bestimmte Parameter wie Breite, Höhe, Modus und die Tiefe festlegen. Die Tiefe gibt an, wie viele Farben man verwenden kann. Als Tiefe kann man Werte zwischen 1 und 5, bei bestimmten Modi auch nur maximal 4 angeben. Die Anzahl der Farben errechnet man mit 2 hoch Tiefe. Mit dem Modus gibt man die Auflösung des Bildschirms an. Es gibt vier verschiedene Modi.

Modus	Auflösung
1	320*200
2	640*200
3	320*400
4	640*400

In den meisten unserer Programme werden wir den Modus eins benutzen. Breite und Höhe des Screens dürfen nicht über die Werte der Auflösung hinausgehen.

Keine Ausgabe auf den Bildschirm geht direkt auf den Screen. Wenn wir den Bildschirm geöffnet haben, müssen wir noch ein Fenster öffnen. In diesem Fenster werden dann automatisch alle Texte und Grafiken ausgegeben.

```
REM 32-Farben DEMO

REM Screen oeffnen
SCREEN 1,320,200,5,1
REM Bildschirm oeffnen
WINDOW 2,"Farbtopf",(0,0)-(311,185),16,1

FOR y= 0 TO 186
  FOR x= 0 TO 311
    PSET (x,y), (x+y) MOD 32
  NEXT x
NEXT y

WHILE INKEY$="": WEND

REM beides wieder schliessen
WINDOW CLOSE 2
SCREEN CLOSE 1
```

Auf dem Bildschirm sehen Sie alle 32 möglichen Farben. Am Programmende werden das Fenster und der neue Screen wieder geschlossen, denn sie werden nicht mehr gebraucht.

Neben diesem recht anspruchslosen Farben-Demo können wir natürlich auch sehr wirkungsvolle Muster zeichnen lassen, z.B.:

```
REM Farbendemo

SCREEN 1,320,200,5,1
WINDOW 2,"Farbdemo",(0,0)-(62,62),16,1

FOR m=0 TO 31
  FOR x=-m TO m
    FOR y=-m TO m
      PSET (31+x,31+y),((ABS(x) AND ABS(y))+32-m) MOD 32
    NEXT y
  NEXT x
NEXT m

WHILE INKEY$="": WEND

WINDOW CLOSE 2
SCREEN CLOSE 1
```

oder

```
REM Pyramide

SCREEN 1,320,200,5,1
WINDOW 2,,(0,0)-(20,10),16,1

FOR y=0 TO 19
  FOR x=0 TO 19
    f1=ABS(x -10)
    f2=ABS(y -10)
    IF f1<f2 THEN SWAP f1,f2
    PSET (x,y),31-f1
  NEXT x
NEXT y

WHILE INKEY$="" : WEND

WINDOW CLOSE 2
SCREEN CLOSE 1
```

1.1.4 Noch ein Wort zu PSET und PRESET

Wir haben oben gesagt, daß PRESET die Punkte löscht, die PSET setzt. Das ist nur solange richtig, wie bei PRESET keine Farbangabe gemacht wird:

```
PSET (100,100)
PRESET (100,100)
```

Diese Zeilen entsprechen dem oben genannten Beispiel. Der Punkt wird gesetzt und sofort wieder gelöscht.

Dagegen ist das Resultat der folgenden Zeilen ein weißer Punkt auf dem Bildschirm, der Punkt wird also nicht gelöscht:

```
PSET (100,100),1
PRESET (100,100),1
```

Sobald ein Farbregister im Befehl genannt wird, sind beide Befehle gleich, und es ist Geschmackssache, welchen der beiden Befehle Sie verwenden.

Statt die Punkte mit PRESET (x,y) zu löschen, kann man auch PSET (x,y),0 schreiben. Wieso gibt es aber überhaupt zwei praktisch gleiche Befehle? Die Default-Farbe (das ist die Farbe, in der die Punkte gezeichnet werden, wenn vom Benutzer keine Farbe genannt wird) von PSET entspricht immer der Vordergrundfarbe, die von PRESET immer der Hintergrundfarbe. Das trifft auch zu, wenn die Hinter- oder Vordergrundfarbe geändert wird. Der Benutzer braucht sich also nicht um diese Werte zu kümmern, während er, wenn er PSET auch zum Löschen benutzt, sich die Hintergrundfarben selber merken muß.

Außerdem erhöht die Verwendung von beiden Befehlen die Übersichtlichkeit im Programm. Man sieht sofort, wo Punkte gesetzt oder gelöscht werden.

1.1.5 Die Umkehrung von PSET: Der POINT-Befehl

Das Wort Umkehrung ist hier so zu verstehen, daß, statt Punkte zu setzen, abgefragt werden kann, ob ein Punkt gesetzt ist und welche Farbe er hat. Das macht nämlich der POINT-Befehl:

```
PRINT POINT(x,y)
```

Gibt entweder die Farbe an oder zeigt an, daß der Punkt (x,y) nicht im aktuellen Ausgabefenster liegt. Im ersten Fall wird das

entsprechende Farbregister angegeben, im zweiten ist das Ergebnis von POINT (x,y) gleich -1.

Das nächste Programm erstellt Grafiken, die anfangs Ähnlichkeiten mit Schnittmustern oder Stadtplänen haben. Nach einer Weile ist das ganze Window ausgemalt, und diese Ähnlichkeiten verschwinden. Das Bild scheint schließlich aus zufällig gesetzten Punkten entstanden zu sein.

```
REM Schnittmuster

SCREEN 1,320,200,5,1
WINDOW 2,,(0,0)-(80,80),16,1
RANDOMIZE TIMER
x=40
y=40

Steigung:
dx=INT (RND*5)-2
dy=INT (RND*5)-2
IF dx=0 AND dy=0 THEN Steigung
IF ABS(dx)>ABS(dy) THEN
  st=ABS(dx)
ELSE
  st=ABS(dy)
END IF

WHILE INKEY$=""
  IF POINT (x+dx,y+dy)=-1 THEN Steigung
  FOR i= 1 TO st
    x=x+dx/st
    y=y+dy/st
      PSET (x,y),POINT(x,y) MOD 31+1
  NEXT i
WEND

WINDOW CLOSE 2
SCREEN CLOSE 1
```

Das Schema, nach dem dieses Bild aufgebaut wurde, ist recht einfach: Ein Punkt wandert über den Bildschirm. Dabei wird bei jedem Bildpunkt das Farbregister abgefragt und um eins erhöht. Stößt der wandernde Punkt an den Fensterrahmen, ändert er seine Richtung.

Beenden Sie das Programm, indem Sie eine beliebige Taste drücken.

1.1.6 Relative Adressierung

Bei PSET und fast allen noch folgenden Grafik-Befehlen des AmigaBASIC gibt es zwei unterschiedliche Adressierungsarten: Die erste läßt sich sehr gut mit Hausnummer und Straßennamen vergleichen. Wenn man sie auf einem Brief angibt, wird der Brief immer an der gewünschten Adresse ankommen. Dies entspricht der gebräuchlicheren Art, die wir bis jetzt ausschließlich in unseren Programmen verwendet haben:

```
PSET (20,30)
```

Die Werte 20 und 30 sind die absoluten Koordinaten des zu setzenden Punktes und entsprechen seiner Zeile und Spalte. Deshalb wird diese Adressierungsart "absolute Adressierung" genannt.

Die zweite Adressierungsart nennt sich "relative Adressierung", denn die angegebenen Koordinaten geben noch nicht die Zeile und Spalte an. Im praktischen Leben tritt sie beispielsweise auf, wenn man jemandem den Weg beschreibt: "Gehen Sie drei Blocks geradeaus und biegen dann links ab...". Im Computer heißt es: Gehe von deinem Standpunkt drei Pixel nach rechts und zwei nach unten. Der Standpunkt ist dort, wo sich der Grafik-Cursor befindet; nämlich dort, wo zuletzt ein Punkt auf den Bildschirm ausgegeben wurde. Als Kennzeichnung für relative Adressierung wird das Wort STEP vor die Klammer gesetzt:

```
PSET STEP (3,2)
```

Leider kann man die Koordinaten des Grafik-Cursors nicht abfragen. Dafür kann man ihn aber sehr einfach setzen, ohne einen Punkt auf dem Bildschirm zu setzen.

```
v=POINT (x,y)
```

Nach dieser Zeile befindet sich der Grafik-Cursor in Zeile y und Spalte x, denn der Cursor wird nicht nur beim Setzen von Punkten geändert, sondern auch bei der Abfrage. "v" ist eine beliebige, aber unbenutzte Variable.

Am Programmstart befindet sich der Grafik-Cursor immer in der Mitte des Ausgabefensters.

Bei relativer Adressierung gibt man nicht immer den gleichen Punkt an, denn sobald sich der Grafik-Cursor bewegt, wird ein ganz anderer Punkt gesetzt. Dafür hat man aber, wenn man mehrere Punkte setzt, immer die gleichen Abstände zwischen den Punkten, unabhängig vom Grafik-Cursor. Das haben wir uns beim folgenden Programm zunutze gemacht. Das Programm gleicht unserem allerersten Programm, dem kleinen Malprogramm. Aber statt eines Punktes werden auf Tastendruck immer gleich mehrere Punkte gezeichnet.

```
REM Relative Adressierung

WHILE INKEY$=""

IF MOUSE(0)<>0 THEN
  x=MOUSE(1)
  y=MOUSE(2)
  PSET (x,y)
  PSET STEP (10,10)
  PSET STEP(-10,10)
  PSET STEP (-10,-10)
  PSET STEP (10,0)
END IF

WEND
```

Der erste Punkt wird absolut angegeben. Die anderen vier Punkte werden relativ angegeben und haben deshalb immer den gleichen Abstand voneinander.

1.2 Der LINE-Befehl

Der LINE-Befehl bietet dem Benutzer zwei in der Schreibweise ähnliche, aber in ihren Auswirkungen vollkommen unterschied-

liche Möglichkeiten. Zum einen kann man, wie es der Name schon verrät, Linien zeichnen. Außerdem kann man mit ihm aber auch Kästchen zeichnen. Letzteres stellt praktisch einen zweiten Befehl dar, und deshalb wird er später gesondert behandelt.

Die Syntax des LINE-Befehls ist ähnlich der von PSET. Allerdings braucht man immer noch zwei Punkte, um eine Linie zu bestimmen.

```
LINE (20,10)-(200,100),2
```

Es wird dann eine schwarze (Farbregister 2) Linie vom Punkt (20,10) zum Punkt (200,100) gezeichnet.

1.2.1 Der Moiré-Effekt

Immer, wenn mehrere Linien dicht nebeneinander oder übereinander gezeichnet werden, ist der Moiré-Effekt auf dem Bildschirm zu beobachten. Mit ihm kann man erstaunliche Bilder konstruieren. Schauen Sie sich doch einmal die von diesem Programm erstellten Grafiken an.

```
REM Moiré-Gitter

a=182              'groesse des quadrats

FOR s=1 TO 10
  CLS

  FOR i=0 TO a STEP s
    LINE (140,1)-(140+2*a,i),2
    LINE (140,1)-(140+2*i,a),2
    LINE (140+2*a,a)-(140,i),2
    LINE (140+2*a,a)-(140+2*i,1),2
  NEXT i

  WHILE INKEY$="": WEND

NEXT s
```

Es werden von zwei Ecken des Quadrats Linien an die jeweils gegenüberliegenden Seiten gezogen. Der Moiré-Effekt entsteht in der Gegend der Verbindungslinie beider Ecken, denn dort werden viele Linien gezeichnet.

Der Moiré-Effekt tritt aber auch schon dann auf, wenn nur von einem Punkt Linien ausgehen. Diesen Effekt kann man noch verstärken, indem man die Linien abwechselnd in zwei verschiedenen Farben zeichnet:

```
REM Moiré-Demo II

xmax=618   'Groesse des Ausgabefensters
ymax=186

COLOR 1,2  'Hintergrund schwarz
start:
CLS
xm=INT(RND*xmax)  'Koordinaten des Mittelpunkts
ym=INT(RND*ymax)

FOR i=0 TO ymax
  LINE (xm,ym)-(0,i),i MOD 2+1
  LINE (xm,ym)-(xmax,i),i MOD 2+1
NEXT i
FOR i=0 TO xmax
  LINE (xm,ym)-(i,0),i MOD 2+1
  LINE (xm,ym)-(i,ymax),i MOD 2+1
NEXT i

WHILE INKEY$="": WEND

GOTO start
```

Wenn man sich die Grafik anschaut, die dieses Programm erstellt, ist es nicht leicht zu erkennen, daß sie auf so einfache Weise entstanden ist.

1.2.2 Quix, das Linienbündel

Kommen wir doch noch einmal zur Geschwindigkeit des LINE-Befehls zurück: In diesem Programm lassen wir den Quix über den Bildschirm jagen. Der Quix besteht aus mehreren Linien, die alle mehr oder weniger parallel liegen. Die Bewegung des

Quix entsteht, weil Linien am Ende des Quix gelöscht werden und dafür vorne angefügt werden. Allerdings lassen sich Anfang und Ende des Quix schlecht bestimmen, denn die neuen Linien, die vorne angefügt werden, übernehmen die Koordinaten der zuvor gezeichneten Linie. Diese Koordinaten werden zufällig etwas verändert, wodurch sich ständig die Orientierung und die Länge der Linien verändern.

```
REM Quix

DEFINT a-z
SCREEN 1,320,200,5,1
WINDOW 2,"Quix",(0,0)-(297,185),31,1
RANDOMIZE TIMER

a=20
DIM x(1,a),y(1,a)
x(0,0)=150
y(0,0)=100
x(1,0)=170
y(1,0)=100

WHILE INKEY$=""
  FOR z=0 TO a
    LINE (x(0,z),y(0,z))-(x(1,z),y(1,z)),0

    FOR i= 0 TO 1
neux: x(i,z)=ABS(x(i,alt)+RND*20-10)
      IF x(i,z)>WINDOW(2) THEN neux
neuy: y(i,z)=ABS(y(i,alt)+RND*20-10)
      IF y(i,z)>WINDOW(3) THEN neuy
    NEXT i

    f1=f1 MOD 31 +1
    LINE (x(0,z),y(0,z))-(x(1,z),y(1,z)),f1
    alt=z
  NEXT z
WEND

WINDOW CLOSE 2
SCREEN CLOSE 1
```

Während der Quix sich über den Bildschirm bewegt, kann man ihn einfangen, indem man mit der Maus auf das Größen-Gadget an der rechten unteren Ecke des Fensters zeigt und mit gedrückter Maustaste die Maus verschiebt. Die Koordinaten des Quix können nie größer als die jeweilige Fenstergröße werden.

Die Größe des aktuellen Ausgabefensters kann man mit WINDOW(2) (die Breite) und WINDOW(3) (für die Höhe) abfragen.

1.2.3 Funktionsplotter

Der Funktionsplotter ist eine der vielen mathematischen Anwendungen der Grafik. Er kann eine große Hilfe bei der Betrachtung von Funktionen sein. Es macht aber auch Spaß, mit Parametern und Funktionen zu experimentieren und nach interessanten Kurven zu suchen. Dieser Funktionsplotter ist recht umfangreich und bietet dem Benutzer sehr viel Komfort. Beispielsweise braucht man sich nicht um die Funktionswerte zu kümmern, denn die Kurve wird immer so gezeichnet, daß das Fenster in seiner vollen Höhe ausgenutzt wird.

```
REM Funktionsplotter

DEFDBL x,y,m,f
DIM y(618) 'maximale Anzahl der Funktionswerte
x1=-10: x2=10 'Wertebereich
funktion=1    'erste Funktion
koordinaten=1 'Koordinatenkreuz an

MENU 1,0,1,"Dienst"
MENU 1,1,1,"Funktion zeichnen"
MENU 1,2,1,"Koordinateneingabe"
MENU 1,3,1,"Koordinatenkreuz aus"
MENU 1,4,1,"Ende"
MENU ON
MENU 2,0,1,"Funktion"

DEF FNy1(x)=SIN(x)/(x^2+1)
a$(1)="y=sinx/(x^2+1)"
MENU 2,1,1,a$(1)

DEF FNy2(x)=SIN(x)*10-1/x+x^2
a$(2)="y=sin(x)*10-1/x+x^2
MENU 2,2,1,a$(2)

DEF FNy3(x)=SIN(1/x)/x
a$(3)="y=sin (1/x) /x"
MENU 2,3,1,a$(3)

DEF FNy4(x)=(EXP(x)-1)/(EXP(x)+1)
a$(4)="y=(e^x-1)/(e^x+1)"
MENU 2,4,1,a$(4)
```

```
WINDOW 1, a$(1),,23
GOSUB Rechnen

Warte:        SLEEP
              ON MENU GOSUB Verzweigung
              GOTO Warte

Dienst:       ON MENU(1) GOSUB Rechnen, Eingabe, Kreuz, Quit
              RETURN

Verzweigung:  ON MENU(0) GOTO Dienst
              funktion=MENU(1)
              WINDOW 1, a$(funktion)

Eingabe:      WINDOW 2,"Koordinaten-Eingabe",(0,0)-(250,9),16
              INPUT "Anfangswert : ";x1
              INPUT "Endwert     : ";x2
              IF x2<x1 THEN SWAP x1,x2
              WINDOW CLOSE 2

Rechnen:      IF x1=x2 THEN Eingabe
              breite=WINDOW(2)
              WINDOW 2,"Bitte Geduld! Ich rechne",(0,0)-(300,0),0
              min=0
              max=0
              ON ERROR GOTO Fehler
              FOR i=0 TO breite
                fwert=x1+(x2-x1)*i/breite
                ON funktion GOSUB F1,F2,F3,F4
                IF y(min)>y(i) THEN min=i
                IF y(max)<y(i)OR y(max)=9999 THEN max=i
Weiter:       NEXT i
              ON ERROR GOTO 0
              min=y(min)
              max=y(max)
              WINDOW CLOSE 2
              GOSUB Titel
Zeichnen:     CLS
              hoehe = WINDOW(3)-8
              IF koordinaten=1 THEN
                IF min<=0 AND max=>0 THEN
                  h=hoehe+min*hoehe/(max-min)
                  LINE (0,h)-(breite,h),2
                END IF
                IF x1<=0 AND x2=>0 THEN
                  b=-x1*breite/(x2-x1)
                  LINE (b,0)-(b,hoehe),2
                END IF
              END IF
              IF min=max THEN      ' Wenn min=max, dann zeichne eine
                                     Gerade'
```

```
            IF max=9999 THEN  ' Funktionswert nicht definiert '
              CLS
            ELSE
              LINE (0,hoehe/2)-(breite,hoehe/2)
            END IF
          END IF
          j=0
          WHILE (y(j)=9999 AND j<618) ' suche ersten definierten
                                        Funktionswert'
            j=j+1
          WEND
          IF j=618 THEN RETURN
          PSET (j,hoehe-(y(j)-min)*hoehe/(max-min))
          FOR i=j+1 TO breite
            IF y(i)<>9999 THEN
              IF flag THEN
                PSET (i,hoehe-(y(i)-min)*hoehe/(max-min))
                flag=0
              ELSE
                LINE -(i,hoehe-(y(i)-min)*hoehe/(max-min))
              END IF
            ELSE
              flag=1
            END IF
          NEXT i
          RETURN

Fehler:   y(i)=9999
          RESUME Weiter

F1:       y(i)=FNy1(fwert)
          RETURN
F2:       y(i)=FNy2(fwert)
          RETURN
F3:       y(i)=FNy3(fwert)
          RETURN
F4:       y(i)=FNy4(fwert)
          RETURN

Titel:    MENU 3,0,1,MID$(STR$(x1),1,5)+"<x<"+MID$(STR$(x2),1,5)
          MENU 4,0,1,MID$(STR$(min),1,5)+"<y<"+MID$(STR$(max),1,5)
          RETURN

Kreuz:    IF koordinaten=1 THEN
            koordinaten=0
            MENU 1,3,1,"Koordinaten an"
          ELSE
            koordinaten=1
            MENU 1,3,1,"Koordinaten aus"
          END IF
          GOTO Zeichnen
```

```
Quit:       WINDOW 1,"plotter",(0,0)-(617,184),31
            MENU RESET
            END
```

Vielleich fragen Sie sich, was der Funktionsplotter mit dem LINE-Befehl zu tun hat, da eine Funktion ja nicht aus Linien, sondern aus Punkten besteht. Unser Programm arbeitet so, daß es genau für jede Spalte einen Punkt errechnet. Häufig besteht zwischen benachbarten Punkten eine große Höhendifferenz. Wenn man nur diese Punkte zeichnen würde, könnte man kaum von einer Kurve sprechen, da die Punkte nicht immer miteinander verbunden wären. Die Verbindung schafft uns der LINE-Befehl. Er zeichnet also in so einem Fall nur fast senkrechte Linien.

1.2.3.1 Funktionsweise und Menüsteuerung

Das Programm besitzt zwei gewöhnliche und zwei Pseudo-Menüs. Letztere haben keine Unterpunkte und bestehen nur aus der Überschrift. Zweck dieser Menüs ist es, an den Benutzer Informationen zu übergeben. Das erste der beiden Pseudo-Menüs gibt die Grenzen des sichtbaren Ausschnitts der Kurve in X-Richtung und das zweite den maximalen und minimalen darstellbaren Y-Wert des sichtbaren Funktionsteils an. Diese beiden Menüs erscheinen erst nach dem Zeichnen und ändern sich bei jeder Werteänderung entsprechend.

Das erste der beiden gewöhnlichen Menüs mit dem Namen "Dienst" enthält vier Unterpunkte. Der erste Punkt zeichnet die Kurve neu. Man kann ihn aufrufen, wenn man die Größe des Fensters verändert hat. Außerdem kann ein neuer Wertebereich eingegeben, das Koordinatenkreuz sichtbar oder unsichtbar gemacht und viertens das Programm beendet werden.

Mit dem zweiten Menü kann der Benutzer zwischen vier verschiedenen mathematischen Funktionen wählen. Leider ist es mit BASIC nicht möglich, Funktionen vom Benutzer im Programm-Modus eingeben zu lassen. Dagegen ist es recht unkompliziert, vier Funktionen fest vorzugeben. Wenn Sie andere Funktionen

untersuchen möchten, brauchen Sie nur vor dem Programmstart die Funktionen am Anfang des Programms zu ändern. Außerdem sollten Sie auch den Text von a$(n) ändern. Dieser Text erscheint im Menü und bildet den Window-Titel, wenn diese Funktion ausgewählt wurde.

1.2.3.2 Scaling

Der Benutzer kann durch Vergrößern oder Verkleinern direkt auf die Größe und das Format der Kurve einwirken. Wenn man die Größe des Fensters verstellt, wird die Kurve beim nächsten Zeichnen dementsprechend gestreckt oder gestaucht.

Unter Scaling versteht man, daß die Kurve immer so dargestellt wird, daß das ganze Fenster ausgenutzt wird. Die Einheiten an x- und y-Achse sind nicht gleich, sondern hängen von dem Verhältnis von Höhe zu Breite ab. Durch die Streckung der Kurve können manchmal verwirrende Eindrücke über deren Verlauf entstehen. So wirkt z.B. eine fast gerade verlaufende Kurve viel steiler.

Diese Darstellungsweise hat aber den Vorteil, daß der Benutzer sich weniger um den Kurvenverlauf kümmern muß.

Nähere Auskünfte über die Kurve liefert, wie oben schon erwähnt, die Menüleiste.

1.2.4 Rechtecke zeichnen

Wie oben schon angekündigt, kann man mit dem LINE-Befehl neben Linien auch Rechtecke zeichnen, und zwar mit einem einzigen LINE-Befehl. Die Schreibweise für beide Aufgaben des LINE-Befehls ist fast identisch. Sollen Kästchen gezeichnet werden, hängt man an den Befehl einfach ein ",B" an.

```
LINE (20,10)-(200,100),2,B
```

Diese Zeile zeichnet ein unausgefülltes schwarzes Rechteck auf den Bildschirm. Das erste Koordinatenpaar gibt die linke obere Ecke an, das zweite die rechte untere Ecke. Daraus sieht man leicht, daß man nur gleichmäßige Rechtecke mit zum Fensterrahmen parallelen Seiten zeichnen kann. Aber gerade diese Rechtecke kann man sehr oft gebrauchen.

Neben ungefüllten kann man auch ausgefüllte Rechtecke zeichnen. Statt ",B" muß das Anhängsel ",BF" heißen.

```
LINE (30,10)-(300,100),3,BF
```

Wenn statt des Farbparameters eine Leerstelle angegeben wird, erscheint das Viereck in der Vordergrundfarbe.

Auch bei diesem Befehl können wir wieder einmal einen kleinen Geschwindigkeitstest ausführen. Unser Testprogramm ähnelt dem Boxes-Demo der Workbench. Es werden ausgefüllte Rechtecke mit zufälligen Farben und Koordinaten gezeichnet:

```
REM Geschwindigkeitstest
REM des LINE Befehls

WHILE INKEY$=""
  x=WINDOW(2)
  y=WINDOW(3)
  LINE (RND*x,RND*y)-(RND*x,RND*y),INT(RND*4),BF
WEND
```

Auch wenn dies nur ein ganz einfaches Programm ist, kann es doch begeistern und "Computer-Heiden" vom Amiga überzeugen. Die Geschwindigkeit, mit der die Rechtecke ausgefüllt werden, ist sehr hoch, sogar so hoch, daß man nicht mehr mitzählen kann, wie viele Fenster gemalt werden. Dabei handelt es sich ja nicht nur um kleine Rechtecke. Auch bei Rechtecken mit mehr als tausend Bildpunkten ist keine Zeitverzögerung zu erkennen.

1.2.5 Relative Adressierung beim LINE-Befehl

Genau wie bei PSET lassen sich die Koordinaten beim LINE-Befehl relativ angeben, und zwar bei beiden Anwendungen.

Bei relativer Adressierung erfolgt die erste Koordinatenangabe relativ zum Grafik-Cursor und die zweite relativ zum Anfangspunkt der Linie.

Es brauchen nicht beide Koordinaten relativ angegeben zu werden. Die folgende Zeile zeichnet eine Linie von (30,20) bis (20,120).

```
LINE (30,20)- STEP(-10,100)
```

Wenn man direkt von der Stelle, an der sich der Grafik-Cursor gerade befindet, weiter zeichnen möchte, kann man die erste Koordinatenangabe weglassen, wie beim folgenden Programm, das mehrere ineinander verschachtelte Quadrate ausgibt:

```
REM Geschachtelte Vierecke

SCREEN 1,320,200,2,1
WINDOW 2,"gedrehte Vierecke",(0,0)-(311,185),16 ,1
COLOR 2,1
CLS

d=5                ' Abstand (1-10)
b=60              'Breite des ersten Vierecks

FOR x=0 TO 311 STEP b
  FOR y=0 TO 185 STEP b
    x1=x: x2=x+b
    y1=y: y2=y
    FOR a= 0 TO .7 STEP ATN(d/b)  'a< PI/4
      IF ((x+y)/b)MOD 2=0 THEN      'Drehrichtung
        LINE (x1,y1)-(x2,y2)
        LINE -(2*x+b-x1,2*y+b-y1)
        LINE -(2*x+b-x2,2*y+b-y2)
        LINE -(x1,y1)
      ELSE
        LINE (2*x+b-x1,y1)-(2*x+b-x2,y2)
        LINE -(x1,2*y+b-y1)
        LINE -(x2,2*y+b-y2)
        LINE -(2*x+b-x1,y1)
      END IF
```

```
      x1=x1+COS(a)*d  'Berechnung des naechsten Vierecks
      x2=x2-SIN(a)*d
      y1=y1+SIN(a)*d
      y2=y2+COS(a)*d
    NEXT a
  NEXT y
NEXT x

WHILE INKEY$="": WEND

WINDOW CLOSE 2
SCREEN CLOSE 1
```

Wenn das Programm fertig ist, erkennt man die einzelnen Vierecke kaum noch. Sie sind in dem großen Muster aufgegangen. Der Effekt wird verstärkt, weil benachbarte Vierecke sich gegenläufig eindrehen.

Nur bei der jeweils ersten Linie eines Quadrates oder einer anderen geometrischen Figur müssen beide Koordinatenpaare angegeben werden.

Beim LINE-Befehl gibt das hintere Koordinatenpaar bei beiden Anwendungen die Stelle an, an der sich nach dem Befehl der Grafik-Cursor befindet. Wenn man das beachtet, kann man Schreibarbeit sparen: Wird der LINE-Befehl zum Zeichnen von Kästchen benutzt, dann kann man jeweils die x- und die y-Koordinaten untereinander vertauschen. Damit kann man selbst bestimmen, an welchem der vier Eckpunkte sich der Grafik-Cursor nach dem Zeichnen befindet. Das kann man für die relative Adressierung ausnutzen.

Das folgende Beispielprogramm zeichnet eine zufällige Balkengrafik, wie sie sehr häufig zu Statistikzwecken gebraucht wird. Unsere Balken sollen dreidimensional erscheinen. Deshalb zeichnen wir um den Balken noch einen Schatten, begrenzt durch mehrere Linien.

```
REM Balkengrafik

RANDOMIZE TIMER
SCREEN 1,320,200,4,1
WINDOW 2,"Balkengrafik",,31,1
```

```
FOR i=0 TO 7
  x=30+i*37
  y=INT(RND*160)+1
  LINE (x,180-y)-STEP(6,-6),i+1
  LINE -STEP (0,y),i+1
  LINE -STEP (-6,6),i+1
  LINE -STEP (-20,-y),i+1,bf
  LINE -STEP (6,-6),i+1
  LINE -STEP (20,0),i+1
NEXT i

WHILE INKEY$="": WEND

WINDOW CLOSE 2
SCREEN CLOSE 1
```

Außer dem ersten Koordinatenpaar jedes Balkens werden alle Koordinaten relativ angegeben, auch die des Kastens. Auf diese Weise braucht man selbst nur ganz wenig Koordinaten zu errechnen, was Zeit spart und die Übersicht fördert.

1.3 Der CIRCLE-Befehl

Bei einem Computer, der auch für professionelle Grafikanwendungen geeignet ist, darf im BASIC natürlich ein so elementarer Befehl wie CIRCLE nicht fehlen.

In der Schule lernt man, daß man den Mittelpunkt und den Radius braucht, um einen Kreis zu zeichnen. So ist das auch beim AmigaBASIC. Probieren wir es aus:

```
CIRCLE (200,100),100
```

Fast genauso können wir auch farbige Kreise zeichnen. Den Wert für das Farbregister hängen wir einfach hinten an:

```
CIRCLE (200,100),100,2
```

Bei beiden Kreisen fallen uns zwei Dinge auf:

1. Die Figuren bilden häufig keine Kreise, sondern eher Ellipsen.

2. Der Radius ist zumindest in der Höhe nicht gleich der Anzahl der Bildpunkte, denn dann dürfte bei einem Radius von 100 der Kreis nicht mehr ganz im Fenster zu sehen sein.

Beide Punkte gehören natürlich zusammen und hängen von dem sogenannten Bildverhältnis ab.

1.3.1 Das Bildverhältnis

Sicher haben Sie schon mal an den Rädchen Ihres Monitors gedreht. Da gibt es hinten einen Regler, mit dem man die vertikale Höhe verstellen kann. (Wenn man auf die Rückwand guckt, ist es der dritte von rechts.) Mit diesem Regler könnten wir den ersten Punkt unserer Beobachtung beheben, was aber unter Umständen den Nachteil hat, daß nicht mehr das ganze Window auf dem Bildschirm zu sehen ist. Und außerdem erscheint spätestens, wenn Sie eine andere Bildschirmauflösung wählen, wieder ein "Ei" auf dem Bildschirm.

Aber es geht auch anders. Denn man kann an den CIRCLE-Befehl einen Parameter übergeben, der das Verhältnis von Breite zu Höhe widerspiegelt, eben das Bildverhältnis. Und so wird's gemacht:

```
CIRCLE (100,100),100,,,,2
```

Zwischen dem Radius und dem Bildverhältnis stehen vier Kommas, denn wir haben drei Werte ausgelassen: die Farbe, die wir schon erwähnt haben, und zwei Winkelangaben, denen wir uns erst später zuwenden wollen.

Man muß das Bildverhältnis aber nicht nur dafür nutzen, einen perfekten Kreis zu zeichnen. Man kann natürlich auch gewollt Ellipsen formen, wie bei unserem nächsten Programm, das gleich

ganz viele Ellipsen und Kreise auf einmal auf den Bildschirm zeichnet. Die äußere Form der mit diesem Programm erzeugten Gebilde reicht vom Kreis über Karos zu vierzackigen Sternen. Alle Figuren sind mit mehreren Kreisbögen durchzogen:

```
REM Ellipsen

SCREEN 1,320,200,2,1
WINDOW 2,,(0,0)-(311,185),16,1

FOR g= 0 TO 80 STEP 5
  CLS
  FOR f= .0001 TO 1 STEP .1
    CIRCLE (100,100),(80-g*f),,,,f
    CIRCLE (100,100),(80-g*f),,,,1/f
  NEXT f
  WHILE INKEY$="": WEND
NEXT g

WINDOW CLOSE 2
SCREEN CLOSE 1
```

Bei diesem Programm haben wir den kleineren Bildschirm mit 320*200 Punkten Auflösung gewählt, weil da das Bildverhältnis annähernd eins ist. Deswegen können die breiten, waagerechten Ellipsen fast genauso erzeugt werden wie die länglichen, horizontalen. Der einzige Unterschied besteht darin, daß das Bildverhältnis einmal gleich dem Wert f und damit kleiner als eins ist, und das zweite Mal gleich 1/f und damit immer größer als eins ist. Die beiden Ellipsen, die bei einem Durchgang gezeichnet werden, sind deshalb auch bis auf ihre Orientierung identisch: Eine ist länglich, die andere breit.

Das Bildverhältnis und der Radius hängen bei allen Figuren voneinander ab. Je größer der Wert f, desto runder werden die Kreise, und gleichzeitig wird auch der Radius kleiner. Um wieviel der Radius kleiner wird, ist bei jeder Figur unterschiedlich.

1.3.1.1 Animierte Grafik mit CIRCLE

Das nächste Programm enthält einfache animierte Grafik. Es simuliert einen springenden Ball auf dem Bildschirm. Bei jedem

Aufprall verformt sich der Ball leicht, als wäre er aus Gummi. Auf diese Weise bekommt er Schwung und springt wieder hoch.

```
REM Springender Ball

SCREEN 1,320,200,2,1
WINDOW 2,,(0,0)-(100,100),16,1

WHILE INKEY$=""
  FOR i=0 TO 3.14 STEP .08
    f=1
    y=70*SIN(i)
    IF y<10 THEN f=.5+y/20
    CLS
    CIRCLE (50,100-y),20,1,,,f
  NEXT
WEND

WINDOW CLOSE 2
SCREEN CLOSE 1
```

Das Bildverhältnis ist abhängig von der Höhe des Balles, die wir mit der Sinusfunktion errechnen. Wenn der Ball tiefer als der Radius des Kreises ist, wird f (das Bildverhältnis) verändert, und der Ball verformt sich.

Das Prinzip unserer Animation ist einfach: Ein Kreis wird gezeichnet, dann ein neuer berechnet, und bevor der neue Kreis gezeichnet wird, wird der ganze Bildschirm gelöscht. Es ist wesentlich unkomplizierter und schneller, den ganzen Bildschirm zu löschen, als den Kreis mit CIRCLE und Hintergrundfarbe zu übermalen. Wenn der ganze Bildschirm gelöscht wird, brauchen die Kreispunkte nicht erst errechnet zu werden, was schon einmal eine Menge Zeit spart. Außerdem übernimmt der sogenannte Blitter, ein besonderer Grafik-Coprozessor, diese Arbeit, wenn wir CLS verwenden. Bildschirme zu löschen stellt ihn nicht vor allzu große Probleme. Er kann Flächen mit bis zu einer Millionen Bildpunkten in nur einer Sekunde füllen, und bei uns sind es ja viel weniger.

Im nächsten Programm zeichnen wir eine Art Drahtmodell einer Schachfigur. Auf Tastendruck wird die Schachfigur um ihre Querachse gedreht. Sie werden die Figur sicher sofort erkennen, es ist die Dame.

Auch beim CIRCLE-Befehl entsteht der Moiré-Effekt, den wir schon beim LINE-Befehl kennengelernt haben und der immer auftritt, wenn mehrere Linien oder wie hier Kreisbögen zusammenfallen.

```
REM Schachfigur

FOR f=0 TO .5 STEP .05
CLS
  READ l
  FOR i= 1 TO l
    READ a
    CIRCLE (320,150-i*3*2*(.5-f)),a*2,2,,,f
  NEXT i
  RESTORE
  WHILE INKEY$="": WEND
NEXT f

REM dame
DATA 39,31,29,29,31,26,23
DATA 21,27,22,19,16
DATA 14,13,13,12,12,12,11,11,11,11,22
DATA 16,16,20,16,16
DATA 17,18,19,21,23,26,29
DATA 27,25,10,10,8
```

Für die Drehung der Figur benutzen wir ausschließlich den CIRCLE-Befehl. Bei diesem Programm bestimmt f nicht nur das Bildverhältnis, sondern ist auch für die Höhe der Mittelpunkte der einzelnen Kreise zuständig. Anfangs sieht man die Figur ganz von der Seite. Je mehr die Dame von oben (oder von unten; ganz nach eigener Vorstellung) zu sehen ist, desto runder werden die Kreise und desto dichter fallen die Mittelpunkte zusammen.

1.3.1.2 Das Bildverhältnis in der Kreisformel

Für das Bildverhältnis ist der Wert 0.44 voreingestellt. Dieser Wert ist nur für Bildschirme mit einer Auflösung von 640*200

Punkten geeignet. Je nachdem, wie Sie Ihren Monitor eingestellt haben, malt CIRCLE Ihnen für diesen Wert Kreise oder Ellipsen. Deshalb empfiehlt es sich, bei jedem Programm, das CIRCLE-Anweisungen benutzt, am Programmanfang in einer Variablen das Bildverhältnis festzusetzen und diese Variable an jeden CIRCLE-Befehl anzuhängen. Auf diese Weise kann man Programme schnell auf anders eingestellte Monitore angleichen. Man braucht ja nur die Variable am Programmbeginn zu verändern.

Werte für das Bildverhältnis sollten zwischen 0 und 200 liegen. 0 würde einen waagerechten Strich ergeben. Nach oben ist der Wertebereich zwar offen, aber der Wert 200 gibt meistens schon einen senkrechten Strich, also das Äquivalent zum Wert Null.

Ist das Bildverhältnis kleiner als eins, dann sind die Strecken vom Mittelpunkt zum seitlichen Rand und der Radius immer gleich. Bei einem Wert größer eins ist diese Strecke kleiner als der Radius, dafür ist die Höhe gleich dem Radius. Bei eins sind sowohl Breite als auch Höhe gleich dem Radius. Wie weit die Kreispunkte bei anderen Werten tatsächlich an bestimmten Stellen vom Mittelpunkt entfernt sind, kann man errechnen.

Dafür muß man aber wissen, wie der CIRCLE-Befehl die Kreispunkte berechnet. Das folgende Programm ersetzt den CIRCLE-Befehl, und zwar mit allem, was wir bis jetzt von ihm kennengelernt haben. Es ist aber in BASIC und deshalb langsamer, aber dafür versteht man so den Ursprung des Bildverhältnisses:

```
REM Simulation des CIRCLE-Befehls

CIRCLE (130,100),100,2,,,.2
CALL kreis (130!,100!,100!,1!,.2)
END

SUB kreis (mx,my,radius,farbe,f) STATIC
FOR w=0 TO 2*3.1415296# STEP .01
  IF f<1 THEN
    x=COS(w)*radius
    y=-f*SIN(w)*radius
  ELSE
    x=COS(w)*radius/f
```

```
    y=-SIN(w)*radius
  END IF
  PSET (mx+x,my+y),farbe
NEXT w
END SUB
```

Natürlich wollen wir den CIRCLE-Befehl nicht auf Dauer ersetzen. Mit diesem Programm wollen wir nur zeigen, wo das Bildverhältnis in der Formel auftaucht. Es ist der Faktor, mit dem der x-Wert (für f kleiner eins) multipliziert oder durch den der y-Wert geteilt wird, wenn f größer als eins ist. Wenn wir das wissen, können wir genau errechnen, wie weit ein beliebiger Kreispunkt vom Mittelpunkt entfernt ist. Das folgende Programm nimmt uns diese Arbeit ab. Auf diese Weise werden Speichen in den Kreis gezeichnet.

```
REM Speichen zeichnen

CIRCLE (200,100),100,2,,,.2
x1=200
y1=100
FOR winkel= 0 TO 6 STEP .5
  x=x1
  y=y1
  CALL koordinaten (x,y,100!,winkel,.2)
  LINE (x1,y1)-(x,y)
NEXT winkel
END


SUB koordinaten (mx,my,radius,w,f) STATIC
IF f<1 THEN
    mx=mx+COS(w)*radius
    my=my-f*SIN(w)*radius
  ELSE
    mx=mx+COS(w)*radius/f
    my=my-SIN(w)*radius
  END IF
END SUB
```

Die Berechnungen der Schnittpunkte mit dem Kreis führt das Unterprogramm aus. Die errechneten Werte werden dann an das Hauptprogramm zurückgegeben. Das gleiche Unterprogramm können Sie für alle Kreise benutzen, denn alle wichtigen Parameter werden an das Unterprogramm übergeben. Der Punkt, den man errechnen will, wird durch einen Winkel bestimmt, den

man ebenfalls übergeben muß. In unserem Beispielprogramm werden zwölf Winkel zwischen 0 und 6, was etwa dem Umfang eines Kreises entspricht, berechnet. Die berechneten Werte stehen nach dem Aufruf in den Variablen, mit denen man den Kreismittelpunkt an das Unterprogramm übergeben hat.

1.3.2 Die Winkel des CIRCLE-Befehls

Als nächstes wollen wir die beiden noch fehlenden Parameter der CIRCLE-Anweisung erklären. Mit diesen beiden Parametern kann man bestimmen, daß nur ein Ausschnitt des Kreises gezeichnet wird. Der erste Wert gibt den Winkel an, mit dem der Kreisausschnitt anfängt, und der zweite seinen Endwert. Es wird immer im mathematisch-positiven Sinn, also gegen den Uhrzeigersinn gezeichnet. Wenn alle Parameter angegeben werden, sieht der CIRCLE-Befehl folgendermaßen aus:

```
CIRCLE (mx,my),radius,farbe,start,ende,Bildverh
```

Wenn man zwei CIRCLE-Befehle mit entgegengesetzten Start- und Endwerten und ansonsten gleichen Werten aufruft, dann wird ein ganzer Kreis gezeichnet.

Wie man dem BASIC-Handbuch entnehmen kann, sind bei der CIRCLE-Anweisung Anfangs- und Endwinkel zwischen -2*PI und 2*PI erlaubt. Aus dem Mathematikunterricht wissen Sie vielleicht noch, daß im Bogengrad 2*PI genau eine volle Kreisumdrehung bedeutet. Jetzt könnte man auf die Idee kommen, der Computer zeichnet bei Angabe der beiden Maximalwerte (-2*PI und +2*PI) zwei volle Kreisumdrehungen. Das ist zwar mathematisch logisch, hier aber falsch. Das Minuszeichen vor einer Zahl hat keine mathematische, sondern eine technische Bedeutung. Steht vor dem Anfangswinkel ein Minuszeichen, wird zusätzlich zu dem angegebenen Kreisbogen eine Linie vom Kreismittelpunkt bis zum Anfang des Kreisbogens gezeichnet; bei negativem Endwinkel wird entsprechend eine Linie zum Ende des Kreisbogens gezeichnet.

Negativ bedeutet kleiner als null. Auch wenn man vor die Null ein Minuszeichen setzt, ist sie nicht negativ. Um bei einem Winkel von 0 Grad eine Linie zu zeichnen, muß man statt dessen den Wert -0.0001 einsetzen. Dieser Wert verkleinert den Winkel nicht merklich, reicht aber aus, den Computer zu veranlassen, das Gewünschte auszuführen.

1.3.3 Relative Adressierung

Auch beim CIRCLE-Befehl gibt es die relative Adressierung. Wenn man mit relativer Adressierung Kreise zeichnet, bildet der Grafik-Cursor den Mittelpunkt des Kreises. Und im Gegensatz zu PSET und LINE wird der Grafikcursor nach dem Zeichnen der Kreise nicht an die Stelle gesetzt, an der der letzte Punkt gezeichnet wurde, sondern immer an den Mittelpunkt des Kreises.

1.3.4 Tortengrafik

Sicher haben Sie schon einmal eine Tortengrafik gesehen; sie gehört zu den Präsentationsgrafiken und ist gut geeignet, Zahlenverhältnisse anschaulich zu machen. Beispielsweise wird oft in Tortengrafiken angegeben, wer nach einer Wahl wie viele Sitze erhält. Die Torte symbolisiert die Gesamtheit aller möglichen Sitze. Die Sitze einer Partei sind dann "Kuchenstücke" in den Farben der jeweiligen Partei.

Selbstverständlich kann man Tortengrafiken auch auf dem Amiga erstellen. Unser Programm zeichnet sie fast wie die Tortengrafiken aus den Wahlsendungen: farbig und dreidimensional.

```
REM Tortengrafik

SCREEN 1,320,200,5,1
WINDOW 2,"Tortengrafik",,,1

f=.3
pi=3.141529
```

```
start:
CLS

summe=0
INPUT "Wieviel Werte ";n

IF n<2 THEN
  CLS
  PRINT "Demoprogramm"
  n=INT(RND(1)*10)+3
  DIM wert(n),farbe(n)

  FOR i=1 TO n
    wert(i)=RND(1)*20+1
    farbe(i)=INT(RND(1)*31)
    summe=summe+wert(i)
  NEXT i
ELSE
  DIM wert(n),farbe(n)
  FOR i=1 TO n
    wert(i)=1
    PRINT i". Wert";
    INPUT wert(i)
    INPUT "Farbe ";farbe(i)
    farbe(i)=farbe(i) MOD 32
      summe=summe+wert(i)
   NEXT i
   CLS
END IF

REM Tortengrafik zeichnen
mx=WINDOW(2)/2
my=WINDOW(3)/2
w1=0
radius=mx -10
CIRCLE (mx,my+20),radius,1,pi,2*pi,f
LINE (mx-radius,my)-(mx-radius,my+20)
LINE (mx+radius,my)-(mx+radius,my+20)
LINE (mx,my)-(mx+radius,my)

FOR i=1 TO n
  w2=w1+2*pi*wert(i)/summe
  CIRCLE (mx,my),radius,1,-w1,-w2 ,f
  REM Segment faerben
  x=COS(w1+(w2-w1)/2)*radius/2
  y=-f*SIN(w1+(w2-w1)/2)*radius/2
  PAINT STEP(x,y),farbe(i),1
  IF w2>pi THEN
    REM Seitenstriche zeichnen
    x=COS(w2)*radius
    y=-f*SIN(w2)*radius
    LINE (mx+x,my+y)-(mx+x,my+y+20)
```

```
    REM Seitenbereiche Faerben
    IF w2-.1>pi THEN
      x=COS(w2-.1)*radius
      y=-f*SIN(w2-.1)*radius
      PAINT (mx+x,my+y+18),farbe(i),1
    END IF
  END IF
  w1=w2
NEXT i

INPUT "Neue Grafik ";a$
ERASE wert,farbe
IF a$<>"n" THEN start

WINDOW CLOSE 2
SCREEN CLOSE 1
```

Dieses Programm bietet Ihnen die Möglichkeit, eine Tortengrafik mit zufälligen Werten zu bestaunen oder selbst Werte einzugeben. Das Demonstrationsprogramm wird gestartet, wenn Sie bei der Frage nach der Anzahl der Werte eine Null eingeben. Wenn Sie selbst Werte eingeben, können Sie zusätzlich eine von 32 Farben zu jedem Wert eingeben.

Jedes Segment der Torte wird einzeln gezeichnet. Wie man Kreissegmente zeichnet, haben wir oben schon angedeutet. Alles, was man braucht, um statt des vollen Kreises nur ein Segment zu zeichnen, sind ein Anfangs- und ein Endwinkel. Vor beide Winkel setzen wir dann noch ein Minuszeichen. Auf diese Weise werden der erste und letzte Punkt des Kreisbogens mit dem Mittelpunkt durch eine Linie verbunden.

Um die Farbe ins Segment zu bringen, bedarf es schon etwas mehr Aufwand. Flächen zu färben ist in der Regel recht einfach. Man braucht nur eine begrenzte Fläche und einen Punkt in dieser Fläche, dann kann man mit PAINT die Fläche füllen. Die erste Bedingung haben wir erfüllt, als wir das Kreissegment gezeichnet haben. Um die zweite Bedingung zu erfüllen, bedienen wir uns der Formel des Kreises, mit der wir auch schon das Bildverhältnis erklärt haben. Den Mittelpunkt eines Segments kann man dann folgendermaßen berechnen:

```
x=COS(W1+(W2-W1)/2)*RADIUS/2
y=-f*SIN(W1+(W2-W1)/2)*RADIUS/2
```

In dieser Formel ist W1 der Anfangs-, W2 der Endwinkel und f das Bildverhältnis, das bei uns immer kleiner als eins ist. X und Y sind keine absoluten Bildschirmkoordinaten, sondern relativ zum Kreismittelpunkt, so daß wir mit der folgenden Zeile und den errechneten Werten das Segment füllen können:

```
PAINT STEP (X,Y),farbe(i),1
```

Die Variable farbe(i) enthält die Farbe, mit der das Segment gefüllt werden soll. Die 1 dahinter bedeutet, daß die Fläche durch weiße Punkte begrenzt wird.

Die Formel zum Errechnen der Mittelpunkte wird in etwas abgewandelter Form in diesem Programm noch einmal benötigt, nämlich um den Rand der Torte zu unterteilen und zu färben.

1.3.5 Punkte und Linien mit CIRCLE

So unsinnig das auch klingen mag, neben Ellipsen und Kreisen kann man auch Punkte und Linien mit CIRCLE zeichnen. Und zwar nicht nur die Linien, die sich ergeben, wenn man das Bildverhältnis auf 0 setzt, wie wir es oben schon gesehen haben. Wir meinen auch nicht die Punkte, die der Computer beim Radius null zeichnen würde. Diese beiden Sonderfälle sind nämlich ohne praktischen Nutzen. Um wirklich sinnvolle Linien und Punkte zu zeichnen, muß man die Winkelangaben manipulieren.

Punkte erzeugt man, indem man den Anfangswinkel gleich dem Endwinkel setzt. Linien erhält man, wenn beide Winkel betragsgleich, aber einer von beiden negativ ist. Die Entfernung der Punkte vom "Mittelpunkt" und die Länge der Linien werden durch den Radius und das Bildverhältnis bestimmt.

Linien auf diese Art zu zeichnen, das ist in bestimmten Fällen recht nützlich, denn es reichen ein Punkt, die Länge und der

Winkel einer Geraden aus, um eine Linie zu zeichnen. Beim LINE-Befehl bräuchte man zwei Punkte und müßte deshalb erst Linie und Winkel in einen zweiten Punkt umrechnen.

Der Vorteil dieser etwas unkonventionellen Art wird recht gut im folgenden Programm deutlich. Es ist eine Analoguhr, die nur mit dem CIRCLE-Befehl gezeichnet wird und sonst keinen anderen Grafik-Befehl enthält. Dadurch, daß die Koordinaten der Zeiger und der Einheiten nicht vom Programm ausgerechnet werden müssen, spart man viel Rechenaufwand und -zeit.

```
REM Analoguhr

pi=3.1415926#
f=.5                    ' Bildverhaeltnis '

REM Kreis Zeichenen
CIRCLE (100,100),100,1,,,f

REM Punkte fuer Minuten
FOR i=.0001 TO 2*pi STEP pi/30
CIRCLE (100,100),97,1,i,i,f
NEXT i

REM Punkte fuer Stunden
FOR i=.0001 TO 2*pi STEP pi/6
CIRCLE (100,100),93,1,i,i,f
CIRCLE (100,100),90,1,i,i,f
NEXT i

st:  INPUT "Stunden ";stunden
     IF stunden >12 GOTO st
INPUT "Minuten ";minuten

swinkel=-((12-stunden)*60-minuten)*pi/360-pi/2.0001
mwinkel=-(60-minuten)*pi/30-pi/2.0001
IF swinkel<-2*pi THEN swinkel=swinkel+2*pi
IF mwinkel<-2*pi THEN mwinkel=mwinkel+2*pi

REM Zeiger
CIRCLE (100,100),85,2,mwinkel,-mwinkel,f
CIRCLE (100,100),70,3,swinkel,-swinkel,f

ON TIMER(60) GOSUB Zeit
TIMER ON

WHILE 1:WEND
```

```
Zeit:
REM Alte Zeiger loeschen
CIRCLE (100,100),70,0,swinkel,-swinkel,f
CIRCLE (100,100),85,0,mwinkel,-mwinkel,f
swinkel=swinkel+pi/360
mwinkel=mwinkel+pi/30
IF swinkel>0 THEN swinkel=swinkel-2*pi
IF mwinkel>0 THEN mwinkel=mwinkel-2*pi
REM Neue Zeiger
CIRCLE (100,100),85,2,mwinkel,-mwinkel,f
CIRCLE (100,100),70,3,swinkel,-swinkel,f
RETURN
```

Die im Programm verwendeten TIMER-Befehle sorgen dafür, daß jede Minute zum Unterprogramm gesprungen wird. Mit diesen Befehlen kann man die Interrupt-Programmierung nachvollziehen, die mancher vielleicht noch von den Homecomputern kennt. Diese Art ist wesentlich komfortabler als die herkömmliche Interrupt-Programmierung, denn Interrupt-Programme werden beim Amiga in BASIC geschrieben.

Statt der Endlosschleife WHILE1:WEND kann man auch ein beliebiges anderes Hauptprogramm einfügen. Für diesen Fall ist es zweckmäßig, die Uhr in ein eigenes Fenster zu verbannen.

1.4 Flächen füllen

Inzwischen haben wir schon mehrere Arten kennengelernt, um Flächen auszumalen. Zuerst war da der LINE-Befehl, mit dem man ziemlich schnell farbige Rechtecke auf den Bildschirm bringen kann.

Der CIRCLE-Befehl hatte keine eingebaute Füllfunktion. Deshalb hatten wir dort etwas vorweggegriffen und den PAINT-Befehl eingeführt. Diesen Befehl wollen wir jetzt noch einmal genauer betrachten.

1.4.1 Der PAINT-Befehl

Paint bedeutet malen, und das ist alles, was der PAINT-Befehl macht. Aber dafür erledigt er diese Aufgabe schnell. Auch seine Handhabung ist einfach. Man gibt einen Punkt in der zu füllenden Fläche und die Farbe an, in der gefüllt werden soll. Wenn Rahmenfarbe und Füllfarbe der Fläche nicht gleich sind, gibt man als letztes auch noch die Rahmenfarbe an. Die Rahmenfarbe gibt die Farbe an, die der Computer dann als Begrenzung sieht. Der Punkt kann sowohl absolut als auch relativ adressiert werden. Relative Adressierung hat besonders dann Vorteile, wenn man ganze Kreise füllen will. Denn nach dem CIRCLE-Befehl befindet sich der Grafik-Cursor automatisch am Kreismittelpunkt. Im folgenden Programm sieht man, wie einfach dadurch das Färben von Kreisen ist.

```
REM Fuelldemo

SCREEN 1,320,200,5,1
WINDOW 2,,(0,0)-(311,185),16,1
RANDOMIZE TIMER

WHILE INKEY$=""
  f=INT(RND*32)
  CIRCLE (RND*311,RND*185),RND*100,f
  PAINT STEP (0,0),f
WEND

WINDOW CLOSE 2
SCREEN CLOSE 1
```

Dieses Programm wirkt lange nicht so schnell wie das entsprechende Programm mit Kästchen und dem LINE-Befehl. Das liegt zum einen am CIRCLE-Befehl. Aber auch der PAINT-Befehl ist langsamer als etwa der LINE-Befehl mit seiner Kästchenfüllfunktion. Denn schließlich muß beim PAINT-Befehl auch bei jedem Punkt abgefragt werden, ob er einen Rand bildet (also die Farbe hat, die als Rahmenfarbe angegeben wurde) oder ob er einfach gefärbt werden kann.

Bei der PAINT-Anweisung muß man mit Vorsicht arbeiten. Wenn auch nur ein "Loch" in der Begrenzung der Fläche ist,

färbt sich unter Umständen der ganze Bildschirm, und die ganze Grafik ist praktisch vernichtet. Ähnliches passiert auch, wenn der Typ eines Fensters (siehe Handbuch), den man beim Öffnen festlegt, kleiner als 16 ist. Dann wird immer dann, dann wenn die Fläche über den rechten Bildschirmrand hinausragt, der ganze Bildschirm mit der Füllfarbe gefärbt. Wenn Sie es ausprobieren möchten, brauchen Sie nur im WINDOW-Befehl des letzten Programms die 16 durch eine 0 zu ersetzen.

1.4.2 Der dritte Weg: AREA und AREAFILL

Neben LINE und PAINT gibt es noch eine dritte Möglichkeit, Flächen zu füllen. Im Gegensatz zur PAINT-Anweisung sind hierfür keine durchgezogenen Grenzen, sondern nur einzelne Punkte als Eckpunkte der Fläche notwendig.

Dieser dritte Weg besteht aus zwei Befehlen. Mit dem ersten Befehl setzt man alle Eckpunkte. Dieser Befehl ist noch einfacher als der PSET-Befehl, denn er braucht nicht einmal eine Farbkennung:

```
AREA (10,30)
AREA (199,140)
AREA STEP (200,-30)
```

Wie man sieht, kann man den AREA-Befehl auch relativ benutzen.

Mit einem zweiten Befehl teilt man dem Computer mit, daß er die begrenzte Fläche ausmalen soll:

```
AREAFILL
```

Nachdem wir alle vier Befehle eingetippt haben, wird ein weißes Dreieck auf den Bildschirm gemalt. Ein Dreieck zeigt nicht unbedingt die Stärken dieses Befehlspaares. Die Stärken liegen in Figuren mit viel mehr Eckpunkten.

Entscheidend für das Aussehen der Fläche ist die Reihenfolge, in der die Punkte angegeben werden. Bei nur drei Punkten spielt das noch keine Rolle, aber schon bei vier Punkten gibt es drei verschiedene Figuren für dieselben Koordinaten.

```
REM drei moegliche

AREA (10,10)
AREA (30,140)
AREA (60,100)
AREA (50,20)
AREAFILL
WHILE INKEY$="": WEND
CLS

AREA (10,10)
AREA (60,100)
AREA (50,20)
AREA (30,140)
AREAFILL

WHILE INKEY$="":WEND
CLS

AREA (10,10)
AREA(60,100)
AREA(30,140)
AREA(50,20)
AREAFILL
```

Bei noch mehr Ecken nimmt die Anzahl der Verbindungsmöglichkeiten noch zu. Hier haben wir auch gleich ein Beispiel. Wir haben einen Drudenfuß gebildet. Ein Drudenfuß hat fünf Zacken und kann mit einem Stift in einem Zug gemalt werden.

```
AREA (100,20)
AREA (140,100)
AREA (20 ,45)
AREA (180,40)
AREA (40,110)

AREAFILL
```

Wenn man sich den entstandenen Stern anguckt, fällt auf, daß nur die Zacken weiß, die Mitte aber blau geblieben ist. Wie kommt's? Zur Erklärung nehmen wir noch einmal unser Dreieckprogramm und ergänzen es ein wenig:

```
REM Rahmen durch zweimaliges Zeichnen

FOR i=0 TO 3
 AREA (10,30)
 AREA (199,140)
 AREA STEP (200,-30)
NEXT
AREAFILL
```

Durch die Änderungen wird jeder Eckpunkt zweimal gesetzt, so daß die gleiche Fläche mit einem AREAFILL-Befehl zweimal ausgemalt wird. Man könnte meinen, doppelt malt besser, aber so ist es nicht. Statt einer gefüllten Fläche ist nämlich nur der Rahmen zu sehen.

Beim Drudenfuß ist es ähnlich. Wie das Dreieck, so ist die Mitte auch zweifach eingeschlossen und wird deshalb zweimal, was gleichbedeutend mit gar nicht ist, ausgefüllt.

Bei nur fünf Ecken sehen die entstandenen Figuren noch recht unkompliziert aus. Dagegen kann man bei neunzehn Ecken häufig kaum noch alle Eckpunkte entdecken. Die Bilder, die entstehen, wenn man alle Punkte zufällig bestimmt, könnte man fast schon als moderne Kunst verkaufen:

```
REM 19 Ecken

RANDOMIZE TIMER

FOR i= 0 TO 18
AREA (RND*611,RND*185)
NEXT i

AREAFILL

WHILE INKEY$="" :WEND
RUN
```

Neunzehn Eckpunkte sind das Maximum, das AREAFILL verarbeiten kann. Wenn man versucht, mehr als neunzehn Eckpunkte zu setzen, wird überhaupt keine Fläche auf den Bildschirm gezeichnet. Nach AREAFILL sind alle Eckpunkte aus dem Speicher gelöscht, und man kann neue Eckpunkte bestimmen.

1.4.2.1 Verschiedene Modi bei AREAFILL

AREAFILL hat zwei verschiedene Modi. Den ersten haben Sie schon kennengelernt, denn wir haben die ganze Zeit mit ihm gearbeitet. Bei diesem Modus wird die Fläche immer in der aktuellen Vordergrundfarbe gefüllt. Diese Farbe ist mit weiß voreingestellt. Man kann sie mit dem COLOR-Befehl verändern:

```
COLOR 2
```

Nach diesem Befehl wird die nächste Fläche schwarz gezeichnet. Dieses ist die einzige Methode, um auf die Farbe der Fläche direkt Einfluß zu nehmen. Diesen Modus braucht man nicht besonders zu kennzeichnen, denn es ist der Normalmodus. Man kann ihn aber durch die Ziffer Null kenzeichnen:

```
AREAFILL 0
```

Der zweite Modus bietet etwas ganz Besonderes. Hier werden die Flächen nicht ausgefüllt, sondern jeder Punkt der Fläche wird invertiert.

```
REM Invertierdemo

PRINT "Dies ist ein Test!!"
PRINT "Alle Punkte innerhalb"
PRINT "des Dreicks werden"
PRINT "invertiert, jawoll!!"
CIRCLE (100,100),90,2
PAINT STEP (0,0),3,2

AREA (20 ,0)
AREA (180,45)
AREA (40,100)
AREAFILL 1
```

Diesen Modus kennzeichnet man mit einer Eins hinter dem AREAFILL-Befehl. Was bedeutet denn eigentlich Invertieren? Jeder Punkt auf dem Bildschirm wird durch eine Bitfolge im Speicher repräsentiert, die sein Farbregister angibt. Wenn ein Punkt invertiert wird, bedeutet das, daß jedes Bit seiner Bitfolge

einzeln "umgedreht" wird. Ist ein Bit vorher eins, wird es hinterher null und umgekehrt. Wie sich das Farbregister eines Punktes ändert, kann man so ausrechnen:

neueFarbe=(2^Tiefe-1)-alteFarbe

Die Geschwindigkeit, mit der diese Umkehrung vor sich geht, ist enorm, was man auch an folgendem Programm erkennen kann:

```
REM   Geschwindigkeit

LOCATE 10,4
PRINT "Geschwindigkeit ist keine Hexerei"

WHILE INKEY$=""

FOR i= 0 TO 2
  AREA (RND*611,RND*185)
NEXT i
AREAFILL 1

WEND
```

Das Programm bestimmt zufällige Eckpunkte eines Dreiecks und invertiert alles, was innerhalb dieses Dreiecks liegt. Dieser Vorgang wiederholt sich immer und immer wieder. Dabei kommt es sehr häufig zu Überlagerungen von Dreiecken und damit von blauen und orangefarbigen Stellen. Dadurch ist schon nach kurzer Zeit der ganze Bildschirm orange und blau gescheckt. Je länger das Programm läuft, desto weniger Struktur ist in den Flecken zu erkennen.

1.4.3 Muster

Wenn Computer eine Fläche füllen, setzen sie einen Punkt neben den anderen, bis alle Punkte der Fläche die gewünschte Farbe haben. So kann es der Amiga auch, wie wir in den zahlreichen Beispielen schon gesehen haben. Er kann aber noch mehr. Man kann ihn dazu veranlassen, Flächen nach selbstdefinierten Mustern zu füllen.

Muster können die Grafik verschönern oder bestimmte Dinge verdeutlichen oder hervorheben. Man kann mit Mustern Schatten andeuten oder sehr einfach eine Mauer "bauen".

Neben gemusterten Flächen kann man auch gemusterte Linien erzeugen. Beide Muster werden mit ein und demselben Befehl definiert. Da die Muster für Linien und Flächen etwa gleich aufgebaut werden, erklären wir die Methode erst einmal für Linien.

1.4.3.1 Aufbau der Muster

Das Muster einer Linie wird mit einer sogenannten 16-Bit-Maske festgelegt. Eine 16-Bit-Maske besteht aus einer Zahlenfolge von 16 Binärzahlen (nur Nullen und Einsen). Jede Eins in dieser Maske entspricht einem gesetzten Punkt in der Linie, eine Null einer Leerstelle. Nach sechzehn Punkten fängt es bei dem ersten Punkt der Maske wieder an. Die Maske ist vergleichbar mit einer Zeichenschablone. Nur dort, wo Löcher sind, kann auch gemalt werden.

Da AmigaBASIC keine Binärzahlen verarbeiten kann, muß die Binärfolge der Maske in hexadezimale Zahlen umgerechnet werden. Man könnte die Binärzahlen auch in dezimale Zahlen umwandeln, was wesentlich komplizierter ist. Zur Umrechnung in Hexzahlen faßt man immer vier Bits zu einem Block zusammen. Unsere Maske könnte dann so aussehen:

```
1011 0010 0000 1111
```

Jeder Block wird nun einzeln umgerechnet. Dafür nimmt man den Wert des ersten Bits des Blocks und multipliziert ihn mit 8. Dazu addiert man das zweite Bit, multipliziert mit 4, das 3 Bit mit 2 malgenommen und das letzte Bit einfach. Das Ergebnis dieser Operationen liegt zwischen 0 und 15. Statt der Zahlen zehn bis 15 schreibt man im Hexadezimalen die Buchstaben A bis F. Bei unserer Beispielsmaske sieht das so aus:

```
1*8 + 0*4 + 1*2 + 1 = B (11)
0*8 + 0*4 + 1*2 + 0 = 2
0*8 + 0*4 + 0*2 + 0 = 0
1+8 + 1+4 + 1+2 + 1 = F (15)
```

Die Hexzahl unserer Maske kann man nun rechts senkrecht ablesen. Sie lautet B20F. Das können wir auch gleich anhand eines kleinen Programms ausprobieren:

```
REM Gepunktete Linie

PATTERN &HB20F
LINE (0,0)-(614,185)
LINE (20,30)-(104,105),2,B
```

Wie Sie sehen, wirkt unser Muster sowohl auf einfache Linien genau wie auf die Umrandung von Kästchen. Die Zeichen "&H" vor unser Maske kennzeichnen die Zahl als Hexzahl.

1.4.3.2 Gemusterte Flächen

Bei Füllmustern ist der Aufbau, wie gesagt, ähnlich. Da aber Flächen im Gegensatz zu Linien zweidimensional sind, werden mehrere 16-Bit-Masken übereinander gestapelt. Diese Masken werden in einer Feldvariablen zusammengefaßt und an den Befehl PATTERN übergeben.

Das Muster für Linien wird als erster, das Muster für Flächen als zweiter Parameter mit einer ganzzahligen Feldvariablen definiert.

```
REM Mustermacher

DEFINT a
OPTION BASE 1
DIM a(8)

FOR i=1 TO 8
READ a(i)
NEXT i

PATTERN ,a
COLOR 3,1
```

```
LINE (0,0)-(614,185),,bf

WHILE INKEY$="": WEND
COLOR 1,0
CLS

DATA &h0,&h7FFF,&h7FFF,&h7FFF
DATA &h0,&hFF7F,&hFF7F,&hFF7F
```

Als erstes muß man beachten, daß die Feldvariable, mit der das Muster übergeben wird, eine kurze Ganzzahl ist. Das bedeutet, daß sie nur Werte zwischen -32768 und 32767 annehmen kann. Diesen Werten entsprechen im Hexadezimalen die Werte von 0 bis FFFF. Wenn die Feldvariable nicht von diesem Typ ist, kann ein Fehler auftreten. Deshalb bestimmen wir anfangs mit "DEFINT a" unsere Feldvariable als kurze Ganzzahlvariable.

PATTERN erfährt über die DIM-Anweisung, wie viele Ebenen das Muster hat. Deshalb muß die Feldvariable, auch wenn sie weniger als zehn Elemente besitzt, vorher deklariert werden. Die Anzahl der Feldelemente muß genau eine Potenz von zwei besitzen (erlaubte Werte sind z.B. 1, 2, 4, 8, 16,usw.). Besitzen sie nur ein Element mehr oder weniger, wird schon eine "Illegal function call"-Meldung auf dem Bildschirm erscheinen. Deshalb müssen Sie beachten, daß das erste Element einer Feldvariablen normalerweise den Index Null führt. Sie müssen also entweder den Index in der DIM-Anweisung immer um 1 kleiner als 2n halten oder mit Hilfe von

```
OPTION BASE 1
```

den kleinsten Indexwert aller Feldvariablen auf 1 heraufsetzen.

1.4.3.3 Design im Listing

Wie Sie gesehen haben, bereitet das Errechnen der Muster sehr viel Arbeit. Aber ist es nicht auch ein wenig umständlich, diese Werte selber ausrechnen zu müssen, wenn man sowieso einen Computer neben sich stehen hat? Da können wir ihn doch gleich die Arbeit machen lassen.

Alles, was wir dazu brauchen, ist ein kleines Programm, das unsere binäre Mustermaske versteht und in Hexadezimal- oder Dezimalzahlen umrechnen kann. Da AmigaBASIC selbst noch keine Binärzahlen versteht, haben wir für diesen Zweck ein kleines Unterprogramm entwickelt. Dieses Programm wandelt Zeichenketten in kurze Ganzzahlen (Zahlen ohne Komma, die im Speicher mit zwei Byte dargestellt werden) um. In der Zeichenkette wird jede Null durch ein Leerzeichen repräsentiert. Jedes andere Zeichen gilt als Eins.

Beim folgenden Programm haben wir das große Amiga-A als Vorlage für unser Muster gewählt.

```
REM Design im Listing

OPTION BASE 1
a=8
DIM f$(a)

REM     0123456789ABCDEF
f$(1)="             ***
f$(2)="            ****
f$(3)="          *  ***
f$(4)="        *    ***
f$(5)="       **********"
f$(6)="     **       ***"
f$(7)="  *****     ******"
f$(8)=""
REM     0123456789ABCDEF

CALL changeformat(f$(),a)

CIRCLE (400,140),100
PAINT STEP(0,0),2,1
AREA (150,160)
AREA (500,100)
AREA (570,170)
AREAFILL 1

MOUSE ON

WHILE INKEY$=""
IF MOUSE(0)<>0 THEN
  b=MOUSE(1)
  c=MOUSE(2)
  IF b>0 AND b<600 AND c>0 AND c<172 THEN
    LINE (b,c)-(b+4,c+4),1,bf
  END IF
```

```
END IF
WEND


SUB changeformat (feld$(1),g) STATIC

DIM feld%(g)
FOR i=1 TO g
  feld$(i)=feld$(i)+SPACE$(16)
  FOR j=0 TO 3
    h=0
    FOR k=0 TO 3
      IF MID$(feld$(i),j*4+k+1,1)<>" " THEN h=h+2^(3-k)
    NEXT k
    feld%(i)=feld%(i)+VAL("&h"+HEX$(h*2^(4*(3-j))))
  NEXT j
  PRINT i, HEX$(feld%(i)),feld%(i)
NEXT i

PATTERN ,feld%
END SUB
```

Das Muster, das erst nur als Zeichenkette aus Leerstellen und Sternen besteht, wird umgeformt, und dann mit ihm ein Kreis und ein Dreieck gefüllt. Anschließend kann man mit der Maus und diesem Muster zeichnen. Dabei wird auf Tastendruck am Mauszeiger in einem 4*4 Pixel großen Rechteck ein Fragment des Musters auf den Bildschirm gemalt.

An das Sub-Programm, das die Zeichenketten in Hex-Zahlen verwandelt, werden zwei Parameter übergeben: ein Zeichenkettenfeld, in dem die Masken gespeichert sind, und die Anzahl der Feldelemente. Die Umrechnung selbst erfolgt nach dem oben erklärten Prinzip. Im Unterprogramm wird dann auch das Muster an PATTERN übergeben.

Es ist vielleicht nicht die beste Methode, bei jedem Programmstart die Werte neu umzurechnen, denn das benötigt doch immer eine gewisse Zeit. Dafür ist dieses Programm aber hervorragend als Mustereditor zu benutzen. Vertauschen Sie im Listing das große Amiga-A einfach mit Ihren eigenen Mustern. Probieren Sie herum, und wenn das Muster Ihnen noch nicht gefällt, beenden Sie das Programm und verbessern es im Listing. Damit man das Programm auch als Editor gebrauchen kann, werden die er-

rechneten Daten im Hex-Code und in Dezimalzahlen ausgedruckt. Wenn Ihr Muster so ist, wie Sie es haben wollen, schreiben Sie sich einfach diese Daten ab und übertragen sie in Ihr eigenes Programm.

1.4.3.4 Änderungen am Cursorstrich

Die Muster sind eigentllich nur dafür gedacht, die Füllfunktionen zu beinflussen. Aber auch den Cursor kann man mit dem PATTERN-Befehl beeinflussen. Das ist ein Nebeneffekt, der jedesmal auftritt, wenn ein Muster definiert wurde. Das beste Beispiel dafür ist das oben abgedruckte Programm. Nachdem das Programm die Daten ausgegeben hat und das Ausgabefenster angeklickt wird, muß sich der Cursor bekanntlich im Ausgabefenster befinden. Tut er auch. Er ist aber nur noch als kleiner Punkt zu sehen. Drücken Sie nun mehrmals kurz die Leertaste, dann sehen Sie statt des gewohnten Cursors mal einen dicken Punkt, mal eine gepunktete Linie. Diese Punkte stammen alle von den As des Musters. Wen diese veränderten Cursor stören, der kann den Cursor dadurch normalisieren, daß er folgende Zeilen an das Musterprogramm anhängt:

```
REM Cursor reset
DIM norm%(2)
norm%(1)=&HFFF
norm%(2)=&HFFF
PATTERN ,norm%
```

(Es ist sehr wichtig, daß vor diesen Programmzeilen der Befehl OPTION BASE 1 ausgeführt wurde. Wenn Sie OPTION BASE nicht benutzt haben, müssen alle Feldindizes um eins verringert werden.)

Natürlich kann man nun auch den umgekehrten Weg gehen und den Cursor absichtlich verändern. Auf diese Weise kann man beispielsweise bei INPUT den Cursor verschwinden lassen oder ihn stricheln. Eine gestrichelte Cursorlinie erzeugt man, indem man im obenstehenden Programm, mit dem wir den Cursor normalisiert haben, norm%(2) auf null setzt. Eine halbe Cursorlinie

entsteht durch ein kurzes Ganzzahlfeld mit acht Elementen, von denen die ersten vier Elemente alle null oder alle &HFFFF und die anderen vier Elemente alle genau entgegengesetzte Werte enthalten.

1.5 Allerlei Buntes

Der Amiga verfügt über eine Farbpalette von 4096 Farben. Davon kann man mit reinem BASIC 32 Farben gleichzeitig benutzen. (Später zeigen wir Ihnen, wie man noch mehr Farben nutzen kann.) Auf welche von diesen Farben man gerade zugreifen kann, steht in den Farbregistern. Bei allen Befehlen gibt man genaugenommen nicht die Farbe, sondern immer nur das Farbregister an.

Man kann nicht in jedem Screen alle 32 Farben benutzen. Die Anzahl der Farben hängt von der Tiefe des Bildschirms ab. Der normale Workbenchscreen hat die Tiefe 2. Mit ihm kann man 2*2, also 4 Farben darstellen. Für jeden Punkt gibt es im Speicher zwei Bits. Aus den Kombinationen der zwei Bits lassen sich die vier Farben darstellen.

00 Farbregister 0
01 Farbregister 1
10 Farbregister 2
11 Farbregister 3

Um 32 Farben darzustellen, braucht man fünf Bit pro Punkt. Deshalb müssen wir erst einen neuen Screen mit Tiefe 5 öffnen:

```
SCREEN 1,320,200,5,1
WINDOW 2,"Titel",(0,0)-(311,185),16,1
```

Nun können wir alle Befehle, die wir schon kennengelernt haben, mit 32 Farben benutzen, wie wir es ja auch schon in einigen Programmen gemacht haben.

Von den Farben, die im jeweiligen Screen zur Verfügung stehen, kann man eine Farbe als Vordergrundfarbe und eine als Hintergrundfarbe bestimmen.

```
COLOR 1,0
```

Das sind die beiden voreingestellten Werte. Der vordere Wert gibt die Vordergrundfarbe an, der zweite die Hintergrundfarbe. Diese Werte werden von den Grafikbefehlen berücksichtigt. Bis auf PRESET wird, solange kein Farbparameter angegeben ist, der Befehl mit der Vordergrundfarbe ausgeführt, bei PRESET der mit der Hintergrundfarbe.

Nachdem man die Vorder- und/oder Hintergrundfarbe geändert hat, ändern sich die Farben des Bildschirms noch nicht. Erst wenn man den Befehl CLS aufruft, färbt sich der Bildschirm entsprechend.

```
COLOR 2,3
CLS
```

Durch diese Befehle erhält man einen orangefarbigen Bildschirm mit schwarzer Vordergrundfarbe.

1.5.1 Die ganze Palette

Sicher werden die 32 voreingestellten Farben nicht immer zum Programm passen. Häufig braucht man ganz andere Farben, als zur Verfügung stehen. Das macht aber rein gar nichts, denn man kann die Farben, die in den Farbregistern angegeben sind, ändern.

Jede Farbe besteht aus drei Werten. Sie geben die Rot-, Grün- und Blauanteile der Farbe an. Jeder Anteil kann einen von 16 Werten annehmen. Das macht dann 4096 Farben (16 hoch 3).

Der BASIC-Befehl, mit dem man die Farben ändert, heißt PALETTE. Außer den RGB-Werten (Rot, Grün und Blau) muß

man auch das Farbregister angeben, das geändert werden soll. Die Werte der Farbanteile müssen zwischen null und eins liegen.

```
PALETTE 0,.75,1,0
```

Diese Zeile schafft einen neongelben Hintergrund. Die Farbe ergibt sich aus einer Mischung von rot und gelb. Blau ist gar nicht vertreten. Den gewohnten blauen Hintergrund bekommt man mit

```
PALETTE 0,0,.3,.6
```

zurück.

Ist nur der Rotanteil eins und sind die anderen beiden null, ist die Farbe rot. Entsprechend ist es bei den anderen beiden Anteilen.

Wenn alle drei Farbanteile gleich sind, ist die Farbe grau. Je niedriger alle drei Farbanteile sind, desto dunkler ist die Farbe; sind alle drei Anteile eins, hat man weiß.

Wie schon gesagt, so gliedert sich jeder Wert in 16 Schritte auf. In der folgenden Tabelle haben wir noch einmal kurz die einzelnen Schritte angedeutet.

1. Schritt	0	bis 0.0625
2. Schritt		bis 0.125
3. Schritt		bis 0.1875
4. Schritt		bis 0.25
usw.		

1.5.2 Farbe gesucht

Wenn man mit den drei Farbanteilen weiter herumjongliert, kann man alle Farben finden, die man braucht. Es wird aber sehr mühsam sein, einen bestimmten Farbton durch planloses Ausprobieren zu finden. Mit dem folgenden Programm kann man Farben gezielt suchen und finden, zwei Farben mischen, die Farbwerte ausgeben, ausdrucken oder abspeichern und noch

etliches mehr. Dabei ist das Programm noch so komfortabel gehalten, daß es sehr viel Spaß macht, mit ihm zu arbeiten.

```
REM Farbdesigner

DEF FNfarbtab=PEEKL(PEEKL(PEEKL(WINDOW(7)+46)+48)+4)
DEF FNrgb(f,rgb)=(PEEKW(FNfarbtab+2*f) AND (15*2^(12-rgb*4)))/(15*2^(12-
    rgb*4))

DIM w(6),wa(6),t(6),af(6),nv(6),a$(3)
FOR i= 1 TO 6:t(i)=1:NEXT i:t(4)=6
a$(1)="scrn:":a$(3)="lpt1:"
farbe=5

MENU 1,0,1,"Ein-/Ausgabe"
MENU 1,1,1,"Bildschirm  "
MENU 1,2,1,"Speichern   "
MENU 1,3,1,"Drucker     "
MENU 1,4,1,"Laden       "
MENU 1,5,1,"Ende"

ON MENU GOSUB ausgabe
MENU ON

SCREEN 1,320,200,5,1
WINDOW 1,"Farbendesign",(0,0)-(297,185),31,1

GOSUB start

WHILE 1
IF MOUSE(0)<>0 THEN
  GOSUB IsWasPassiert
ELSE
  t=0
END IF
WEND

ende:  WINDOW 1,"Auf Wiedersehen !",,,-1
       SCREEN CLOSE 1
       END

start:  CLS  '**** Bildschirm aufbauen ****
        RESTORE
        FOR i=1 TO 6:wa(i)=0:NEXT
        FOR i=0 TO 31
          LINE (i*9+14,0)-(i*9+23,50),i,bf
        NEXT i
       LINE (13,0)-(303,51),1,b
       FOR i= 1 TO 15
         READ x1,y1,x2,y2
         LINE (x1,y1)-(x2,y2),1,bf
```

```
        NEXT i
        DATA 35,61,55,73,35,77,55,89,35,93,55,105
        DATA 60,56,278,105,283,61,303,73,283,77,303,89
        DATA 283,93,303,105,35,109,55,121,35,125,55,137
        DATA 35,141,55,153,283,109,303,121,283,125,303,137
        DATA 283,141,303,153,60,109,278,158,13,55,30,158

        COLOR 0,1
        LOCATE 9,1
        FOR i= 1 TO 6
          READ x$
          PRINT TAB(6);"-";TAB(9);x$;TAB(34);x$;TAB(37);"+"
          PRINT
        NEXT i
        DATA R,G,B,H,S,V
        FOR i=1 TO 6
          FOR j=1 TO 15*t(i)
LINE (76+(j-1)*12/t(i),48+i*16)-(76+j*12/t(i),55+i*16),2,b
LINE (77+(j-1)*12/t(i),49+i*16)-(75+j*12/t(i),54+i*16),0,bf
          NEXT j
        NEXT i
        LINE (14,56)-(29,157),farbe,bf
        FOR i=1 TO 3
          w(i)=FNrgb(farbe,i)
          af(i)=w(i)
        NEXT i
        GOSUB werte
        LOCATE 22,3
        PRINT " Tausche ";TAB(13);" Kopiere ";TAB(23);"Uebergang";
        PRINT TAB(33);"Undo";
        COLOR 1,0
RETURN

IsWasPassiert:
WHILE MOUSE(0)<>0 AND t<150:t=t+1:WEND
x=MOUSE(1)
y=MOUSE(2)
IF y<51 THEN            '**** Farbe ****
  farbe=POINT (x,y)
  LINE (14,56)-(29,158),farbe,bf
  FOR i=1 TO 3
    w(i)=FNrgb(farbe,i)
  NEXT i
  IF af<0 THEN
    af(1)=w(1):af(2)=w(2):af(3)=w(3)
    GOTO werte
  END IF
  ON tool GOTO tausche,kopiere,uebergang
ELSEIF y>167 AND y<176 THEN
  IF x>15 AND x<248 THEN
    tool=INT((x-15)/77)+1
    af=farbe
```

```
    LINE (16,180)-(247,184),1,bf
    RETURN
  ELSEIF x>257 AND x<294 THEN
    FOR i= 1 TO 3     '**** Undo ****
      w(i)=af(i)
    NEXT
    GOTO werte
  END IF
ELSEIF y>61 AND y<153 THEN
  i=INT((y-46)/16)
  IF x>34 AND x<56 THEN
    w(i)=w(i)-1/15:IF w(i)<0 THEN w(i)=0
  ELSEIF x>282 AND x<304 THEN
    w(i)=w(i)+1/15:IF w(i)>t(i) THEN w(i)=t(i)
  ELSEIF x>75 AND x<255 THEN
    w(i)=INT(t(i)*15*(x-76)/178+.9)/15
  END IF
  IF i>3 THEN
    CALL rgb(w(4),w(5),w(6),w(1),w(2),w(3))
    GOTO werte2
  END IF
END IF

werte:
  CALL hsv(w(1),w(2),w(3),w(4),w(5),w(6))
werte2: af=-1
    LINE (16,180)-(247,184),0,bf
  PALETTE farbe,w(1)*15/16,w(2)*15/16,w(3)*15/16
  FOR i= 1 TO 6
    FOR j=INT(wa(i)*15)+1 TO INT(w(i)*15)
        LINE (77+(j-1)*12/t(i),49+i*16)-(75+j*12/t(i),54+i*16),3,bf
    NEXT j
    FOR j=INT(w(i)*15)+1 TO INT(wa(i)*15)
        LINE (77+(j-1)*12/t(i),49+i*16)-(75+j*12/t(i),54+i*16),0,bf
    NEXT j
    wa(i)=w(i)
  NEXT i
RETURN

tausche:
FOR i=1 TO 3
  af(i)=w(i)
  w(i)=FNrgb(af,i)
NEXT
PALETTE af,af(1)*15/16,af(2)*15/16,af(3)*15/16
GOTO werte

kopiere:
FOR i=1 TO 3
  w(i)=FNrgb(af,i)
NEXT i
GOTO werte
```

```
uebergang:
  IF af=farbe THEN RETURN
  FOR i=1 TO 3
    af(i)=FNrgb(af,i)
  NEXT i
  CALL hsv(af(1),af(2),af(3),af(4),af(5),af(6))
  CALL hsv(w(1),w(2),w(3),w(4),w(5),w(6))
  s=1:h2=af
  IF farbe <af THEN s=-1:h2=farbe
  h=ABS(farbe-af)
  FOR i= 4 TO 6
    nv(i)=(w(i)-af(i))/h
  NEXT
  IF s=-1 THEN nv(4)=(w(4)-6-af(4))/h
  FOR i=1 TO h-1
    FOR j=4 TO 6
      af(j)=af(j)+nv(j)
      IF af(4)<0 THEN af(4)=af(4)+t(4)
      IF af(4)>=t(4) THEN af(4)=af(4)-t(4)
    NEXT j
  CALL rgb(af(4),af(5),af(6),af(1),af(2),af(3))
  PALETTE af+i*s,af(1)*15/16,af(2)*15/16,af(3)*15/16
  NEXT i
GOTO werte

ausgabe:
IF MENU (0)<>1 THEN RETURN
p=MENU(1)
IF p=5 THEN ende
CLS
IF p=2 OR p=4 THEN INPUT "Dateiname : ",a$(2)
PRINT "Erstes Register"
INPUT "(0-31 / -1=Abbruch) :",a
IF a>31 OR a<0 THEN start
IF p=4 THEN
  OPEN a$(2) FOR INPUT AS 1
    INPUT #1,a1,b
    FOR i=0 TO b-a1
      INPUT #1,r,g,b
      PALETTE a+i,r,g,b
    NEXT i
  CLOSE 1
  GOTO start
ELSE
  b=31
  INPUT "Endregister (0-31) : ",b
  IF b<a OR a>31 THEN start
  OPEN a$(p) FOR OUTPUT AS 1
    PRINT #1,a,b
    FOR i= a TO b
      PRINT #1,USING " #.##";FNrgb(i,1),FNrgb(i,2),FNrgb(i,3)
```

```
    NEXT i
  CLOSE 1
  PRINT "Taste druecken"
  WHILE INKEY$="":WEND
  GOTO start
END IF


SUB hsv (r,g,b,h,s,v) STATIC
  v=r
  IF v<g THEN v=g
  IF v<b THEN v=b
  min=r
  IF min>g THEN min=g
  IF min>b THEN min=b
  IF min<>v THEN
    s=(1-min/v)
    IF r=v THEN
      h=(g-b)/(v-min)
    ELSEIF g=v THEN
      h=2+(b-r)/(v-min)
    ELSEIF b=v THEN
      h=4+(r-g)/(v-min)
    END IF
    IF h<0 THEN h=h+6
  ELSE
    s=0
    h=0
  END IF
END SUB


SUB rgb(h,s,v,r,g,b) STATIC
   r=v
   g=v
   b=v
   IF s<>0 THEN
     h1=v*(1-s)
     h2=v*(1-s*(h-INT(h)))
     h3=v*(1-s*(1-h+INT(h)))
     IF INT(h)=0 THEN
       g=h3:b=h2
     ELSEIF INT(h)=1 THEN
       r=h2:b=h1
     ELSEIF INT(h)=2 THEN
       r=h1:b=h3
     ELSEIF INT(h)=3 THEN
       r=h1:g=h2
     ELSEIF INT(h)=4 THEN
       r=h3:g=h1
     ELSEIF INT(h)=5 THEN
       g=h1:b=h2
```

```
    END IF
  END IF
END SUB
```

Dieses Programm ist ein klassischer Beweis dafür, daß sich die Länge von einen Programm mit mehr Komfortabilität drastisch erhöht. Aber wenn das Programm erst einmal im Computer ist, freut man sich über jede Arbeitserleichterung.

1.5.2.1 Programmbedienung

In der oberen Bildschirmhälfte sind 32 Farben aufgereiht. Jede Farbe entspringt den Werten eines Registers. Wenn Sie eine von diesen Farben ändern wollen, brauchen Sie sie nur mit der Maus anzuklicken.

Auf der linken Seite befindet sich ein größerer Kasten. Hier wird die jeweils aktuelle Farbe angezeigt.

Die Werte der Farbe werden auf dem Bildschirm durch drei Balken dargestellt. Man kann sie durch Anklicken der "+" und der "-" Felder oder durch Anklicken der Balken ändern.

Damit man die Farbänderungen auch in die eigenen Programme einbauen kann, bietet das Programm drei verschiedene Möglichkeiten an, die Daten herauszugeben. Man kann sie sich auf den Bildschirm oder den Drucker ausgeben lassen oder auf Diskette abspeichern. Diese Möglichkeiten erreichen Sie über das Menü.

Unter den drei Balken für die Rot-, Grün- und Blauanteile finden Sie drei weitere Balken. Diese Balken enthalten ebenfalls die Informationen der aktuellen Farbe. Aber in diesen Balken werden die Farben durch ein anderes Farbmodell ausgedrückt.

1.5.2.2 Das HSV-Farbmodell

Es gibt viele verschiedene Möglichkeiten, eine Farbe darzustellen. Sehr viele dieser Möglichkeiten finden auch irgendwo eine

Verwendung, haben aber für den Amiga keinen praktischen Nutzen. Beim Amiga wie auch bei anderen Computern wird intern ausschließlich das RGB-Modell (für Rot, Grün und Blau) benutzt. Ein anderes Farbmodell ist das HSV-Modell. Hier stehen die Buchstaben für die englischen Worte Hue (Farbe), Saturation (Sättigung) und Value (Wert).

Saturation und Value nehmen nur Werte zwischen 0 und 1 an. Es gibt, genau wie bei den RGB-Werten, 16 Schritte, d.h. man hat die Wahl zwischen 16 verschiedenen Werten für Value and Saturation. Bei Hue hat man wesentlich mehr Schritte. Es gibt also sehr viel mehr Kombinationen aus HSV-Werten als beim RGB-Modell. Deshalb geben nicht alle Kombinationen unterschiedliche Farben, nicht jede Veränderung eines HSV-Wertes bewirkt auch eine Farbänderung.

Will man mit dem HSV-Modell arbeiten, dann muß man die HSV-Werte in RGB-Werte umrechnen. Diese etwas komplizierte Arbeit übernehmen zwei Routinen am Programmende unseres Farbdesigners. Das Unterprogramm RGBHSV() wandelt HSV- in RGB-Werte um, HSVRGB() erledigt die andere Richtung.

Wenn man sich soviel Arbeit mit dem Hin- und Herrechnen von Werten macht, muß sich das natürlich auch lohnen. Was ist der große Vorteil des HSV-Modells?

Anfangs hatten wir angekündigt, daß man mit dem Programm auch Farben mischen kann. Hierbei setzen wir das HSV-Modell ein. Wenn wir die HSV-Werte zweier Farben haben, brauchen wir nur ihre Mittelwerte zu bilden und erhalten so die gemischte Farbe. Auf diese Weise kann man nicht nur eine Mischfarbe, sondern gleich verschiedene Farben zurechtmischen und bekommt so feine Farbabstufungen zwischen zwei Farben.

Wollen Sie mit dem Farbendesigner diese Farbabstufungen erzeugen, sollten Sie zuerst die erste Farbe, die in die Vermischung eingeht, bestimmen. Danach müssen Sie das Wort "Übergang" mit der Maus anklicken. Bis jetzt ist noch nichts passiert. Erst wenn Sie eine weitere Farbe auswählen, berechnet das Pro-

gramm die Farbübergänge und schreibt sie in die Farbregister zwischen die der ausgewählten Farben. Bei der Auswahl der Farben ist es nicht egal, welche Farbe Sie zuerst anklicken. Es entsteht eine total andere Farbtabelle, wenn Sie die jeweils andere Farbe zuerst anklicken. Hier müssen Sie ausprobieren, welche der beiden verschiedenen Farbabstufungen Sie gerne benutzen möchten.

1.5.2.3 Die Umkehrung von PALETTE

Leider gibt es in BASIC keine Umkehrung zum PALETTE-Befehl. Man kann mit keinem Befehl die Farbanteile abfragen. Doch gerade das ist bei einigen Programmen, wie etwa beim Farbdesigner, sehr wichtig. Deshalb greifen wir an dieser Stelle etwas voraus und gucken direkt in den Speicher, um uns von dort die Werte zu holen. Wie die Adresse zu dieser Farbtabelle zustande kommt, erklären wir später, wenn wir uns vom reinen BASIC lösen.

Damit Sie nicht mit den Peeks belästigt werden, haben wir die Abfrage als Funktion am Programmanfang abgelegt. Sie brauchen sich nicht um Adressen zu kümmern, sondern nur das Farbregister an die Funktionen zu übergeben. Das folgende Programm befindet sich, in leicht abgeänderter Form, auch schon im Farbdesigner. Es gibt drei einzelne Funktionen, für jeden Farbanteil eine.

```
REM Palette-Umkehrung

SCREEN 1,320,200,5,1
WINDOW 2,,,16,1

DEF FNfarbtab=PEEKL(PEEKL(PEEKL(WINDOW(7)+46)+48)+4)
DEF FNrot(f)=(PEEKW(FNfarbtab+2*f) AND 3840)/3840
DEF FNgruen(f)=(PEEKW(FNfarbtab+2*f) AND 240)/240
DEF FNblau(f)=(PEEKW(FNfarbtab+2*f) AND 15)/15

PRINT "RGB-Farbwerte:"

FOR i = 0 TO 31
  LOCATE 5+i MOD 16,1+INT (i/16)*20
  COLOR i
```

```
  PRINT USING "##";i,
  COLOR 1
  PRINT USING " #.##";FNrot(i);FNgruen(i);FNblau(i)
NEXT i

WHILE INKEY$="":WEND

WINDOW CLOSE 2
SCREEN CLOSE 1
```

Dieses Programm druckt zu jeder Farbe die entsprechenden Farbwerte aus. Farbregister und Farbe sind jeweils vor den drei Werten zu lesen und zu sehen.

1.6 Rund um PUT und GET

Damit man seine eigenen Grafiken auch auf Dauer erhalten kann, gibt es den PUT- und den GET-Befehl. Mit diesen Befehlen kann man beispielsweise eine Grafik auf Diskette speichern. Man kann aber noch viel mehr aus diesen Befehlen herausholen, z.B. mit ihnen Animation betreiben.

Eins haben alle Anwendungen von PUT und GET gemeinsam: Bildinformationen werden verarbeitet.

1.6.1 Arbeitsweise von PUT und GET

Mit PUT liest man eine Grafik aus einem bestimmten Bereich, den man im PUT-Befehl angibt. Die Daten werden in einer Feldvariablen abgelegt. Wir benutzen als Feldvariable immer ein Feld, mit dem man nur ganze Zahlen abspeichern kann, alle ganzen Zahlen zwischen 32767 und -32768. Bei diesem Typ ist der Aufbau der Daten denkbar einfach:

1. Feldplatz = Breite
2. Feldplatz = Höhe
3. Feldplatz = Tiefe
4. Feldplatz = Bitplanes

Ab dem vierten Feldplatz folgen alle Bitplanes. Die Tiefe in einer Grafik gibt die Anzahl der verwendeten Bits pro Bildpunkt an. Die erste Bitplane enthält alle ersten Bits aller Punkte, die zweite alle zweiten usw. Deshalb gibt die Tiefe auch die Anzahl der Bitplanes an. Daran kann man schon sehen, daß die Anzahl der Feldplätze, die man für eine Grafik reservieren muß, immer unterschiedlich ist. Neben der Tiefe hängt die Anzahl auch von Breite und Höhe ab.

Bei diesen Daten werden 16 Bits einer Bitplane und einer Bildschirmzeile zusammengefaßt, denn soviel läßt sich in einer kurzen Ganzzahl speichern. Deshalb müssen wir bei der Bestimmung der Speicherplätze die Breite durch 16 teilen. Wenn dabei ein Rest entsteht, die Breite also nicht genau durch sechzehn zu teilen ist, muß auf jeden Fall aufgerundet werden, denn sonst würde man immer die letzten Bildpunkte einer Zeile abschneiden. Diesen Wert muß man nur noch mit der Höhe und der Tiefe multiplizieren. Außerdem muß in je einem Speicherplatz die Breite, die Höhe und die Tiefe angegeben werden. Daraus ergibt sich folgende Formel, mit der man immer die Anzahl der Feldplätze ausrechnen kann:

Speicherplätze=3+Hoehe*Tiefe*INT((Breite+15)/16)

Das erste Programm dieses Kapitels macht nichts weiter, als den Bildschirminhalt mit GET zu retten, den Bildschirm zu löschen und dann das Gerettete wieder auszugeben.

```
REM Demo fuer GET und PUT

OPTION BASE 1
DEFINT f

CIRCLE (170,60),110
PAINT STEP (0,0),2,1
COLOR 1,2
LOCATE 5,10
PRINT "Dieses Demo zeigt"
PRINT TAB(10)"wie man Grafiken"
PRINT TAB(10)"im Speicher ablegen"
PRINT TAB(10)"und auch wieder"
PRINT TAB(10)"Im Bildshirm aus-"
PRINT TAB(10)"geben kann !!!"
```

```
AREA (140,20)
AREA (80,60)
AREA (300,80)
AREAFILL 1
COLOR 1,0

REM Grafik speichern
x1=40 :y1=10
x2=300:y2=120
DIM feld(3+2*(y2-y1+1)*INT((x2-x1+16)/16))
GET (x1,y1)-(x2,y2),feld

REM Grafik wieder ausgeben
FOR i=0 TO 140
  CLS
  PUT (i*3,i),feld
NEXT i
```

Die ersten etwa zwanzig Zeilen des Programms dienen nur dazu, eine Grafik zu erstellen. Von dem so entstandenen Bild haben wir die Koordinaten der oberen linken und der unteren rechten Ecke genommen und damit ausgerechnet, wie viele Speicherplätze wir reservieren müssen.

Ob die Formel, die die Anzahl der benötigten Speicherplätze errechnet, stimmt, können wir ganz einfach überprüfen: Addieren Sie in dieser Formel statt der drei benötigten nur zwei Speicherstellen. Sobald wir das Programm starten, wird eine "Illegal function call"-Meldung auf dem Bildschirm ausgegeben. Diese Meldung wird immer ausgegeben, wenn Sie zuwenig Speicher für GET reserviert haben. Also immer, wenn man genug Speicher zur Verfügung hat, ruhig etwas mehr Speicher zurücklegen!

An dieser Demo können Sie nicht nur die Arbeitsweise der beiden Befehle sehen. Man erkennt auch die hohe Geschwindigkeit, mit der sich die Grafik über den Bildschirm bewegt. Die Bewegung wirkt fließend, und auch während der Bewegung kann man den Schriftzug deutlich lesen. Daran sieht man schon, daß man mit PUT und GET animierte, also bewegte Grafiken auf einfache Weise erstellen kann.

1.6.2 Speichern auf Diskette

Wie wir die Bildinformationen in den Speicher bekommen, haben wir nun schon gesehen. Dort können sie aber nicht ewig bleiben, denn nach dem Ausschalten wären sie gelöscht. Um eine Grafik zu erhalten, muß man sie auf Diskette ablegen.

Das folgende Programm enthält jeweils ein Unterprogramm zum Laden und zum Speichern. Diese Programmteile arbeiten unabhängig vom Programm und können deshalb auch für andere Programme benutzt werden.

Außerdem kann man mit dem Programm auch ganz gut malen. Anfangs hat man nur den kleinen Punkt als Pinsel. Dieser Pinsel arbeitet mit vier Farben. Das Gemalte kann man auf Diskette abspeichern, und man kann das Gespeicherte auch wieder laden und damit auf dem Bildschirm malen.

```
REM Malprogramm

OPTION BASE 1
DEFINT a-z

PRINT "Malen Sie mit der Maus."
PRINT "Wenn Sie einen Teil der"
PRINT "Grafik speichern wollen, druecken"
PRINT "Sie 's'."
PRINT "Mit 'l' koennen Sie die Grafik"
PRINT "wieder laden und mit ihr malen."

WHILE a$<>"e"
  a$=INKEY$
  WHILE a$=""
    IF MOUSE(0)<>0 THEN
      PSET (MOUSE(1),MOUSE(2))
    END IF
    a$=INKEY$
  WEND
  IF a$="l" THEN GOSUB bildmalen
  IF a$="s" THEN GOSUB bildretten
  IF a$>="0" AND a$<"4" THEN COLOR VAL(a$)
WEND

bildmalen:
    DIM feld(10000)
    WINDOW 2,,(0,0)-(600,0),16
```

```
    INPUT "Dateiname :";b$
    IF b$<>"" THEN
      CALL laden(b$,feld())
    END IF
    WINDOW CLOSE 2
    WHILE INKEY$=""
      IF MOUSE (0)<>0 THEN
        PUT(MOUSE(1),MOUSE(2)),feld
      END IF
    WEND
    ERASE feld
RETURN

bildretten:
    WINDOW 2,,(0,0)-(600,0),16
    INPUT "Name ";b$
    IF b$<>"" THEN
      PRINT "Setzen Sie mit der Maus ";
      PRINT "die Eckpunkte der Grafik.";
      IF MOUSE(0)=0 THEN 1
      ax=MOUSE(1):ay=MOUSE(2)
      IF MOUSE(0)<>0 THEN 2
      IF MOUSE(0)=0 THEN 3
      bx=MOUSE(1):by=MOUSE(2)
      WINDOW CLOSE 2
      CALL speichern (b$,ax,ay,bx,by,2)
    ELSE
      WINDOW CLOSE 2
    END IF
RETURN

' Die folgenden Unterprogramme
' sind programmunabhaengig.

SUB speichern (n$,x1,y1,x2,y2,tiefe) STATIC

  e=3+(y2-y1+1)*tiefe*INT((x2-x1+16)/16)
  DIM grafik%(e)
  GET (x1,y1)-(x2,y2),grafik%
  OPEN n$ FOR OUTPUT AS #1
  FOR i=1 TO e
    WRITE #1, grafik%(i)
  NEXT i
  CLOSE #1
  ERASE grafik%
END SUB

SUB laden (Dateiname$,grafik%(1)) STATIC

OPEN Dateiname$ FOR INPUT AS 1
INPUT #1,grafik%(1),grafik%(2),grafik%(3)
e=3+grafik%(3)*grafik%(2)*INT((grafik%(1)+15)/16)
```

```
FOR i=4 TO e
  INPUT #1, grafik%(i)
NEXT i
CLOSE 1

END SUB
```

Wenn man das ganze Bild abspeichern will, reicht der Speicher höchstwahrscheinlich nicht aus. Abhilfe kann man schaffen, indem man einmalig vor Programmstart den Speicher heraufsetzt:

```
CLEAR ,40000
```

Damit man das Bild anschließend auch wieder laden kann, muß man gleichzeitig im Programm bei der Marke BILDMALEN die Speicherplätze von FELD auf ca. 15000 erhöhen.

Wenn Sie die beiden Unterprogramme "Laden" und "Speichern" in Ihre Programme übernehmen, müssen Sie bestimmte Dinge beachten: An das Unterprogramm "Speichern" müssen der Dateiname, unter dem die Grafik gespeichert werden soll, der linke obere und der rechte untere Eckpunkt der Grafik sowie die Tiefe übergeben werden.

An "Laden" muß der Dateiname und ein Feld, das groß genug ist, um die Grafik in sich aufzunehmen, übergeben werden. Um sicherzustellen, daß kein Fehler auftritt, sollten Sie lieber zuviel als zuwenig Speicher für das Feld abzweigen.

Beide Programmteile kann man beliebig oft hintereinander aufrufen. Dafür sorgt beispielsweise auch der ERASE-Befehl am Ende des Unterprogramms "Speichern". Da die Variablen der Subs zwischen zwei Aufrufen ihre Werte behalten, würde ohne ERASE ein "Duplicate definition"-Fehler auftreten.

Das Laden und Speichern auf Diskette wird mit den allgemein üblichen Dateiverwaltungsbefehlen OPEN, INPUT#, WRITE# und CLOSE erledigt. Mit OPEN und CLOSE öffnet und schließt man eine Datei auf Diskette. Beim Öffnen gibt man jeweils an, ob man aus einer Datei lesen oder in eine schreiben möchte.

Letzteres macht man mit WRITE#, was ähnlich der Ausgabe auf den Bildschirm ist. Das Einlesen der Daten ist mit der Tastatureingabe zu vergleichen.

Bei "Laden" wird, bevor die Bitplanes geladen werden, die Information über Breite, Höhe und Tiefe geladen. Mit diesen Werten wird dann errechnet, wieviel Daten geladen werden müssen.

1.6.3 Noch mehr Möglichkeiten mit PUT

Wie Sie vielleicht beim Malen mit PUT im letzten Programm festgestellt haben, überdeckt eine mit PUT in den Bildschirm geschriebene Grafik den schon vorhandenen Inhalt nicht, sondern verbindet sich irgendwie mit dem alten Bildschirminhalt. Vielleicht ist Ihnen auch aufgefallen, daß, wenn man eine Grafik zweimal an dieselbe Stelle setzt, die Grafik wieder verschwindet.

1.6.3.1 Der Standardmodus von PUT

Beide Phänomene gehören zusammen. Die Ursache liegt im PUT-Befehl. Statt den Bildschirm einfach zu überdecken, wird jeder Punkt der Grafik mit den alten Bildschirminhalten durch ein XOR verknüpft. Die Verknüpfungstabelle von XOR sieht so aus:

```
0 XOR 0  = 0
0 XOR 1  = 1
1 XOR 0  = 1
1 XOR 1  = 0
```

Wenn dabei zwei gesetzte Punkte übereinander fallen, wird der Punkt gelöscht. Es wird nur ein Punkt gesetzt, wenn ein ungesetzter und ein gesetzter Punkt übereinander fallen.

Sicher ist dies nicht für jeden Anwendungszweck optimal. Aber es geht auch anders. Hinter den PUT-Befehl kann man verschiedene Befehlsworte hängen, die Sie auch schon als Befehl oder Funktion kennen und mit denen man vollkommen andere Effekte erzielt (siehe unten).

Für den voreingestellten, oben beschriebenen Modus ist das Befehlswort XOR. Ausgeschrieben würde der Standard-PUT-Befehl folgende Syntax haben:

```
PUT (x,y),feld,XOR
```

Mit Hilfe von XOR kann man sehr gut Animation betreiben, denn man kann ein Objekt über einen Hintergrund bewegen, ohne ihn zu verändern. So macht es auch die Kugel im folgenden Programm.

```
REM Bewegte Bilder mit PUT

DEFINT g
COLOR 2,0
LOCATE 10,10
PRINT "Dieses Programm laesst eine Kugel ueber"
LOCATE 11,10
PRINT "den Bildschirm wandern, ohne dass sie"
LOCATE 12,10
PRINT "diese Schrift veraendert!!"

DEFINT g
DIM g(250)
CIRCLE (20,20),20,1,,,.5
PAINT (20,20),3,1
GET (0,10)-(40,30),g

PUT (0,10), g, XOR

WHILE INKEY$=""
  FOR i=0 TO 180
    PUT (2*i,i), g ,XOR
    PUT (2*i,i),g,XOR
  NEXT i
WEND
```

Wenn man den PUT-Befehl zweimal an derselben Stelle ausführt, wird genau das alte Bild wieder hergestellt.

Wie Sie sehen, kann unser simples Animationsobjekt ziemlich viel von dem, was kompliziert aufgebaute Sprites des 64er oder anderer Computer geschafft haben. Andere Effekte der Sprites kann man mit anderen Modi des PUT-Befehls erreichen.

Das Geheimnis der verschiedenen Modi und der Geschwindigkeit des PUT-Befehls liegt in einem Grafik-Coprozessor mit dem Namen Blitter. Dieser schiebt in ungeahnter Geschwindgkeit Daten im Speicher umher.

Außerdem kann er auch noch etwas anderes: z.B. mit den Daten, die er verschiebt, gleichzeitig auch bestimmte Verknüpfungen durchführen, wie etwa die XOR-Verknüpfung.

1.6.3.2 Der direkte Weg

Wenn man es will, kann man den Blitter auch dazu veranlassen, alle Verknüpfungen zu unterlassen und die Daten einfach an die gewünschte Stelle auf dem Bildschirm zu bringen. Diesen direkten, aber nicht unbedingt schnelleren Weg begeht man im PSET-Modus.

Wenn wir im letzten Programm einen PUT-Befehl löschen und beim anderen PSET mit XOR vertauschen, sehen wir genau seine Wirkungsweise. So, wie die Grafik erstellt wurde, so zieht sie auch über den Bildschirm. Alles, was ihr im Weg ist, wird übermalt. Dabei wird nicht nur das gelöscht, was sich unter unserer Kugel befindet, sondern das ganze Viereck, das wir mit GET gespeichert hatten. Denn auch weiße Stellen übertünchen bei PUT-PSET den Bildschirm.

1.6.3.3 Grafiken invertieren

Wie beim Setzen der Punkte, so gibt es zu PUT-PSET auch ein Gegenstück mit dem Namen PRESET. Im PRESET-Modus kann man auf einfache Weise eine Grafik invertieren. Dazu reichen zwei Zeilen aus:

```
GET (X1,Y1)-(X2,Y2),g
PUT (X1,Y1),g, PRESET
```

g ist ein Ganzzahlfeld, das man vor diesen beiden Zeilen definiert und dem genügend Speicherplatz zur Verfügung gestellt sein muß.

Eine Grafik zu invertieren bedeutet, jedes Bit der Grafik zu invertieren. Alle Einsen in den Bit-Ebenen werden dann zu Nullen. Dadurch werden die Farben aus anderen Farbregistern entnommen als vorher. Das neue Farbregister kann man folgendermaßen errechnen:

Neues Register=2 ^ Tiefe der Grafik - altes Register

Wie sich die Farben ändern, können wir auch in einem kleinen Programm zeigen:

```
REM Farbwechsel mit PRESET

DEFINT f
DIM f(400)

SCREEN 1,320,200,5,1
WINDOW 2,,(0,0)-(287,30),16,1

FOR i=0 TO 31
LINE (i*9,0)-(i*9+8,40),i,bf
NEXT i

WHILE INKEY$=""
FOR i=31 TO 0 STEP -1
  GET (i*9,0)-(i*9+8,40),f
  PUT (i*9,0),f,PRESET
NEXT i
WEND

WINDOW CLOSE 2
SCREEN CLOSE 1
```

Es gibt zwei verschiedene Möglichkeiten, ein Bild, das mit PUT-PRESET invertiert wurde, zu restaurieren. Die erste Möglichkeit invertiert einfach den gleichen Bereich ein zweites Mal.

Diese Technik haben wir im folgenden Programm ausgenutzt. Dabei wandert ein Rechteck über den Bildschirm. Überall, wo es auftaucht, wird dieser Bereich invertiert. Bevor das Rechteck weiterzieht, wird die gleiche Stelle ein zweites Mal invertiert, so daß der alte Hintergrund wieder zu sehen ist:

```
REM Invertierdemo

DEFINT f
DIM f(100)

CIRCLE (160,80),100,1
PAINT STEP(0,0),2,1
LINE(120,50)-(180,160),1,b
PAINT (121,51),3,1
PAINT (179,159),1,1

WHILE INKEY$=""
  FOR i=0 TO 160
    GET (i,i)-(i+20,i+20),f
    PUT (i,i),f,PRESET
    FOR t=0 TO 200: NEXT
    GET (i,i)-(i+20,i+20),f
    PUT (i,i),f,PRESET
  NEXT i
WEND
```

Die zweite Möglichkeit ist etwas kürzer und einfacher. Nach dem Invertieren gibt man die Grafik mit PUT-PSET noch einmal an derselben Stelle aus.

1.6.3.4 Und oder Oder

Es gibt noch zwei weitere Modi der Verknüpfung bei PUT. Dabei handelt es sich um die einfache Und- und Oder-Verknüpfung.

Bei diesen Modi werden - wie bei XOR - nicht die Punkte, sondern die einzelnen Bits eines Punktes, die seine Farbe angeben, verknüpft. Bei der Und-Verknüpfung wird nur dann ein Bit eines Punktes gesetzt, wenn sowohl bei der mit GET ge-

speicherten Grafik als auch beim vorhandenen Bild eine 1 vorliegt. Dabei kann es bei einer Tiefe von zwei mit vier maximalen Farben zu folgenden Kombinationen kommen:

```
0 AND 0  = 0
0 AND 1  = 0
1 AND 0  = 0
1 AND 1  = 1
```

Es kommen also nur folgende Farben zustande:

- Ist ein Punkt blau (00), dann spielt die zweite Farbe keine Rolle. Der Punkt ist immer blau (00).

- Orange (11) und eine zweite Farbe ergibt die zweite Farbe.

- Bei zwei gleichen Farben bleibt diese Farbe.

- Weiß (01) und schwarz (10) wird zu blau (00).

Daß genau diese Kombinationen auftreten, kann man ganz einfach zeigen:

```
REM Und-Verknüpfung

DEFINT f
DIM f(100)

CIRCLE (160,80),100,1
PAINT STEP(0,0),2,1
LINE(120,50)-(180,160),1,b
PAINT (121,51),3,1
PAINT (179,159),1,1

PRINT "test"
GET (0,0)-(39,8),f

FOR i=20 TO 160 STEP 8
  PUT (i,i),f,AND
NEXT i
```

Mit GET legen wir den Schriftzug "Test" als Grafik im Speicher ab. Mit dieser Grafik gehen wir dann über einen vierfarbigen

Bildschirm. Wie vorausgesagt, ist die Grafik, in unserem Fall der weiße Text, nur bei orangefarbigem und weißem Untergrund zu lesen.

Wenn wir statt AND die OR-Verknüpfung anwenden, tritt ein fast gegensätzlicher Effekt ein. Der Text ist bei blauen und schwarzem Untergrund zu lesen. Außerdem überdeckt der blaue Hintergrund der Schrift diesmal nicht die Bildschirminhalte. Die Verknüpfungstabelle zu OR lautet:

```
0 OR 0   = 0
0 OR 1   = 1
1 OR 0   = 1
1 OR 1   = 1
```

1.7 Animation in BASIC

Animation ist eine zu Recht viel gerühmte Fähigkeit, auf die Sie als Besitzer des Amiga stolz sein können. Sogar in BASIC kann man sehr leicht Animation betreiben. Dafür sorgen die zahlreichen eingebauten Befehle, die alle mit dem Wort OBJECT beginnen.

Um tolle Effeke erzielen zu können, muß man wissen, wie man mit diesen Befehlen umgehen muß und kann. Das BASIC-Handbuch erwähnt zwar alle Befehle, zeigt aber nicht, was man aus ihnen herausholen kann. So werden zwar das Erstellen von Bobs mit einem mitgelieferten Programm und die Handhabung der Befehle kurz angeschnitten, aber über einige wirklich interessante Dinge, wie die COLLISION-Maske, ist nichts gesagt.

1.7.1 Sprites und Bobs

Die Animationsbefehle des BASIC sind sowohl für Sprites als auch für Bobs zuständig. Es gibt keine Befehle, die nur für eins

von beiden gelten. Manchen unter Ihnen mögen die Begriffe Bob und Sprite vielleicht nicht viel sagen, deshalb hier eine kurze Definition:

Wie schon erwähnt, sind beides bewegte Grafiken. Sprites kennen einige von Ihnen vielleicht vom 64er oder anderen Homecomputern. Sprites sind beim Amiga bis zu sechzehn Pixel breite, beliebig hohe, frei definierbare, hochaufgelöste Grafiken mit bis zu drei verschiedenen Farben. Sie bewegen sich sehr schnell über den Bildschirm. Ihre Bedienung ist einfach, denn sie benötigen nur wenig Programmieraufwand und sind einfach zu kontrollieren.

Bobs sind ähnlich, nur für etwas andere Anwendungen, denn ihre Größe ist beliebig, läßt man die Frage des freien Speicherplatzes mal außer acht. Sie besitzen, dem jeweiligen Screen entsprechend, bis zu 32 verschiedene Farben, bewegen sich aber langsamer als Sprites. Sie werden auch mit anderer Technik auf den Bildschirm gebracht.

Außerdem unterscheiden sich Bobs und Sprites noch durch ein gesetztes bzw. nicht gesetztes Bit in einer Datei, aber dazu später mehr. Benutzt man mehr als vier Sprites auf einmal, kann es schon zu Komplikationen kommen, während die Zahl der Bobs bei grenzenloser Speicherkapazität auch grenzenlos ist.

1.7.2 Am Anfang war der OBJECT.SHAPE-Befehl

Im Gegensatz zum guten alten Commodore 64 muß man die Daten der Bobs und Sprites nicht selber in den Speicher poken. Dafür gibt es den OBJECT.SHAPE-Befehl. Man schreibt alle wichtigen Daten einfach in eine Zeichenkette und übergibt diese an OBJECT.SHAPE. Das ist aber auch der einzige Weg, mit BASIC-Befehlen die Form des Objektes zu bestimmen.

Aber OBJECT.SHAPE versteht natürlich nicht jede Zeichenkette. Eine Zeichenkette, die die Daten eines Bobs oder Sprites

enthält, muß in ganz bestimmter Weise aufgebaut sein. Deshalb im folgenden nähere Erläuterungen dazu, wie man ein Objekt selbst definiert.

1.7.3 Erstellen der Objekte

Damit man diese Schwierigkeiten nicht hat, gibt es Programme, die aus einer von Ihnen erstellten Grafik grafische Objekte formen. OBJEDIT, das mitgelieferte BASIC-Programm, ist dieser Aufgabe nur begrenzt gewachsen. Es ist unter anderem nicht fähig, selbstdefinierte Kollisionsmasken oder Schattenbilder zu verarbeiten (was das ist und wofür man das braucht, erklären wir weiter hinten).

Wofür hat man alle diese phantastischen Möglichkeiten, wenn man sie mangels Software nicht nutzen kann? Nicht verzweifeln, Sie haben ja noch uns.

1.7.4 Eddi III macht es möglich

Als erstes haben wir ein sehr komfortables Programm geschrieben, mit dem man Objekte aller Art herstellen kann. Es bietet wesentlich mehr Möglichkeiten als OBJEDIT, der mit dem Computer mitgelieferte Objekt-Editor. Mit unserem Editor, Eddi III, kann man Objekte mit einer Breite von bis zu 309 und einer Höhe von bis zu 180 Pixeln konstruieren. Man kann die Tiefe verändern, den Objekten die verschiedensten Eigenschaften mitgeben und seine Objekte im Programm ausprobieren. Zum Malen hat man sehr viele Befehle zur Auswahl. Wenn einem eine Grafik gefällt, kann man sie als Objekt oder auch als einfache Grafik abspeichern, die man mit dem PUT-Befehl wieder auf den Bildschirm bringen kann. Zusätzlich enthält das Programm auch einen Programmgenerator. Wenn man ihn aufruft, werden die Daten so auf Diskette gespeichert, daß man sie nachher als Programm in den BASIC-Editor einladen kann.

Mehr als alle Worte kann das Programm selbst Sie bestimmt überzeugen. Wegen seiner Länge wird es wohl nur auf Computern mit 512 KByte und mehr Speicher laufen. Hier ist es:

```
REM *****************
REM Grafik-Editor III
REM *****************
REM
REM Jens Trapp  12/88
REM

CLEAR ,43000&    'Arbeitsspeicher vergroessern
OPTION BASE 1
DEFINT a-e,fe,g,h,j-r,t-z
DEF FNe(b,h,t)=(3+t*h*INT((b+15)/16))

REM Farben
DIM f(32,3)

REM Menues aendern

MENU 1,0,1,"Windows"
MENU 1,1,1,"Laden    "
MENU 1,2,1,"Speichern"
MENU 1,3,1,"Probe    "
MENU 1,4,1,"Loeschen "
MENU 1,5,1,"Farben   "
MENU 1,6,1,"Prog.Gen."
MENU 1,7,1,"Ende"

MENU 2,0,1,"Tools"
MENU 2,1,2,"  Punkte 0"
MENU 2,2,1,"  Linien "
MENU 2,3,1,"  Rahmen "
MENU 2,4,1,"  Kaesten"
MENU 2,5,1,"  Kreise "
MENU 2,6,1,"  Fuellen"


MENU 3,0,1,"Tiefe"
MENU 3,1,1,"  1"
MENU 3,2,1,"  2"
MENU 3,3,1,"  3"
MENU 3,4,1,"  4"
MENU 3,5,2,"  5"

MENU 4,0,1,"Flags"
MENU 4,1,1,"  Sprite   "
MENU 4,2,1,"  Collision"
MENU 4,3,1,"  Shadow   "
MENU 4,4,2,"  Saveback "
```

```
MENU 4,5,2,"  Overlay  "
MENU 4,6,1,"  Savebob  "
MENU 4,7,1,"PlanePick  "
MENU 4,8,1,"PlaneOnOff "

ON MENU GOSUB verzweigen   'Menues aktivieren
MENU ON
MOUSE ON

REM voreingestellte Werte
stiefe=5        'Screentiefe
tiefe=stiefe    'Tiefe der Grafik
breite=100
hoehe=100
dicke=0

DIM feld (FNe(breite,hoehe,tiefe))
farbe=1         'Farbe = weiss
farbealt=1      'Rahmenfarbe = weiss
tool=1          'Anfangsbefehl = "punkte"

REM flags fuer standartobjekte
planepick=2^tiefe-1
saveback=1
overlay=1

REM Bildschirm aufbauen
SCREEN 1,320,200,stiefe,1
WINDOW 2,,,16,1
FOR i=2 TO 4
  FOR j=1 TO 3
    READ f(i,j)
  NEXT j
PALETTE i-1,f(i,1),f(i,2),f(i,3)
NEXT i

DATA 1,1,1,0,0,0,1,.53,0

GOSUB schirm


Abfrage:
    a$=INKEY$
    WHILE a$=""
      x=MOUSE(1)
      y=MOUSE(2)
      LOCATE 24,32
      IF x<breite AND y<hoehe THEN
        PRINT USING " ###";x,y;
      ELSE
        PRINT "        ";
      END IF
```

```
      IF MOUSE(0)<>0 THEN GOSUB zeichnen
      a$=INKEY$
    WEND


REM Format aendern
IF ASC(a$)=28 AND hoehe>1 THEN
  LINE (0,hoehe+1)-(breite+1,hoehe+1),8
  hoehe=hoehe-1
  feld(2)=hoehe
  LINE (0,0)-(breite+1,hoehe+1),farbealt,b
END IF
IF ASC(a$)=29 AND hoehe<178 THEN
  LINE (1,hoehe+1)-(breite,hoehe+1),0
  hoehe=hoehe+1
  LINE (0,0)-(breite+1,hoehe+1),farbealt,b
END IF
IF fvsprite=0 THEN
  IF ASC(a$)=31 AND breite>1 THEN
    LINE (breite+1,0)-(breite+1,hoehe+1),8
    breite=breite-1
    feld(1)=breite
    LINE (0,0)-(breite+1,hoehe+1),farbealt,b
  END IF
  IF ASC(a$)=30 AND breite<WINDOW(2) THEN
    LINE (breite+1,1)-(breite+1,hoehe+1),0
    breite=breite+1
    LINE (0,0)-(breite+1,hoehe+1),farbealt,b
  END IF
END IF
IF ASC(a$)>47 AND ASC(a$)<58 THEN
   dicke=ASC(a$)-48: MENU 2,1,1-(tool=1)," Punkte"+STR$(ASC(a$)-48)
END IF
IF a$<>"q" THEN Abfrage

speichern:  b=1
     ERASE feld
     DIM feld(FNe(breite,hoehe,stiefe))
     GET (1,1)-(breite,hoehe),feld
     CALL dateiname("Save",n$,b)
     IF n$="" OR b=2 THEN RETURN
     OPEN n$ FOR OUTPUT AS 2
       IF b=1 THEN
       GOSUB BFormat
       PRINT #2,a$;
       ELSE
         PRINT #2,breite,hoehe,tiefe
         FOR i=4 TO FNe(breite,hoehe,tiefe)
           PRINT #2,feld(i)
         NEXT i
       END IF
       CLOSE 2
```

```
RETURN

ProgGen:  b=1
     ERASE feld
     DIM feld(FNe(breite,hoehe,stiefe))
     GET (1,1)-(breite,hoehe),feld
     CALL dateiname("ProGen",n$,b)
     IF n$="" OR b=2 THEN RETURN
     OPEN n$ FOR OUTPUT AS 2
       IF b=1 THEN
         GOSUB BFormat
         Laenge =LEN(a$)
         PRINT #2,STR$(Laenge/2)
         a$=a$+MKL$(0)
         FOR i= 1 TO Laenge STEP 2
           IF (i-1) MOD 24 =0 THEN
             PRINT #2,
             PRINT #2,"DATA ";
           ELSE
             PRINT #2,",";
           END IF
           PRINT #2,STR$(CVI(MID$(a$,i,2)));
         NEXT i
       ELSE
         PRINT #2,STR$(FNe(breite,hoehe,tiefe))
         feld(3)=tiefe
         FOR i=1 TO FNe(breite,hoehe,tiefe)
           IF (i-1) MOD 24 =0 THEN
             PRINT #2,
             PRINT #2,"DATA ";
           ELSE
             PRINT #2,",";
           END IF
           PRINT #2,STR$(feld(i));
         NEXT i
         feld(3)=stiefe
       END IF
       PRINT #2,
       CLOSE 2
RETURN

BFormat:
    a$=MKL$(0)+MKL$(0)
    a$=a$+MKI$(0)+MKI$(tiefe)
    a$=a$+MKI$(0)+MKI$(breite)
    a$=a$+MKI$(0)+MKI$(hoehe)
    flags=fvsprite+2*collmask+4*shadmask+8*saveback
    flags=flags+16*overlay+32*savebob
    a$=a$+MKI$(flags)
    a$=a$+MKI$(planepick)
    a$=a$+MKI$(planeonoff)
    FOR i=4 TO FNe(breite,hoehe,tiefe)
```

```
    a$=a$+ MKI$(feld(i))
  NEXT i
  IF shadmask THEN
    IF shad$="" THEN GOSUB shad2
    OPEN shad$ FOR INPUT AS 1
    INPUT #1,b,h,t
    IF b<>breite OR h<>hoehe THEN
      LOCATE 10,4
      PRINT "Ungleiches Format der ShadowMask"
      CLOSE 1
      WHILE INKEY$="":WEND
      GOTO schirm
    END IF
    FOR i=4 TO FNe(breite,hoehe,1)
      INPUT #1,a
      a$=a$+MKI$(a)
    NEXT i
    CLOSE 1
  END IF
  IF collmask THEN
    IF coll$="" THEN coll2
    OPEN coll$ FOR INPUT AS 1
    INPUT #1,b,h,t
    IF b<>breite OR h<>hoehe THEN
      LOCATE 10,4
      PRINT "Ungleiches Format der CollisionMask"
      CLOSE 1
      WHILE INKEY$="":WEND
      GOTO schirm
    END IF
    FOR i=4 TO FNe(breite,hoehe,1)
      INPUT #1,a
      a$=a$+MKI$(a)
    NEXT i
    CLOSE 1
  END IF
  IF fvsprite THEN
    FOR i=2 TO 4 'Sprite Farben
      a$=a$+MKI$(INT(f(i,1)*15)*256+INT(f(i,2)*15)*16+f(i,3)*15)
    NEXT i
  END IF
RETURN
```

Die wichtigsten Variablen:

feld Das ist ein kurzes Integerfeld. In dieses Feld werden die Bildinformationen geschrieben. Es kann einfach mit PUT oder GET auf den Bildschirm gebracht werden.

f	Enthält Farbwerte. Wichtig für die Sprites!
breite	Breite der Grafik. Da unsere Grafik immer bei (1,1) anfängt, gleichzeitig maximaler x-Wert beim Zeichnen.
hoehe	Höhe und letzter y-Wert der Grafik.
tiefe	Aktuelle Tiefe der Grafik. Dieser Wert beeinflußt nicht die Tiefe des Bildschirms.
stiefe	Tiefe des Bildschirms. Dieser Wert bleibt immer gleich.
x	x-Koordinate beim Zeichnen.
y	y-Koordinate zum Zeichnen. Der x- und y-Wert werden neben dem Farbstrahl angezeigt.
xalt	Anfangskoordinate beim Zeichnen von Linien etc.
yalt	y-Koordinate zu xalt.
farbe	Aktuelle Zeichenfarbe.
farbealt	Farbe des Rahmens beim Füllen. Wird bei jedem Aufruf von "Füllen" gleich farbe gesetzt.
tool	Gibt die Nummer des aktuellen Zeichenbefehls an.
titel	Nummer des aktivierten Menüs.
punkt	Nummer des aktivierten Menüpunktes.
b	Hilfsvariable zur Formatunterscheidung.
dicke	Pinselbreite für den Befehl "Punkte".
n$	Dateiname zum Laden und Speichern.
coll$	Dateiname für die COLLISION-Maske (siehe unten).
Shad$	Dateiname für die Shadowmask (siehe unten).

Alle Objektvariablen und Flags siehe unten.

1.7.4.1 Der Bildschirm

Wenn Sie das Programm gestartet haben, erscheint Ihr neuer Arbeitsplatz auf dem Fenster: Eddis eigenes Fenster in seinem eigenen Bildschirm. In diesem Fenster sehen Sie zwei Dinge: Zum einen alle 32 Farben in kleinen Kästchen am unteren Bildschirmrand, zum anderen eine leere obere Ecke, die durch einen weißen Rahmen vom sonst weinroten Fenster abgegrenzt ist. Dies ist das Gebiet, in dem sich Ihre Phantasie austoben soll. Dort soll die Grafik hineingezeichnet werden.

1.7.4.2 Ein Programm mit Format

Das erste, was man für ein Objekt machen sollte, ist seine Größe bestimmen. Zwar kann die Größe des Objekts auch später noch geändert werden, doch wird man sich am Anfang schon etwas über Form und Ausmaß des Objektes im Klaren sein. Wenn Sie das Objekt zu groß wählen, also noch sehr viel ungenutzten Rand um die Grafik stehen lassen, haben Sie mehr Daten, die verwaltet werden wollen. Das wirkt sich natürlich negativ auf die Geschwindigkeit aus und sollte darum vermieden werden.

Die Größe ändert man mit den Cursortasten: Die Größe des Objektes wird jeweils um eins vergrößert oder verkleinert.

Die linke obere Ecke des Objekts bleibt immer in der linken oberen Ecke des Bildschirms und läßt sich nicht verändern.

Sowohl die Höhe als auch die Breite kann man bis auf eins heruntersetzen. Da haben wir schon den ersten Superlativ dieses Programms. Sogar eine Grafik dieser Größe, sofern man noch von Größe reden kann, kann man als Objekt definieren. Damit hätten wir das kleinstmögliche Objekt.

Bei der maximalen Ausdehnung haben wir zwar keinen Superlativ zu bieten, aber die 312*180 Punkte dürften, wenn man genügend Speicher hat, für die meisten Gelegenheiten ausreichen.

1.7.4.3 Die Tiefe

Die Tiefe gehört auch zum Format des Objekts. Sie gibt an, wie viele Farben ein Objekt maximal haben darf. Bei Tiefe 5 sind es 32 (2 hoch 5) Farben. Die Tiefe erhält in diesem Programm eine gesonderte Stellung, weil man sie auf andere Weise manipuliert. Sie hat nämlich ein eigenes Menü. Dort kann man die Tiefe abfragen oder ändern.

Die aktuelle Tiefe ist im Menü durch ein Häkchen gekennzeichnet. Die Tiefe ändert man genauso, wie man auch in der Workbench Menüpunkte auswählt.

Wenn man sie verändert, ändert sich auch die Farbenvielfalt der Farbenleiste unten auf dem Bildschirm, denn mit geringerer Tiefe lassen sich ja weniger Farben darstellen. Wenn man vorher schon ein Bild in voller Farbenpracht auf dem Bildschirm hatte, werden auch diese Farben bei Tiefenänderung entsprechend verändert.

1.7.4.4 Die Farbe

Wenn Sie sich den Farbenstrahl einmal genauer angeschaut haben, werden Sie festgestellt haben, daß das erste Rechteck im Verhältnis zu allen anderen viel größer ist. Dieses Kästchen gibt nämlich die aktuelle Zeichenfarbe an.

Die Zeichenfarbe kann man ändern, indem man mit der Maus in der Farbenleiste die Farbe seiner Wahl anklickt.

1.7.4.5 Bilder malen

Nun kommen wir zum Kernstück des Programms: dem Malen. Das Programm hat sechs verschiedene Zeichenmodi. Für die Zeichenmodi gibt es auch wieder ein Menü. Im Menü "Tools" (englisch für Werkzeuge) gibt es eigentlich alles, was man braucht, um ein gutes Bild zu malen. Wenn Sie sich die ent-

sprechenden Zeilen im Listing angucken, werden Sie ein Wiedersehen mit allen Grafikbefehlen feiern können, die wir Ihnen schon vorgestellt haben. Alle Zeichenbefehle arbeiten mit der aktuellen Zeichenfarbe:

PUNKTE N PUNKTE ist ein Zeichenbefehl, wie wir ihn schon in einigen Demos kennengelernt haben. Wenn man auf die linke Maustaste drückt, wird an der Stelle, an der sich der Mauszeiger gerade befindet, ein Punkt zurückgelassen. Dieser Befehl hat aber noch eine Besonderheit. Statt des einfachen Punktes kann man auch mit Quadraten bis zu 9*9 Punkten malen. Die Zahl hinter PUNKTE gibt immer diese Pinselstärke an. Man kann sie jederzeit verändern, indem man auf der Tastatur eine Zahl eingibt.

LINIEN Dieser Befehl entspricht dem BASIC-Befehl LINE. Beim gewünschten Anfangspunkt der Linie drückt man auf die linke Maustaste und bewegt die Maus, während man die Maustaste gedrückt hält, zum Endpunkt. Dort läßt man die Taste wieder los. Solange man die Maustaste gedrückt hält, blinken der Anfangs- und der augenblickliche Endpunkt. Auf diese Weise kann man Ziel und Richtung der Geraden besser abschätzen.

RAHMEN Dieser Befehl malt ein unausgefülltes Rechteck in das Fenster. Die linke obere und rechte untere Ecke werden genau wie oben erklärt bestimmt.

KAESTEN Im Gegensatz zu RAHMEN zeichnet dieser Befehl gefüllte Rechtecke. Ansonsten genau wie RAHMEN.

KREISE Bei diesem Befehl kann man die Parameter für den CIRCLE-Befehl aus BASIC bedienerfreundlich mit dem Mauszeiger bestimmen. Das funktioniert ähnlich wie beim LINIEN-Befehl. Dort, wo

man die Taste herunterdrückt, ist der Kreismittelpunkt. Der Punkt, an dem man die Taste wieder losläßt, gibt die maximale Breite und Höhe des Kreises an. Er selbst ist aber nie ein Kreispunkt. Die Differenz dieses Punktes vom Mittelpunkt in x- und y-Richtung gibt jeweils den horizontalen und vertikalen Radius an. Die beiden Punkte, die man mit der Maus festlegt, markieren also genau ein Viertel des Kreises.

FUELLEN Der letzte Befehl füllt begrenzte Flächen aus. Dabei kann man im Gegensatz zu OBJEDIT die Flächen auch mit einer anderen Farbe als der Rahmenfarbe füllen. Die Rahmenfarbe legt man automatisch beim Anschalten dieses Befehls fest. Es ist immer die zu diesem Zeitpunkt aktuelle Zeichenfarbe. Wenn man die Farbe ändert, während FUELLEN aktiv ist, bleibt die Rahmenfarbe erhalten. Die Rahmenfarbe kann man ändern, indem man einfach ein zweites Mal den FUELLEN-Befehl aufruft. Welches die aktuelle Rahmenfarbe ist, sieht man an der Umrandung des Kästchens mit der aktuellen Zeichenfarbe und dem Rahmen des Objekts.

Wenn einem eine Grafik nicht gefällt, kann man mit dem LÖSCHEN-Befehl aus dem ersten Menü den ganzen Bildschirm löschen.

1.7.4.6 Laden und speichern

Damit man mit den Meisterwerken überhaupt etwas anfangen kann, braucht man einen Lade- und einen Speicher-Befehl. Bei diesen Befehlen sind der GET- und PUT-Befehl des BASIC natürlich unverzichtbar.

Eigentlich gibt es hier je zwei unterschiedliche Lade- und Speicherbefehle. Wie oben schon angedeutet, kann man die Daten in zwei verschiedenen Formaten laden oder speichern.

Das erste haben wir PUT-Format getauft, denn die Daten sind als kurze Integer-Werte abgespeichert. Diese Werte kann man ganz normal, wie in den GET- und PUT-Demos dieses Buches, laden und auch wieder mit PUT in alter Frische auf den Bildschirm ausgeben.

Das zweite Format ist nach den Bobs benannt. Mit ihm werden die Files für Bobs und Sprites erstellt. Diese Files enthalten neben den reinen Bildinformationen noch einige andere Werte für Objekte und können von OBJECT.SHAPE verstanden werden.

Das Programm merkt sich den jeweils letzten Dateinamen eines Aufrufs zum Laden oder Speichern. Wenn man dann noch einmal auf die gleiche Datei zugreifen will, braucht man nur die Return-Taste zu drücken.

1.7.4.7 Objekte ausprobieren

Damit man nicht jedesmal den Editor verlassen muß, um sein Objekt auszuprobieren, kann man es auch gleich vom Editor heraus testen. Wenn dann nicht alles nach Wunsch ist, kann man es sehr schnell wieder ändern.

Objekte, die man austesten will, müssen vorher nicht auf Diskette abgespeichert werden. Die Daten werden direkt vom Bild in das für den OBJECT.SHAPE-Befehl verständliche Format umgewandelt. Man kann ein Objekt also richtig austesten, bevor man es speichert.

1.7.4.8 Objekte im eigenen Programm laden

Um die mit dem Editor erstellten Objekte laden zu können, muß man die Grafik im Bob-Format abgespeichert haben. Dann kann man sie ganz einfach wieder laden.

```
OPEN "Dateiname" FOR INPUT as 1
OBJECT.SHAPE 1,INPUT$(LOF(1),1)
CLOSE 1
```

Nun hat man das Objekt gebrauchsbereit.

1.7.4.9 Der Programmgenerator

Wenn man die Daten nicht von Diskette laden möchte, kann man sie auch als Daten im Programm ablegen. Das geht folgendermaßen vor sich:

Rufen Sie den Programmgenerator ProGen vom Menü aus auf. Nun wird zuerst die Anzahl der Daten abgespeichert. Danach werden alle Daten, wahlweise als Objekt- oder PUT-Grafik, in DATA-Zeilen verpackt auf Diskette abgespeichert. Beenden Sie nun das Programm, dann können Sie die eben abgespeicherten DATA-Zeilen als BASIC-Programm einladen.

Um nun die Grafik oder das Objekt auf den Bildschirm zu bringen, brauchen Sie nur folgende Zeilen ergänzen. Bei Objekt-Format:

```
FOR i=1 TO laenge
READ a
a$=a$+MKI$(a)
next i

OBJECT.SHAPE 1,a$
```

Mit den anderen OBJECT.-Befehlen können Sie das Objekt auf dem Bildschirm sichtbar machen. (In den folgenden Programmen

finden Sie entsprechende Beispiele.) Den Wert, den Sie für laenge einsetzen müssen, finden Sie beim Einladen immer über der ersten DATA-Zeile.

Für normale Grafiken (im Programm PUT-Format genannt) müssen Sie folgende Zeilen zu den DATA-Zeilen ergänzen:

```
OPTION BASE 1
DEFINT f
DIM feld(laenge)

FOR i=1 TO laenge
  read feld(i)
next i

PUT (0,0),feld
```

Daten, die Sie mit ProGen abgespeichert haben, können Sie nicht wieder mit Eddi einladen! Speichern Sie die Grafik zusätzlich noch mit der normalen Speicherfunktion ab.

1.7.5 Die Flags

Die Flags bestimmen, wie das Objekt auf dem Bildschirm erscheint. Ein Flag nimmt nur zwei verschiedene Zustände ein: Entweder es ist gesetzt, also 1, oder ungesetzt und damit 0. Mehrere dieser Flags werden in einem Byte zusammengesetzt. Alle Flags werden dem Computer in der Zeichenkette, die man an OBJECT.SHAPE übergibt, mitgeteilt. Danach kann man sie in BASIC nicht mehr verändern.

In unserem Editor gibt es ein eigenes Menü für die Flags. Gesetzte Flags werden mit einem kleinen Fähnchen im Menü symbolisiert.

Neben den Flags gibt es auch noch zwei andere Funktionen in diesem Menü: PlanePick und PlaneOnOff. Was es mit diesen Funktionen auf sich hat, erklären wir später.

1.7.5.1 Das SaveBack-Flag

Dieses Flag wollen wir zuerst erklären, weil es wohl das einfachste ist. Grundsätzlich tragen alle Flags Namen, die den Benutzer an ihre Bedeutung erinnern sollen. So ist es auch bei SaveBack. Saveback ist die Abkürzung für "Save the Background".

Wenn ein Bob gezeichnet wird, wird es zu einem Teil des Bildschirms. Das, was vorher an dieser Stelle auf dem Bildschirm zu sehen war, wird einfach übermalt. Damit der Hintergrund gespeichert und nachher, nachdem sich das Objekt bewegt hat, wieder restauriert wird, muß man dieses Flag setzen.

Dagegen bleibt das Bild des Objekts auf dem Bildschirm zurück, auch wenn es bewegt wird, wenn das Flag nicht gesetzt wurde. Probieren Sie es doch einmal mit einem eigenen Objekt und dem Editor aus.

Wenn dieses Flag gesetzt ist, kann es bei großen Objekten und bei großer Bildschirmtiefe zu einem Flackern kommen. Das Flackern entsteht, wenn das Objekt über der zu restaurierenden Fläche liegt. Dieses Flackern kann man leider mit BASIC nicht verhindern. Deshalb sollte man dieses Flag nur dann setzen, wenn man es wirklich benötigt.

An diese Stelle gehört wohl das erste Beispielprogramm. Da wir die Objekte im Buch als Daten im Programm abspeichern müssen und die Anzahl der Daten bei großen Objekten noch viel größer ist, verwenden wir im Beispielprogramm nur kleine Objekte.

```
REM Flugzeug

DEFINT a
SCREEN 1,320,200,2,1
WINDOW 2,,,16,1

PRINT "Ich lese die Daten"

FOR i=1 TO 313
```

```
  READ a
  a$=a$+MKI$(a)
NEXT i

LOCATE 10,1
PRINT "Dies ist ein Objekt-Demo: Sie werden ein
PRINT "Flugzeug sehen, das ueber den"
PRINT "Bildschirm fliegt, ohne diese Schrift"
PRINT "zu zerstoeren!!!"

OBJECT.SHAPE 1,a$
nocheinmal:
OBJECT.X 1,1
OBJECT.Y 1,80
OBJECT.VX 1,3
OBJECT.VY 1,20
OBJECT.AX 1,4
OBJECT.AY 1,-2
OBJECT.ON
OBJECT.START

WHILE INKEY$="":WEND
GOTO nocheinmal

REM Daten fuer Flugzeug
DATA 0,0,0,0,0,2,0,88,0,25
DATA 8 :REM Hier werden die Flags gespeichert
DATA 3,0,8160,0,0,0,0,0,16368,0,0,0,0
DATA 0,16376,0,0,0,0,0,16380,0,0,896,0
DATA 0,16382,0,0,-1,-32768&,0,16382,0,1,-28668,-16384
DATA 0,16383,0,3,4100,24576,0,16383,0,6,4100,12288
DATA 0,16383,-32768,12,4100,6144,0,16383,-16384,24,4100,3072
DATA 0,16383,-16384,48,4100,2044,0,16383,-2048,127,-1,-1
DATA 0,16383,-1,-1,-1,-14337,0,16383,-1,-1,-2048,12543
DATA 0,16383,-1,-4,1023,-385,0,4095,-1,-31,-1,-129
DATA -32768,4095,-1,-497,-1,-129,-32768,1023,-1,-257,-1,-129
DATA -32768,1023,-1,-257,-1,-129,-32768,1023,-1,-129,-1,-385
DATA -32768,511,-1,-249,-1,-257,0,7,-1,-32,0,504
DATA 0,0,0,16383,-1,-32,0,0,0,0,2044,0
DATA 0,0,0,0,0,0,0,0,0,0,0,0
DATA 0,0,0,0,0,0,0,0,0,0,0,0
DATA 0,0,0,0,0,0,0,0,8176,0,0,0,0
DATA 0,1536,0,0,28667,0,1024,1648,0,0,-4101,-32768
DATA 1024,1736,0,0,-4101,-32768,1024,1728,0,0,28667,8192
DATA 1024,1728,0,0,28667,-32768,1024,1728,0,0,28667,-32768
DATA 1024,1728,0,0,0,0,1024,1736,0,0,0,0
DATA 2048,1648,0,0,0,0,2048,0,0,0,0,0
DATA 14336,0,0,0,0,127,-2048,0,0,0,0,127
DATA -2048,3,-1,-512,0,0,26624,1023,-1,-512,0,0
DATA 2048,0,0,256,0,0,3072,0,0,0,0,0
DATA 1024,0,0,0,0,0,1024,0,0,0,0,0
DATA 1024,0,0,0,0,0,1024,0,0,0,0,0,1024
```

In diesem Programm sind die gebräuchlichsten OBJECT.-Befehle enthalten. Die meisten dieser Befehle muß man für jedes Objekt angeben. Hier die Bedeutungen im einzelnen:

OBJECT.SHAPE Die Zeichenkette a$, die alle notwendigen Informationen enthält, wird dem Computer übergeben. Wenn wir nun auf das Bob zugreifen wollen, geben wir immer die Nummer an, die wir dem Objekt bei OBJECT. SHAPE zugewiesen haben. In unserem Fall ist es die Nummer 1. Oben haben wir gezeigt, wie man die Daten des Objekts von Diskette lädt.

OBJECT.X/Y Mit diesen beiden Befehlen gibt man die Startposition an. Der erste Parameter gibt jeweils die Nummer des Objekts an.

OBJECT.VX/Y Für das angegebene Objekt legt man eine Geschwindigkeit fest.

OBJECT.AX/Y Die voreingestellte Geschwindigkeit muß nicht konstant bleiben. Mit diesem Befehl kann man eine Beschleunigung angeben.

OBJECT.ON Macht das angegebene Objekt auf dem Bildschirm sichtbar.

OBJECT.START Ohne diesen Befehl würde sich das Objekt trotz Geschwindigkeitsangabe nicht bewegen. Dieser Befehl startet die angegebenen Objekte.

An diesem Beispielprogramm kann man sehr gut die Eigenschaften des SaveBack-Flags erkennen. Obwohl das Flugzeug mitten über den Text fliegt, bleibt dieser erhalten.

Wenn Sie sehen wollen, was passieren würde, wenn das Flag nicht gesetzt wäre, brauchen wir nur den elften Wert unser Da-

tenliste, er steht ganz alleine in der zweiten DATA-Zeile, von 8 auf 0 zu ändern. Wenn Sie das Programm nun starten, bleibt immer eine Ecke vom Flugzeug auf dem Bildschirm stehen.

1.7.5.2 SaveBob

Das Flag SaveBob ist in etwa die Umkehrung zu SaveBack. Wenn SaveBob gesetzt wird, bleibt das Objekt auf dem Bildschirm stehen. Vielleicht ist Ihnen diese Funktion von den großen Malprogrammen bekannt. Man kann auf diese Weise mit dem Objekt auf den Bildschirm malen.

1.7.5.3 Overlay

Wie Ihnen vielleicht bei unserem kleinen Flugzeug aus dem letzten Beispielprogramm aufgefallen ist, überdeckt nicht nur das Flugzeug die Schrift, sondern das ganze Rechteck des Objekts, auch die Stellen, an denen keine Punkte gesetzt wurden. Dieser Effekt ist sicher sehr häufig unerwünscht. Mit dem Overlay-Flag kann man Abhilfe schaffen. Wenn dieses Flag gesetzt ist, werden alle nicht gesetzten Punkte des Objekts transparent. Man kann die Punkte des Hintergrunds sehen, wenn das Objekt an einer Stelle keine Punkte hat. Wir können das an unserem Flugzeug ausprobieren. Das Overlay-Flag können wir nachträglich setzen, indem wir 16 zur 8 des SaveBack-Flags addieren. Der neue Wert der Flags ist also 24. Nun sind beide Flags aktiviert.

Das ist eine von vielen Möglichkeiten, die sich mit Overlay bieten. Die anderen Möglichkeiten ergeben sich in Verbindung mit der Schattenmaske.

1.7.5.4 Die Schattenmaske

Die Schattenmaske ist eine von zwei Masken, die man pro Objekt definieren kann. Mit der Schattenmaske kann man bestimmen, welche Punkte den Hintergrund überdecken und wel-

che nur die Farbe des Hintergrunds verändern. Wenn zusätzlich das Overlay-Flag gesetzt ist, entscheidet die Schattenmaske, welche Punkte des Objektes zu sehen sind.

Eine Maske muß die gleiche Breite und Höhe wie das zugehörige Objekt haben. Vom Format sind Maske und Objekt also gleich, aber die Tiefe der Maske ist, unabhängig von der Tiefe des Objekts, immer eins. Jedes Bit in der Maske entspricht einem Bildpunkt des Objekts. Ist ein Bit in der Maske gesetzt, dann überdeckt der entsprechende Punkt des Objekts den Bildschirmhintergrund. Bei ungesetztem Punkt entscheidet das Overlay-Flag, was zu tun ist. Ist es gesetzt, dann beeinflußt der jeweilge Punkt den Bildschirm in keiner Weise, egal, ob im Objekt ein Bildpunkt gesetzt ist oder nicht. Wenn Overlay nicht gesetzt ist, verändert sich die Farbe des Bildschirmpunktes.

Wenn man keine eigene Schattenmaske bestimmt hat, aber das Overlay-Flag gesetzt ist, wird automatisch eine Maske vom Computer erstellt. In dieser Maske ist jedes Bit gesetzt, bei dem auch der entsprechende Bildpunkt des Objekts gesetzt ist. Auf diese Weise sind alle ungesetzten Punkte transparent.

Wenn man beim Editor eine Schattenmaske für ein Objekt wünscht, kann man folgendermaßen vorgehen:

1. Voraussetzung ist, daß das dazugehörige Objekt schon besteht. Dieses laden wir nämlich als erstes ein. Damit haben wir schon mal das richtige Format für die Maske.
2. Wir stellen die Tiefe auf eins.
3. Wir malen die Maske. Dabei können uns vielleicht noch verbliebene Punkte des Objekts Anhaltspunkte liefern.
4. Diese Maske speichern wir im P-Format auf Diskette ab.
5. Wir laden die Grafik des zukünftigen Objekts noch einmal herein.

6. Nun setzen wir das Flag Shadowmask im Flags-Menü und geben anschließend den Namen an, unter dem wir die Maske abgespeichert haben.
7. Wir speichern das Objekt im B-Format ab oder probieren das Objekt aus.

Wenn man schon eine Maske vorbereitet hat, kann man natürlich auf die ersten fünf Punkte dieser Liste verzichten.

1.7.5.5 Auf Kollisionskurs

Die zweite versprochene Maske ist die COLLISION-Maske. Diese Maske hat nichts mit dem OBJECT.HIT-Befehl zu tun, den wir später noch erklären werden. Die Kollisionsmaske gibt an, bei welchen Punkten der Computer eine Kollision "spürt". Wäre die Kollisionsmaske ganz leer, würde der Computer nie eine Kollision registrieren. Wenn man eine Kollisionsmaske definiert, sind alle Punkte mit entsprechend gesetztem Bit in der Maske sensitiv.

Aufbau und Installierung sind mit der Shadowmask identisch. Wenn man selber keine Kollisionsmaske definiert, benutzt der Computer trotzdem eine Maske. Diese automatisch definierte Maske entsteht wieder durch eine logische Oder-Verknüpfung aller Bitplanes des Bobs (genau wie die Schattenmaske). Dadurch reagiert der Computer auf jede Berührung mit irgendeinem Bildpunkt des Objekts.

In unserem nächsten Programm haben wir neben der Kollisionsmaske noch viele weitere Möglichkeiten ausgeschöpft, Zusammenstöße zu überwachen. Alle diese Befehle probieren wir an zwei Quadraten aus, die sich über den Bildschirm bewegen.

```
REM Kollisionen

DEFINT a
RANDOMIZE TIMER

SCREEN 1,320,200,2,1
```

```
WINDOW 2,,,16,1

PRINT "Ich lese die Daten"
FOR j= 1 TO 2
FOR i=1 TO 13
  READ a
  a$(j)=a$(j)+MKI$(a)
NEXT i
  a$(j)=a$(j)+MKI$(-1)+MKI$(-1)
  FOR i=1 TO 30
    a$(j)=a$(j)+MKI$(-32768&)+MKI$(1)
  NEXT
  a$(j)=a$(j)+MKI$(-1)+MKI$(-1)
NEXT j
FOR i=1 TO 30
a$(2)=a$(2)+MKI$(0)
NEXT i
a$(2)=a$(2)+MKI$(1)+MKI$(-32768&)
a$(2)=a$(2)+MKI$(1)+MKI$(-32768&)
FOR i=1 TO 30
a$(2)=a$(2)+MKI$(0)
NEXT i

CLS
OBJECT.CLIP (70,30)-(230,190)
LINE (70,30)-(230,190),3,b

ON COLLISION GOSUB zusammenstoss
COLLISION ON

OBJECT.SHAPE 1,a$(1)
OBJECT.SHAPE 2,a$(2)
OBJECT.PRIORITY 1, 1
OBJECT.HIT 1,3,2
OBJECT.HIT 2,2,2
OBJECT.X 1,150
OBJECT.Y 1,80
OBJECT.X 2,155
OBJECT.Y 2,85
OBJECT.VX 1,8
OBJECT.VY 1,4

OBJECT.START 1,2
OBJECT.ON 1,2

WHILE INKEY$=""
LOCATE 2,15
PRINT OBJECT.X(1);TAB(21);OBJECT.Y(1)
PRINT TAB(15);OBJECT.X(2);TAB(21);OBJECT.Y(2)
WEND

WINDOW CLOSE 2
```

```
SCREEN CLOSE 1
END

zusammenstoss:
n=COLLISION(0)
m=COLLISION(n)
mehr:
IF n=1 AND m<0 THEN
  BEEP
  IF ABS(m) MOD 2=0 THEN
    OBJECT.VX 1,(m+3)*(RND*20+1)
  ELSE
    OBJECT.VY 1,(m+2)*(RND*20+1)
  END IF
END IF
OBJECT.VX 2,OBJECT.VX(1)+3*SGN(OBJECT.X(1)-1-OBJECT.X(2))
OBJECT.VY 2,OBJECT.VY(1)+3*SGN(OBJECT.Y(1)-1-OBJECT.Y(2))
OBJECT.START
n=COLLISION(0)
m=COLLISION(n)
IF m<>0 THEN mehr
RETURN

DATA 0,0,0,0,0,1,0,32,0,32
DATA 24,1,0
DATA 0,0,0,0,0,1,0,32,0,32
DATA 10,1,0
```

Während die beiden Quadrate über den Bildschirm huschen, werden ihre Koordinaten auf den Bildschirm ausgegeben. Dazu gibt es die OBJECT.X-Funktion. Neben den Koordinaten kann man auch die Geschwindigkeiten der Objekte abfragen. In Anlehnung an den entsprechenden Befehl heißt diese Funktion OBJECT.VX-Funktion. Die gleiche Funktion gibt es natürlich auch für die Y-Richtung. Diese Funktionen sind sehr wichtig, zum Beispiel, wenn es darum geht, auf Kollisionen zu reagieren.

Zu einem Zusammenstoß gehören bekanntlich immer zwei. Deshalb gibt es im Programm auch zwei Objekte. Die beiden Objekte haben beide die Form eines Quadrats. Beide sind gleich groß, haben die gleiche Farbe und bestehen nur aus dem Rahmen. Trotz dieser äußeren Ähnlichkeiten sind die Objekte verschieden. Für das erste haben wir keine Kollisionsmaske erstellt. Wie oben erwähnt, nimmt der Computer dann immer alle Bitplanes, mit OR verknüpft, als Kollisionsmaske. Bei unserem Bob ist die Kollisionsmaske der Rahmen eines Quadrats.

Dem zweiten Objekt haben wir selber eine Maske "verpaßt". In dieser Maske sind nur die vier mittleren Punkte des Objekts gesetzt. Im Programm halten wir das zweite Objekt im ersten gefangen. Immer, wenn es sich zu weit vom ersten Objekt entfernen will, tritt eine Kollision auf, und seine Flucht wird gestoppt.

Aber auch das Quadrat des ersten Objekts kann sich nicht frei bewegen. Es kann sich nur in einem abgesteckten Bereich auf dem Bildschirm bewegen. Den Bereich, in dem sich die Objekte bewegen können, kann man mit OBJECT.CLIP begrenzen.

Daß man überhaupt auf Zusammenstöße reagieren kann, verdankt man den COLLISION-Anweisungen. Ihrer gibt es drei. Sie sind die einzigen Befehle, die nicht mit dem Wort OBJECT beginnen. Mit dem ersten sagt man dem Computer, wohin er verzweigen soll (ON COLLISION GOSUB). Doch erst der zweite Befehl sorgt dafür, daß verzweigt wird. Mit COLLISION ON stellt man die Unterbrechungsfähigkeit nämlich erst an. Ohne diesen Befehl würden die Objekte einfach am Rand des ihnen erlaubten Bereichs stehenbleiben.

Die dritte COLLISION-Anweisung dient uns zum Unterscheiden, welches Objekt mit wem zusammengestoßen ist. Die Nummer des Objekts erhält man durch COLLISION(0). Den Partner bei der Karambolage ermittelt man durch COLLISION(n). n muß die Nummer des Objekts sein, das an der Karambolage beteiligt war. Statt n kann man also auch COLLISION(0) in COLLISION einsetzen. Der Wert, den man erhält, gibt den Kollisionspartner an. Ist der Wert kleiner null, trat eine Kollision mit dem Rand auf.

COLLISION(COLLISION(0))	Rand
-1	oben
-2	links
-3	unten
-4	rechts

Manchmal gelingt unserem zweiten Bob tatsächlich die Flucht. Das kann immer dann passieren, wenn seine Geschwindigkeit so

groß ist, daß er mit seiner Kollisionsmaske über die Maske des zweiten Objekts springt, ohne sie zu berühren. Wenn das Objekt diese Flucht geschafft hat, kann es sogar aus dem abgesteckten Bereich heraus, denn der Rand macht ihm nichts aus. Wie kommt das? Man kann als Programmierer selber festlegen, bei welchen Karambolagen eine Unterbrechung eintritt. Dafür hat man den OBJECT.HIT-Befehl. Und der funktioniert so: Man gibt zwei 16-Bit-Masken an. Die erste Maske ist die Me-Maske oder Selbstmaske. Die zweite Maske ist die Hit-Maske oder Stoßmaske. Wenn ein Objekt mit einem anderen kollidiert, wird jeweils eine Me- mit der Hit-Maske des anderen Objekts verglichen. Ist bei beiden an der gleichen Stelle eine Eins, wird eine Programmunterbrechung veranlaßt. Wenn in einer Stoßmaske eines Objekts eine Eins gesetzt ist und das Objekt an den Fensterrahmen oder an den Rand des mit OBJECT. CLIP eingestellten Bereichs stößt, wird ebenfalls eine Unterbrechung herbeigeführt.

Die Masken werden nicht als Bitfolge, sondern als korrespondierende Zahl zwischen -32768 und 32767 angegeben. Voreingestellter Wert der Masken ist -1, was einer vollbesetzten Maske entspricht. Das bedeutet, daß bei jedem Zusammenstoß eine Unterbrechung eintritt.

Der letzte neue Befehl in diesem Programm hat eigentlich nicht direkt etwas mit Kollisionen zu tun. Man kann mit ihm die Reihenfolge festlegen, in der Objekte auf den Bildschirm gezeichnet werden. Das ist besonders bei sich überdeckenden Objekten wichtig. Der Befehl lautet OBJECT.PRIORITY. Voreingestellt ist null. Je höher die Priorität, desto eher wird ein Objekt gezeichnet.

1.7.5.6 Animierte Bit-Ebenen

Wir haben schon mehrfach erwähnt, daß die Bildinformationen der Objekte in verschiedene Bitplanes aufgeteilt sind. Was ist mit den Bitplanes eigentlich genau gemeint? Für jeden Bildschirmpunkt gibt es im Speicher mehrere Bits, die die Farbe des

Bildpunkts festsetzen. Die Tiefe ist die Anzahl der Bits, die pro Bildpunkt zur Verfügung stehen. Die ersten Bits aller Punkte bilden die erste Bitplane. Alle zweiten Bits machen die zweite Bitplane aus usw.

Alle Bitplanes sind im Speicher hintereinander angeordnet. Das hat, besonders wenn man mit Objekten arbeitet, große Vorteile. Dadurch kann man beispielsweise sehr einfach eine weitere Bitplane anfügen oder eine löschen. Dadurch ist es aber auch erst möglich, Objekte zu definieren, die eine geringere Tiefe haben als der Bildschirm, in dem sie sich befinden. Z.B. könnte man in einem Bildschirm mit einer Tiefe von 5 ein Objekt mit Tiefe 2 laufen lassen. Natürlich verfügt das Objekt dann nur über die ersten vier Farben. Halt! Die letzte Aussage war ein Trugschluß. Man kann zwar nur über vier Farben verfügen, es müssen aber nicht die ersten vier sein.

Es gibt zwei Befehle, mit denen man mit dem gleichen Objekt aus unserem letzten Beispiel sehr viele Farbkombinationen mit allen 32 Farben erstellen kann. Überzeugen Sie sich selbst:

```
REM Demonstrationsprogramm zum
REM OBJECT.PLANES-Befehl

SCREEN 1,320,200,5,1
WINDOW 2,"Planes-Demo",,31,1

FOR i=1 TO 61
  READ g
  p$=p$+MKI$(g)
NEXT i

OBJECT.SHAPE 1,p$
OBJECT.ON

x=50
y=5

FOR i=0 TO 3
 FOR j=i+1 TO 4
 IF j<>i THEN
   FOR k=0 TO 31
     IF (k AND 2^i OR k AND 2^j)=0 THEN
       OBJECT.X 1,x
       OBJECT.Y 1,y
```

```
        OBJECT.PLANES 1,2^i+2^j,k
        x=x+20
        IF x>244 THEN
          x=50
          y=y+20
        END IF
      END IF
    NEXT
  END IF
  NEXT j
NEXT i

LOCATE 22,5
PRINT "80 verschiedene Farbkombinationen"

WHILE INKEY$=""
WEND

WINDOW CLOSE 2
SCREEN CLOSE 1


DATA 0, 0, 0, 0, 0, 2, 0, 16, 0, 16, 52
DATA 0, 0, -8180,-8082,-8081,-8081,-8177
DATA -1,-1,-1,-1,-1,-1,-1025,-1105,-681
DATA -2049,-1,-4,-2,-1,-1,-1,-1,-1,-1,-1
DATA -1,-16381,-16381,-16381,-16381,-16381
DATA -16381,-16381,-4,-2,-1,-1,-1,-1,-1
DATA -1,-1,-1,-1,-1025,-1105,-681,-2049
DATA -1
```

Wie man zählen und lesen kann, gibt es 80 Kombinationen. Warum es nicht mehr geben kann, rechnen wir Ihnen später vor. Erst einmal möchten wir klären, wie man zu so vielen Farbkombinationen gelangt. Das ermöglichen uns zwei Werte in der Objekt-Struktur. Der erste Wert gibt an, in welche Ebenen die Bitplanes geschrieben werden. Dadurch ändert sich die Reihenfolge der fünf Bits eines Bildpunktes und damit auch die Farbe, in der der Punkt erscheint. Dieser Wert heißt PlanePick und findet sich unter diesem Namen auch im Editor im Menü Flags wieder.

Es gibt zehn verschiedene Kombinationen, die zwei Ebenen des Bobs in fünf Ebenen des Screens zu verteilen.

1. 11000 0, 1, 2, 3
2. 10100 0, 1, 4, 5

3. 10010 0, 1, 8, 9
4. 10001 0, 1, 16, 17
5. 01100 0, 2, 4, 5
6. 01010 0, 2, 8, 9
7. 01001 0, 2, 16, 17
8. 00110 0, 4, 8, 9
9. 00101 0, 4, 16, 17
10. 00011 0, 8, 16, 17

Hinter den zehn Kombinationen stehen die Farbregister, aus denen sich der Computer bedient, wenn man die Bitplanes in andere Ebenen schreibt. In der Zahlenfolge aus Nullen und Einsen bedeutet eine Eins, daß in diese Ebene eine Bitplane geschrieben wird. Dabei wird die erste Ebene des Objekts immer in die erste ausgewählte Ebene des Bildschirms geschrieben. Mit PlanePick kann man natürlich nicht nur zwei Ebenen auf fünf Bitplanes verteilen. Es geht ebensogut mit weniger bzw. mehr Ebenen.

Dem Editor werden die ausgewählten Ebenen genau wie oben durch eine Folge von Einsen und Nullen mitgeteilt.

Wenn Sie sich angeguckt haben, wo Sie PlanePick in unserem Editor finden, ahnen Sie vielleicht schon die zweite Möglichkeit zur Farbänderung. Direkt unterhalb von PlanePick befindet sich im Menü der Befehl PlaneOnOff. Dieser setzt in der Objekt-Struktur einen Wert, der bestimmt, was mit noch nicht benutzten Ebenen des Bildschirms passiert. Alle Ebenen, die nicht von PlanePick genutzt werden, gelten als unbenutzt. Wie der Name schon sagt, kann man die Ebenen aus- und wieder anschalten. Den ausgeschalteten Zustand kennen Sie schon. Das ist nämlich der Normalzustand für "unbenutzte" Ebenen. Wenn man eine vom Bob unbenutzte Ebene anschaltet, bedeutet das, daß in diese Ebene die Schattenmaske geschrieben wird. Mit anderen

Worten, die Farbe eines Punktes ändert sich, wenn eine Bitplane eingeschaltet wird und in der Schattenmaske das dem Bildpunkt entsprechende Bit gesetzt ist.

Der Aufbau ist mit PlanePick identisch. Die Werte, die man im Editor festlegt, kann man im Gegensatz zu den Flags noch nachträglich verändern. Sonst wäre auch unser Farbkombinationsprogramm wesentlich komplizierter. Für beide Werte gibt es den Befehl OBJECT.PLANES. An ihn gibt man in folgender Reihenfolge die Werte Objektnummer, PlanePick und PlaneOnOff ein. Hier kann man allerdings keine binäre Zahlenfolge eingeben wie im Editor, sondern man muß diese Werte umrechnen. Dazu kann man folgende Zeile benutzen:

```
PlanePick=b1+2*b2+4*b3+8*b4+16*b5
```

Für b1 bis b5 brauchen Sie nur noch die binäre Zahlenfolge einzusetzen. Für PlaneOnOff ist die Umrechnung gleich.

Durch PlanePick und besonders PlaneOnOff eröffnen sich ungeahnte Möglichkeiten für die Animation. Aus einem Bob kann man mehrere verschiedene Figuren erzeugen. Auf diese Weise haben wir eine Art Olympiafeuer auf dem Bildschirm erzeugt. Die züngelnden Flammen und die verschiedenen Farben entstehen, wenn wir PlanePick und PlaneOnOff verändern.

```
REM Feuer

DEF FNplanes=CINT(RND)*4+CINT(RND)*8+CINT(RND)*16

FOR i= 1 TO 274
READ a
a$=a$+MKI$(a)
NEXT i

SCREEN 1,320,200,5,1
WINDOW 2,,,,1

PALETTE 0,0,0,0
PALETTE 4,1,.5,0
PALETTE 8,1,0,0
PALETTE 12,1,.26,0
PALETTE 16,1,.4,0
PALETTE 20,.8,0,0
```

```
PALETTE 24,.95,.5,0
PALETTE 28,1,.9,0

LOCATE 2,5
PRINT "Das ewige Feuer"

LINE (96,120)-(110,180),26,bf
CIRCLE (103,70),60,27,4.02,5.4
LINE (66,108)-(140,108),27
PAINT (100,110),27

OBJECT.SHAPE 1,a$
CLOSE 1
OBJECT.X 1,79
OBJECT.Y 1,79
OBJECT.ON 1

feuer:
a=FNplanes
IF a=28 THEN a=24
OBJECT.PLANES 1,a
FOR i=0 TO 10:NEXT
OBJECT.PLANES 1,,FNplanes
FOR i=0 TO 10: NEXT
GOTO feuer

Daten der Flamme
DATA 0,0,0,0,0,2,0,48,0,29
DATA 12,24,0,0,0,0,0,0,0,0
DATA 0,0,0,0
DATA 0,0,0,0,0,0,0,0,0,0
DATA 0,0,0,0,0,0,0,0,96,0,0
DATA 0,0,0,1152,16384,0,1665,-16384,32,1792
DATA -16384,16,1671,-32768,48,902,0,18,5661,-32768
DATA 26,-30194,128,12,-20869,256,7,-14609,-13824,3
DATA 15999,-15872,0,-257,-17664,0,-1554,23040,0,-7434
DATA -8704,0,-18441,5120,0,27399,18432,0,30230,16384
DATA 0,29704,16384,0,-32752,0,0,0,0,0
DATA 0,0,0,0
DATA 0,0,192,0,0,320,0,0,320,128
DATA 0,480,1920,0,416,7680,256,1888,16128,896
DATA 1664,-1024,896,16353,-9216,992,-16607,-208,976,-20702
DATA -2044,504,32727,26652,404,8013,26728,252,4991,29040
DATA 210,15420,31728,122,-21928,-3088,77,-21544,16128,46
DATA 8360,8896,50,56,960,10,-32176,8384,10,-16384
DATA -32640,6,12,-24320,5,271,8192,2,-31244,-32768
DATA 0,-32657,-26624,0,-32745,12288,0,0,8192,0
DATA 2,0,0,6
DATA 0,0,6,0,0,7,0,0,3,0
DATA 0,9,0,0,11,256,0,10,768,1
DATA 28,3328,1,26,4608,2,62,8704,2,206
DATA 8704,6,460,30208,6,460,28160,15,510,17920
```

```
DATA 31,3486,-31744,4,-27235,19968,20,21022,19456,5
DATA 13132,7168,21,-26168,12288,7,-25524,-4096,1,22590
DATA 28672,2,-10628,20480,3,-3279,0,1,29903,-20480
DATA 0,19525,12288,0,11577,0,0,1033,0,0,11777,0
```

In unserer Flamme kommen nur acht verschiedene Farben vor. Es sind die Farben aus den Registern 0, 4, 8, 12, 16, 20, 24, 28. Diese Farben haben wir geändert. Alle anderen Farben sind frei verfügbar.

In der Flamme sind mehrere Kombinationen der beiden Ebenen und der Schattenmaske möglich. Es kann vorkommmen, daß beide Ebenen von PlanePick "gepickt" werden. Es ist aber auch möglich, daß nur eine oder gar keine Bitplane gezeichnet wird. Außerdem kann jede noch nicht belegte Ebene mit PlaneOnOff angeschaltet sein.

Und das alles ist doch mit relativ wenig Aufwand zu arrangieren. Genauso lassen sich noch ganz andere Objekte aufbereiten. Zum Beispiel kann man die Explosion eines Raumschiffs, das Zwinkern eines Auges oder Lippenbewegungen und noch vieles mehr in einem Objekt verpacken.

1.7.6 Die Alternative: Sprites

Bis jetzt sind die Sprites bei unserer Beschreibung der Animation ziemlich kurz gekommen. Wir haben ihr Vorhandensein nur kurz erwähnt. Für Sprites gilt vieles, was wir schon für Bobs erwähnt haben. Alle BASIC-Befehle gelten sowohl für Bobs als auch für Sprites.

1.7.6.1 Der feine Unterschied

Wenn man gerne ein Sprite konstruieren möchte, braucht man in unserem Editor nur das Sprite-Flag zu setzen. Wenn man das gemacht hat, werden alle anderen Punkte dieses Menüs in Geisterschrift erscheinen. Mit diesen Punkten kann man nämlich nichts mehr anfangen, weil sie keine Wirkung auf Sprites haben.

Sprites werden immer mit den Eigenschaften geboren, die durch gesetzte Overlay-Flags und SaveBack-Flags bei Bobs zu erreichen sind.

Auch bei der Tiefe sind die Auswahlmöglichkeiten stark beschränkt. Sprites haben immer nur drei verschiedene Farben. Das entspricht einer Tiefe von 2.

Sprites haben ganz bestimmte Eigenschaften. Ein Sprite ist immer nur 16 Pixel breit. Seine Höhe ist nicht eingeschränkt. Sprites bewegen sich schneller über den Bildschirm als Bobs.

1.7.6.2 Farbige Sprites

Die Farben der Sprites sind im Gegensatz zu Bobs nicht von der Tiefe des Bildschirms abhängig. Sprites bringen nämlich ihre eigenen Farben mit. Diese Farben stehen in keinem Register, sondern es sind ihre eigenen Werte. Trotzdem gibt es keine 35 oder mehr verschiedenen Farben. Die Farben, die ein Sprite mitbringt, verändern auch die Farben auf dem Bildschirm. Allerdings nur Farben, die unterhalb des Sprites sind. Wenn die Punkte eines Farbregisters sowohl ober- als auch unterhalb des Sprites liegen, haben die Punkte unterschiedliche Farben.

Die Register, in die das Sprite seine Farben legt, kann man nicht selber bestimmen. Eddi nimmt für Sprites immer die Farben der Register 1 bis 3. Wollen Sie dem Sprite andere Farben mitgeben, müssen Sie die Werte dieser Register ändern.

1.8 Grafik mit GFA-BASIC

AmigaBASIC gibt es schon genauso lange wie den Amiga selbst. Seit sehr viel kürzerer Zeit gibt es jetzt auch eine GFA-BASIC-Version für diesen Computer. Ein neues Programm verkauft sich nur dann, wenn es auch Vorteile gegenüber den schon vorhandenen bietet, und GFA hat so einiges zu bieten.

1.8.1 Schneller und immer schneller...

Der deutlichste Unterschied zwischen diesen beiden BASIC-Versionen ist die Arbeitsgeschwindigkeit. GFA-BASIC arbeitet viel schneller als AmigaBASIC und eröffnet damit dem Programmierer ganz neue Möglichkeiten. Wenn Sie im Besitz von GFA-BASIC sind, dann können Sie ja einmal folgendes GFA-Programm mit dem entsprechenden AmigaBASIC-Programm aus Kapitel 1.2.3 vergleichen.

```
OPENS 1,0,0,320,256,6,&H800
OPENW 0
a=63
DIM x(1,a),y(1,a)
x(0,0)=150
y(0,0)=100
x(1,0)=170
y(1,0)=100
WHILE INKEY$=""
  FOR z=0 TO a
    COLOR 0
    LINE x(0,z),y(0,z),x(1,z),y(1,z)
    FOR i=0 TO 1
      REPEAT
        x(i,z)=ABS(x(i,alt)+RND*20-10)
      UNTIL x(i,z)<320
      REPEAT
        y(i,z)=ABS(y(i,alt)+RND*20-10)
      UNTIL y(i,z)<256
    NEXT i
    f1=f1 MOD 63+1
    COLOR f1
    LINE x(0,z),y(0,z),x(1,z),y(1,z)
    alt=z
  NEXT z
WEND
```

Dieses Programm zeichnet, genau wie das Programm in Kapitel 1.2.3, ein scheinbar wildgewordenes Linienbündel auf den Bildschirm, halt nur wesentlich schneller.

1.8.2 Die Grafikbefehle

Die beiden Programme unterscheiden sich nicht nur in ihrer Geschwindigkeit, sondern auch in Programmstruktur und Befehlsnamen.

Obwohl man auch hier einen BASIC-Interpreter vor sich hat, scheinen die beiden Programme in unterschiedlichen Sprachen geschrieben worden zu sein. Zum einen ist der Standard-BASIC-Wortschatz von GFA um einige neue Befehle aufgestockt worden. Diese Befehle sind aus anderen Sprachen entlehnt, sie wollen wir hier aber nicht besprechen. Uns interessieren nur die Unterschiede in der Grafikprogrammierung.

Es gibt leider keinen Grafik-Befehl-Standard, und so kommt es, daß die Grafik-Befehle von Amiga- und GFA-BASIC völlig verschieden sind. Mal ist nur das Format neu, mal nur der Name geändert, und so manches Mal ist nichts mehr ähnlich geblieben. Wie sich die Befehle geändert haben, kann man sehr gut der folgenden Tabelle entnehmen. Besonders nützlich dürfte diese Tabelle den Umsteigern sein. Sie können genau sehen, wie sie in GFA-BASIC den selben Effekt erzielen wie mit dem alten BASIC. Umgekehrt gilt dies natürlich genauso.

Die Reihenfolge, mit der die Befehle durchgegangen werden, ist den vorherigen Kapiteln angepaßt.

Aufgabe	AmigaBASIC	GFA-BASIC
Punkte		
setzen	PSET (x,y)	PLOT x,y oder DRAW x,y
löschen	PRESET (x,y)	keine Entsprechung
farbig	PSET (x,y),f	COLOR f PLOT x,y
abfragen	PRINT POINT(x,y)	PRINT POINT(x,y)

Aufgabe	AmigaBASIC	GFA-BASIC
relativ	PSET STEP (x,y)	
Linien		
einfach	LINE (x1,y1)-(x2,y2)	LINE x1,y1,x2.y2 oder DRAW x1,y1 TO x2,y2
farbig	LINE (x1,y1)-(x2,y2),f	COLOR f LINE x1,y1,x2,y2
relativ1	LINE -(x,y)	DRAW TO x,y
relativ2	LINE STEP(x1,y1)-(x2,y2)	
Rechtecke		
gerahmt	LINE (x1,y1)-(x2,y2),,B	BOX x1,y1,x2,y2
gefüllt	LINE (x1,y1)-(x2,y2),,BF	PBOX x1,y1,x2,y2
Kreise		
einfach	CIRCLE (x,y),r	CIRCLE x,y,r
farbig	CIRCLE (x,y),r,f	COLOR f CIRCLE x,y,r
Ellipsen	CIRCLE (x,y),r,,,,v	ELLIPSE x,y,rx,ry
Teilbogen	CIRCLE (x,y),r,,wa,we	
Flächen füllen		
allgemein	PAINT (x,y)	FILL x,y

(Diese Befehle entsprechen sich nicht ganz! Beide Füllfunktionen benutzen unterschiedliche Abgrenzungen.)

farbig	PAINT (x,y),f	COLOR f FILL x,y,f

Aufgabe	AmigaBASIC	GFA-BASIC
Begrenzung	PAINT (x,y),f1,f2	COLOR f1 FILL x,y,f2
Rechtecke	LINE (x1,y1)-(x2,y2),,BF	PBOX x1,y1,x2,y2
Kreise	CIRCLE (x,y),r PAINT (x,y)	PCIRCLE x,y,r
Ellipsen	CIRCLE (x,y),r,,,,v PAINT (x,y)	PELLIPSE x,y,r
Polygone	AREA (x1,y1) AREAFILL	POLYLINE n,x(),y() POLYFILL

(Bei GFA-BASIC werden alle n-Eckpunkte in einem Befehl durch Angabe von Feldvariablen übergeben. Bei AmigaBASIC wird jeder Eckpunkt einzeln mit einem AREA-Kommando eingegeben.)

Muster		
Linie	PATTERN w%	DEFLINE w%
Fläche	PATTERN	DEFFILL

(Hier haben wir die Parameter weggelassen, weil die Unterschiede zwischen beiden Befehlen sehr groß sind.)

Farbe		
setzen	COLOR f1,f2	COLOR f1,f2
ändern	PALETTE f,r,g,b	SETCOLOR f,r,g,b

(Bei PALETTE liegen die RGB-Werte zwischen 0 und 1, bei SETCOLOR zwischen 0 und 15.)

GET/PUT		
GET	GET (x1,y1)-(x2,y2),feld	GET x1,y1,x2,y2,string
PUT	PUT (x1,y1),feld	PUT x1,y1,string

(Bei AmigaBASIC werden die Daten der Grafik in einer Feldvariablen, bei GFA-BASIC in einer Zeichenkette abgelegt.)

Animation

(Fast alle Animationsbefehle sind vollkommen identisch in beiden Dialekten und deshalb nicht noch einmal aufgelistet.)

Zur Bedeutung der Variablen:

x,y,x1,y1,x2,y2	Bildschirmkoordinaten
x(),y()	Felder mit Koordinaten
f,f1,f2	Farben
r	Radius
rx,ry	Entfernung der Kreisbahn vom Mittelpunkt in x-,y-Richtung
v	Bildverhältnis: v=rx/ry
wa,we	Anfangs-/Endwinkel
n	Anzahl der Ecken
feld	Ganzzahlfeld
string	Zeichenkettenvariable

Zu der Tabelle sei noch folgendes gesagt. Nicht alle Befehle haben eine Entsprechung in der anderen Sprache. Auch wenn sich zwei Befehle nur durch Nuancen unterscheiden, zeigt das manchmal schon eine sehr große Wirkung. Auch kommen in diesem Vergleich die jeweiligen stärken eines Dialekts nicht zur Geltung und von denen hat gerade GFA-BASIC bei genauerem Hinsehen ein ganze Menge. Für viele Dinge, die in AmigaBASIC nur über Libraries zu erreichen sind (wie das gemacht wird, zeigen wir in den folgenden Kapiteln), gibt es eingebaute Befehle: Benutzung von Hardware-Sprites, mehrfarbige Muster, Arbeiten mit dem Rastport, mehr als 32 Farben benutzen, einen Hardcopy-Befehl und noch vieles mehr.

Wie oben schon erwähnt, so sind die Animationsbefehle beider BASIC-Interpreter größtenteils identisch. Darum können Bobs und Sprites, die mit dem Programm Grafik-Edit_III konstruiert wurden, ohne Probleme und auf gleiche Weise auch mit GFA-BASIC benutzt werden.

1.8.3 Die Mandelbrot-Menge

Im folgenden Programm wollen wir Ihnen zeigen, wie GFA-BASIC in der Praxis aussieht. Das Programm zeichnet Bilder aus der Mandelbrot-Menge (wegen ihrer Form auch Apfelmännchen genannt) auf den Bildschirm. Auch wenn Sie vielleicht mit diesem Begriff nichts anfangen können, so haben Sie bestimmt schon solche Grafiken gesehen.

In diesem Programm sind mehrere verschiedene Grafik-Komponenten, die wir auch schon in AmigaBASIC kennengelernt haben, eingebaut. Daneben haben wir aber auch einige Grafikbefehle verwendet, die AmigaBASIC nicht kennt. Wir hoffen, daß Sie mit diesem Programm ein kleinen Einblick in die neue Sprache bekommen.

Alle, die sich nicht für das Wie des Programms interessieren, können den folgenden Absatz überspringen. Denn man kann auch mit dem Programm arbeiten, ohne verstanden zu haben, wie es funktioniert. Um dieses Programm zu verstehen, muß man mit komplexen Zahlen rechnen können.

Die Mandelbrot-Menge entsteht durch folgendes Rekursionsverfahren:

X=X hoch 2 + C

Hierbei sind X und C komplexe Zahlen. C bleibt konstant, während X seinen Wert immer verändert. Den obigen Rechenschritt wiederholt das Programm immer wieder. Inzwischen prüft, ob der Betrag von X einen bestimmten Wert überschreitet. Die Anzahl der Durchläufe bis zum Überschreiten des Grenzwertes wird in einen Farbwert umgerechnet, den der Punkt auf dem Bildschirm bekommt, der der Zahl C entspricht. Überschreitet der Betrag von X nach einer bestimmten, von Ihnen angegebenen Anzahl von Durchläufen den Grenzwert nicht, wird abgebrochen, und der Punkt wird auf schwarz gesetzt. Danach wird X gelöscht, und man fängt mit einem neuen C-Wert wieder von vorne an.

```
REM Mandelbrot-Menge
REM 1/89
REM
GOSUB schirm
breite=100
hoehe=100
ar=-2.25
ai=-1.5
br=0.75
bi=1.5
iter=50
REM Hauptprogramm
WHILE 1
  GOSUB mandel
  GET x1,y1,x1+breite,y1+hoehe,b$
  REPEAT
    MOUSE mx,my,mk
    a$=INKEY$
  UNTIL mk<>0 OR a$<>""
  WHILE INKEY$<>""
  WEND
  IF a$="q"
    CLOSEW 0
    CLOSES 1
    END
  ELSE IF a$="s"
    CLOSEW 0
    CLOSES 1
    GOSUB speichern
    GOSUB schirm
  ELSE IF a$="l"
    CLOSEW 0
    CLOSES 1
    GOSUB laden
    GOSUB schirm
  ELSE IF a$="e"
    CLS
    PRINT "Realeteil "
    INPUT "von :",ar
    INPUT "bis :",br
    PRINT "Imaginaerteil"
    INPUT "von :",ai
    INPUT "bis :",bi
    b$=""
  ELSE IF a$="<" AND iter>10
    iter=iter-10
    b$=""
  ELSE IF a$=">"
    iter=iter+10
    b$=""
  ELSE IF a$="d"
    HARDCOPY
```

```
ELSE IF a$=""
  GRAPHMODE 2
  REPEAT
    MOUSE bx,by,mk
    BOX mx,my,bx,by
    BOX mx,my,bx,by
  UNTIL mk=0
  PLOT mx,my
  IF mx<>bx AND my<>by THEN
    IF mx>bx
      SWAP mx,bx
    ENDIF
    IF my>by
      SWAP my,by
    ENDIF
    COLOR 1,0
    GRAPHMODE 1
    BOX 10,210,80,226
    TEXT 14,220,"Fenster"
    BOX 100,210,190,226
    TEXT 104,220,"Ausschnitt"
    BOX 200,210,270,226
    TEXT 204,220,"beides"
    REPEAT
      MOUSE px,py,mk
    UNTIL mk=1
    IF px>10 AND px<80 AND py>210 AND py<226
      breite=ABS(mx-bx)
      hoehe=ABS(my-by)
      bi=ai+(br-ar)*hoehe/breite
      b$=""
    ELSE IF px>100 AND px<190 AND py>210 AND py<226
      px=br-ar
      py=bi-ai
      ar=ar+(mx-x1)*px/breite
      br=br+(bx-x1-breite)*px/breite
      ai=ai+(my-y1)*py/hoehe
      bi=bi+(by-y1-hoehe)*py/hoehe
      IF ABS(br-ar)/breite>ABS(bi-ai)/hoehe
        hoehe=ABS((bi-ai)*breite/(br-ar))
      ELSE
        breite=ABS((br-ar)*hoehe/(bi-ai))
      ENDIF
      b$=""
    ELSE IF px>200 AND px<270 AND py>210 AND py<226
      px=br-ar
      py=bi-ai
      ar=ar+(mx-x1)*px/breite
      br=br+(bx-x1-breite)*px/breite
      ai=ai+(my-y1)*py/hoehe
      bi=bi+(by-y1-hoehe)*py/hoehe
      px=breite
```

```
        py=hoehe
        breite=ABS(mx-bx)
        hoehe=ABS(my-by)
        IF x>x1+px
          x=x1+px
        ENDIF
        x=x-mx+(300-breite)/2
        IF mx<x1 OR my<y1 OR by>y1+py
          b$=""
        ELSE
          GET mx,my,bx,by,b$
        ENDIF
      ENDIF
    ENDIF
  ENDIF
WEND
PROCEDURE mandel
  COLOR 1,0
  CLS
  PRINT AT(1,1);"Realteil    :";
  PRINT USING "-#.######",ar;
  PRINT " bis ";
  PRINT USING "-#.######",br
  PRINT AT(1,2);"Imaginarteil:";
  PRINT USING "-#.######",ai;
  PRINT " bis ";
  PRINT USING "-#.######",bi
  PRINT AT(1,3);"Iteration   :";iter
  x1=(300-breite)/2
  y1=(250-hoehe)/2
  GRAPHMODE 1
  BOX x1-1,y1-1,x1+breite+1,y1+hoehe+1
  PUT x1,y1,b$
  IF x<x1 OR b$=""
    x=x1
  ENDIF
  WHILE x<=x1+breite
    EXIT IF MOUSEK=1 OR INKEY$<>""
    cr=ar+(br-ar)*(x-x1)/breite
    FOR y=y1 TO y1+hoehe
      ci=ai+(bi-ai)*(y-y1)/hoehe
      r=cr
      i=ci
      durchlauf=0
      WHILE (durchlauf<iter) AND (r*r+i*i<4)
        durchlauf=durchlauf+1
        h=r
        r=r*r-i*i+cr
        i=2*h*i+ci
      WEND
      COLOR INT(29*durchlauf/iter)+2
      PLOT x,y
```

```
    NEXT y
    INC x
  WEND
RETURN
PROCEDURE schirm
  RESTORE
  OPENS 1,0,0,320,256,5,0
  OPENW 0
  FOR i=0 TO 31
    READ a
    SETCOLOR i,a
  NEXT i
  DATA 7,4095,522,1036,1551,2319,2831
  DATA 3343,3855,3853,3851,3850,3848,3846
  DATA 3842,3840,3628,3643,3658,3689
  DATA 3703,3974,4005,4019,4050,4065
  DATA 4067,4040,4027,4046,4079,0
RETURN
PROCEDURE speichern
  FILESELECT "Bild speichern","speichern","df0:",name$
  OPEN "O",#1,name$
  WRITE#1,breite,hoehe,ar,br,ai,bi,x,LEN(b$)
  PRINT #1,b$
  CLOSE #1
RETURN
PROCEDURE laden
  FILESELECT "Bild laden","laden","df0:",name$
  OPEN "I",#1,name$
  INPUT #1,a1$,a2$,a3$,a4$,a5$,a6$,a7$,a8$
  b$=INPUT$(VAL(a8$),#1)
  breite=VAL(a1$)
  hoehe=VAL(a2$)
  ar=VAL(a3$)
  br=VAL(a4$)
  ai=VAL(a5$)
  bi=VAL(a6$)
  x=VAL(a7$)
  CLOSE #1
RETURN
```

Sehr viele Funktionen dieses Programms lassen sich durch die Tastatur steuern.

Eine sehr wichtige Funktion ist das Verändern der Anzahl der Iterationen. Mit Iterationen ist die maximale Anzahl der Durchläufe gemeint. Ist der Wert von Iterationen klein, wird das Bild schneller berechnet, dafür gehen aber leider auch alle Feinheiten verloren. Je tiefer man also in die Mandelbrot-Menge "einsteigt", desto größer muß der Wert Iteration gewählt sein. Das hat wie-

derum zur Folge, daß der Rechner mehrere Stunden rechnen muß. Den Wert von Iterationen verändern Sie durch die Tasten "größer als" zum Erhöhen und "kleiner als" zum Verkleinern.

Da das Programm ziemlich lange an einem Bild rechnet, ist es sehr sinnvoll, fertige Bilder abzuspeichern. Diesen Punkt erreichen Sie über die Taste "s". Dagegen kann man mit "l" ein gespeichertes Bild wieder einladen. Es können übrigens nicht nur fertige Bilder abgespeichert werden. Wenn Sie ein angefangenes Bild Abspeichern, wird die Berechnung nach dem Einladen automatisch dort fortgesetzt, wo sie zum abspeichern unterbrochen wurde.

Durch die Taste "e" kann man den Bereich, der auf dem Bildschirm sichtbar ist, manuell eingeben. Dabei sind nur die Randgebiete der Mandelbrot-Menge interessant. Die Mandelbrot-Menge liegt zwischen folgenden Koordinaten:

- Realteil von -2.25 bis 0.75
- Imaginärteil von -1.5 bis 1.5

Eine sehr viel bequemere Art der Gebietsauswahl erfolgt über die Maus. Hierbei kann man ein Gebiet in Abhängigkeit des aktuellen, auf dem Bildschirm sichtbaren Bereichs auswählen. Drücken Sie die linke Maustaste an einem der gewünschten Eckpunkte des Bereichs, und ziehen Sie den Mauszeiger dann bei gedrückt gehaltenem Knopf zum anderen Eckpunkt. Wenn Sie den Mausknopf loslassen, wird der neue Bereich berechnet und auf den Bildschirm gezeichnet. Auf gleiche Weise kann man auch die Größe des Fensters, in dem die Grafik ausgegeben wird, einstellen.

Außerdem kann man die Grafiken durch Druck auf "d" auf dem Drucker ausgeben lassen.

2. Einstieg in das Amiga-Betriebssystem

Bisher kamen alle unsere Programme mit den herkömmlichen BASIC-Befehlen aus. An dieser Stelle haben wir jedoch die Leistungsgrenze des AmigaBASIC-Interpreters erreicht. Viele interessante Projekte, die sich mit AmigaBASIC-Befehlen allein nicht verwirklichen lassen, wollen wir nun in Angriff nehmen: Eine Grafik-Hardcopyroutine zur Ausgabe von beliebigen Grafiken auf einen Drucker, neue, auch selbstdefinierte Zeichensätze, 1024x1024 Punkte Super-Bitmap, um nur einiges zu nennen.

Gleichzeitig finden Sie zu jedem AmigaBASIC-Programm ein entsprechendes Komplement im neuen GFA-BASIC. Dieser Interpreter besitzt eine weit größere Befehlsvielfalt und ermöglicht so Dinge, die in AmigaBASIC nur durch geschickte Manipulationen erreicht werden können. Aber auch GFA-BASIC kann (und muß des öfteren) auf Komponenten des Betriebssystems zurückgreifen, so daß pure GFA-Anwender ebenfalls auf ihre Kosten kommen.

2.1 Die Befehls-Bibliotheken des Betriebssystems

Bei den meisten anderen Computern wäre die Zeit gekommen, umständliche Maschinensprache-Routinen zu erstellen, um die oben angeschnittenen Projekte zu realisieren. Nicht jedoch beim Amiga. Die Lösungen zu unseren Problemen existieren bereits, und zwar liegen sie im Betriebssystem des Amiga. Dieses besitzt eine Reihe von Bibliotheken (engl. Libraries), in denen, sorgsam nach Themenbereichen geordnet, hunderte kleiner Maschinensprache-Routinen abgespeichert sind. Für (fast) jedes Programmierproblem kann man hier die Lösung finden. Es ist also gar nicht nötig, selbst langwierig neue Routinen zu entwickeln; lediglich ein Weg muß gefunden werden, an die Systembibliothe-

ken heranzukommen. Dieser Weg existiert sowohl für AmigaBASIC als auch für GFA-BASIC, wie die folgenden Seiten beweisen.

2.2 Zugriff auf die Bibliotheken aus AmigaBASIC

Der AmigaBASIC-Interpreter stellt die beiden Befehle LIBRARY und DECLARE FUNCTION zur Verfügung, um mit den Systembibliotheken Kontakt aufzunehmen. Wir werden sofort auf sie eingehen. Unbedingte Voraussetzung für die Nutzung der Systembibliotheken ist jedoch eine Datei, die mit dem Suffix .bmap ausgestattet ist. Jede Bibliothek hat ihre eigene bmap-Datei, und so sind die folgenden Dateien ganz besonders wichtig für die weitere Arbeit:

graphics.bmap
exec.bmap
layers.bmap
intuition.bmap
diskfont.bmap
dos.bmap

Diese Dateien sind gefüllt mit einer Reihe wichtiger Daten für jede einzelne Routine innerhalb der jeweiligen Bibliothek. Schließlich wollen Sie sich nicht mit Anfangsadressen, Sprungoffsets und Parameter-Registern herumschlagen, und so sind diese Informationen bereits automatisch in den bmap.-Dateien abgelegt.

Bevor Sie nun weiterlesen, sollten Sie die sechs oben genannten Dateien generieren. Hierzu benötigen Sie die im Lieferumfang Ihres Amiga enthaltene Diskette namens "Extras". Besitzen Sie Ihren Amiga jedoch schon längere Zeit, so ist es ratsam, bei Freunden oder Bekannten eine neuere Version dieser Extras-Diskette zu beschaffen, da die Beispielprogramme in diesem Buch auf dem Stand der Version 1.3 erstellt wurden. Ältere Versionen (Version 1.1 und 1.2) unterstützen noch nicht alle Befehle, die wir verwendet haben. Ferner sollten auch Kickstart

(sofern Sie einen Amiga 1000 besitzen) und Workbench mindestens der Version 1.2 entsprechen. Im Zweifelsfall wird Ihnen Ihr Händler sicher weiterhelfen können.Auf der oben genannten Extras-Diskette befindet sich ein Verzeichnis namens BasicDemos. In ihm befindet sich das BASIC-Programm ConvertFD. Laden Sie dieses. Nun geben Sie bitte ein:

```
CHDIR "Extras:FD1.3"  (für Extras-Disk Version 1.3)
```

Jetzt können Sie die Dateien generieren, indem Sie das Programm starten und auf die erste Frage eingeben:

```
graphics_lib.fd
```

Auf die zweite Frage antworten Sie:

```
LIBS:graphics.bmap
```

Hierdurch wird das bmap.-File direkt in das Library-Verzeichnis der Bootdiskette (meist die Workbench-Disk) kopiert. Sie können die Dateien aber auch direkt in Ihr BASIC-Verzeichnis kopieren lassen:

```
Extras:BasicDemos/graphics.bmap
```

Nun wiederholen Sie diese Prozedur für die anderen fünf Dateien (siehe oben).

Nachdem die Voraussetzungen nun erfüllt sind, kommen wir zur Programmierung. Wie eingangs erwähnt, stehen dazu die beiden Befehle

```
LIBRARY
DECLARE FUNCTION (...) LIBRARY
```

zur Verfügung. Bevor Sie eine (oder mehrere) der Systembibliotheken benutzen können, müssen Sie diese öffnen. Dies geschieht mit dem LIBRARY-Befehl. Für die Grafik-Bibliothek sieht dieser Aufruf wie folgt aus:

```
LIBRARY "graphics.library"
```

Sehen wir uns einmal an, was das AmigaBASIC tut, sobald es auf diesen Befehl stößt: Zunächst sucht es nach der Datei graphics.bmap. Ist diese Datei nicht im aktuellen Diskettenverzeichnis auffindbar, so wird es im allgemein deklarierten LIBS:-Verzeichnis gesucht, das sich gewöhnlich auf der Workbench-Diskette befindet. Wenn Sie also Ihre bmap-Dateien dort speichern, so lagern sie am sichersten und werden immer gefunden, auch wenn Sie einmal Programme von anderen Disketten verwenden. Sollte die Datei graphics.bmap aber in keinem der beiden Diskettenverzeichnisse gefunden werden, so kommt es unweigerlich zu einem "File Not Found"-Error. Sie sehen also, wie wichtig es ist, die entsprechenden bmap-Dateien zu generieren und entweder zusammen mit Ihren BASIC-Programmen oder aber im LIBS:-Verzeichnis abzulegen.

Verlief alles ordnungsgemäß und wurde die Datei gefunden, dann wird die entsprechende Systembibliothek - in unserem Beispiel die Grafik-Bibliothek - von BASIC geöffnet. Von diesem Zeitpunkt an vergleicht der Interpreter bei jedem Programmstart Funktionsaufrufe im Programm mit den in der bmap-Datei abgelegten Funktionsnamen und entnimmt dieser im Bedarfsfall die nötigen Daten wie Library-Offset und Registerzuordnungen. Anschließend ruft der Interpreter die Maschinenroutine auf und läßt sie ausführen. Wir werden das nun an einem kleinen Beispiel demonstrieren. Die Funktion der Grafik-Bibliothek, die wir jetzt probeweise aufrufen wollen, nennt sich "Text". Sie gibt einen Text beliebiger Länge auf den Bildschirm aus. Dabei müssen drei Parameter mitgeliefert werden: die Adresse des Rastports (auf den wir gleich noch zu sprechen kommen), die Speicheradresse, ab der der auszugebende Text gespeichert ist, sowie die Länge des auszugebenden Textes in Zeichen. Hier der komplette Aufruf:

```
LIBRARY "graphics.library"
a$        = "Hello World!"
laenge%   = LEN (a$)
rastport& = WINDOW (8)

CALL Text (rastport&, SADD (a$), laenge%)
LIBRARY CLOSE
```

Tippen Sie das Programm nun sorgfältig ab. Vermeiden Sie unbedingt Tippfehler, denn ganz im Gegensatz zu den herkömmlichen BASIC-Befehlen werden Fehlaufrufe hier nicht erkannt und durch eine Error-Meldung angezeigt, sondern direkt zum Prozessor geschickt, der sich dann eventuell mit einer der gefürchteten Guru-Meditationen revanchiert. (Alles in allem möchten wir aber klipp und klar sagen, daß Ihnen schlimmstenfalls sämtliche im Speicher befindlichen Daten verlorengehen - nach einem Reset ist Ihr Amiga mit Sicherheit wieder der alte -, bleibende Schäden durch ungebändigte Experimentierlust sind also nicht zu befürchten.)

Sobald Sie das Programm starten, wird die Datei graphics.bmap gesucht. Anschließend erscheint in der linken oberen Ecke des Bildschirms der Schriftzug "Hello World!". Der Aufruf "LIBRARY CLOSE" schließt die Bibliothek wieder, ist aber im Grunde nur ein Schönheitsbefehl, der auch weggelassen werden kann, denn BASIC schließt alle geöffneten Bibliotheken auch ohne Aufforderung bei jedem RUN oder NEW.

Doch kommen wir wieder zurück zu obigem kleinen Programm. Zwei Dinge bedürfen der weiteren Erklärung: das Statement "SADD (a$)" und die Variable "rastport&". Der Befehl SADD liefert die Adresse der Speicherstelle, ab der der Text im Speicher liegt. Von dort liest ihn die Routine "Text()". Als Rastport bezeichnet man die Kontaktstelle zu einer Zeichenfläche: einem Fenster oder einem Screen. Im weiteren Verlauf dieses Buches werden wir den Begriff "Rastport" noch wesentlich präzisieren, doch genügt es im Augenblick, wenn Sie sich die Adresse des Rastports als Angabe vorstellen, die der Text-Routine mitteilt, in welches Fenster sie schreiben soll. Für Ihr BASIC-Ausgabefenster findet sich dieser Rastport (bzw. seine Anfangsadresse) immer in der Variablen WINDOW(8):

```
rastport& = WINDOW(8)
PRINT rastport&
```

Obiges Beispielprogramm hatte lediglich die Aufgabe, die Benutzung einer Bibliothek zu demonstrieren - nicht mehr. Den-

selben Effekt hätte man schließlich ebensogut - wenn auch um vieles einfacher - durch ein einfaches PRINT erreichen können. Schauen Sie sich nun einmal das nachfolgende Programm an, das sich alle bisher gesammelten Erkenntnisse zunutze gemacht hat: eine SUB-Routine namens "P". Im wesentlichen nichts weiter als ein Ersatz für PRINT, aber bei genauem Hinsehen doch mehr...

```
Aufruf:     P "text"
entspricht: PRINT "text"

Aufruf:     P "text@"
entspricht: PRINT "text";
```

2.3 Zugriff auf die Bibliotheken aus GFA-BASIC

Der Benutzer des GFA-BASIC hat hier einen ganz entscheidenden Vorteil: Alles ist wesentlich unkomplizierter als bei Amiga-BASIC, denn der GFA-Interpreter hat alle wichtigen Bibliotheken bereits von selbst geöffnet, so daß die entsprechenden LIBRARY-Befehle unnötig werden. Auch werden keine bmap-Dateien gebraucht; ihre etwas aufwendige Generierung kann sich der GFA-Programmierer also ersparen. Schließlich kann auch das Deklarieren der Funktionen entfallen, da GFA-BASIC dies automatisch vornimmt. So könnte der Aufruf der vorangegangenen Text()-Routine folgendermaßen aussehen:

```
OPENW 1
PLOT 100,100
a$="Hello World!"
laenge=LEN(a$)
rastport%=LPEEK(WINDOW(1)+50)
VOID Text(rastport%,LPEEK(*a$),laenge)
~Text(rastport%,LPEEK(*a$),laenge)
END
```

Der PLOT-Befehl ist hierbei lediglich eingefügt worden, um den Text im sichtbaren Bereich des GFA-Fensters zu positionieren, da Text immer an die Stelle schreibt, an der der aktuelle Grafikcursor gerade steht.

Wie Sie sehen, benötigt GFA keinerlei Überbau, um Systembefehle in seinen Programmen verwalten zu können. Der Befehl VOID (optional auch ~) genügt, um eine Routine (die also keinen Wert zurückliefern soll) aufzurufen.

Der Befehlsumfang des GFA-Interpreters ist allerdings so umfassend, daß hier die Verwendung des Betriebssystems-Text() überflüssig ist, weil erstens das normale GFA-PRINT schnell genug ist und zweitens der GFA-Befehl Text existiert, der dem eben behandelten Text() stark entspricht:

```
OPENW 0
WHILE INKEY$=""
  TEXT RAND(640),RAND(256),"Hello World!"
WEND
END
```

Hier werden wahllos Texte an die verschiedensten Bildschirmpositionen gedruckt, bis eine beliebige Taste gedrückt wird.

3. Intuition - Das Benutzer-Interface

Wir beginnen unsere Reise durch die Grafikwelt des Amiga mit der Systemkomponente "Intuition". Hinter diesem Namen verbirgt sich, wie könnte es anders sein, eine Bibliothek des Betriebssystems (siehe Kapitel 2), die ganz analog zu der bereits probehalber verwendeten Grafik-Bibliothek aufgebaut ist. Intuition ist zuständig für Fenster, Screens, Requester, Alerts (z.B. die Guru Meditationen) sowie einiges mehr, was für AmigaBASIC aber nicht interessant genug ist.

3.1 Intuition-Fenster

Sofern Sie nicht mit einem eigenen Betriebssystem arbeiten, werden alle Fenster des Amiga von Intuition verwaltet. So auch die beiden Standardfenster des AmigaBASIC-Interpreters: "LIST" und "BASIC" ebenso wie das Fenster des GFA-Interpreters "GFA-BASIC". Eigens für Intuition-Fenster gibt es eine Datenkomponente. Sie umfaßt 124 Bytes und enthält die wichtigsten Daten eines jeweiligen Fensters. Die Anfangsadresse dieser Datenstruktur für das aktuelle BASIC-Ausgabefenster ist für AmigaBASIC immer in der Variablen WINDOW(7) gespeichert und für GFA-BASIC in der Variablen WINDOW(x) zu finden, wobei x die Fensternummer ist. Die dort abgelegte Adresse zeigt auf einen Datenblock, der folgendermaßen aufgebaut ist:

```
fenster&=WINDOW(8)
```

Datenstruktur "Window"/Intuition/124 Bytes

Offset	Typ	Bezeichnung
+ 000	Long	--> nächstes Fenster
+ 004	Word	X-Koordinate der linken oberen Ecke
+ 006	Word	Y-Koordinate der oberen Kante
+ 008	Word	Breite des Fensters
+ 010	Word	Höhe des Fensters

Offset	Typ	Bezeichnung
+ 012	Word	Y-Koordinate der Maus, rel. zum Fenster
+ 014	Word	X-Koordinate der Maus, rel. zum Fenster
+ 016	Word	minimale Breite des Fensters
+ 018	Word	minimale Höhe des Fensters
+ 020	Word	maximale Breite des Fensters
+ 022	Word	maximale Höhe des Fensters
+ 024	Long	Fenster-Modi
		Bit 0: 1=Vergrößerungsgadget vorhanden
		Bit 1: 1=Verschiebe-Gadget vorhanden
		Bit 2: 1=Vorder/Hintergrund-Gadgets vorh.
		Bit 3: 1=Schließgadget vorhanden
		Bit 4: 1=Vergrößerungsgadget ist rechts
		Bit 5: 1=Vergrößerungsgadget ist unten
		Bit 6: 1=Simple Refresh
		Bit 7: 1=Superbitmap
		Bit 8: 1=Backdrop-Fenster
		Bit 9: 1=Report Maus
		Bit 10: 1=GimmeZeroZero
		Bit 11: 1=Borderless
		Bit 12: 1=Activate
		Bit 13: 1=Dieses Fenster ist aktiv
		Bit 14: 1=Dieses Fenster ist in Request-Mode
		Bit 15: 1=Aktives Fenster mit aktivem Menü
+ 028	Long	--> Menü-Header
+ 032	Long	--> Titeltext für dieses Fenster
+ 036	Long	--> erster aktiver Requester
+ 040	Long	--> Double-Click-Requester
+ 044	Word	Anzahl der das Fenster block. Request
+ 046	Long	--> Screen, in dem Fenster ist
+ 050	Long	--> Rastport des Fensters
+ 054	Byte	Linker Rahmen
+ 055	Byte	Oberer Rahmen
+ 056	Byte	Rechter Rahmen
+ 057	Byte	Unterer Rahmen
+ 058	Long	--> Rahmenrastport
+ 062	Long	--> erstes Gadget
+ 066	Long	--> Eltern-Fenster
+ 070	Long	--> Kind-Fenster
+ 074	Long	--> Sprite-Data für Pointer
+ 078	Byte	Höhe des Sprite-Pointers
+ 079	Byte	Breite des Sprite-Pointers
+ 080	Byte	X-Offset des Pointers
+ 081	Byte	Y-Offset des Pointers

Offset	Typ	Bezeichnung
+ 082	Long	IDCMP-Flags
+ 086	Long	--> User Message Port
+ 090	Long	--> Fenster Message Port
+ 094	Long	IntuiMessage Message Key
+ 098	Byte	Detail-Pen
+ 099	Byte	Block-Pen
+ 100	Long	--> Menü-Haken
+ 104	Long	--> Screen-Titeltext
+ 108	Word	GZZ-MausX
+ 110	Word	GZZ-MausY
+ 112	Word	GZZ-Breite
+ 114	Word	GZZ-Höhe
+ 116	Long	--> externe Daten
+ 120	Long	--> User Daten

Jedes Fenster besitzt einen solchen Datenblock, gefüllt mit den entsprechenden Parametern. Um mit diesem Datenblock arbeiten zu können, gehen Sie bei AmigaBASIC folgendermaßen vor: Zunächst bestimmen Sie das gewünschte Ausgabefenster mit Hilfe des Befehls WINDOW OUTPUT. Anschließend finden Sie die Anfangsadresse des Datenblockes für dieses Fenster in der Variablen WINDOW(7). Bei GFA-BASIC entnehmen Sie die Anfangsadresse der Variable WINDOW(x), wobei wie oben gesagt x die Nummer des Fensters ist, dessen Struktur Sie benötigen. Nun addieren Sie zu dieser Adresse den jeweiligen Offset-Wert des gesuchten Datenfeldes. Wie Sie sehen, gibt es drei verschiedene Feldarten: Byte, Word und Long. Ein Byte-Feld umfaßt genau ein Byte. Sie fragen es mittels PEEK ab, verändern es durch POKE. Ein Word-Feld ist zwei Bytes groß. Es wird mittels PEEKW ausgelesen (bei GFA: DPEEK) und durch POKEW (bei GFA: DPOKE) manipuliert. Ein Long-Feld schließlich besteht aus vier Bytes und wird analog durch PEEKL (bei GFA: LPEEK) ausgelesen und mit Hilfe von POKEL (bei GFA: LPOKE) manipuliert. Wir werden das gleich an mehreren Beispielen verdeutlichen.

3.2 Die Fenster-Datenstruktur im Detail

Sie wissen nun, wie Sie an die entsprechende Datenstruktur für Ihr aktuelles Ausgabefenster herankommen. Jetzt werden wir diese Datenstruktur Eintrag für Eintrag näher unter die Lupe nehmen. Sehen wir uns an, was mit ihrer Hilfe alles bewerkstelligt werden kann:

Offset 0: Zeiger zu nächstem Fenster

Sobald Intuition ein neues Fenster eröffnet (und damit verbunden auch eine neue Window-Datenstruktur schafft), hinterlegt es automatisch die Anfangsadresse dieser neuen Struktur in dem Datenfeld Nr. 1 der Window-Datenstruktur des zuletzt geöffneten Fensters. In diesem Datenfeld befindet sich also der Zeiger auf das nächstjüngere Fenster. Der Aufruf,

```
fenster& = PEEKL (WINDOW (7) + 0)
```

GFA:

```
OPENW 0
fenster% = LPEEK (WINDOW (0) + 0)
```

liefert in der Variablen fenster& die Anfangsadresse der Window-Datenstruktur des nächsten Fensters. Ist fenster& = Null, so gibt es keine weiteren Fenster und Ihr augenblickliches Fenster wurde zuletzt geschaffen, ist also das jüngste.

Offset 4/6: Position der linken oberen Ecke

Offset 8/10: Fensterbreite und -höhe

Offset 12/14: Koordinaten der Maus

In diesen beiden Word-Feldern ist die Position der Maus relativ zur linken oberen Ecke des Fensters angegeben. Untypischerweise enthält dabei das erste Feld die y- und das zweite Feld die

x-Koordinate. Das folgende Programm zeigt, wie sich diese Informationen für ein kleines Zeichenprogramm umsetzen lassen:

```
WHILE INKEY$ = ""
  f&         = WINDOW(7)
  m.y%       = PEEKW (f& + 12)
  m.x%       = PEEKW (f& + 14)
  r.links%   = PEEK  (f& + 54)
  r.oben%    = PEEK  (f& + 55)

  m.y% = m.y% - r.oben%
  m.x% = m.x% - r.links%

  PSET (m.x%, m.y%)
WEND
```

GFA:

```
OPENW 0
WHILE INKEY$=""
  f%=WINDOW(0)
  my=DPEEK(f%+12)
  mx=DPEEK(f%+14)
  rl=PEEK(f%+54)
  ro=PEEK(f%+55)
  my=my-ro
  mx=mx-rl
  PLOT mx,my
WEND
```

Unbekannt sind unter Umständen die Zeilen 5 und 6. Dort werden die Variablen r.links% und r.oben% definiert. Sie finden eine eingehende Beschreibung dieser beiden Datenfelder ein paar Seiten weiter. Da diese Felder die Mausposition relativ zur linken oberen Ecke des Fensters (inkl. Rahmen) angeben, muß die Breite des oberen und des linken Rahmens abgezogen werden, damit korrekte Werte entstehen.

Wie Sie unschwer erkennen können, entstehen durch dieses primitive Malprogramm Punkte anstatt wirklicher Linien, wenn die Maus schnell bewegt wird. Dies liegt an der begrenzten Abfragegeschwindigkeit der Maus. Man kann sich aber behelfen und anstatt des PSET-Kommandos den LINE-Befehl verwenden, wie das nachfolgende Programm zeigt:

```
TRUE = -1: FALSE = 0

WHILE INKEY$ = ""
  f&         = WINDOW(7)
  m.y%       = PEEKW (f& + 12)
  m.x%       = PEEKW (f& + 14)
  r.links%   = PEEK  (f& + 54)
  r.oben%    = PEEK  (f& + 55)

  m.y% = m.y% - r.oben%
  m.x% = m.x% - r.links%

  IF ok = FALSE THEN
    ok = TRUE
  ELSE
    LINE (ma.x%,ma.y%)-(m.x%,m.y%)
  END IF

  ma.x% = m.x%
  ma.y% = m.y%
WEND
```

Im Grunde ähneln sich die Programme sehr. Beachtenswert ist hierbei jedoch die Variable ok. Beim ersten Durchlauf der Schleife ist diese Variable noch FALSE, wird dann aber TRUE. Erst wenn dies geschehen ist, beginnt das Programm damit, Linien zu ziehen. Dies ist notwendig, weil wegen des LINE-Befehls zwei Koordinaten anstelle einer benötigt werden. Somit produziert der erste Durchlauf (mit ok = FALSE) die Anfangskoordinate.

Wenn Sie bei diesem Programm die Maus extrem schnell bewegen, fallen gezackte Linien auf. Diese rühren von der Tatsache, daß es zeitweise nicht möglich war, beide (x- und y-) Werte rechtzeitig zu aktualisieren und somit einer der beiden noch der alten Position entspricht.

Offset 18/20/22/24: Fenster-Limits

Viele Fenster lassen sich mit der Maus vergrößern und auch verkleinern. Hierzu dient bekanntlich ein kleines Gadget in der rechten unteren Ecke des Fensters. Wird dieses Element mit dem Mauspointer angeklickt und die Maus dann bewegt, so läßt sich

die Größe des Fensters nahezu frei bestimmen. Nahezu frei, weil jedem Fenster gewisse, individuell festgelegte Minimal- und Maximalgrenzen gesetzt sind. Die Mindestbreite und die Mindesthöhe liegen in den ersten beiden Feldern als Punkteanzahl vor, die Höchstbreite und die Höchsthöhe entsprechend in den anderen beiden Feldern. Das folgende Programm zeigt, wie diese Werte ausgelesen werden und wie man selbst eigene Grenzen bestimmen kann. Zu letzterem Zweck wird die Intuition-Funktion WindowLimits() aufgerufen - zwar könnte man eigene Werte auch direkt in diese Speicherstellen "poken", doch übernimmt WindowLimits() selbständig einen Fehlercheck und filtert falsche Parameter heraus.

```
DECLARE FUNCTION WindowLimits% LIBRARY

LIBRARY "intuition.library"


TRUE = 1: FALSE = 0

f&              = WINDOW (7)
min.x%          = PEEKW (f& + 16)
min.y%          = PEEKW (f& + 18)
max.x%          = PEEKW (f& + 20)
max.y%          = PEEKW (f& + 22)

PRINT "Parameter fuer Window-Sizing"
PRINT "============================"
PRINT
PRINT "Mindestbreite: "; min.x% ;" Punkte."
PRINT "Mindesthoehe:  "; min.y% ;" Punkte."
PRINT
PRINT "Hoechstbreite: "; max.x% ;" Punkte."
PRINT "Hoechsthoehe:  "; max.y% ;" Punkte."
PRINT
PRINT "Wollen Sie diese Werte aendern (j/n) ?"

WHILE jn$ = ""
  lazyline
  jn$ = INKEY$
WEND

IF jn$ = "j" THEN
  PRINT SPACE$ (60)
  INPUT "Neue Mindestbreite"; min.x%
  INPUT "Neue Mindesthoehe "; min.y%
  INPUT "Neue Hoechstbreite"; max.x%
  INPUT "Neue Hoechsthoehe "; max.y%
```

```
        status% = WindowLimits% (WINDOW(7), min.x%, min.y%, max.x%,
max.y%)
        IF status% = TRUE THEN
          PRINT
          PRINT "Error-frei. Neue Daten implementiert."
        ELSE
          PRINT
          PRINT "ERROR: DIMENSION(EN) FALSCH GEWAEHLT."
        END IF
      ELSE
        PRINT
        PRINT "Ok. Ende"
      END IF
      END

SUB lazyline STATIC
        LOCATE ,1
        IF plopp = 0 THEN
          plopp = 1
          PRINT "--.--.--.--.--.--.--.--.--.--.--.--.--";
        ELSEIF plopp = 1 THEN
          plopp = 2
          PRINT "-.--.--.--.--.--.--.--.--.--.--.--.--.";
        ELSEIF plopp = 2 THEN
          plopp = 0
          PRINT ".--.--.--.--.--.--.--.--.--.--.--.--.-";
        END IF
END SUB
```

Die Funktion WindowLimits() wird mittels fünf Parametern aufgerufen:

```
status% = WindowLimits% (fenster&, mix%, miy%, max%, may%)
```

fenster&:	<WINDOW(7) (Anfangsadresse der Window-Struktur)
mix%,miy%:	Mindestbreite, Mindesthöhe
max%,may%:	Höchstbreite, Höchsthöhe

Nach der Ausführung liefert die Routine einen Statusreport zurück. Ist dieser Wert =1 (TRUE), so verlief alles ordnungsgemäß. Ist er jedoch Null, so konnte mindestens einer der angegebenenen Größen nicht ausgeführt werden. Dies kann daran gelegen haben, daß eine Mindestbreite größer war als die augenblickliche Breite des Fensters, etc.

GFA-BASIC besitzt für obige Aufgabe einen eigenen Befehl, der dem WindowLimits()-Befehl sehr ähnlich ist (und wahrscheinlich auf ihm beruht):

```
LIMITW Fensternr, minx, miny, maxx, maxy
```

Offset 24: Fenster-Modi

Intuition kennt verschiedene Fensterarten. Zunächst läßt sich jedes Fenster mit einer Reihe Extras ausstatten:

- Vergrößerungs-Verkleinerungs-Gadget
- Verschiebe-Kopfleiste
- Vodergrund-Hintergrund-Knöpfe
- Schließknopf

Zudem läßt sich die Refresh-Art festlegen. Darunter versteht man die Methode, mit der der Inhalt eines Fensters gespeichert wird, falls das Fenster (z.B. durch ein anderes Fenster) verdeckt wird. Hier gibt es:

- SimpleRefresh
- SmartRefresh
- SuperBitmap

Im SimpleRefresh-Modus wird der Inhalt des Fensters gar nicht gespeichert. Wird Ihr Fenster kurzfristig von einem anderen überschattet, so bleibt anschließend an dieser Stelle in Ihrem Fenster ein "Loch". Es ist dann an Ihnen, dieses zu flicken. Anders im SmartRefresh-Modus, der zwar speicheraufwendiger ist, aber ohne weiteres Zutun dafür sorgt, daß Ihr Fensterinhalt unbeschadet von wildestem Windowing immer ordnungsgemäß erscheint. Hierzu wird Intuition automatisch alarmiert, sobald ein anderes Fenster damit droht, einen Teil des Fensters zu überlappen. Sofort speichert Intuition den gefährdeten Teil Ihres Fensters in einem separaten Stück Speicher. Wird später der bedeckte Teil wieder ganz oder teilweise freigegeben, so kopiert Intuition unaufgefordert diesen oder ein Bruchstück davon zurück. Die dritte und speicherplatzaufwendigste Methode bedient

sich einer völlig separaten Bitmap: Der gesamte Fensterinhalt wird also zu allen Zeiten an separater Stelle im Speicher abgelegt und kann dort auch nicht beschädigt werden. Außerdem kann auf diese Weise eine viel größere Zeichenfläche geschaffen werden, als das Fenster groß ist. Diese bis zu 1024 x 1024 Punkte große Zeichenfläche kann dann innerhalb des Fensters hin- und hergescrollt werden (wie dies programmiert wird, werden wir Ihnen an anderer Stelle in diesem Buch verraten).

Neben diesen Grundattributen eines Fensters gibt es Spezialfeatures: Ein "Backdrop"-Fenster liegt immer hinter allen anderen und kann niemals hervorgehoben werden. So ist der Workbench-Screen in Wirklichkeit ein solches Backdrop-Fenster, das über dem eigentlichen Screen liegt. Ein "GimmeZeroZero"-Fenster ist zweigeteilt: Es besteht aus einem Rahmen und einer Zeichenfläche. Das normale BASIC-Fenster ist beispielsweise ein Fenster solchen Typs. Dieser Typ ermöglicht völlig ungezwungenes Zeichnen, denn es besteht niemals die Gefahr, versehentlich aus der Zeichenebene in den Rahmen zu zeichnen (allerdings verhindert der BASIC-Interpreter auch bei anderen Typen solcherlei Mißgeschick). Ein "Borderless"-Fenster schließlich besitzt keinen Rahmen. Das gerade erwähnte Workbenchfenster vom Typ "Backdrop" ist zugleich vom Typ "Borderless" - womit klarsteht: Diese Attribute können selbstverständlich auch in Kombination auftreten. Das folgende Beispiel demonstriert, wie man aus einem ganz normalen BASIC-Fenster ein Fenster des Typs "Borderless" macht - durch Manipulation des enstprechenden Bits in diesem Kontrollfeld und anschließendem Aufruf der Funktion "RefreshWindowFrame":

```
LIBRARY "intuition.library"

Borderless 1


SUB Borderless (nr%) STATIC
    '-----------------------------------------------
    'Aufruf: NewWindow (Fensternummer)
    '                   ^------------1=Basicfenster
    'wieder normal:  WINDOW Fensternummer,,,,-1
    '-----------------------------------------------
```

```
    WINDOW nr%,,,0,-1
    WINDOW OUTPUT nr%
    POKEL WINDOW(7) + 24, PEEKL (WINDOW(7) + 24) OR 2^11
    CALL RefreshWIndowFrame (WINDOW(7))
END SUB
```

Offset 28: Der Menü-Header

Eine besondere Eigenschaft aller Intuition-Fenster ist die Möglichkeit, ein Menü zu schaffen. Dieses Feld enthält den Zeiger auf die Kopfstruktur des Intuition-Menü-Systems für dieses Fenster. Für BASIC-Anwendungen sind diese Strukturen jedoch zu aufwendig, um in gesundem Verhältnis zum Effekt zu stehen. Hier empfehlen sich weiterhin die MENU-Befehle des Interpreters.

Offset 32: Titel-Text dieses Fensters

Jedes Fenster verfügt über einen eigenen Namen, der normalerweise in der Kopfleiste des Fensters eingeblendet ist. Dieser Name ist im ASCII-Code abgelegt. Die Anfangsadresse auf diesen Speicherbereich finden Sie in hier. Somit ist Ihnen die Möglichkeit gegeben, sowohl den augenblicklichen Namen Ihres Fensters auszulesen als auch einen eigenen Namen anzugeben. Das folgende Programm demonstriert beide Möglichkeiten und verwendet zum Setzen eines eigenen Namens die Intuition-Funktion SetWindowTitles(), so daß der neue Name sofort in der Kopfzeile erscheint. Auf diese Weise könnte man beispielsweise zu Anfang eines eigenen Programmes den Namen auslesen und speichern, um anschließend über die Kopfzeile auf markante Weise mit dem Anwender kommunizieren zu können (oder um einen Copyright-Hinweis auszugeben). Bei Programmschluß kann dann der zu Anfang abgelegte Originalname wiedereingesetzt werden. Hier das Programm:

```
LIBRARY "intuition.library"

WindowName alter.name$
PRINT "Bisheriger Name des Fensters ist: ";alter.name$
```

```
        LINE INPUT "Ihr neuer Name: ";neuer.name$
        WindowName neuer.name$

        LIBRARY CLOSE
        END



SUB WindowName (text$) STATIC
    '-----------------------------------
    'Format: WindowName NeuerName$
    '        neuen Namen setzen
    '
    'Format: WindowName Leerstring$
    '        Leerstring$ = Namen auslesen
    '-----------------------------------
    count% = 1
[1] par.1%  = 32

[2] IF text$ = "" THEN
      name.adr&   = PEEKL (WINDOW(7) + par.1%)

      in$ = CHR$ (PEEK( name.adr&))
      WHILE in$ <> CHR$(0)
        text$  = text$ + in$
        in$    = CHR$( PEEK( name.adr& + count%))
        count% = count% + 1
      WEND
    ELSE
[3]   text$ = text$ + CHR$(0)
      CALL SetWindowTitles (WINDOW(7), SADD(text$), -1)
    END IF
END SUB


[1]   Hier steht der Offset für dieses Datenfeld
[2]   Ist der angegebene String leer? Dann den Namen des
      Fensters auslesen und dort hinterlegen!
[3]   Der String war nicht leer. Also nullterminieren (mit
      einem 0-Charakter als Endkennzeichen abschließen).
```

Auch das GFA-Komplement soll Ihnen nicht vorenthalten werden. Das Auslesen des Namens geschieht durch diese Zeilen:

```
OPENW 0
fn%=LPEEK(WINDOW(0)+32)
fname$=CHAR{fn%}
PRINT fname$
```

Der Name des Fensters läßt sich unter GFA hierdurch verändern:

```
TITLEW Fensternr, "Neuer Titel"
```

Offset 36/40/44: Requester-Handling

Hier hinterlegt Intuition interne Informationen bezüglich der Auswertung von Requestern. Für BASIC unergiebig.

Offset 46: Kontakt zum Screen

An dieser Stelle begegnen Sie einem sehr wichtigen Datenfeld, auf das wir in Zukunft noch oft zu sprechen kommen. Natürlich gibt es keine Fenster ohne Screen. Zu Anfang existiert zumindest der Workbench-Screen als Umgebung Ihres Fensters, und auch wenn Sie mit Hilfe des SCREEN-Kommandos weitere Screens erschaffen haben, hat jedes Fenster "seinen" eigenen Screen, in dem es erscheint. Hier findet sich die Anfangsadresse auf eine Datenstruktur namens "Screen", auf die wir ein wenig später genauestens eingehen werden. Sicher haben Sie es sich schon gedacht: Wie jedes Fenster über seine Window-Struktur verfügt, gibt es auch für jeden Screen eine eigene Screen-Struktur gefüllt mit interessanten Daten.

Offset 50: Der Rastport des Fensters

Hier treffen wir endlich auf eine Spur des in Kapitel 2 andeutungsweise erwähnten Rastports. An dieser Stelle liegt der Zeiger auf den Rastport dieses Fensters, der sich als just eine weitere Datenstruktur entpuppt. Er stellt den Kontakt her zwischen der höheren Intuition-Ebene und der elementaren Welt der Graphics-Primitives, jener Grundbausteine der Amiga-Grafik, auf die wir später eingehen werden.

Offset 54/55/56/57: Fensterrahmen

In diesen vier Byte-Feldern finden sich die Abmessungen des Fensterrahmens. Bei der Behandlung der Mauskoordinaten traten diese Daten schon einmal in den Vordergrund. Neben dem Auslesen dieser Daten steht es Ihnen selbstverständlich frei, andere Werte einzusetzen und mittels RefreshWindowFrame() zu aktivieren. Hierbei wird jedoch keine sichtbare Veränderung eintreten, wenn nicht gleichzeitig das im Normalfall in der Fenstermitte liegende Layer miteinbezogen und entsprechend manipuliert wird.

Offset 58: Der Rahmenrastport des Fensters

Wie bereits angeschnitten, verfügen alle GimmeZeroZero-Fenster über zwei getrennte Zeichenebenen: Den Fensterrahmen sowie den eigentlichen Fensterinhalt. Die hier abgelegte Adresse zeigt auf den Rastport des Rahmens. Zwar wird die Struktur "Rastport" erst später eingehend behandelt, aber bereits in Kapitel 2 tauchte wenigstens eine Funktion aus der Grafik-Bibliothek auf, die mit dem Rastport zusammenarbeitet: Die Funktion Text() zur Ausgabe einer beliebigen Zeichenkette. Mit ihrer Hilfe installiert das folgende Programm eine Statuszeile im Kopf Ihres GimmeZeroZero-Fensters:

```
        LIBRARY "graphics.library"
        LIBRARY "intuition.library"

main:   CLS
        status "Diese Statuszeile faellt ins Auge! (TASTE!) ", 60

        WHILE INKEY$ = ""
        WEND

        WHILE jn$ <> "j"
          status "Eingabemodus: Bitte Namen angeben!", 60
          CLS
          LOCATE 1,1
          LINE INPUT "---> "; n$

          status "Kontrollmodus: " + n$ + " Korrekt (j/n) ?", 60
          LOCATE 1,1
          PRINT SPACE$ (50)     '{3}
          LOCATE 1,1
```

```
        LINE INPUT "---> "; jn$
      WEND

      status "Experiment abgeschlossen. Taste druecken!", 0
      CLS

      WHILE INKEY$ = ""
      WEND

      CALL RefreshWindowFrame(WINDOW(7))
      LIBRARY CLOSE
      END


SUB status (text$, weite%) STATIC

    '-------------------------------------------
    'Format: status "texttexttext...", breite%
    '        status "texttexttext...", 0 (breite =
    '                                    Textbreite)
    '        status text$, breite%
    '        status text$, 0 (s.o.)
    '-------------------------------------------

    border.rast&     = PEEKL (WINDOW(7) + 58)
    IF border.rast&  = 0 THEN
      BEEP
      PRINT "Dies ist kein Fenster vom Typ GimmeZeroZero."
      EXIT SUB
    END IF

    fenster.weite% = PEEKW (WINDOW(7) + 8)
    max.zeichen%   = INT ((fenster.weite% - 86) / 8)
    text.laenge%   = LEN(text$)
    IF weite%      = 0 THEN weite% = text.laenge%
    IF weite% < max.zeichen% THEN max.zeichen% = weite%
    IF text.laenge% < weite% THEN
      text$ = text$ + SPACE$ (weite% - text.laenge%)
    END IF

    CALL Move (border.rast&, 32, 7)
    CALL text (border.rast&, SADD(text$), max.zeichen%)
END SUB
```

Zum Programm:

In der Variable border.rast& wird der Inhalt dieses Datenfeldes abgelegt. Ist der Eintrag null, so handelt es sich nicht um ein

GimmeZeroZero-Fenster und besitzt somit auch keinen Rahmenrastport; das Programm endet dann.

Andernfalls wird der Text durch die Text()-Funktion der Graphics-Bibliothek ausgegeben. Die vorangestellte, noch unbekannte Funktion Move() setzt den Grafikcursor in die gewünschte Position.

In GFA-BASIC ist derselbe Effekt sogar noch einfacher programmierbar:

```
    OPENW 1
    FULLW 1
    '
    status_zeile("Diese Statuszeile faellt ins Auge!")
    WHILE INKEY$=""
    WEND
    WHILE jn$<>"j"
      status_zeile("Eingabemodus: Bitte Namen eingeben!")
      CLS
      LOCATE 1,1
      LINE INPUT "---> ";n$
      '
      status_zeile("Kontrollmodus: "+n$+" Korrekt (j/n) ?")
      LOCATE 1,1
      PRINT SPACE$(50)
      LOCATE 1,1
      LINE INPUT "---> ";jn$
    WEND
    '
status_zeile("Experiment beendet. Bitte bel. Taste druecken!")
WHILE INKEY$=""
WEND
END
'
PROCEDURE status_zeile(text$)
  f%=WINDOW(1)
  r%=LPEEK(f%+58)
  o%=LPEEK(f%+50)
  IF r%<>0 THEN
    breite=DPEEK(f%+8)
    max=INT((breite-86)/8)
    text$=LEFT$(text$,max)
    RASTPORT r%
    TEXT 32,7,text$
    RASTPORT o%
  ENDIF
RETURN
```

Offset 62: Erstes Gadget

Gadgets sind kleine (oder auch größere) "Schalt-Elemente", die von der Maus betätigt werden können. Dazu zählen beispielsweise der Ein/Aus-Schalter eines Fensters oder seine Vergrößerungsschalter. Dies ist die Adresse auf die erste Gadget-Struktur in einer ganzen Kette. Für BASIC ist das aber ohne Belang.

Offset 66 und 70: Vater- und Kind-Fenster

Wir erwähnten es bereits bei Offset 0: Intuition verwaltet alle Fenster in einer Datenkette. Jedes Fenster besitzt dabei eine eigene Fenster-Datenstruktur. In jeder Datenstruktur wiederum findet sich jeweils ein Zeiger auf das vorangegangene (Vater-) Fenster sowie auf das nächstfolgende (Kind-) Fenster. Das erste Fenster in der Datenkette besitzt keinen Vater-Zeiger (=0), das letzte keinen Kind-Zeiger (=0).

Diese beiden Felder sind sehr wichtig. Bisher hatten wir lediglich die Möglichkeit, die Adresse des jeweiligen Ausgabefensters in der Variablen WINDOW(7) zu erfragen. Nun können wir von jedem beliebigen Fenster jedes beliebige andere Fenster erreichen. Das folgende Beispiel macht dies deutlich:

```
'#######################################
'#
'# Programm: Fenster-Sucher
'# Datum: 5. April 87
'# Autor: tob
'# Version: 1.0
'#
'#######################################

init:       fenster& = WINDOW(7)

'* Nun wird das Ende der Datenkette gesucht.
'* Das Eltern-Feld des ersten Elementes ist =0

            WHILE found% = 0
              eltern.fenster& = PEEKL(fenster&+66)
              IF eltern.fenster&=0 THEN
                found% = 1
              ELSE
                fenster& = eltern.fenster&
```

```
        END IF
      WEND
      found% = 0

'* fenster& enthaelt nun die Adresse des Fenster-
'* Datenblocks des ersten Fensters in der Daten-
'* Kette. Nun wird diese Kette abgeklappert, bis
'* das Kind-Feld =0 ist.

      WHILE found%=0
        zaehler% = zaehler%+1
        PRINT zaehler%;
        PRINT ". Fenster:"
        PRINT "Adresse der Datenstruktur: ";fenster&

'* Nun wird der Name des gefundenen Fensters ausgegeben
'* Offset +32

        PRINT "Name des Fensters: ";
        fenster.name& = PEEKL(fenster&+32)
        WHILE ende% = 0
          gef$ = CHR$(PEEK(fenster.name&))
          IF gef$ = CHR$(0) THEN
            ende% = 1
            PRINT
          ELSE
            PRINT gef$;
            fenster.name& = fenster.name&+1
          END IF
        WEND
        PRINT
        ende% = 0
        kind.fenster& = PEEKL(fenster&+70)
        IF kind.fenster& = 0 THEN
          found% = 1
        ELSE
          fenster& = kind.fenster&
        END IF
      WEND
```

Hier das Programm in GFA:

```
OPENW 0
fenster%=WINDOW(0)
zaehler=0
WHILE gefunden$<>"gefunden!!!"
  eltern.fenster%=LPEEK(fenster%+66)
  IF eltern.fenster%=0
    gefunden$="gefunden!!!"
  ELSE
    fenster%=eltern.fenster%
```

```
    ENDIF
  WEND
'
  WHILE gefunden$<>"ende erreicht!!!"
    zaehler=zaehler+1
    PRINT ""
    PRINT zaehler;
    PRINT ". Fenster: "
    PRINT "Adresse der Datenstruktur: ";fenster%
    PRINT "Name des Fensters: ";
    fenster.name%=LPEEK(fenster%+32)
    PRINT CHAR{fenster.name%}
    '
    kind.fenster%=LPEEK(fenster%+70)
    IF kind.fenster%=0
      gefunden$="ende erreicht!!!"
    ELSE
      fenster%=kind.fenster%
    ENDIF
  WEND
  END
```

Durch diese Technik erhalten Sie vollen Zugriff auf alle unter Intuition verwalteten Fenster. Sie können so in fremde Fenster schreiben, deren Namen verändern etc. etc. Wir werden das später an entsprechenden Programmen zeigen.

Offset 74, 78, 79 und 80: Das Sprite-Image

Jedes Fenster hat die Möglichkeit, einen eigenen, völlig individuell gestalteten Mauszeiger zu erzeugen. Dazu dienen diese Felder. Sie legen fest, wo das neue Sprite-Image gespeichert ist, wie hoch der neue Pointer (beliebig) und wie breit (max 16 Punkte) er sein soll. Es ist jedoch nutzlos, hier direkt andere Werte einzupoken. Wer einen für sein Fenster individuell gestalteten Sprite-Pointer benötigt, kann diesen nur via Intuition ("SetPointer") implementieren. Wir zeigen Ihnen im Anschluß, wie es gemacht wird:

```
LIBRARY "intuition.library"
          image$=""
          sprite.hoehe%   =  14
          sprite.breite%  =  16
          sprite.xOff%    = -7
          sprite.yOff%    = -6
```

```
RESTORE daten
FOR loop%=0 TO 31
  READ info&
  hi%    = INT(info&/256)
  lo%    = info&-(256*hi%)
  image$ = image$+CHR$(hi%)+CHR$(lo%)
NEXT loop%
CALL SetPointer(WINDOW(7),SADD(image$),sprite.hoehe%,
     sprite.breite%,sprite.xOff%,sprite.yOff%)

CLS
PRINT "Beliebige Taste = Abbruch"
PRINT "Linke Maustaste = Zeichnen"

DIM area.pat%(3)
area.pat%(0) = &H1111
area.pat%(1) = &H2222
area.pat%(2) = &H4444
area.pat%(3) = &H8888
PATTERN ,area.pat%
COLOR 2,3

WHILE INKEY$=""
  state%=MOUSE(0)
  IF state%<0 THEN
    mouseOldX% = mouseX%
    mouseOldY% = mouseY%
    mouseX%    = MOUSE(1)
    mouseY%    = MOUSE(2)
    IF lplot% = 0 THEN
      lplot% = 1
      PSET (mouseX%,mouseY%)
    ELSE
      LINE (mouseOldX%,mouseOldY%)-(mouseX%,mouseY%), 1,bf
    END IF
  ELSE
    lplot% = 0
  END IF
WEND

COLOR 1,0

CALL ClearPointer(WINDOW(7))
LIBRARY CLOSE
END


daten:  DATA 0,0
        DATA 256,256
        DATA 256,256
        DATA 256,256
```

```
DATA 896,0
DATA 3168,0
DATA 12312,0
DATA 256,49414
DATA 256,49414
DATA 12312,0
DATA 3168,0
DATA 896,0
DATA 256,256
DATA 256,256
DATA 256,256
DATA 0,
```

Offset 82, 86, 90 und 94: IDCMP-Flags und Message Ports

IDCMP steht für "Intuition Direct Communications Message Port". Über diese Nachrichtenkanäle kann Intuition mit anderen Tasks, zum Beispiel dem BASIC-Task, kommunizieren. Für Sie als Programmierer ist das jedoch uninteressant, denn die Nachrichten werden ohnehin von BASIC abgefangen und weiterverarbeitet.

Offset 98 und 99: Fenster-Farben

In diesen beiden Byte-Feldern sind die Farbregister abgespeichert, aus denen das Fenster seine Farben bezieht. Sie können durch POKE die Farbwerte verändern. Die Veränderung wird jedoch von Intuition nicht sofort durchgeführt, sondern erst, sobald das Fenster nachgezeichnet werden muß. Das passiert beispielsweise, wenn es verschoben oder vergrößert wird.

Offset 100: Das Check-Mark-Image

Hier findet sich die Adresse auf eine Kleingrafik. Sicherlich kennen Sie die Möglichkeit, im Menü einen kleinen Haken auftauchen zu lassen, der angibt, welche Selektion augenblicklich gilt. Dieses Häkchen bezeichnet man im englischen als "Check-Mark", und hier findet sich die Adresse auf den Speicherbereich, in dem das Aussehen dieses Hakens definiert ist (bzw. =0, wenn das Standard Checkmark Image verwendet wird).

Offset 104: Der Screen-Titel

Der Screen, in dem Ihr Fenster haust, kann verschiedene Namen besitzen. Sein Name hängt ab vom selektierten (=aktiven) Fenster. Jedes Fenster kann den Screen anders nennen. Es erscheint jeweils der Name, der im aktiven Fenster an der durch diese Adresse angegebenen Stelle abgespeichert ist.

Offset 108, 110, 112 und 114: GimmeZeroZero-Parameter

Diese Felder werden nur im Falle eines GimmeZeroZero-Fensters benötigt. GZZ-MausX und GZZ-MausY verhalten sich absolut analog zu den Offset-Feldern 12 und 14: Sie geben die Koordinaten des Maus-Pointers an. Diese Koordinaten verstehen sich jedoch relativ zur Zeichenebene, nicht zum Fenster. Der Nullpunkt liegt in der oberen linken Ecke des Fensterinhaltes, nicht in der oberen linken Ecke des Fensterrahmens.

Die anderen beiden Felder verhalten sich analog zu den Offset-Feldern 8 und 10: Hier ist die Breite und die Höhe des Fensterinhaltes exklusive Rahmen gespeichert, nicht die des Fensters inklusive Rahmens.

Offset 116 und 120: Optionale Zeiger

Mit Hilfe dieser beiden Zeiger lassen sich weitere Datenblöcke anderer Natur mit dieser Standard-Struktur verbinden. Der erste Zeiger ist dabei für Intuition reserviert, der zweite steht dem Anwender zur Verfügung.

3.3 Die Funktionen der Intuition-Bibliothek

Sie wissen nun, wie Intuition Fenster verwaltet. Wir können deshalb jetzt damit beginnen, die Routinen der Intuition-Bibliothek vorzustellen, die zuständig sind für Fenster:

SetPointer()
ClearPointer()
MoveWindow()
SizeWindow()
WindowLimits()
WindowToBack()
WindowToFront()

3.3.1 Ein individueller Maus-Pointer

Die Funktion "SetPointer" erlaubt es Ihnen, einen völlig individuellen Maus-Pointer für Ihr Fenster zu kreieren. Er wird dann immer an Stelle des "normalen" Pointers auf dem Bildschirm erscheinen, sobald Ihr Fenster aktiv ist.

Die Funktion verlangt sechs Parameter:

```
SetPointer(fenster,image,höhe,breite,xOff,yOff)
```

fenster:	Adresse der Fenster-Datenstruktur des entsprechenden Fensters
image:	Adresse eines Sprite-Image-Blockes
höhe:	Höhe des Sprites
breite:	Breite des Sprites (max. 16)
xOff:	Markiert den "Hot Spot"
yOff:	Markiert den "Hot Spot"

Bevor wir weitere Worte verlieren, ein entsprechendes Demo-Programm:

```
'#######################################
'#
'# Programm: SetPointer/ClearPointer
'# Datum: 5.April 87
'# Autor: tob
'# Version: 1.0
'#
'#######################################

' Der Amiga Standard Mauspointer wird durch
' einen selbstdefinierten Pointer ersetzt.
```

```
PRINT "Suche das .bmap-File..."

'INTUITION-Bibliothek
'SetPointer()
'ClearPointer()

LIBRARY "intuition.library"

init:       image$=""
            sprite.hoehe%   =  14
            sprite.breite%  =  16
            sprite.xOff%    = -7
            sprite.yOff%    = -6

image:      '* Einlesen des Sprite-Images
            RESTORE daten
            FOR loop%=0 TO 31
              READ info&
              hi%    = INT(info&/256)
              lo%    = info&-(256*hi%)
              image$ = image$+CHR$(hi%)+CHR$(lo%)
            NEXT loop%

setpoint:   '* Neues Image einbauen
            CALL
SetPointer(WINDOW(7),SADD(image$),sprite.hoehe%,sprite.breite%,
   sprite.xOff%,sprite.yOff%)

mainDemo:   '* Hier eine Demonstration des neuen Pointers
            CLS
            PRINT "Beliebige Taste = Abbruch"
            PRINT "Linke Maustaste = Zeichnen"

           '* mit Muster zeichnen
            DIM area.pat%(3)
            area.pat%(0) = &H1111
            area.pat%(1) = &H2222
            area.pat%(2) = &H4444
            area.pat%(3) = &H8888
            PATTERN ,area.pat%
            COLOR 2,3

            WHILE INKEY$=""
              state%=MOUSE(0)
              IF state%<0 THEN
                mouseOldX% = mouseX%
                mouseOldY% = mouseY%
                mouseX%    = MOUSE(1)
                mouseY%    = MOUSE(2)
                IF lplot% = 0 THEN
                  lplot% = 1
                  PSET (mouseX%,mouseY%)
```

```
            ELSE
              LINE (mouseOldX%,mouseOldY%)-(mouseX%,mouseY%),1,bf
            END IF
          ELSE
            lplot% = 0
          END IF
        WEND

        COLOR 1,0

ende:   '* Demo ende, alten Pointer
        CALL ClearPointer(WINDOW(7))
        LIBRARY CLOSE
        END


daten:  '* Die Sprite-Datas
        DATA 0,0
        DATA 256,256
        DATA 256,256
        DATA 256,256
        DATA 896,0
        DATA 3168,0
        DATA 12312,0
        DATA 256,49414
        DATA 256,49414
        DATA 12312,0
        DATA 3168,0
        DATA 896,0
        DATA 256,256
        DATA 256,256
        DATA 256,256
        DATA 0,0
```

Programm-Beschreibung:

Unser neuer Maus-Pointer soll 16 Pixel breit und 14 Pixel hoch werden. Der "Hot Spot", der Punkt, an dem unser Maus-Pointer empfindlich ist, liegt in unserem Beispiel 7 Pixel rechts und sechs Pixel unterhalb der linken oberen Sprite-Ecke.

Im Programmteil "image" wird das neue Äußere unseres Pointers definiert. Die Daten dazu liegen im Teil "Daten". Jede Zeile eines Sprites darf bis zu 16 Punkte breit sein. Der Datenblock besteht nun aus zwei 16-Bit-Werten pro Sprite-Zeile. Da unser Beispiel-Sprite 14 Zeilen hoch sein soll, existieren auch 14x2=28 Sprite Daten (zzgl. 2x0 am Anfang und am Ende, um DMA aus-

zuschalten). Für jeden möglichen Punkt des Sprites gibt es also zwei Bits. Ist keines der beiden Bits gesetzt, dann erscheint dieser Punkt des Sprites transparent. Die anderen drei Kombinationen entsprechen den möglichen drei Sprite-Farben.

Die Sprite-Daten werden in der String-Variable image$ gespeichert. Dazu müssen die 16-Bit-Werte zunächst in zwei 8-Bit-Werte (lo- und hi-Byte) verwandelt werden.

Schließlich erfolgt der Aufruf "SetPointer", der augenblicklich den neuen Pointer aktiviert.

Am Ende der Zeichendemo steht der Aufruf der Routine "ClearPointer". Er aktiviert den normalen Pointer wieder.

3.3.2 Fenster-Verschieben leicht gemacht

Sicherlich kennen Sie die Möglichkeit, mit Hilfe der Maus Fenster umherzuverschieben. Dasselbe läßt sich mittels Intuition auch durch eine Programmanweisung bewerkstelligen. Dazu wird die Intuition-Routine "MoveWindow" benötigt. Sie verlangt drei Argumente:

```
MoveWindow(fenster,deltaX,deltaY)
```

fenster:	Adresse der Fenster-Datenstruktur
deltaX:	Anzahl der Pixel, um die das Fenster nach rechts geschoben werden soll (negativ = links)
deltaY:	Entsprechend, jedoch vertikale Verschiebung

Diese Routine überprüft Ihre Angaben nicht auf Richtigkeit. Liegen also Ihre Delta-Werte einige Screen-Breiten außerhalb des Monitors, dann versucht Intuition, das Fenster aus dem Monitor zu schieben. Das klappt natürlich nicht, die Kiste hängt. Wir haben uns deshalb erlaubt, Ihre Eingaben auf Richtigkeit zu überprüfen. Dazu dient eine kleine Error-Check-Routine, die auf den Informationen basiert, die in der Fenster-Datenstruktur zu finden sind (siehe Kapitel 2.2)

Hier das Programm: Es ist ein SUB namens Move.

```
'#####################################
'#
'# Programm: Fenster verschieben
'# Datum: 10.4.87
'# Autor: tob
'# Version: 1.0
'#
'#####################################

'Intuition kann programmgesteuert beliebige
'Fenster verschieben. Dieses Programm demon-
'striert den WindowMove()-Befehl (incl. Error-
'Check)

PRINT "Suche das .bmap-File..."

'INTUITION-Bibliothek
'MoveWindow()

LIBRARY "intuition.library"


demo:       CLS
            WINDOW 2,"Test-Fenster",(10,10)-(400,100),16
            WINDOW OUTPUT 2

            PRINT "Original-Position! Bitte Taste druecken!"
            WHILE INKEY$="":WEND

            Move 10,20 '10 rechts, 20 unten
            PRINT "Neue Position! Taste druecken!"
            WHILE INKEY$="":WEND

            Move -10,-20 '10 links, 20 hoch
            PRINT "Zurueck!"

            FOR t=1 TO 3000:NEXT t

            Move 10000,10000 'FEHLER
            '(nix passiert, dank error-check!)

            WINDOW CLOSE 2
            LIBRARY CLOSE

SUB Move(x%,y%) STATIC
      fen&            = WINDOW(7)
      screen.breite%  = 640
      screen.hoehe%   = 256
      fenster.x%      = PEEKW(fen&+4)
```

```
    fenster.y%      = PEEKW(fen&+6)
    fenster.breite% = PEEKW(fen&+8)
    fenster.hoehe%  = PEEKW(fen&+10)
    min.x%          = fenster.x%*(-1)
    min.y%          = fenster.y%*(-1)
    max.x%          = screen.breite%-fenster.breite%-fenster.x%
    max.y%          = screen.hoehe%-fenster.hoehe%-fenster.y%
    IF x%<min.x% OR x%>max.x% THEN x%=0
    IF y%<min.y% OR y%>max.y% THEN y%=0
    CALL MoveWindow(fen&,x%,y%)
END SUB
```

3.3.3 Setzen der Fenster-Limits

Für alle Fenster, deren Größe variabel ist, gibt es eine Mindest- und eine Höchstgröße. Die Intuition-Routine "WindowLimits" setzt diese Grenzen Ihren Angaben entsprechend. Dabei werden die Datenfelder der Fenster-Datenstruktur (siehe Kapitel 2.2) ab Offset 16 direkt manipuliert. Diese Routine kontrolliert außerdem, ob die mitgelieferten Argumente stimmen. Entsprechend wird ein Wert zurückgeliefert, der TRUE (=1) ist, falls alles geklappt hat. Andernfalls ist er FALSE (=0).

"WindowLimits" verlangt fünf Argumente und liefert einen Wert zurück:

```
resultat%=WindowLimits%(fenster,minX,minY,maxX,maxY)
```

resultat%:	1 = alles OK
	0 = Mindestgrößen größer Maxigrößen etc.
minX,minY:	Mindestausdehnungen des Fensters
maxX,maxY:	Höchstausdehnungen des Fensters

Hier ein Beispiel:

```
DECLARE FUNCTION WindowLimits% LIBRARY
LIBRARY "intuition.library"

minX%=5
minY%=5
maxX%=640
maxY%=200
res%=WindowLimits%(WINDOW(7),minX%,minY%,maxX%,maxY%)
```

```
IF res%=0 THEN
  PRINT "Etwas stimmte nicht..."
END IF

LIBRARY CLOSE
END
```

3.3.4 Vergrößern und Verkleinern von Fenstern

Die Intuition-Funktion "SizeWindow" ist in der Lage, ein Fenster nach Belieben zu verkleinern oder vergrößern. Sie verlangt drei Argumente:

```
SizeWindow(fenster,deltaX,deltaY)
```

fenster:	Adresse der Fenster-Datenstruktur
deltaX:	Anzahl der Pixel, um die das Fenster in horizontaler Richtung vergrößert werden soll (negativ = verkleinern)
deltaY:	Entsprechend, jedoch vertikal

Auch hier wird kein Fehler-Check gemacht. Sollten Ihre Delta-Werte ein Fenster derart deformieren, daß es kleiner als nichts oder größer als der bestehende Screen wird, kommt es zu einem System-Crash. Deshalb haben wir auch hier eine Sicherung eingebaut, die falsche Werte erkennt und unschädlich macht. Das SUB nennt sich "Size":

```
'#######################################
'#
'# Programm: Fenster-Limits
'# Datum: 10.4.87
'# Autor: tob
'# Version: 1.0
'#
'#######################################

'Demonstriert das Setzen der Fenster Min-
'dest- und Hoechstgrenzen fuer das Ver-
'schieben.

PRINT "Suche das .bmap-File..."

'INTUITION-Bibliothek
DECLARE FUNCTION WindowLimits% LIBRARY
```

```
LIBRARY "intuition.library"

demo:       CLS
            WINDOW 2,"Test-Fenster",(10,10)-(400,100),16
            WINDOW OUTPUT 2

           '* Fenster-Grenzen setzen
            r%=WindowLimits%(WINDOW(7),0,0,600,200)
            IF r%=0 THEN ERROR 255

            PRINT "Original-Groesse! Bitte Taste druecken!"
            WHILE INKEY$="":WEND

            Size 60,40 '60 rechts, 40 unten
            PRINT "Neue Groesse! Taste druecken!"
            WHILE INKEY$="":WEND

            Size -60,-40 '60 links, 40 hoch
            PRINT "Zurueck!"

           '* warten
            FOR t=1 TO 3000:NEXT t

           '* fehlerhafte Eingabe wird abgefangen
            Size 10000,10000 'FEHLER
            '(nix passiert, dank error-check!)

            WINDOW CLOSE 2
            LIBRARY CLOSE

SUB Size(x%,y%) STATIC
      fen&=WINDOW(7)
      fenster.breite% = PEEKW(fen&+8)
      fenster.hoehe%  = PEEKW(fen&+10)
      fenster.minX%   = PEEKW(fen&+16)
      fenster.minY%   = PEEKW(fen&+18)
      fenster.maxX%   = PEEKW(fen&+20)
      fenster.maxY%   = PEEKW(fen&+22)
      min.x%          = fenster.minX%-fenster.breite%
      min.y%          = fenster.minY%-fenster.hoehe%
      max.x%          = fenster.maxX%-fenster.breite%
      max.y%          = fenster.maxY%-fenster.hoehe%
      IF x%<min.x% OR x%>max.x% THEN x%=0
      IF y%<min.y% OR y%>max.y% THEN y%=0
      CALL SizeWindow(fen&,x%,y%)
END SUB
```

3.3.5 Programmgesteuertes Tiefen-Arrangement

Fenster lassen sich - relativ zu anderen Fenstern - in den Hintergrund schieben oder in den Vordergrund holen. Das geschieht normalerweise mit der Maus. Aber es gibt auch zwei Intuition-Funktionen, die genau dasselbe tun, sich aber in Programmen verwenden lassen: WindowToFront und WindowToBack.

Hier eine kleine Demonstration:

```
LIBRARY "intuition.library"

FOR loop%=1 to 10
  CALL WindowToBack(WINDOW(7))
  PRINT "Hinten!"
  FOR t=1 TO 2000:NEXT t
  CALL WindowToFront(WINDOW(7))
  PRINT "Vorn!"
  FOR t=1 TO 2000:NEXT t
NEXT loop%
```

3.4 Der Intuition-Screen

Neben den Fenstern werden auch die Screens von Intuition verwaltet. Ähnlich wie die Intuition-Fenster besitzen auch die Intuition-Screens eine eigene Datenstruktur. Den Zeiger darauf finden Sie in der Fenster-Datenstruktur ab Offset 46. Für Ihr aktuelles Ausgabefenster lautet die Basisadresse dieser Datenstruktur also bei AmigaBASIC:

```
screen&=PEEKL(WINDOW(7)+46)
```

GFA-BASIC liefert die Adresse in der Variablen SCREEN(x), wobei x die Kennzahl des gewünschten Screens ist:

```
OPENS 1
PRINT SCREEN(1)
```

Die Adressen der einzelnen Datenfelder erhalten Sie wieder durch Addieren der Offsets zu obiger Basisadresse. Es folgt nun die Belegung dieser Struktur:

Datenstruktur "Screen"/Intuition/342 Bytes

Offset	Typ	Bezeichnung
+ 000	Long	--> nächster Screen
+ 004	Long	--> erstes Fenster in diesem Screen
+ 008	Word	X-Koordinate der linken oberen Ecke
+ 010	Word	Y-Koordinate der linken oberen Ecke
+ 012	Word	Breite des Screens
+ 014	Word	Höhe des Screens
+ 016	Word	Y-Koordinate des Maus-Pointers
+ 018	Word	X-Koordinate des Maus-Pointers
+ 020	Word	Flags
	Bit 0:	1=Workbench-Screen
	Bit 0-3:	1=Custom Screen
	Bit 4:	1=Show Title
	Bit 5:	1=Screen beept gerade
	Bit 6:	1=Custom Bit Map
+ 022	Long	--> Screen-Namenstext
+ 026	Long	--> Standard-Titeltext
+ 030	Byte	Kopfleisten-Höhe
+ 031	Byte	vertikale Grenze der Kopfleiste
+ 032	Byte	horizontale Grenze der Kopfleiste
+ 033	Byte	vertikale Grenze des Menüs
+ 034	Byte	horizontale Grenze des Menüs
+ 035	Byte	oberer Fensterrahmen
+ 036	Byte	linker Fensterrahmen
+ 037	Byte	rechter Fensterrahmen
+ 038	Byte	unterer Fensterrahmen
+ 039	Byte	unbenutzt
+ 040	Long	--> Standard Font TextAttr
+ 044	----	Viewport des Screens
+ 084	----	Rastport des Screens
+ 184	----	Bitmap des Screens
+ 224	----	Layerinfo des Screens
+ 326	Long	--> erstes Screen-Gadget
+ 330	Byte	Detail Pen
+ 331	Byte	Block Pen
+ 332	Word	Backup-Register für Beep(), speichert Col0
+ 334	Long	--> externe Daten
+ 338	Long	--> User Daten

Sicherlich werden Sie viele Ähnlichkeiten zwischen dieser und der Fenster-Datenstruktur aus Kapitel 2.1 gefunden haben. Ei-

nige Felder sind jedoch ganz neu. Wieder werden wir die Felder der Reihe nach durchgehen und ihre Bedeutung klären.

3.4.1 Die Screen-Daten unter der Lupe

Offset 0: Nächster Screen

Auch die Screens werden von Intuition in der Form einer Datenkette organisiert. Wenn es neben Ihrem Screen noch weitere geben sollte, dann finden Sie hier die Adresse des Screen-Datenblocks für den nächsten Screen.

Offset 4: Erstes Fenster

Sicherlich erinnern Sie sich noch an die Datenkette, in der die Fenster gehalten wurden: Zwei Felder, das Eltern- und das Kind-Feld, gaben jeweils die Adresse des vorangegangenen und des nachfolgenden Fensters an. So konnte man von jedem Fenster aus die Datenkette auf oder ab wandern und gelangte zu jedem beliebigen Fenster. Diese Methode hatte einen kleinen Schönheitsfehler, denn um die gesamte Datenkette entlanglaufen zu können, mußte man zunächst ihren Anfang suchen, denn das "Einstiegsfenster" lag meist nicht zufällig dort.

Interessieren Sie sich lediglich für Fenster innerhalb eines Screens, dann gibt es eine einfachere Methode: In diesem Feld ist die Adresse der ersten Fenster-Datenstruktur abgespeichert. Die Adresse der nachfolgenden Struktur finden Sie im jeweils ersten Feld einer jeden Fenster-Datenstruktur (Offset +0):

```
fenster&=WINDOW(7)
scr&=PEEKL(fenster&+46)
sucher&=scr&+4
WHILE flag%=0
  sucher&=PEEKL(sucher&)
  IF sucher&=0 THEN
    flag%=1
  ELSE
    zaehler%=zaehler%+1
```

```
    PRINT zaehler%;
    PRINT ". Fenster-Datenblock: Adresse ";
    PRINT sucher&
  END IF
WEND
END
```

Nochmals: Diese Methode ist programmtechnisch einfacher, listet aber nur diejenigen Fenster, die zusammen mit dem aktuellen Ausgabefenster in ein- und demselben Screen liegen.

Offset 8, 10, 12 und 14: Dimensionen des Screens

Ganz analog zur Fenster-Datenstruktur finden Sie hier die Koordinaten der linken oberen Ecke des Screens relativ zur obersten Ecke des Displays, sowie Breite und Höhe. In der augenblicklichen Version der Amigas 500-2000 lassen sich Screens nicht horizontal verschieben. Feld Offset +8 ist damit lediglich der Kompatibilität wegen vorhanden.

Offset 16 und 18: Die Maus-Koordinaten

Hier finden Sie die Y- und X-Koordinaten des Maus-Pointers relativ zur linken oberen Ecke des Screens. Während des Herunter- oder Heraufziehens des Screens kann es beim Y-Wert zu kleinen Schwankungen kommen.

Offset 20: Flags

Die Bit-Belegungen erklären sich von selbst. "Show Title" bedeutet, daß der Titeltext des Screens sichtbar ist. Eine Custom Bitmap ist eine vom Anwender beim Erzeugen eines neuen Screens mitgelieferte eigene Zeichenebene.

Offset 22 und 26: Die Namen des Screens

Hier findet sich a) die Adresse auf den Namensstring dieses Screens sowie b) ein Standard-Text, der sich von Fenstern übernehmen läßt, wenn dort kein anderer festgelegt wurde.

Offset 30 - 39: Default-Parameter

Diese Byte-Felder enthalten diverse Standardparameter, wie z.B. die Abmessungen der Kopfleiste etc. Alle Fenster in diesem Screen richten sich nach diesen Werten und übernehmen sie für sich selbst.Offset 40: Der Standard-Zeichengenerator

Sobald ein Fenster innerhalb Ihres Screens geöffnet wird, besitzt es einen standardmäßigen Character-Generator (eine vorbestimmte Schriftart). Die Adresse auf diesen Standard-Zeichengenerator liegt in diesem Feld.

Wir werden das Thema "Zeichengenerator" später ausführlich behandeln.

Offset 44: Der Viewport

Und wieder haben wir einen neuen Begriff: Viewport. In der Datenstruktur des Screens ist dieses hier ausnahmsweise kein Zeiger. Ab Offset 44 liegt der eigentliche Viewport des Screens. Bei ihm handelt es sich um eine eigenständige kleine Datenstruktur von 40 Bytes. Ohne an dieser Stelle weiter auf ihn eingehen zu wollen, handelt es sich bei ihm um den Knotenpunkt zu Amigas Grafik-Hardware, dem Grafik-Coprozessor "Copper". Sie müssen sich jedoch noch eine Weile gedulden, bevor wir ihn genauer unter die Lupe nehmen.

Offset 84: Der Rastport

Auch hierbei handelt es sich nicht um einen Zeiger, sondern um den Rastport höchstselbst. Sie hatten bereits bei der Fenster-Datenstruktur mit ihm das Vergnügen. Da der Screen - wie auch

das Fenster - eine Zeichenebene ist, findet sich auch hier ein solcher Rastport. Nähere Erläuterungen folgen auch hier in Kürze.

Offset 184: Die Bitmap

Erneut eine eigenständige Datenstruktur namens Bitmap, 40 Bytes groß. Dies ist der Knotenpunkt des Screens mit den eigentlichen Speicherbereichen, in denen der Screen-Inhalt abgelegt wird, den sogenannten "Bitplanes". Auch dies ist ein eigenständiges Thema. Es wird an späterer Stelle aufgegriffen.

Offset 224: Das LayerInfo

Die letzte interne Datenstruktur des Screens. Hierbei handelt es sich um das Kernstück des Windowing-Systems, den Layers. Wir werden auch das noch eingehend behandeln.

Offset 326: Zeiger auf Screen-Gadgets

Auch Ihr Screen kennt Gadgets, mit deren Hilfe er sich in den Vordergrund holen oder in den Hintergrund schieben läßt. Dieses Feld ist allerdings für den internen Systemgebrauch bestimmt.

Offset 330 und 331: Die Screen-Farben

Wie auch bei den entsprechenden Feldern in der Fenster-Datenstruktur machen sich Manipulationen dieser Farben erst bemerkbar, wenn der Screen neu gezeichnet wird, also z.B. die Screen-Menüs benutzt werden.

Offset 332: Backup-Register

Hier hinterlegt Intuition die Farbe des Registers 0, wenn es diesen Screen beepen läßt. Das geschieht z.B. durch:

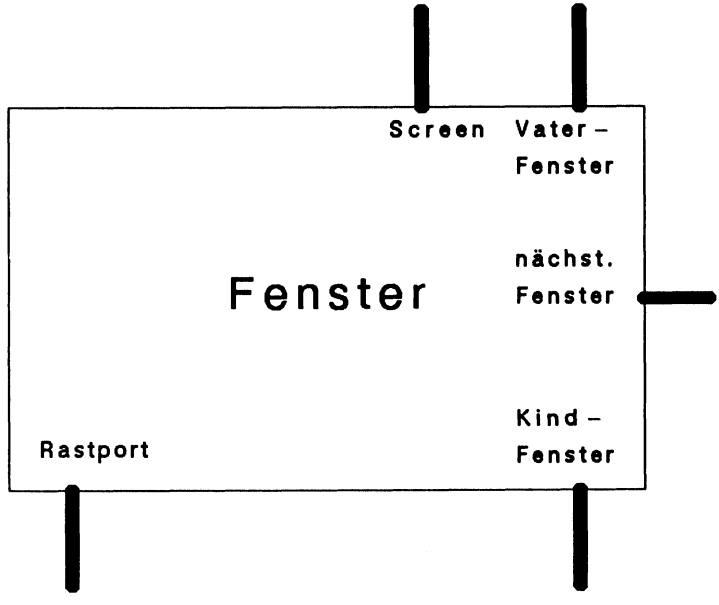
```
PRINT CHR$(7)
```

Offset 334 und 338: Externe und User Daten

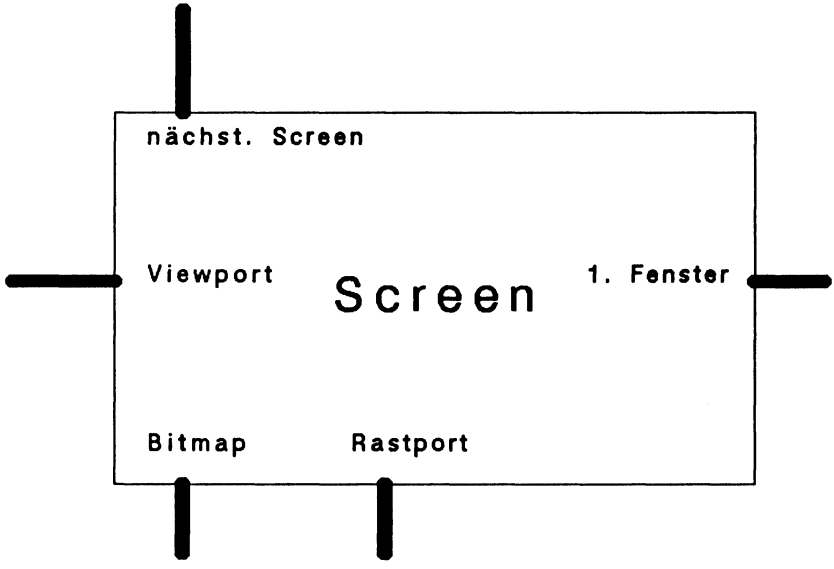
Wieder die Möglichkeit, weitere Datenblöcke mit dieser Standard-Struktur zu verbinden.

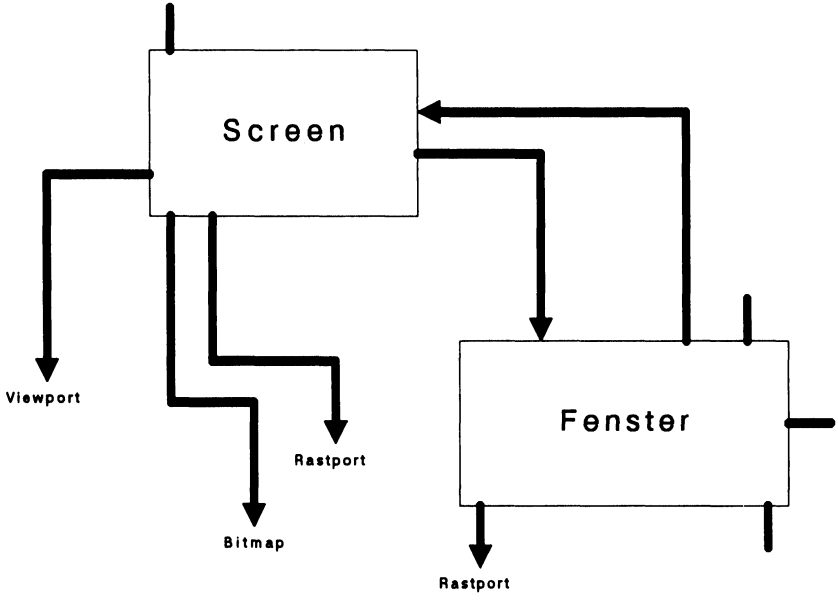
3.5 Intuition und der Rest der Welt

Sie haben bis jetzt die Datenstrukturen "Fenster" und "Screen" kennengelernt. Es ist an der Zeit, ein Resümee zu ziehen und diese Datenblöcke in Zusammenhang zu bringen. Sehen Sie sich dazu bitte einmal die folgende Zeichnung an. Sie symbolisiert eine Fenster-Datenstruktur. Die Ausgänge sind Zeiger auf andere Komponenten, die innerhalb dieser Struktur zu finden sind:



Ebenso läßt sich die "Screen"-Struktur verbildlichen:


nächst. Screen
Viewport
Screen
1. Fenster
Bitmap
Rastport


Screen
Viewport
Rastport
Bitmap
Fenster
Rastport

Die obenstehende Zeichnung stellt ein System im Zusammenhang dar, bestehend aus einem Screen und einem Fenster.

Noch ist das Bild unvollständig. Es fehlen gänzlich Kenntnisse über die Strukturen "Rastport", "Viewport" und "Bitmap". Als nächstes nehmen wir uns daher den Rastport vor.

3.6 Der Rastport

Mit diesem Datenblock nähern wir uns den Grundelementen der Amiga-Grafik, den sogenannten "Graphic Primitives". Wir verlassen damit die Welt der "Intuition" und begeben uns eine Stufe tiefer in der Amiga System-Architektur, auf die Stufe der Grafik-Bibliothek.

Der Rastport verwaltet eine Zeichenebene. Er enthält die Daten darüber, wie eine Zeichnung vonstatten zu gehen hat. Die Anfangsadresse dieses Datenblockes für das aktuelle Fenster liegt für AmigaBASIC immer in der Variablen WINDOW(8). Ebenso gut läßt sie sich aber auch direkt aus der Fenster-Datenstruktur auslesen:

```
PRINT WINDOW(8)
PRINT PEEKL(WINDOW(7)+50)
```

Letzteres ist auch bei GFA nötig:

```
OPENW 0
PRINT LPEEK(WINDOW(0)+50)
```

Der Rastport-Block ist wie folgt aufgebaut:

Datenstruktur "Rastport"/graphics/100 Bytes

Offset	Typ	Bezeichnung
+ 000	Long	--> die Layer-Struktur
+ 004	Long	--> die Bitmap-Struktur
+ 008	Long	--> das AreaFill-Muster
+ 012	Long	--> TmpRas-Struktur

Offset	Typ	Bezeichnung
+ 016	Long	--> AreaInfo-Struktur
+ 020	Long	--> GelsInfo-Struktur
+ 024	Byte	Mask: Schreibmaske für dieses Raster
+ 025	Byte	Vordergrundfarbe
+ 026	Byte	Hintergrundfarbe
+ 027	Byte	AreaFill-Outline-Farbe
+ 028	Byte	Zeichen-Modi
		JAM1 = 0
		JAM2 = 1
		COMPLEMENT = 2
		INVERSEVID = 4
+ 029	Byte	AreaPtSz: 2n Words für AreaFill-Muster
+ 030	Byte	unbenutzt
+ 031	Byte	line draw pattern preshift
+ 032	Word	verschiedene Kontroll-Bits
		FIRST DOT = 1: zeichne ersten Punkt?
		ONE DOT = 2: one dot mode für Linien
		DBUFFER = 4: double buffered gesetzt
+ 034	Word	LinePtrn: 16 Bits für Linienmuster
+ 036	Word	X-Koordinate des Grafik-Cursors
+ 038	Word	Y-Koordinate des Grafik-Cursors
+ 040	----	8x1 Byte minterms
+ 048	Word	Cursorbreite
+ 050	Word	Cursorhöhe
+ 052	Long	--> Zeichengenerator
+ 056	Byte	Zeichensatz-Modus (fett, kursiv, etc.)
+ 057	Byte	textspezifische Flags
+ 058	Word	Höhe des Zeichensatzes
+ 060	Word	durchschnittl. Zeichenbreite
+ 062	Word	Texthöhe ohne Unterlängen
+ 064	Word	Zeichenabstand
+ 066	Long	--> User Daten
+ 070	Word	reserviert (7x)
+ 084	Long	reserviert (2x)
+ 092	Byte	reserviert (8x)

3.6.1 Ausführlicher Kommentar zur Datenstruktur Rastport

Ebenso wie bei den beiden Strukturen "Fenster" und "Screen" werden wir Ihnen auch hier Stück für Stück die Bedeutung dieses Datenblocks näherbringen:

Offset 0: Das Layer

Layer bedeutet zu deutsch "Schicht". Der Amiga benutzt Layers, um eine einzige Zeichenebene unter vielen unabhängigen Benutzern aufzuteilen. Im Grunde sind die Layers nämlich nichts anderes als die Seele eines jeden Intuition-Fensters. Obwohl man sich ihnen nur sehr widerwillig nähert, denn sie erscheinen auf den ersten Blick sehr kompliziert und vergleichsweise nutzlos (schließlich gibt es die Fenster bereits), werden wir diesem Thema an späterer Stelle ein ganzes Kapitel widmen. Bei näherer Betrachtung entpuppen sich diese Layers nämlich als wahre Grafik-Fundgrube. Doch lassen Sie sich überraschen!

Offset 4: Die Bitmap

Schon einmal haben Sie eine Bitmap kennengelernt. Sie war Bestandteil der Screen-Struktur. Dies ist ein Zeiger auf nichts anderes als diese Bitmap-Struktur. Somit können Sie indirekt die Adresse des Screens auch über den Rastport bestimmen:

```
scr&=PEEKL(WINDOW(8)+4)-184
```

GFA:

```
scr%=LPEEK(LPEEK(WINDOW(0)+50)+4)-184
```

Die Bitmap ist, wir sprachen darüber, der Knotenpunkt zwischen Datenstruktur und den RAM-Bänken, in denen der Fenster- und Screen-Inhalt gespeichert ist.

Offset 8: --> das AreaFill-Muster

Sicherlich ist Ihnen die Möglichkeit bekannt, Flächen nicht nur einfarbig, sondern auch mit Mustern auszufüllen. Doch die Mustervorgabe muß irgendwo gespeichert sein. Dieses Feld enthält die Adresse des Speicherbereiches.

Sie erhalten weitere Informationen und Beispiele im Anschluß an diesen Kommentar. Dann nämlich werden wir mit Hilfe einiger dieser Register auch das letzte in Sachen Muster aus dem Amiga herauskitzeln: Bis zu 32-farbige Multicolor-Muster!

Offset 12: Der TmpRas

TmpRas steht vermutlich für Temporäres Raster. Es handelt sich bei ihm um eine Datenstruktur, die vornehmlich aus freiem RAM besteht. Wann immer Sie Füll-Befehle wie PAINT oder LINE bf verwenden, wird diese Struktur notwendig. Sie muß in der Lage sein, das gesamte Füllobjekt aufzunehmen und zwischenzuspeichern.

Offset 16: AreaInfo

Dies ist eine Datenstruktur, die zur Polygonzeichnung verwendet wird. Für BASIC ist sie uninteressant. Wir werden sie aber später am Rande erläutern.

Offset 20: GelsInfo

"Gels" steht für Graphics Elements. Dazu gehören Sprites und Bobs (Blitter Objects), aber auch das vollautomatische Amiga Animationssystem. Bevor dieses System aktiviert werden kann, muß eine GelsInfo-Struktur geschaffen werden, die einige wichtige Parameter enthält.

Offset 24: Schreibmaske

Mit dieser Variablen lassen sich einzelne Bitplanes der Zeichenebene ausblenden. Normalerweise finden Sie hier den Wert 255, alle Bits sind gesetzt. Es werden also alle vorhandenen Bitplanes verwendet. Der POKE

```
POKE WINDOW(8)+24,0
```

sorgt dafür, daß keine Bitplane mehr aktiv ist; auf den Bildschirm wird nichts mehr gezeichnet. Entsprechend können Sie ausgewählte Bitplanes aktivieren etc.

Offset 25, 26 und 27: Zeichenfarben

Mit diesen Registern werden die Zeichenfarben bestimmt: Das erste Register enthält die Nummer des Farbregisters für die Zeichenfarbe, das zweite die des Hintergrundes, das dritte die des AreaOutline-Modus.

Offset 28: Zeichen-Modus

Amiga kennt vier grundsätzliche Zeichenmodi, mit denen Sie arbeiten können. Es sind dies die Modi

JAM 1	= 0
JAM 2	= 1
COMPLEMENT	= 2
INVERSEVID	= 4

Der normale Zeichenmodus ist JAM2. Dabei wird in die Zeichenebene mit der Vordergrundfarbe gezeichnet. Der Rest wird mit Hintergrundfarbe ausgemalt. Das folgende Beispiel macht das deutlich:

(Alle Programmbeispiele im Direktmodus eingeben!)

```
LINE (0,0)-(100,100),2,bf
LOCATE 1,1:PRINT "HALLO!"
```

Die weiße Schrift erscheint auf blauem Hintergrund. In den ursprünglich schwarzen Hintergrund wurde ein Loch geschnitten.

Anders ist es bei JAM1. Hier wird lediglich die Vordergrundfarbe benutzt, der Hintergrund bleibt unberührt:

```
LINE (0,0)-(100,100),2,bf
POKE WINDOW(8)+28,0
```

```
LOCATE 1,1:PRINT "HALLO!"
POKE WINDOW(8)+28,1
```

COMPLEMENT komplemetiert die Grafik mit dem Hintergrund: Wo früher ein Punkt gesetzt war, wird er nun gelöscht und umgekehrt:

```
LINE (0,0)-(100,100),2,bf
POKE WINDOW(8)+28,2
LINE (50,50)-(150,150),3,bf
POKE WINDOW(8)+28,1
```

INVERSEVID invertiert die Grafik: Hinter- und Vordergrundfarbe werden vertauscht. Das sieht so aus:

```
POKE WINDOW(8)+28,4
PRINT "INVERSE!"
POKE WINDOW(8)+28,1
```

Diese "Pokereien" funktionieren im Direktmodus wunderbar. Es hapert jedoch, wenn Sie versuchen, die POKEs in Programme einzubinden. Dann nämlich passiert gar nichts.

Abhilfe schafft die Routine "SetDrMd" der Grafik-Bibliothek, die den gewünschten Zeichenmodus sicher und zuverlässig einschaltet:

```
LIBRARY "graphics.library"
CALL SetDrMd(WINDOW(8),modus%)

modus%=0 - 255
```

Problemlos lassen sich verschiedene Modi miteinander kombinieren (lediglich JAM1 und JAM2 beißen sich...).

Einfacher ist es bei **GFA**: Hier gibt es den Befehl GRAPHMODE, der SetDrMd() sehr ähnlich ist: GRAPHMODE modus

Offset 29: AreaPtSz

Wann immer Sie mit Mustern arbeiten (PATTERN-Befehl), ist es nötig anzugeben, wie hoch denn das Muster sein soll. Zulässig

sind lediglich Musterhöhen in Zweierpotenz-Schritten: 1, 2, 4, 8,... In diesem Feld ist die Potenz gespeichert.

Durch eine besondere Programmierung läßt sich über dieses Register außerdem der Multicolor-Muster-Modus aktivieren, mit dem man nicht nur ein- sondern bis zu 32-farbige Muster kreieren kann. Mehr dazu im nächsten Kapitel!

Offset 30, 31 und 32: für Systembenutzung

Offset 34: Linienmuster

Nicht nur Flächen lassen sich gemustert ausfüllen, auch Linien können dergestalt gezeichnet werden. Die Technik ist einfach: 16 aufeinanderfolgende Punkte einer gedachten Linie können von Ihnen gesetzt oder gelöscht werden. Der Amiga zeichnet dann mit diesem "Beispielstück" alle anderen Linien. Nehmen wir an, Sie möchten das folgende Linienmuster erstellen:

*****.*.*****.*.

- eine gestrichpunktete Linie also. Sie ermitteln zunächst die Bitwerte dieser Linie:

```
bit&=215+214+213+212+211+29+27+26+25+24+23+21
```

Nun wird dieser Wert in dieses Register geschrieben:

```
POKEW WINDOW(8)+34,bit&
```

Ein Test:

```
LINE (10,10)-(600,10)
```

Sie sehen, es funktioniert. GFA-BASIC bietet hier den Befehl DEFLINE, der die Pokerei ersetzt:

```
OPENW 0
DEFLINE &X1100110011110000
```

```
COLOR 1,2
LINE 0,0,640,256
```

zeichnet eine gemusterte Linie diagonal über den Schirm.

Offset 36 und 38: Koordinaten des Grafik-Cursors

Diese beiden Felder sind von außergewöhnlicher Wichtigkeit. Text auf dem Amiga ist bekanntlich nichts anderes als textförmige Grafik. Demnach läßt sich Text an beliebiger Position auf dem Bildschirm verteilen. Das folgende Beispiel beweist es:

```
WHILE INKEY$=""
  x%=RND(1)*600
  y%=RND(1)*160
  POKEW WINDOW(8)+36,x%
  POKEW WINDOW(8)+38,y%
  PRINT "Commodore AMIGA!"
WEND
```

GFA hat diese Möglichkeit bereits in seinen Text-Befehl eingebaut:

```
WHILE INKEY$=""
  Text RAND(640),RAND(256),"Commodore AMIGA!"
WEND
```

produziert denselben Effekt.

Offset 40-51: minterms, interner Gebrauch

Offset 52: Der Zeichengenerator

Wie bereits in der Screen-Struktur ist auch dies ein Zeiger auf den gerade aktiven Zeichengenerator. Der Zeichengenerator bestimmt das Aussehen der Textzeichen. Wir kommen später auf ihn zurück.

Offset 56: Aktueller Text-Stil

Amiga kann Text eines Zeichensatzes in verschiedenen Arten auf den Bildschirm bringen:

normal	**= 0**
unterstrichen	**= Bit 0 gesetzt**
fett	**= Bit 1 gesetzt**
kursiv	**= Bit 2 gesetzt**

Die letzten drei Modi lassen sich selbstverständlich auch untereinander mischen.

Offset 57: Text-Flags, interner Gebrauch

Offset 58: Texthöhe

In diesem Feld ist die Höhe der augenblicklich aktiven Textzeichen gespeichert. Dadurch wird nach einem "Wagenrücklauf" errechnet, wo die nächste Zeile Text beginnt.

Es hindert Sie nichts daran, einen eigenen Zeilenabstand festzulegen. Er kann enger sein als normal:

```
POKEW WINDOW(8)+58,5
```

oder aber weiter:

```
POKEW WINDOW(8)+58,12
```

Offset 60: Zeichenbreite

Hier finden Sie die durchschnittliche Breite eines jeden Textzeichens. Da der Amiga auch Proportionalschrift unterstützt (Zeichen sind verschieden breit), kann hier nur ein Durchschnittswert geliefert werden.

Offset 62: Texthöhe ohne Unterlängen

Offset 64: Zeichenabstand

Mit Hilfe dieser Routine läßt sich der Abstand zwischen den einzelnen Buchstaben eines Textes variieren. Der Normwert in diesem Feld ist 0. Größere Werte bewirken eine gesperrte Schrift, wie das folgende Beispielprogramm deutlich macht:

```
text$="Hallo Welt!"
text%=LEN(text$)

FOR loop%=1 TO 40
  POKEW WINDOW(8)+36,280-(loop%*text%*.5)
  '(zentrieren)
  POKEW WINDOW(8)+38,90
  POKEW WINDOW(8)+64,loop%
  PRINT text$
NEXT loop%

FOR loop%=39 TO 0 STEP -1
  POKEW WINDOW(8)+36,280-(loop%*text%*.5)
  POKEW WINDOW(8)+38,90
  POKEW WINDOW(8)+64,loop%
  PRINT text$
NEXT loop%

END
```

Offset 66: User Daten

Hier wieder ein Zeiger auf mögliche Benutzerdatenblöcke

Offset 70 und folgende: reservierte Datenfelder für Aufwärtskompatibilität

3.7 Einstieg in die Grafik-Primitives

Mit dem Rastport haben wir nun eine Kontaktadresse zu der untersten softwaremäßigen Grafikebene, den Graphic Primitives.

Als erstes wollen wir das gerade gewonnene Wissen über die Möglichkeiten des Rastports nutzbringend anbringen. Hier einige Projekte:

3.7.1 Multicolor-Muster

Am Anfang dieses Buches hatten wir Ihnen den PATTERN-Befehl vorgestellt. Mit ihm lassen sich beliebige Flächen mit einem frei definierbaren Muster ausfüllen. Statt einfacher Flächen, wie sie dieses Programm beschert,

```
CIRCLE (310,100),100
PAINT (310,100),2,1
konnten also auch gemusterte Flächen erzeugt werden:
DIM area.pat%(3)
area.pat%(0)=&HFFFF
area.pat%(1)=&HCCCC
area.pat%(2)=&HCCCC
area.pat%(3)=&HFFFF

CIRCLE (310,100),100
PATTERN ,area.pat%
PAINT (310,100),3,1
```

Es funktioniert, der Kreis füllt sich in apartem Orange mit einem dezenten Lochmuster.

Einen Nachteil hatten die Muster bisher: Sie waren stets einfarbig.

Das soll nun anders werden. Wir haben ein ganzes Musterpaket für Sie zusammengestellt, mit dem Sie erstens Ihre Muster ganz einfach und bequem entwerfen und zweitens sogar Farbe ins Spiel bringen können.

Aus dem vorangegangenen Kapitel wissen Sie, daß das Muster in einem Speicherbereich abgelegt sein muß, dessen Anfangsadresse im Rastport ab Offset 8 gespeichert wird. Die Höhe des Musters wird als Potenz in das Rastport-Feld ab Offset 29 geschrieben. Das ist das gesamte Geheimnis der Patterns. Es steht uns also nichts mehr im Wege, ein eigenes kleines PATTERN-SUB zu

schreiben, mit dem sich auch der Multi-Color-Modus aktivieren läßt. Dieser wird eingeschaltet, sobald die Potenz ab Offset 29 nicht positiv, sondern negativ angegeben wird (also 256-Potenz).

Das folgende Programm verwaltet Muster ohne PATTERN-Befehl. Es besteht aus diesen sechs Unterprogrammen:

1. *InitPattern wieviele%*

   Dieses Unterprogramm initialisiert das neue PATTERN-System. Sie geben als Parameter ein, wieviel Zeilen Ihr Beispielmuster hoch sein soll.

   Die Routine berechnet anhand des Parameters den benötigten Speicherplatz und ruft GetMemory auf. Die Anfangsadresse des neuen Puffers wird in muster& zurückgeliefert.

2. *SetPat nummer%,pat$*

   Mit diesem neuen Befehl können Sie die einzelnen Zeilen Ihres Beispielmusters in einfacher binärer Darstellung angeben: Ein A entspricht einem ungesetzten, ein B einem gesetzten Punkt. Die nummer% gibt die Nummer der Zeile Ihres Beispielmusters an, für die die nachfolgende Definition gelten soll. pat$ muß immer 16 Zeichen enthalten. Für eine ununterbrochene Linie sieht pat$ zum Beispiel so aus:

   ```
   "BBBBBBBBBBBBBBBB"
   ```

3. *MonoPattern*

   Diese Routine wird ohne Argument aufgerufen und aktiviert das Mustersystem. Intern werden die beiden Rastport-Adressen mit den korrekten Werten initialisiert.

4. *EndPattern*

   Auch diese Routine verlangt kein Argument. Durch sie wird dem Amiga mitgeteilt, daß Ihr Beispielmuster nun nicht mehr benutzt werden soll. Außerdem wird der Spei-

cherplatz wieder freigegeben, der durch das Beispielmuster belegt worden war. Sie sollten EndPattern am Ende Ihres Programms aufrufen, um dem System wirklich allen Speicherplatz zurückzugeben.

5. *GetMemory size&*

Dies ist eine Allround-Speicher-Besorg-Routine. Wann immer Sie Speicherplatz benötigen - GetMemory besorgt es. Sie weisen einer &-Variablen lediglich die Größe des gewünschten Speicherplatzes in Bytes zu und rufen mit ihr als Argument diese Routine auf. Sie bekommen in derselben Variablen die Anfangsadresse des Speicherstückes zurückgeliefert.

Ein 1245 Bytes großes Speicherstück bekommt man beispielsweise so:

```
DECLARE FUNCTION AllocMem& LIBRARY
LIBRARY "exec.library"
myMem&=1245
GetMemory myMem&
PRINT "Anfangsadresse: ";myMem&
```

Achtung: Erhalten Sie als Adresse 0 zurück, so hat etwas nicht geklappt. Sie haben dann keinen Speicher erhalten und dürfen diesen auch nicht mit FreeMemory zurückgeben.

6. *FreeMemory add&*

Wenn Sie Ihren so erlangten Speicherplatz nicht mehr brauchen, sollten Sie ihn via FreeMemory zurück ans System geben. Ein vollständiger Aufruf sieht so aus:

```
DECLARE FUNCTION AllocMem& LIBRARY
LIBRARY "exec.library"

speicher&=100  '100 Bytes, bitte!
GetMemory speicher&

(...)
FreeMemory speicher&  'wieder freigeben
```

```
LIBRARY CLOSE
```

Hier nun die angekündigten Unterprogramme (ColorPattern entspricht Monopattern, aktiviert jedoch den MC-Modus). Die folgenden Farben stehen zur Verfügung:

Farbe 1 A	**Farbregister 0 (Hintergrund)**
Farbe 2 B	**Farbregister 1**
Farbe 3 C	**Farbregister 2**
Farbe 4 D	**Farbregister 3**

(für den Workbench-Screen)

```
'######################################
'#
'# Programm: Mono-Color-Pattern
'# Datum: 27.12.86
'# Autor: tob
'# Version: 1.0
'#
'######################################

' Ermoeglicht mehrfarbige "Multi-Color"
' Muster (via Rastport Manipulation)

PRINT "Suche die .bmap-Dateien..."

'EXEC-Bibliothek
DECLARE FUNCTION AllocMem& LIBRARY
'FreeMem()

LIBRARY "exec.library"


init:       CLS
            zeilen%=8
            InitPattern zeilen%
            '          0123456789ABCDEF
            SetPat 0,"DCDAAAAAAABBAAAA"
            SetPat 1,"DCDAAAAAABBBBAAA"
            SetPat 2,"DCDAAAAABAABBAAA"
            SetPat 3,"DCDAAAABAAAABBAA"
            SetPat 4,"DCDAAABBBBBBBBBA"
            SetPat 5,"DCDAABAAAAAAABBA"
            SetPat 6,"DCDBBBBAAAAABBBB"
            SetPat 7,"CCCCCCCCCCCCCCCC"

zeichnen:   PRINT TAB(5);"MONO";TAB(40);"COLOR!"
```

```
            MonoPattern
            CIRCLE (60,60),60
            PAINT  (60,60),farben%,1

            ColorPattern
            CIRCLE (310,100),100
            PAINT  (310,100),farben%,1

ende:       EndPattern
            LIBRARY CLOSE
            END


SUB ColorPattern STATIC
      SHARED muster&,plane2%
      planes% = LOG(plane2%)/LOG(2)
      POKEL WINDOW(8)+8,muster&
      POKE  WINDOW(8)+29,256-planes%
END SUB

SUB InitPattern(wieviele%) STATIC
      SHARED muster&,plane1%,plane2%,farben%

     '* Na, ist das eine Zweierpotenz...?
      IF LOG(wieviele%)/LOG(2)<>INT(LOG(wieviele%)/LOG(2)) THEN
        PRINT "2^x! Eine Zweierpotenz fuer InitPattern! 1,2,4,8,16..."
        ERROR 17
      END IF

     '* Parameter auslesen
      planes% = PEEK(PEEKL(WINDOW(8)+4)+5)
      DIM SHARED p&(wieviele%*planes%)

      plane1% = planes%
      plane2% = wieviele%
      farben% = 2^plane1%-1

     '* Definitionsmuster-Buffer besorgen
      muster& = wieviele%*2*planes%
      GetMemory muster&
END SUB

SUB SetPat(nummer%,pat$) STATIC
      SHARED muster&,plane1%,plane2%

     '* Zu viele Zeilen?!
      IF nummer%>=plane2% THEN
        PRINT "Mehr Zeilen als mit InitPattern definiert!"
        EndPattern
        ERROR 17
      END IF
```

```
      '* Error-Handling: String auf 16 Bytes stutzen
       IF LEN(pat$)<16 THEN
         pat$=pat$+STRING$(16-LEN(pat$),"A")
       END IF

      '* Das Definitionspattern auslesen
       FOR loop1% = 0 TO 15
         check$ = UCASE$(MID$(pat$,loop1%+1,1))
         col%   = ASC(check$)-65
         IF col%>=2^plane1% OR col%<0 THEN col%=0
         FOR loop2% = col% TO 0 STEP -1
           IF col%> = 2^loop2% THEN
             col%                          = col%-2^loop2%
             p&(nummer%+loop2%*plane2%) = p&(nummer%+loop2%*plane2%)+
                                           2^(15-loop1%)
           END IF
         NEXT loop2%
       NEXT loop1%

      '* Werte in Buffer schreiben
       FOR loop3% = 0 TO plane2%*plane1%
         POKEW muster&+2*loop3%,p&(loop3%)
       NEXT loop3%
END SUB

SUB MonoPattern STATIC
       SHARED muster&,plane2%
       planes% = LOG(plane2%)/LOG(2)
       POKEL WINDOW(8)+8, muster&
       POKE  WINDOW(8)+29,planes%
END SUB

SUB EndPattern STATIC
       SHARED muster&

      '* Pattern aus und Memory freigeben
       POKEL WINDOW(8)+8, 0
       POKE  WINDOW(8)+29,0
       FreeMemory muster&
END SUB

SUB GetMemory(size&) STATIC
         mem.opt&  = 2^0+2^1+2^16
         RealSize& = size&+4
         size&     = AllocMem&(RealSize&,opt&)
         IF size&  = 0 THEN ERROR 255
         POKEL   size&,RealSize&
         size& = size&+4
END SUB

SUB FreeMemory(add&) STATIC
```

```
        add&      = add&-4
        RealSize& = PEEKL(add&)
        CALL FreeMem(add&,RealSize&)
END SUB
```

Reichen Ihnen die vier möglichen Farben des Workbench-Screens nicht aus, dann haben Sie die Möglichkeit, einen eigenen Screen mit mehr als 2 Bitplanes Tiefe einzurichten. Das folgende Programm tut genau dies. Nach der Initialisierung erscheint ein Ihnen sicherlich nicht unbekanntes 11-farbiges Logo als Füllmuster. Mit der Maus können Sie zudem mit dem Muster malen, wenn Sie die linke Maustaste gedrückt halten:

```
'#######################################
'#
'# Programm: Multi-Color-Pattern
'# Datum: 27.12.86
'# Autor: tob
'# Version: 1.0
'#
'#######################################

'Demonstriert die Verwendung eines Multi-Color-Patterns mit
'bis zu 16 Farben (Screen-Tiefe = 4); bis zu 32 Farben moeg-
'lich (farbwerte: A=0 bis Z=25, Farben 26-32 = chr$(91)-chr$(97) )

'bei OUT OF HEAP SPACE andere Fenster schliessen!

PRINT "Suche die .bmap-Dateien..."

'EXEC-Bibliothek
DECLARE FUNCTION AllocMem& LIBRARY
'FreeMem()

LIBRARY "exec.library"


init:       SCREEN 1,640,200,4,2
            WINDOW 1,"Hallo!",,,1

            LOCATE 4,15
            PRINT "*** Geduld! ***"

            zeilen%=8
            InitPattern zeilen%
            '          0123456789ABCDEF
            SetPat 0,"AAAAAAAAAAABBABB"
            SetPat 1,"AAAAAAAAAABBABBA"
            SetPat 2,"AAAAAAAAACCACCAA"
```

```
            SetPat 3,"AAAAAAAADDADDAAA"
            SetPat 4,"FFAFFAAEEAEEAAAA"
            SetPat 5,"AGGAGGHHAHHAAAAA"
            SetPat 6,"AAKKAIIAIIAAAAAA"
            SetPat 7,"AAAJJJJJJAAAAAAA"

farben:     '* Farbauswahl fuer das Muster
            PALETTE 0,0,0,0     'A
            PALETTE 1,.9,.3,.4  'B
            PALETTE 2,.8,.5,.4  'C
            PALETTE 3,.8,.6,0   'D
            PALETTE 4,1,.8,0    'E
            PALETTE 5,0,0,.6    'F
            PALETTE 6,0,.3,.6   'G
            PALETTE 7,.7,.9,0   'H
            PALETTE 8,.3,.9,0   'I
            PALETTE 9,0,.5,0    'J
            PALETTE 10,0,.3,0!  'K

zeichnen:   ColorPattern
            LOCATE 3,10

            PRINT "Linke Maus-Taste druecken!"
            PRINT TAB(10);"Linken Screen-Rand beruehren = ENDE!"
            CIRCLE (310,100),100
            PAINT  (310,100),farben%,1

mausContr:  test% = MOUSE(0)
            WHILE MOUSE(1)<>0
              x%  = MOUSE(1)
              y%  = MOUSE(2)
              IF test%<>0 THEN
                LINE (x%,y%)-(x%+10,y%+5),farben%,bf
              END IF
              test% = MOUSE(0)
            WEND

ende:       EndPattern

            WINDOW 1,"Demo beendet.",,,-1

            SCREEN CLOSE 1
            LIBRARY CLOSE
            END


SUB ColorPattern STATIC
      SHARED muster&,plane2%
      planes% = LOG(plane2%)/LOG(2)
      POKEL WINDOW(8)+8,muster&
      POKE  WINDOW(8)+29,256-planes%
```

```
END SUB

SUB InitPattern(wieviele%) STATIC
      SHARED muster&,plane1%,plane2%,farben%

     '* Na, ist das eine Zweierpotenz...?
      IF LOG(wieviele%)/LOG(2)<>INT(LOG(wieviele%)/LOG(2)) THEN
        PRINT "2^x! Eine Zweierpotenz fuer InitPattern! 1,2,4,8,16..."
        ERROR 17
      END IF

     '* Parameter auslesen
      planes% = PEEK(PEEKL(WINDOW(8)+4)+5)
      DIM SHARED p&(wieviele%*planes%)

      plane1% = planes%
      plane2% = wieviele%
      farben% = 2^plane1%-1

     '* Definitionsmuster-Buffer besorgen
      muster& = wieviele%*2*planes%
      GetMemory muster&
END SUB

SUB SetPat(nummer%,pat$) STATIC
      SHARED muster&,plane1%,plane2%

     '* Zu viele Zeilen?!
      IF nummer%>=plane2% THEN
        PRINT "Mehr Zeilen als mit InitPattern definiert!"
        EndPattern
        ERROR 17
      END IF

     '* Error-Handling: String auf 16 Bytes stutzen
      IF LEN(pat$)<16 THEN
        pat$=pat$+STRING$(16-LEN(pat$),"A")
      END IF

     '* Das Definitionspattern auslesen
      FOR loop1% = 0 TO 15
        check$ = UCASE$(MID$(pat$,loop1%+1,1))
        col%   = ASC(check$)-65
        IF col%>=2^plane1% OR col%<0 THEN col%=0
        FOR loop2% = col% TO 0 STEP -1
          IF col%> = 2^loop2% THEN
            col%                          = col%-2^loop2%
            p&(nummer%+loop2%*plane2%) = p&(nummer%+loop2%*plane2%)+2^
(15-loop1%)
          END IF
        NEXT loop2%
      NEXT loop1%
```

```
      '* Werte in Buffer schreiben
       FOR loop3% = 0 TO plane2%*plane1%
         POKEW muster&+2*loop3%,p&(loop3%)
       NEXT loop3%
END SUB

SUB MonoPattern STATIC
       SHARED muster&,plane2%
       planes% = LOG(plane2%)/LOG(2)
       POKEL WINDOW(8)+8, muster&
       POKE  WINDOW(8)+29,planes%
END SUB

SUB EndPattern STATIC
       SHARED muster&

      '* Pattern aus und Memory freigeben
       POKEL WINDOW(8)+8, 0
       POKE  WINDOW(8)+29,0
       FreeMemory muster&
END SUB

SUB GetMemory(size&) STATIC
         mem.opt&  = 2^0+2^1+2^16
         RealSize& = size&+4
         size&     = AllocMem&(RealSize&,opt&)
         IF size&  = 0 THEN ERROR 255
         POKEL   size&,RealSize&
         size& = size&+4
END SUB

SUB FreeMemory(add&) STATIC
         add&      = add&-4
         RealSize& = PEEKL(add&)
         CALL FreeMem(add&,RealSize&)
END SUB
```

GFA-BASIC unterstützt ebenfalls die Arbeit mit selbstdefinierten Mustern. Neben einer ganzen Reihe bereits vordefinierter Musterungen können Sie Ihre ganz eigenen Muster mit dem DEFFILL-Kommando definieren. Das besondere: Der Multicolormodus ist bereits implementiert und läßt sich durch einen kleinen Trick mitbenutzen. Hier ein Demoprogramm:

```
' arbeiten mit selbstdef. mustern
OPTION BASE 1
basis=4
DIM a(basis)
```

```
DIM a$(basis)
'
OPENW 0
GRAPHMODE 1
a(1)=&X1111111100000000
a(2)=&X1111111100000000
a(3)=&X111111110000
a(4)=&X111111110000
'
change
'
' hier das  v   Ausrufungszeichen entfernen!!!
DEFFILL 2,g$!+CHR$(1)
'
r%=LPEEK(WINDOW(0)+50)
md=PEEK(r%+29)
a%=LPEEK(r%+8)
FOR loop=0 TO basis-1
  PRINT "Feld: ";a(loop+1),"Rastport: ";DPEEK(a%+2*loop)
NEXT loop
PRINT " Modus: ";md
STOP
PBOX 0,0,400,200
WHILE INKEY$=""
WEND
END
'
PROCEDURE change
  nr=DIM?(a())
  FOR loop=1 TO nr
    a$(loop)=MKI$(a(loop))
  NEXT loop
  g$=""
  FOR loop=1 TO nr
    g$=g$+a$(loop)
  NEXT loop
RETURN
```

In der vorliegenden Fassung produziert das Programm einfarbige Muster. Wenn Sie jedoch das Ausrufungszeichen an der angegebenen Stelle entfernen, kommen Sie auf einfache Weise ebenfalls in den Genuß vielfarbiger Muster. Sie aktivieren den MC-Modus bei GFA nämlich, indem Sie an den Definitionsstring des DEFFILL-Kommandos ein beliebiges Byte anhängen.

Ist der MC-Modus aktiviert, so müssen Sie Ihr Muster ebenenweise definieren. Soll (wie in diesem Beispiel) ein zwei Zeilen hohes Muster entstehen, benötigen Sie somit vier Definitionszei-

len (denn der Workbench-Screen umfaßt zwei Bitplanes). Zunächst definieren also zwei Zeilen die erste Ebene, dann weitere zwei Zeilen die zweite Ebene. Entsprechend stehen die so möglichen Kombinationen 0|0, 0|1, 1|0 und 1|1 für die vier möglichen Farben.

3.7.2 Mit Cursorpositionierung Schattendruck

Die mehrfarbigen Muster aus dem vorangegangenen Absatz wurden ausschließlich durch geschickte Manipulation des Rastports realisiert. Es steckt aber noch viel mehr in dieser Datenstruktur, lediglich ein bißchen Kreativität ist nötig. Als Anregung wollen wir Ihnen einmal zeigen, was sich mit Hilfe der Offsetfelder 28, 36 und 38 bewerkstelligen läßt.

Bei diesen Feldern handelt es sich um:

Offset	Typ	Bezeichnung	
+ 028	Byte	Zeichen-Modus	
		JAM1	= 0
		JAM2	= 1
		COMPLEMENT	= 2
		INVERSEVID	= 4
+ 036	Word	X-Koordinate des Grafik-Cursors	
+ 038	Word	Y-Koordinate des Grafik-Cursors	

Wir wollen nun mit ihrer Hilfe eine schattierte Textausgabe realisieren. Diese Methode ist bei Fernsehgesellschaften seit langer Zeit eingeführt und funktioniert so: Text wird in schwarzer Farbe ausgegeben. Anschließend wird derselbe Text um einige Bildschirmpixel verschoben in weiß darübergedruckt. Es ergibt sich der Schatten-Effekt, der Text besonders lesbar macht, denn ob der Hintergrund dunkel oder hell ist - es spielt keine Rolle; der Kontrast ist sichtbar.

Zur Verwirklichung unserer Routine werden wir drei Routinen der Grafik-Bibliothek einsetzen:

SetDrMd()
Text()
Move()

Die erste Routine setzt den Zeichen-Modus und beeinflußt somit direkt Rastport-Offset 28 (siehe Kapitel 3.6.1). Der Text-Befehl wurde bereits in Kapitel 2 behandelt: Er gibt Text auf den Bildschirm aus. Das Move-Kommando schließlich setzt den Grafik-Cursor auf eine beliebige Position. Dazu beeinflußt diese Routine direkt die Rastport-Offsets 36 und 38. Statt des Move-Befehls könnten Sie auch direkt in die Speicherstellen poken.

Unsere Routine soll "Schatten" heißen. Sie verlangt zwei Argumente:

```
Schatten text$,mode%

text$:      Der Text, der ausgegeben werden soll
mode%:      0 = PRINT text$
            1 = PRINT text$;
```

Hier zunächst das Programm:

```
'#######################################
'#
'# Programm: Schatten-Druck
'# Datum: 25.12.86
'# Autor: tob
'# Version: 1.0
'#
'#######################################

PRINT "Suche die .bmap-Datei..."

'GRAPHICS-Bibliothek
'Text()
'Move()
'SetDrMd()

LIBRARY "graphics.library"

main:       '* Kontrastfarben
            PALETTE 0,.5,.5,.5
            CLS
            LOCATE 5,1
```

```
            PRINT    "Dies ist der langweilige und kontrastarme"
            PRINT    "Normaldruck. Nicht sehr hervorstechend..."
            PRINT
            Schatten "Schatten-Print ist genauso schnell wie PRINT!",0
            Schatten "Das klappt nur durch konsequenten Einsatz der",0
            Schatten "Text()-Funktion aus der Grafik-Bibliothek!",0
            PRINT
            Schatten "Der Text scheint effektreich VOR DEM SCHIRM zu",0
            Schatten "schweben.",0

ende:       LIBRARY CLOSE
            END

SUB Schatten(Text$,mode%) STATIC
      '* Parameter festlegen
       textlen% = LEN(Text$)
       tiefe%   = 2
       cX%      = PEEKW(WINDOW(8)+36)
       cY%      = PEEKW(WINDOW(8)+38)

      '* Schatten zeichnen
       COLOR 2,0
       CALL  Move(WINDOW(8),cX%+tiefe%,cY%+tiefe%)
       CALL  Text(WINDOW(8),SADD(Text$),textlen%)

      '* JAM1 und Vordergrund zeichnen
       CALL  SetDrMd(WINDOW(8),0)
       COLOR 1,0
       CALL  Move(WINDOW(8),cX%,cY%)
       CALL  Text(WINDOW(8),SADD(Text$),textlen%)

      '* CR nach Bedarf
       IF mode% = 0 THEN
         PRINT
       END IF

      '* und wieder JAM2 und fertig!
       CALL SetDrMd(WINDOW(8),1)
END SUB
```

Während des Zeichenprozesses ist es nötig, den Zeichenmodus von JAM2 auf JAM1 umzuschalten, denn sonst würde der weiße Text den schwarzen völlig auslöschen.

Wir benutzten in diesem Programm die Text-Funktion an Stelle des PRINT-Befehls, weil es hier auf Geschwindigkeit ankommt. Text ist mehr als dreimal so schnell wie PRINT. Damit ist un-

sere Schatten-Textausgabe schneller als ein normales PRINT-Kommando! Wenn Sie den Unterschied einmal "sehen" wollen, dann tauschen Sie die Zeile:

```
CALL Text(WINDOW(8),SADD(text$),textlen%)
```

gegen die Zeile

```
PRINT text$
```

aus. Der Unterschied ist enorm. Hier nun dasselbe in GFA-BASIC:

```
schatten(100,40,"Hallo liebe Welt!")
'
PROCEDURE schatten(x,y,text$)
  GRAPHMODE 0
  COLOR 3
  TEXT x+1,y+1,text$
  COLOR 1
  TEXT x,y,text$
  GRAPHMODE 1
RETURN
```

Der enorme Befehlsumfang macht eine wesentlich effizientere Programmierung möglich - wir benötigen nicht eine Systemroutine!

3.7.3 Outline-Druck - der besondere Flair

Wenn man ihn sieht, denkt man zunächst an komplizierte Algorithmen und Maschinensprache - die Rede ist vom "Outline"-Druck. Hierbei wird nur die Silhouette des Textes gedruckt. Diese Schriftart fällt sofort ins Auge und eignet sich besonders gut für Überschriften.

Realisiert wird dieser Modus durch folgende Technik: Im Zeichenmodus JAM1 wird der auszugebende Text in alle Himmelsrichtungen jeweils um einen Pixel verschoben ausgegeben. Das Ergebnis ist ein "verschmierter" Textausdruck. Nun wird an die

Originalposition mit der Hintergrundfarbe der Text geschrieben. Man erhält den Outline-Effekt.

Hier unsere Routine namens Outline. Sie entspricht im wesentlichen der vorangegangenen Schattenroutine:

```
'#######################################
'#
'# Programm: Outline-Druck
'# Datum: 25.12.86
'# Autor: tob
'# Version: 1.0
'#
'#######################################

PRINT "Suche die .bmap-Datei..."

'GRAPHICS-Bibliothek
'Text()
'Move()
'SetDrMd()

LIBRARY "graphics.library"

main:       CLS
            LOCATE 8,1
            Outline "  Outline-Print hat ein wahrhaft ins Auge
                       stechendes Aeusseres!",0
            Outline "  Trotz eines sehr aufwendigen Zeichenprozesses
                       ist die Outline-Routine",0
            Outline "  aeusserst schnell durch konsequenten Einsatz der
                       Text()-Funktion.",0
            Outline "  OUTLINE funktioniert natuerlich auch mit anderen
                       Zeichensaetzen...!",0

ende:       LIBRARY CLOSE
            END

SUB Outline(text$,mode%) STATIC
      '* Parameter
       textlen% = LEN(text$)
       cX%      = PEEKW(WINDOW(8)+36)
       cY%      = PEEKW(WINDOW(8)+38)

      '* JAM1 und Text verschmieren
      '* eine Schleife macht's schneller und unuebersichtlicher
       CALL SetDrMd(WINDOW(8),0)
       CALL Move(WINDOW(8),cX%+1,cY%)
       CALL text(WINDOW(8),SADD(text$),textlen%)
       CALL Move(WINDOW(8),cX%-1,cY%)
```

```
        CALL text(WINDOW(8),SADD(text$),textlen%)
        CALL Move(WINDOW(8),cX%,cY%+1)
        CALL text(WINDOW(8),SADD(text$),textlen%)
        CALL Move(WINDOW(8),cX%,cY%-1)
        CALL text(WINDOW(8),SADD(text$),textlen%)
        CALL Move(WINDOW(8),cX%-1,cY%-1)
        CALL text(WINDOW(8),SADD(text$),textlen%)
        CALL Move(WINDOW(8),cX%+1,cY%+1)
        CALL text(WINDOW(8),SADD(text$),textlen%)
        CALL Move(WINDOW(8),cX%+1,cY%-1)
        CALL text(WINDOW(8),SADD(text$),textlen%)
        CALL Move(WINDOW(8),cX%-1,cY%+1)
        CALL text(WINDOW(8),SADD(text$),textlen%)

       '* Hintergrundfarbe und Text lochen
        COLOR 0,0
        CALL  Move(WINDOW(8),cX%,cY%)
        CALL  text(WINDOW(8),SADD(text$),textlen%)

       '* Reset Modes und Farbe, CR nach Bedarf
        COLOR 1,0
        IF mode%=0 THEN
          PRINT
        END IF
        CALL SetDrMd(WINDOW(8),1)
END SUB
```

Und wieder das Beispiel in GFA:

```
WHILE INKEY$=""
  outline(RAND(640),RAND(256),"Hallo liebe Welt!")
WEND
'
PROCEDURE outline(x,y,text$)
  GRAPHMODE 0
  COLOR RAND(64)
  FOR loop1=-1 TO 1
    FOR loop2=-1 TO 1
      TEXT x+loop1,y+loop2,text$
    NEXT loop2
  NEXT loop1
  GRAPHMODE 2
  TEXT x,y,text$
  GRAPHMODE 1
RETURN
```

3.7.4 Softwaremäßige Schriftmodi

Die Textausgabe des Amiga läßt sich programmieren: vier verschiedene Modi gibt es. Wir sprechen von dem Rastport-Offset 56. In Abhängigkeit von seinem Inhalt stellt der Amiga Text

a) normal
b) fett
c) unterstrichen
d) kursiv

dar. Außerdem lassen sich mehrere Modi miteinander kombinieren. Es gibt grundsätzlich zwei Möglichkeiten, zwischen den Modi umzuschalten:

a) Direkte Rastport-Manipulation

Bei dieser Methode wird der Modus durch direktes POKEn in den Rastport umgeschaltet. Das funktioniert so:

```
normal%=0
unterstrichen%=2^0
fett%=2^1
kursiv%=2^2
[bei GFA nach OPENW 0 anstatt WINDOW(8): LPEEK(WINDOW(0)+50) ]

POKE WINDOW(8)+56,unterstrichen%
PRINT "Unterstrichener Text!"

POKE WINDOW(8)+56,fett%
PRINT "Fettdruck."

POKE WINDOW(8)+56,kursiv%+unterstrichen%
PRINT "Kombiniert: Kursiv und unterstrichen"

POKE WINDOW(8)+56,normal%
```

b) Via Grafik-Bibliothek

In dieser Bibliothek gibt es zwei Funktionen, die mit dieser Thematik zu tun haben:

```
AskSoftStyle()
```

und

```
SetSoftStyle()
```

Das Prinzip ist ähnlich: Wieder stehen die vier Grundtypen zur Verfügung, wieder lassen sie sich mischen. Der Aufruf des SetSoftStyle-Befehls besitzt jedoch ein drittes Feld:

```
newStyle%=SetSoftStyle%(rastport,modus,enable)
```

rastport:	Adresse des Rastports
modus:	Der gewünschte Stil
enable:	Die zur Verfügung stehenden Modi

Es kann vorkommen, daß ein Zeichensatz sich mit einem bestimmten SoftStyle nicht verträgt und unleserlich wird. Deshalb hat die Grafik-Bibliothek das Enable-Feld ins Spiel gebracht. Die AskSoftStyle-Funktion erfragt eine Maske, die alle legalen Typen des augenblicklichen Zeichensatzes zurückliefert. Dieser Wert wird dann SetSoftStyle übergeben. Hier ein Programmbeispiel:

```
'########################################
'#
'# Programm: SoftStyle
'# Datum: 20.12.86
'# Autor: tob
'# Version: 1.0
'#
'########################################
' Demonstriert die Verwendung verschiedener Schrift-
' arten, sogenannter "SoftStyles", die softwarege-
' steuert, also algorithmisch zustande kommen.

PRINT "Suche das .bmap-File..."

'GRAPHICS-Bibliothek
DECLARE FUNCTION AskSoftStyle& LIBRARY
DECLARE FUNCTION SetSoftStyle& LIBRARY
'SetDrMd()

LIBRARY "graphics.library"

init:     normal        = 0
          unterstrichen = 2^0
          fett          = 2^1
          kursiv        = 2^2
          CLS
```

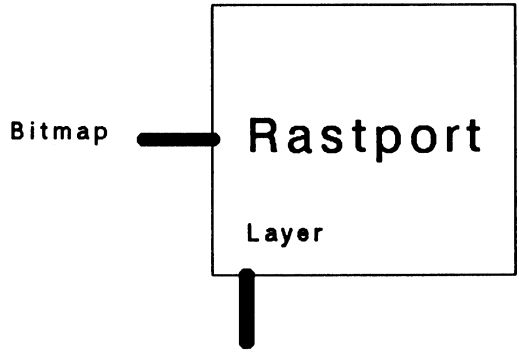
```
main:    '* JAM1 fuer lesbare Schraegschrift
         CALL   SetDrMd(WINDOW(8),0)
         LOCATE 4,1
         SetStyle unterstrichen+fett
         PRINT TAB(8);"ALGORITHMISCH GENERIERTE SCHRIFTARTEN"
         PRINT
         SetStyle normal
         PRINT "Amiga kann eine Menge machen mit den bestehenden
               Zeichensaetzen."
         PRINT "Ohne eine ";
         SetStyle unterstrichen
         PRINT "Definitionsaenderung";
         SetStyle normal
         PRINT " lassen sich folgende Aenderungen vornehmen:"
         PRINT
         SetStyle fett
         PRINT "FETT-Druck"
         SetStyle kursiv
         PRINT "SCHRAEG-Druck"
         SetStyle unterstrichen
         PRINT "UNTERSTRICHENER Text"
         SetStyle unterstrichen+kursiv
         PRINT "und GEMISCHT."
         PRINT
         SetStyle normal
         PRINT "Damit lassen sich Texte wesentlich professioneller
               gestalten."
         CALL SetDrMd(WINDOW(8),1)

LIBRARY CLOSE
END

SUB SetStyle(mode) STATIC
  '0!   = normal
  '2^0  = unterstrichen
  '2^1  = fett
  '2^2  = kursiv
  mode% = CINT(mode)

 '* Font kontrollieren und wenn moeglich verformen
  enable%   = AskSoftStyle&(WINDOW(8))
  newStyle% = SetSoftStyle&(WINDOW(8),mode%,enable%)

END SUB
```

Vielleicht ist Ihnen etwas aufgefallen: Der Kursivdruck des ersten Beispiels sah etwas unförmig aus. Bei diesem Programm erschien er jedoch leserlich. Das Geheimnis ist schnell gelüftet: Der Kursivdruck funktioniert nur im Zeichenmodus JAM1 korrekt, denn durch die Schrägstellung der Zeichen fällt jeweils der

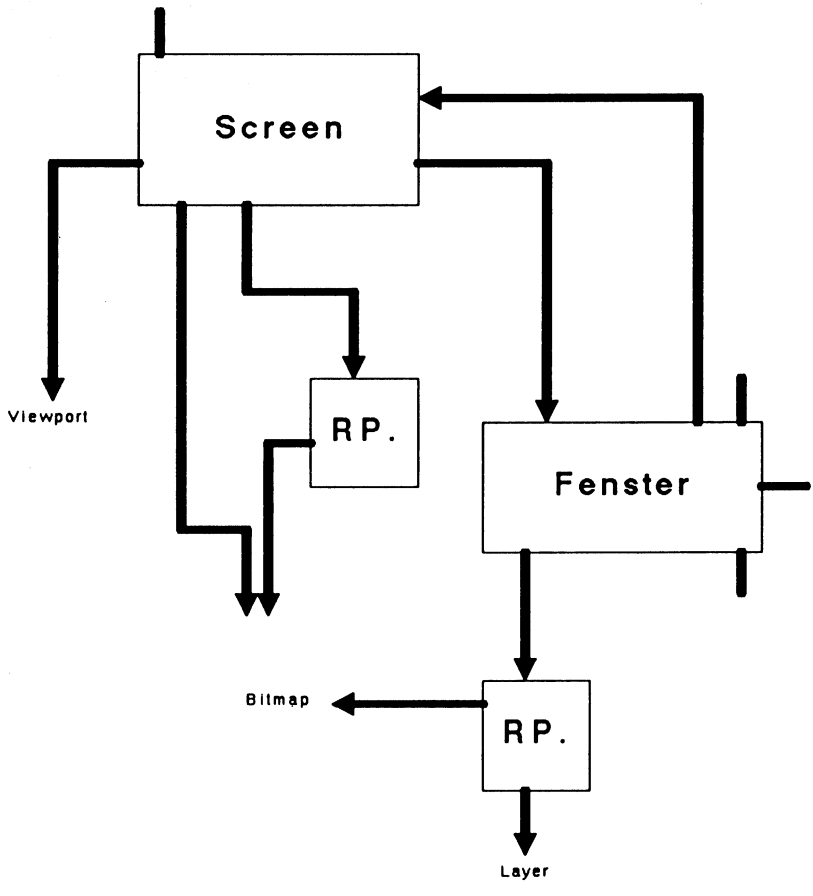
rechte Teil in den Einflußbereich des nächsten Buchstabens. Ist dann der normale Modus JAM2 aktiv, wird dieser Teil durch die Hintergrundfarbe überdeckt, und die Zeichen erscheinen abgehackt.

3.8 Der Rastport als Teil des Grafik-Betriebssystems

Sie haben nun Ihre ersten Erfahrungen mit dem Rastport gemacht und sollten einen ungefähren Einblick in seine Möglichkeiten bekommen haben. Wieder ist es an der Zeit, einen Blick auf das Gesamtkonzept des Amiga zu werfen. In Kapitel 3.5 hatten wir noch erhebliche Schwierigkeiten, ein vollständiges Bild des Systems zu bekommen. Jetzt gesellt sich der Rastport als Vertrauter zu Fenster und Screen:



Damit läßt sich das System schon wesentlich besser darstellen. Wir setzen den Rastport in die Zeichnung aus Kapitel 3.5:



Noch immer fehlen Informationen über Viewport und Bitmap. Es ist sogar noch ein weiterer Unbekannter hinzugekommen: ein Layer. Aber das Bild, das sich uns vom Amiga-System bietet, wird immer schärfer.

Als nächstes werden wir uns die Bitmap-Datenstruktur vornehmen.

3.9 Die Bitmap-Struktur

Mit dieser Struktur erhalten wir Kontakt zu den RAM-Bänken, in denen der Screen-Inhalt abgespeichert ist. Es handelt sich um eine 40 Bytes große Datenstruktur:

Datenstruktur "Bitmap"/graphics/40 Bytes

Offset	Typ	Bezeichnung
+ 000	Word	Bytes pro Display-Zeile
+ 002	Word	Anzahl der Display-Zeilen
+ 004	Byte	System-Flag (bzw. unbenutzt)
+ 005	Byte	Anzahl der Bitplanes ("Tiefe")
+ 006	Word	unbenutzt
+ 008	Long	Zeiger auf 1. Bitplane
+ 012	Long	Zeiger auf 2. Bitplane
+ 016	Long	Zeiger auf 3. Bitplane
+ 020	Long	Zeiger auf 4. Bitplane
+ 024	Long	Zeiger auf 5. Bitplane
+ 028	Long	Zeiger auf 6. Bitplane
+ 032	Long	Zeiger auf 7. Bitplane
+ 036	Long	Zeiger auf 8. Bitplane

Die Anfangsadresse dieser Struktur läßt sich beispielsweise so ermitteln:

```
bitmap&=PEEKL(WINDOW(8)+4)
```

GFA:

```
OPENW 0
bitmap%=LPEEK(LPEEK(WINDOW(0)+50)+4)
```

Die ersten beiden Felder enthalten die Abmessungen des Displays, das die Bitplanes speichern:

```
bitmap&=PEEKL(WINDOW(8)+4)   'GFA: s.o.
x%=PEEKW(bitmap&)*8
y%=PEEKW(bitmap&+2)
PRINT "Ausdehnung: horiz. ";x%;

PRINT "vert. ";y%
```

Das vierte Feld enthält die Anzahl der benutzten Bitplanes. Im Augenblick lassen sich bis zu 6 Bitplanes aktivieren, die Zeiger auf die 7. und 8. Bitplane existieren wegen der Aufwärtskompatibilität zu späteren Amigas.

4. Der Viewport

Fast haben wir die Grafik-Hardware des Amiga nun erreicht. Der Viewport repräsentiert das elementarste Display des Amiga. Dabei handelt es sich um einen Datenblock von 40 Bytes:

Datenstruktur "Viewport"/graphics/40 Bytes

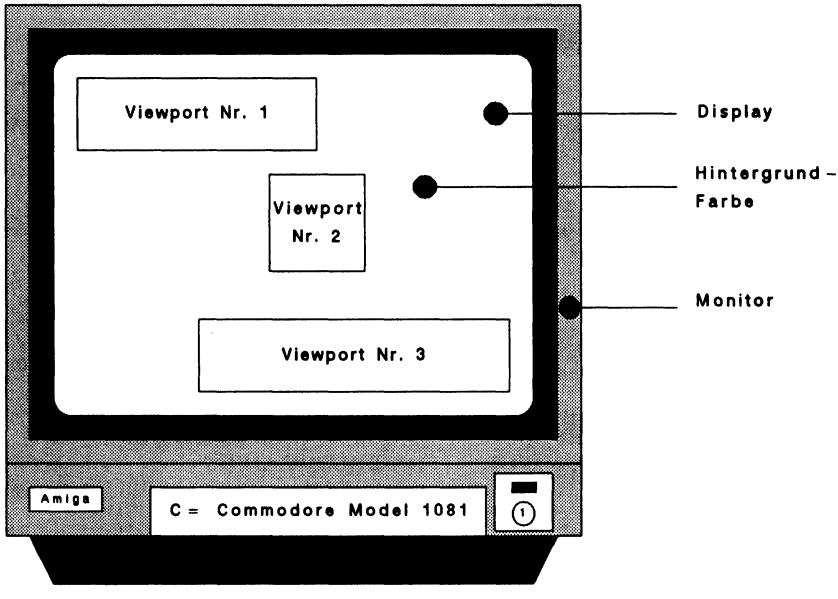
Offset	Type	Bezeichnung
+ 000	Long	Zeiger auf nächsten Viewport
+ 004	Long	Zeiger auf ColorMap
+ 008	Long	DspIns: Copper-Liste von MakeView
+ 012	Long	SprIns: Copper-Liste für Sprites
+ 016	Long	ClrIns: Copper-Liste für Sprites
+ 020	Long	UCopIns: User Copper-Liste
+ 024	Word	Breite des Displays
+ 026	Word	Höhe des Displays
+ 028	Word	X-Offset von (0,0) der Bitplanes
+ 030	Word	Y-Offset von (0,0) der Bitplanes
+ 032	Word	Viewport-Modi
		Bit 1: 1=GENLOCK VIDEO
		Bit 2: 1=INTERLACE
		Bit 6: 1=PFBA
		Bit 7: 1=EXTRA HALFBRITE
		Bit 8: 1=GENLOCK AUDIO
		Bit 10: 1=DUALPF
		Bit 11: 1=HAM
		Bit 13: 1=VP-HIDE
		Bit 14: 1=SPRITES
		Bit 15: 1=HIRES
+ 034	Word	reserviert
+ 036	Long	Zeiger auf RasInfo-Struktur

Bevor wir die einzelnen Komponenten dieser Struktur einer näheren Untersuchung unterwerfen, ist es notwendig, die Bedeutung des Viewports zu klären.

Ein Viewport ist nichts weiter als eine Datenstruktur im RAM-Speicher. Sie repräsentiert jedoch einen Teil des Displays, also

einen Teil dessen, was Sie auf dem Bildschirm sehen. Bei näherer Betrachtung der Screen-Struktur aus Kapitel 3.4 werden Sie feststellen, daß der Viewport Teil dieser Struktur ist. Die Vermutung liegt also nahe, daß ein Intuition-Screen nichts anderes ist als ein Viewport nebst etwas Beiwerk. Das ist tatsächlich der Fall. Das Herz eines jeden Screens ist ein Viewport.

Ein Display besteht aus einem oder mehreren Viewports. Die folgende Zeichnung demonstriert dies:



Viewports unterliegen gewissen Einschränkungen. So ist es nicht möglich, Viewports nebeneinander darzustellen. Viewports dürfen sich außerdem nicht überschneiden und müssen mindestens eine Pixelzeile von einander Abstand halten.

Jeder Viewport kann seine eigene Grafikauflösung, eigene Farben, eigene Bitplanes besitzen. Der Viewport selbst läßt sich

wiederum in separate Zeichenflächen unterteilen, die Fenster. Diese Fenster unterliegen selbstverständlich keinen Beschränkungen und dürfen sich überlappen.

Kommen wir nun zur detaillierten Beschreibung der Datenstruktur.

4.1 Kommentar zur Datenstruktur Viewport

Offset 0: Nächster Viewport

Ein Display kann aus einem oder mehreren Viewports bestehen. Alle existierenden Viewports sind in einer Kette organisiert. Dieses Feld zeigt zum nächstfolgenden Viewport des Displays. Ist dieses Feld =0, dann existieren keine weiteren Viewports.

Offset 4: Colormap

Jeder Viewport kann seine eigenen Farben definieren. Dies ist ein Zeiger zu einer Datenstruktur namens "Colormap", die die RGB-Werte dieser Farben enthält. In Abhängigkeit von der Anzahl der vorhandenen Bitplanes kann ein jeder Viewport bis zu 32 völlig individuelle Farben nutzen (ohne Hinzunahme von Spezialmodi versteht sich).

Offset 8, 12, 16 und 20: Copper-Listen

Der Copper ist einer der drei Amiga Grafik-Coprozessoren. Er beherrscht das gesamte Display und manipuliert Register, bewegt Sprites, programmiert den Blitter (der Kopier-Prozessor). Für den Copper wurde eine eigene Programmiersprache entwickelt, die aus nur drei Befehlen besteht. Diese Felder der Viewport-Struktur enthalten die Copper-Befehlslisten, die der Prozessor braucht, um den durch den Viewport repräsentierten Teil des

Displays korrekt darstellen zu können. Die erste Liste stellt eine Zusammenfassung der anderen drei Listen dar und wird für das Display des Viewports verwendet.

Mehr darüber erfahren Sie einige Kapitel später, wenn wir uns mit der Programmierung des Coppers beschäftigen.

Offset 24 und 26: Breite und Höhe

Hier finden Sie die Breite und die Höhe des durch diesen Viewport kontrollierten Display-Teils.

Offset 28 und 30: Bitmap-Offset

Hier finden Sie die Koordinaten der linken oberen Ecke des Viewports relativ zum gesamten Display. Mit diesen Werten läßt sich der Viewport positionieren. DyOffset kann zwischen -16 und +200 variieren (bei Interlace -32 bis +400), DxOffset zwischen -16 bis +352 (bei Hi-Res -32 bis +704).

Offset 32: Die Viewport-Modi

Amiga kennt verschiedene Grafik-Modi, wovon Hi-Res (640 Punkte horizontal) und Interlace (400 Punkte vertikal) wohl die bekanntesten sein dürften. In diesem Feld ist der augenblickliche Modus zu finden.

Offset 36: Der RasInfo-Block

Jeder Viewport besitzt mindestens eine an ihn gebundene RasInfo-Datenstruktur. Wir werden sie Ihnen ein paar Seiten später im Detail vorstellen.

4.2 Die Grafik-Modi des Amiga

Insgesamt kennt der Amiga neun Spezial-Grafik-Modi. Es sind dies:

Genlock Video
Interlace
PFBA
Extra Halfbrite
DUALPF
HAM
VP-Hide
Sprites
Hi-Res

Zumindest Hi-Res und Interlace werden Ihnen bereits bekannt sein, denn AmigaBASIC unterstützt diese beiden. Der AmigaBASIC-Befehl SCREEN ist in der Lage, Screens vom Typ Lo-Res (normal, 320 Pixel breit), Hi-Res (640 Pixel breit) sowie Interlace (512 statt 256 Pixel hoch) zu schaffen.

Das Sprites-Flag muß gesetzt werden, wenn innerhalb des Viewports Sprites oder VSprites erscheinen sollen. Dies ist normalerweise der Fall.

VP-Hide ist gesetzt, wenn dieser Viewport gerade von anderen Viewports überdeckt ist (also beispielsweise ein Screen unter einem anderen liegt). Dadurch wird dieser Viewport nicht dargestellt.

Genlock Video bewirkt, daß an Stelle der Hintergrundfarbe das Videosignal einer externen Quelle dargestellt wird. Das könnte beispielsweise ein Videorecorder oder eine Kamera sein. Um diesen Modus nutzen zu können, ist ein Genlock-Interface nötig.

DUALPF steht für "Dual Playfield". In diesem Modus lassen sich innerhalb eines Viewports zwei Display-Ebenen schaffen, wobei die Hintergrundfarbe der oberen Ebene transparent ist. Wir kommen darauf zurück.

PFBA arbeitet mit dem Dual Playfield Modus. Es bestimmt die Videoprioritäten der beiden Ebenen.

HAM steht für "Hold and Modify". Mit Hilfe dieses Modus lassen sich alle 4096 Farben des Amiga gleichzeitig auf dem Screen darstellen. Diese Darstellungsart ist jedoch extrem schwierig zu programmieren. Wir kommen gleich auf sie zurück.

Extra Halfbrite ist ein neuer Grafik-Modus, mit dessen Hilfe sich anstatt der bisher 32 nun bis zu 64 Farben gleichzeitig darstellen lassen.

4.2.1 Der Halfbrite-Modus

Extra Halfbrite ist einer der Grafik-Spezialmodi, die nicht vom SCREEN-Befehl des AmigaBASIC unterstützt werden. Es ist daher unmöglich, einen Screen mit diesem Grafik-Modus durch SCREEN zu erzeugen.

Es gibt jedoch die Möglichkeit, einen bereits bestehenden Screen in einen Halfbrite-Screen zu verwandeln. Bevor wir Ihnen zeigen, wie das funktioniert, erläutern wir erst einmal die Halfbrite-Technik.

Im Normalfall ist der Amiga in der Lage, bis zu 32 Farben gleichzeitig darzustellen. Diese Zahl resultiert zum einen aus der maximal zulässigen Zahl von Bitplanes (5, 25=32), zum anderen aus der Tatsache, daß der Amiga lediglich 32 Farbregister besitzt, in denen die Farben mittels des AmigaBASIC-Befehls PALETTE definiert werden können.

Ist der Halfbrite-Modus aktiviert, dann lassen sich sechs Bitplanes verwenden. Dadurch erhöht sich die Anzahl der darstellbaren Farben von 25=32 auf 26=64. Bleibt noch das Problem der 32 Farbregister. Wo sollen die zusätzlichen 32 Farben definiert werden? Zu diesem Zweck wird jedes der 32 existierenden Farbregister doppelt benutzt: Die Farben 0 bis 31 werden wie bisher direkt aus den Farbregistern 0 - 31 bestimmt. Die Farben 32 bis 63 benutzen dieselben Farbregister 0 - 31, jedoch mit ei-

nem Unterschied: Die in den Registern gespeicherten Rot-, Grün- und Blau-Werte werden um ein Bit nach rechts verschoben.

Daraus ergeben sich drei Konsequenzen: Erstens lassen sich die zusätzlichen 32 Halfbrite-Farben nicht frei definieren. Sie hängen ab von den entsprechenden ersten 32 Farben. Zweitens werden die zusätzlichen Farben Kopien der existierenden Farben, die jedoch dunkler erscheinen (deshalb Halfbrite=Half Bright). Drittens: Sind die von Ihnen definierten ersten 32 Farben an sich bereits dunkel, dann unterscheiden sich die Halfbrite-Farben gegebenenfalls nicht von den ersten 32 Farben.

So schwerwiegend diese Einschränkungen auf den ersten Blick erscheinen, Halfbrite lohnt sich trotzdem! Sie bekommen zu Ihren 32 Farben 32 etwas dunklere Varianten. Damit läßt sich viel anfangen!

Da Halfbrite von BASIC aus nicht ohne weiteres zu aktivieren ist, erzeugen wir zunächst einen ganz normalen Screen der Tiefe 5 (5 Bitplanes). Aus den vorangegangenen Kapiteln kennen wir uns bereits sehr gut aus im Amiga-Grafiksystem. Es wird uns nicht schwerfallen, eine sechste Bitplane in die Bitmap-Struktur des Screens einzufügen. Anschließend braucht nur noch das Halfbrite-Flag des Viewports gesetzt zu werden, und (fast) sind wir am Ziel (es gibt dann noch ein kleines Problem, auf das wir aber gleich noch zu sprechen kommen).

Zur Realisierung unseres Problems benötigen wir Zugriff auf zwei System-Bibliotheken: exec und intuition. Wir brauchen die Funktionen

RemakeDisplay()
AllocMem()
FreeMem()

Es folgt nun das Programm, der Halfbrite-Aktivator. Neben dem Demoprogramm besteht es aus den beiden SUBs "HalfBriteEin" und "HalfBriteAus". Beide verlangen kein Argument.

```
'#####################################
'#
'# Programm: Halfbrite-Aktivator
'# Datum: 17.1.87
'# Autor: tob
'# Version: 1.1
'#
'#####################################

' Aktiviert den von AmigaBASIC sonst nicht zugaenglichen
' Amiga Grafik-Spezialmodus "Halfbrite". Bei 6 Bitplanes
' stehen insgesamt 64 verschiedene Farben zur Verfuegung.
' Die Funktionsweise und effektivste Programmierung dieses
' Modus' ist im Grafik-Buch genau erlaeutert. ACHTUNG: Die-
' ser Modus funktioniert nur im LoRes (Low Resolution)
' Display!

PRINT "Suche die .bmap-Dateien..."

'EXEC-Bibliothek
DECLARE FUNCTION AllocMem& LIBRARY
'FreeMem

'INTUITION-Bibliothek
'RemakeDisplay()

LIBRARY "intuition.library"
LIBRARY "exec.library"

main:  '* Einen SCREEN der Tiefe 5 eroeffnen
       loRes              = 1
       screen.nr%         = 1
       screen.x%          = 320
       screen.y%          = 200
       screen.tiefe%      = 5 '5 Planes erforderlich!
       screen.aufloesung% = loRes
       SCREEN screen.nr%,screen.x%,screen.y%,screen.tiefe%,screen.
              aufloesung%

      '* Ein FENSTER im neuen Screen eroeffnen
       fenster.nr%   = 1
       fenster.name$ = "Halfbrite!"
       WINDOW fenster.nr%,fenster.name$,,,screen.nr%

demo:  '* HalfBrite aktivieren!
       HalfBriteEin

       PRINT TAB(10);"Der HalfBrite-Modus!"

      '* Die Originalfarben...
       LOCATE 3,2:COLOR 1,0
```

```
PRINT "A ";
FOR loop%=0 TO 31
  COLOR 0,loop%
  PRINT " ";
NEXT loop%

'* ...und die HalfBrite-Farben!
LOCATE 4,2:COLOR 1,0

PRINT "B ";
FOR loop%=32 TO 63
  COLOR 0,loop%
  PRINT " ";
NEXT loop%

LINE   (22,15)-(280,32),1,b
LOCATE 7,2:COLOR 1,0
PRINT "A: Die 32 Originalfarben, gespeichert"
PRINT "     in den Hardware-Farbregistern"

LOCATE 10,2
PRINT "B: Die zusaetzlichen 32 HalfBrite-"
PRINT "     Farben, entsprechend den Original-"
PRINT "     Farben mit halber Intensitaet."

LOCATE 14,2
PRINT   "Das blinkende Beispiel zeigt: Wird"
PRINT "  das Farbregister der Originalfarbe"
PRINT "  veraendert, aendert sich auch die "
PRINT "  HalfBrite-Farbe entsprechend!" -

LOCATE 19,4
PRINT "[Linke Maustaste druecken!]"

WHILE check% = 0
  check% = MOUSE(0)
  PALETTE 30,.7,.2,.9
  FOR t  = 1 TO 500:NEXT t
  PALETTE 30,.3,.8,.1
  FOR t  = 1 TO 500:NEXT t
WEND

FOR loop% = 0 TO 31
  COLOR  loop%,loop%+32
  LOCATE 20,1
  PRINT "TEST FARBE ";loop%
  PRINT "Schriftfarbe = Originalfarbe"
  PRINT "Hintergrundfarbe = HalfBrite-Farbe"
  FOR t = 1 TO 500:NEXT t
NEXT loop%
CLS
COLOR 1,0
```

```
ende:  '* HalfBrite ausschalten und SCREEN schliessen
       HalfBriteAus
       WINDOW fenster.nr%,fenster.name$,,,-1
       SCREEN CLOSE screen.nr%
       PRINT "DEMO ist beendet!"
       LIBRARY CLOSE
       END


SUB HalfBriteEin STATIC
     SHARED screen.modus%
     SHARED screen.viewport&

    '* Variablen definieren
     MEM.CHIP  = 2^1
     MEM.CLEAR = 2^16
     memory.option&   = MEM.CHIP+MEM.CLEAR
     window.base&     = WINDOW(7)
     screen.base&     = PEEKL(window.base&+46)
     screen.bitmap&   = screen.base&+184
     screen.viewport& = screen.base&+44
     screen.rastport& = screen.base&+84
     screen.weite%    = PEEKW(screen.bitmap&)
     screen.hoehe%    = PEEKW(screen.bitmap&+2)
     screen.groesse&  = screen.weite%*screen.hoehe%
     screen.tiefe%    = PEEK(screen.bitmap&+5)
     screen.modus%    = PEEKW(screen.viewport&+32)

    '* SCREEN hat schon 6 BitPlanes?
     IF screen.tiefe%>5 THEN screen.tiefe%=2^8

    '* die fehlenden Bitplanes einbauen
     FOR loop1%=screen.tiefe%+1 TO 6
       plane&(loop1%)    = AllocMem&(screen.groesse&,memory.option&)
       IF plane&(loop1%) = 0 THEN
         FOR loop2% = screen.tiefe%+1 TO loop1%-1
           CALL FreeMem(plane&(loop2%),screen.groesse&)
         NEXT loop2%
         ERROR 7
       END IF
       POKEL screen.bitmap&+4+4*loop1%,plane&(loop1%)
     NEXT loop1%
     POKE screen.bitmap&+5,6

    '* HalfBrite einschalten
     POKEW screen.viewport&+32,(screen.modus% OR 2^7)
     CALL  RemakeDisplay
END SUB

SUB HalfBriteAus STATIC
     SHARED screen.modus%
```

```
    SHARED screen.viewport&

    '* HalfBrite-Flag zuruecksetzen
    POKEW screen.viewport&+32,screen.modus%
    CALL  RemakeDisplay
END SUB
```

Arbeiten mit Halfbrite:

Nachdem Sie das SUB "HalfBriteEin" aufgerufen haben, stehen Ihnen 64 verschiedene Farben zur Verfügung. Die ersten 32 Farben können Sie frei definieren. Benutzen Sie dazu den PALETTE-Befehl des AmigaBASIC:

```
PALETTE register,rot,grün,blau

register: 0-31
rot, gruen, blau: 0.0 - 1.0
```

Die Farben 32 bis 63 werden entsprechend mitdefiniert (sie sind halb so hell).

Mit Hilfe des COLOR-Befehls können Sie nun frei zwischen den Farben 0 - 63 wählen, damit zeichnen, schreiben, füllen! Lediglich ein Hinweis ist wichtig: Für AmigaBASIC besitzt der Screen nach wie vor nur 5 Bitplanes. Wenn AmigaBASIC also den Screen-Inhalt scrollt (wenn Sie beispielsweise in die unterste Zeile des Fensters schreiben), dann scrollen nur fünf Planes, die sechste steht still. Vermeiden Sie daher, in die unterste Fensterzeile zu printen.

Wenn Sie den Halfbrite-Modus nicht mehr brauchen, können Sie ihn durch Aufruf des SUBs "HalfBriteAus" deaktivieren.

Programm-Hinweis: Am Anfang dieses Kapitels haben wir von einem Problem gesprochen, das es gibt, sobald das Halfbrite-Flag im Viewport verändert wird. Bei dem Problem handelt es sich um die Tatsache, daß sich gar nichts tut, wenn man dieses Flag setzt. Es ist überhaupt völlig egal, was für Manipulationen Sie im Viewport vornehmen, es wird sich im Display nicht das Geringste verändern.

Diese etwas merkwürdige Feststellung ist jedoch eine logische Konsequenz: Viewport ist lediglich ein Datenblock im RAM, kein Hardware-Register. Das Display wird aber nur durch die Hardware-Register verändert. Vielmehr enthält der Viewport lediglich die Anweisungen, wie das Display beschaffen sein soll. Diese Anweisungen müssen aber erst zum Copper geschickt werden, der sie dann ausführt und die Hardware entsprechend programmiert.

Änderungen im Viewport werden erst ausgeführt, wenn die Intuition-Funktion "RemakeDisplay" aufgerufen wird. Durch sie werden neue Copper-Listen erstellt, die die Änderungen in der Viewport-Struktur reflektieren. Diese Listen werden anschließend zum Copper geschickt.

Viel einfacher sieht obiges Programm aus, wenn man es unter GFA-BASIC programmiert. Dann nämlich entfallen all die langwierigen Systemmanipulationen, denn GFA unterstützt sämtliche mögliche Videodarstellungen des Amigas, so auch Extra_Halfbrite:

```
' extra_halfbrite Screen
OPENS 1,0,0,320,256,6,128
OPENW 0,0,0,320,256,0,0
'
WHILE INKEY$=""
  COLOR RAND(64),RAND(64),RAND(64)
  DEFFILL 1,RAND(2)+2,RAND(24)+1
  PBOX RAND(320),RAND(256),RAND(320),RAND(256)
WEND
'
DEFFILL 1,1,0
WHILE INKEY$=""
  COLOR RAND(64),RAND(64),RAND(64)
  x=RAND(320)
  y=RAND(256)
  PBOX x,y,x+30,y+30
WEND
'
FOR x=0 TO 31
  COLOR x,x,x
  PBOX x*10,50,x*10+9,100
  COLOR x+32,x+32,x+32
  PBOX x*10,150,x*10+9,200
NEXT x
```

```
'
WHILE INKEY$=""
WEND
CLOSES 1
```

4.2.2 Der Hold-And-Modify Modus: 4096 Farben

Auch der Hold-And-Modify-Modus (kurz: HAM) wird nicht von AmigaBASIC unterstützt. Er läßt sich nicht durch SCREEN einschalten.

Sie werden es sich schon gedacht haben: Auch dieser Modus läßt sich nachträglich in einen bereits existierenden Screen einbauen. Bevor wir das tun, wollen wir uns das Prinzip dieses Modus vor Augen führen:

Ist der HAM-Modus aktiv, dann lassen sich bis zu 4096 Farben gleichzeitig darstellen. Es ist klar, daß dazu ein besonderes Verfahren angewendet werden muß, denn unter den herkömmlichen Bedingungen wären zur Darstellung von 4096 Farben 12 Bitplanes erforderlich. Erstens würde dies einem immensen Speicherplatzverbrauch gleichkommen (1 Bitplane = 64.000 Bytes in Lo-Res, 12 Bitplanes = 768,000 Bytes!), zweitens ist Amiga's DMA (Direct Memory Access) gar nicht schnell genug, aus 12 verschiedenen RAM-Stücken alle 1/50 Sekunde ein neues Bild zu basteln.

In Wirklichkeit arbeitet HAM mit nur sechs Bitplanes, genau wie der Halfbrite-Modus. Die ersten 16 Farben erscheinen in genau der Farbe, in der sie definiert wurden (also analog zu den ersten 16 Farbregistern). Alle anderen Farben werden nach dem HAM-Prinzip bestimmt. Sie nehmen die Farbe des Pixels zur Linken an und verändern jeweils den Rot-, Grün- oder Blauwert.

Bevor wir die etwas komplexe Gestaltung einer HAM-Grafik behandeln, wollen wir den Modus aktivieren. Das geschieht ähnlich wie beim Halfbrite-Modus: Eine sechste Bitplane wird erzeugt, in die Bitmap eingebaut, und schließlich wird das HAM-

Flag im Viewport gesetzt. Der Aufruf "RemakeDisplay" schaltet das Display um. Wieder finden Sie zwei SUBs: HAMein und HAMaus.

```
'#####################################
'#
'# Programm: HAM-Aktivator
'# Datum: 16.2.87
'# Autor: tob
'# Version: 1.4
'#
'#####################################

' Aktiviert den von AmigaBASIC sonst nicht zugaenglichen
' Amiga Grafik-Spezialmodus "HAM" (Hold-And-Modify), mit
' dem sich bis zu 4096 Farben gleichzeitig (bei 6 Bitplanes)
' darstellen lassen. ACHTUNG: Dieser Modus funktioniert
' nur im LoRes (Low Resolution) Display!

PRINT "Suche die .bmap-Dateien..."

'EXEC-Bibliothek
DECLARE FUNCTION AllocMem& LIBRARY
'FreeMem()

'INTUITION-Bibliothek
'RemakeDisplay()

LIBRARY "intuition.library"
LIBRARY "exec.library"

main:  '* Einen SCREEN der Tiefe 5 eroeffnen
       loRes              = 1
       screen.nr%         = 1
       screen.x%          = 320
       screen.y%          = 200
       screen.tiefe%      = 5 '5 Planes noetig
       screen.aufloesung% = loRes
       SCREEN screen.nr%,screen.x%,screen.y%,screen.tiefe%,screen.
              aufloesung%

      '* Ein FENSTER im neuen Screen eroeffnen
       fenster.nr%   = 1
       fenster.name$ = "HAM! 4096 Farben herbei!"
       WINDOW fenster.nr%,fenster.name$,,,screen.nr%

demo:  '* HalfBrite aktivieren!
       HAMein

       PRINT TAB(7) "256 aus 4096 Farben"
```

```
        s = 10  'Kaestchengroesse
        x = 40  'Position der linken
        y = 20  'oberen Ecke der Demo

        PALETTE 3,0,0,0      'Rahmenfarbe
        PALETTE 4,.5,0,.5    'dunkel-rotblau
        PALETTE 5,1,0,1      'hell-rotblau
        PALETTE 6,1,0,0      'hell-rot
        PALETTE 7,0,0,1      'hell-blau

       '* Setzen der Orientierungsmarken
        LINE (5,y)-(5+8,y+8),4,bf
        LINE (240,y)-(240+8,y+8),7,bf
        LINE (5,166)-(5+8,166+8),6,bf
        LINE (240,166)-(240+8,166+8),5,bf

       '* Zeichnen des Rahmens
        LINE (x-1,y-1)-(x+17*s+1,y+16*s+1),3,b

       '* Die ersten 256 HAM-Farben zeichnen
        FOR loop% = 0 TO 15
          LINE (x,loop%*s+y)-(s+x,loop%*s+s+y),32+loop%,bf
          FOR loop2% = 0 TO 15
            LINE (s+loop2%*s+x,loop%*s+y)-(2*s+loop2%*s+x,loop%*s+s+y),
                 loop2%+16,bf
          NEXT loop2%
        NEXT loop%

       '* Den Gruen-Level erhoehen
        FOR loop2% = 0 TO 15
          PALETTE 3,0,loop2%*(1/15),0
          LOCATE  10,28
          PRINT   "Gruenlevel:"
          PRINT   TAB(31) loop2%
          FOR t = 1 TO 3000:NEXT t
        NEXT loop2%

        LOCATE 2,7
        PRINT "Bitte eine Taste druecken!"

        WHILE INKEY$="":WEND

ende:  '* HAM ausschalten und SCREEN schliessen
        HAMaus
        WINDOW fenster.nr%,fenster.name$,,,-1
        SCREEN CLOSE screen.nr%
        PRINT "DEMO ist beendet!"
        LIBRARY CLOSE
        END


SUB HAMein STATIC
```

```
      SHARED screen.modus%
      SHARED screen.viewport&

     '* Variablen definieren
      MEM.CHIP  = 2^1
      MEM.CLEAR = 2^16
      memory.option&   = MEM.CHIP+MEM.CLEAR
      window.base&     = WINDOW(7)
      screen.base&     = PEEKL(window.base&+46)
      screen.bitmap&   = screen.base&+184
      screen.viewport& = screen.base&+44
      screen.rastport& = screen.base&+84
      screen.weite%    = PEEKW(screen.bitmap&)
      screen.hoehe%    = PEEKW(screen.bitmap&+2)
      screen.groesse&  = screen.weite%*screen.hoehe%
      screen.tiefe%    = PEEK(screen.bitmap&+5)
      screen.modus%    = PEEKW(screen.viewport&+32)

     '* SCREEN hat schon 6 BitPlanes?
      IF screen.tiefe%>5 THEN screen.tiefe%=2^8

     '* die fehlenden Bitplanes einbauen
      FOR loop1% = screen.tiefe%+1 TO 6
        plane&(loop1%)    = AllocMem&(screen.groesse&,memory.option&)
        IF plane&(loop1%) = 0 THEN
          FOR loop2% = screen.tiefe%+1 TO loop1%-1
            CALL FreeMem(plane&(loop2%),screen.groesse&)
          NEXT loop2%
          ERROR 7
        END IF
        POKEL screen.bitmap&+4+4*loop1%,plane&(loop1%)
      NEXT loop1%

      POKE screen.bitmap&+5,6

     '* HAM einschalten
      POKEW screen.viewport&+32,(screen.modus% OR 2^11)
      CALL RemakeDisplay
END SUB

SUB HAMaus STATIC
      SHARED screen.modus%
      SHARED screen.viewport&

     '* HalfBrite-Flag zuruecksetzen
      POKEW screen.viewport&+32,screen.modus%
      CALL RemakeDisplay
END SUB
```

Sobald Sie das Programm starten, sehen Sie ein Farbfeld. In ihm befindet sich eine Auswahl von 256 Farben. Diese Farben wer-

den lediglich aus Rot und Blau zusammengesetzt. In der linken oberen Ecke ist Dunkellila, in der rechten unteren Ecke Hell-Lila. Entsprechend findet sich Hellrot links unten, Hellblau rechts oben.

Nun wird diesen Farben gleichmäßig und langsam Grün beigemischt. Insgesamt werden also alle 4096 Farben des Amiga dargestellt.

Diese Farbenvielfalt ist beeindruckend. Leider ist ihre Programmierung nicht ganz leicht. Auf den ersten Blick erscheint sie jedenfalls kompliziert, was sie im Grunde gar nicht ist. Sehen wir uns die Sache einmal näher an:

Zunächst unterscheiden wir zwischen Echtfarben und HAM-Farben. Als Echtfarben bezeichnen wir die Farben 0 - 15. Sie entsprechen genau den Farbregistern 0 - 15. Diese Farben sind unveränderlich und lassen sich nur durch einen PALETTE-Befehl verändern. Anders ist das mit den HAM-Farben. Dies sind die Farben 16 - 63. Die HAM-Farben werden immer durch ihre Nachbarfarbe zur Linken beeinflußt. Eine HAM-Farbe nimmt die Farbe ihres Nachbarn an und verändert die Rot-, Grün- oder Blau-Komponente dieser Farbe. Welche der drei Komponenten verändert wird, hängt von der HAM-Farbe ab:

Farbe 0	**- 15**	**Echtfarbe**
Farbe 16+0	**- 16+15**	**HAM-Typ 1**
Farbe 32+0	**- 32+15**	**HAM-Typ 2**
Farbe 48+0	**- 48+15**	**HAM-Typ 3**

HAM-Farbe des Typs 1 übernimmt die Nachbarfarbe und verändert die Blau-Komponente dieser Farbe. Die Blau-Komponente der HAM-Farbe entspricht dem Wert hinter der 16. Die HAM-Farbe 16+12=28 übernimmt also die Nachbarfarbe und benutzt als Blau-Wert den Wert 12.

HAM-Farbe des Typs 2 übernimmt die Nachbarfarbe und verändert die Rot-Komponente. Die HAM-Farbe 32+8=40 über-

nimmt also die Nachbarfarbe und benutzt den Wert 8 in der Rot-Komponente.

HAM-Farbe des Typs 3 tut dasselbe wie die anderen beiden Typen, manipuliert jedoch die Grün-Komponente.

In unserem Programmbeispiel ziehen wir zunächst einen schwarzen Rahmen. Rot, Grün und Blau sind also =0. Direkt rechts vom Rahmen wird nun eine HAM-Farbe des Typs 2 gezeichnet. Sie überprüft die Farbe zur Linken, den schwarzen Rahmen also, und übernimmt seine Farbe. Rot, Grün und Blau sind also Null. Das Blau-Feld wird jedoch von der HAM-Farbe selbst bestimmt. Der Blau-Wert steigt in einer Schleife pro Bildschirmzeile an. Direkt rechts von dieser HAM-Farbe werden nun sechzehn HAM-Farben des Typs 1 gezeichnet, wobei der Rot-Wert jeweils um eins zunimmt.

Es entsteht so ein Farbmuster, dessen Rot-Intensität nach rechts hin zunimmt, dessen Blau-Intensität nach unten hin größer wird. Nun wird die Farbe des Rahmens verändert: Der ehemals schwarze Rahmen wird mittels PALETTE immer grüner. Der Grün-Wert des Rahmens wird dabei sofort von den HAM-Farben unverändert übernommen: Die gesamte Farbgrafik wird also von immer intensiverem Grün durchflutet.

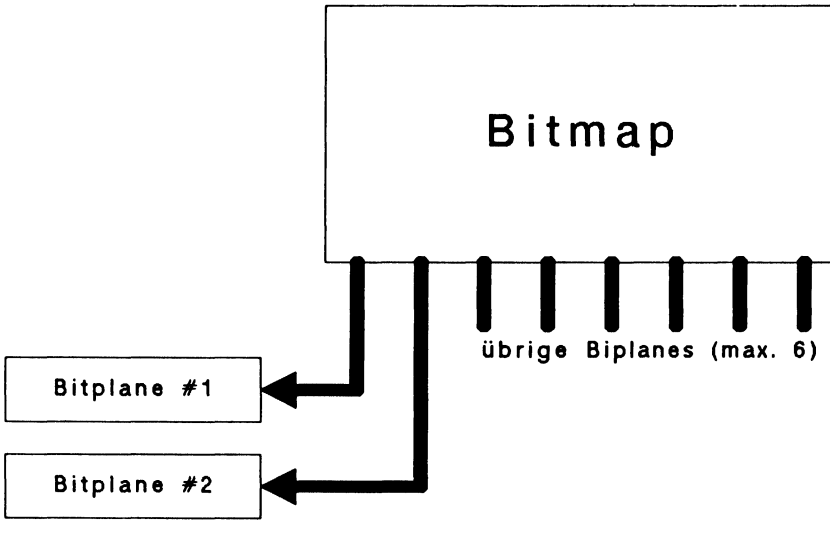
Wieder das Gegenbeispiel in **GFA** programmiert: kürzer und übersichtlicher, wie das Beispiel zeigt:

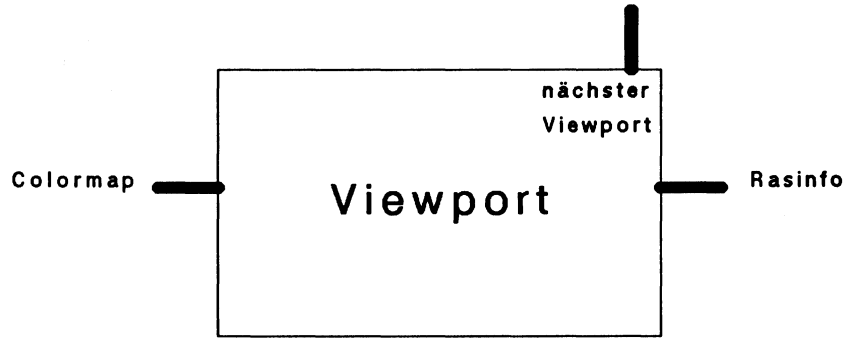
```
' hold_and_modify modus
OPENS 1,0,0,320,256,6,2048
OPENW 0,0,0,320,256,0,0
FOR rot=0 TO 15
  FOR gruen=0 TO 15
    FOR blau=0 TO 15
      x=gruen*20
      y=rot+blau*16
      COLOR rot+48
      LINE x,y,x+18,y
      INC x
      COLOR gruen+32
      LINE x,y,x+18,y
```

```
      INC x
      COLOR blau+16
      LINE x,y,x+18,y
    NEXT blau
  NEXT gruen
NEXT rot
'
WHILE INKEY$=""
WEND
CLOSES 1
```

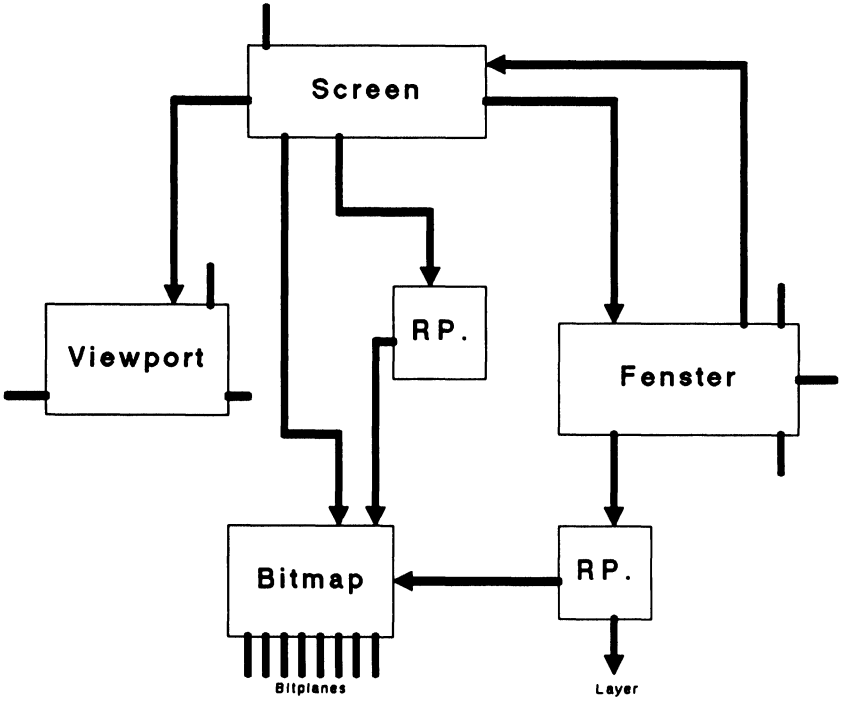
4.3 Der Viewport im System

Unser Bild vom System des Amiga ist nun ganz entschieden klarer geworden. Zwei Komponenten, die Bitmap und der Viewport, sollten Ihnen nun recht vertraut sein. Beide lassen sich wie immer zeichnerisch darstellen:





Nun diese Komponenten im Systemzusammenhang:



Langsam wird die Angelegenheit verständlich: Der Viewport ist die elementarste Stufe eines Displays. Intuition verwaltet den Viewport mit Hilfe des Screens. Sowohl der Screen als auch das Fenster zeichnen über einen privaten Rastport in ein gemeinsames Video-RAM: die Bitplanes.

Aber noch immer nicht sind die Komponenten Layers, Colormap und RasInfo geklärt.

Bevor wir uns an die weitere Erforschung des Grafiksystems machen, müssen wir einen für Sie unter Umständen nicht sofort nachvollziehbaren Schritt machen. Es gibt nämlich eine weitere Datenstruktur, auf die bisher kein einziger Zeiger verwies. Praktisch aus dem Nichts taucht jetzt der "View" auf.

4.4 Der View: Das Grafik-Stammhirn

Es ist gar kein Wunder, daß die Adresse des Views - oder bloß sein Name - noch nirgends aufgetaucht ist. Der View ist die Kontaktstelle zwischen Grafik-Software und Grafik-Hardware Ihres Amigas. Von dort geht alles los. Wir haben also am falschen Ende angefangen, als wir das Fenster einer Untersuchung unterzogen. Die Adresse des Views läßt sich durch Aufruf einer Intuition-Funktion ermitteln. Sie heißt "ViewAddress":

```
DECLARE FUNCTION ViewAddress& LIBRARY
LIBRARY "intuition.library"

view&=ViewAddress&
```

bzw. in GFA:

```
view%=ViewAddress
```

Als kleine Anmerkung: Die Intuition-Routine "ViewPortAddress" liefert die Adresse des Viewports, in dem sich Ihr aktuelles Ausgabefenster befindet:

```
DECLARE FUNCTION ViewPortAddress& LIBRARY
LIBRARY "intuition.library"

vp&=ViewPortAddress&(WINDOW(7))
```

Hier die View-Datenstruktur:

Datenstruktur "View"/graphics/18 Bytes

Offset	Typ	Bezeichnung
+ 000	Long	Zeiger auf ersten Viewport
+ 004	Long	LongFrame Copper-Liste
+ 008	Long	ShortFrame Copper-Liste
+ 012	Word	DyOffset
+ 014	Word	DxOffset
+ 016	Word	Modi

So unscheinbar diese Datenstruktur sein mag: Das gesamte Display (einschließlich aller Screens) hängt von ihr ab! Zunächst die Erläuterungen der Datenfelder:

Offset 0: Nächster Viewport

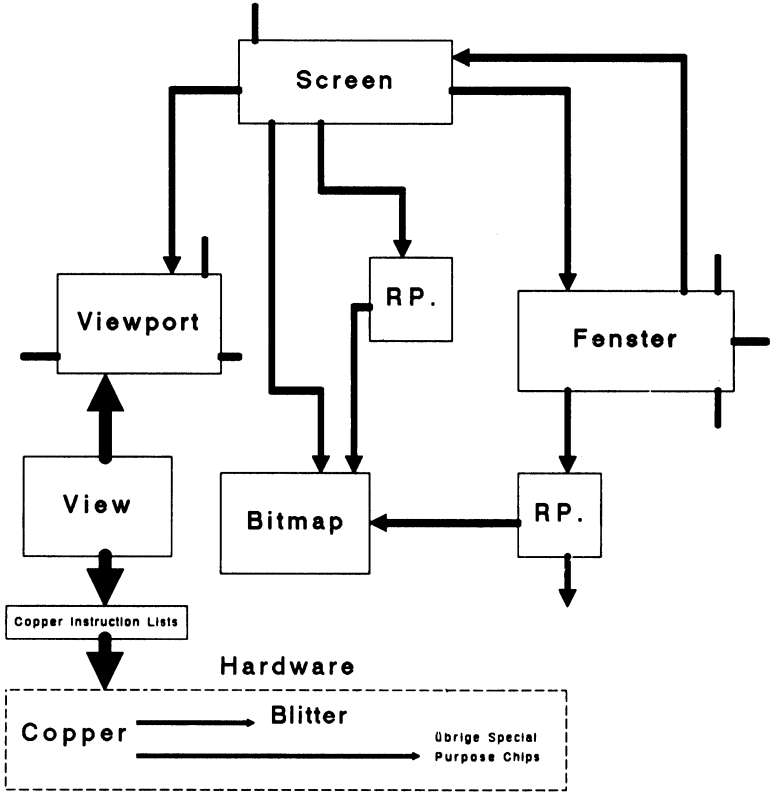
Hier findet sich die Adresse auf die erste Viewport-Struktur des Displays. Von dort findet sich dann die Adresse zu weiteren Viewports, falls es weitere gibt.

Offset 4 und 8: Copper-Listen

Schon einmal hatten wir es mit Copper-Listen zu tun. Das war innerhalb des Viewports. Während die Copper-Listen des Viewports lediglich für die Zeichenregion des Viewports zuständig waren, verwalten diese Copper-Listen das gesamte Display, also alle Viewports. Ein normales Display benötigt lediglich die LongFrame-Liste. Nur bei Interlace ist die zweite Copper-Liste nötig.

Die restlichen Felder entsprechen in ihrer Bedeutung ganz genau den gleichnamigen Feldern des Viewports. Sie sind in Kapitel 4 beschrieben.

Mit dem View kann unser Bild vom Grafiksystem nun mit den wichtigsten Komponenten ausgerüstet werden. Die Verbindung zwischen Hardware und Intuition ist hergestellt:



Bevor wir im Grafiksystem weiterarbeiten, werden wir unser Modell dieses Systems überprüfen. Wir sind nun weit genug fortgeschritten, ein eigenes Display zu erzeugen. Dazu sind einige Funktionen der Grafik-Bibliothek nötig:

InitView()
InitVPort()
GetColorMap()
InitBitMap()
AllocRaster()
LoadRGB4()
MakeVPort()
MrgCop()
LoadView()
FreeRaster()
FreeColorMap()
FreeVPortCopLists()
FreeCprList()

Wir werden in dem folgenden Programm alle Schritte durchexerzieren, die zur Schaffung eines einfachen Displays mit einem Viewport notwendig sind. Betrachten Sie dies als eine Kontrolle, daß unser Modell korrekt ist, sowie als Übung im Umgang mit dem Copper. Das nächstfolgende Kapitel wird sich nämlich eingehend mit seiner Programmierung beschäftigen.

Hier zunächst das Programm:

```
'########################################
'#
'# Programm: Grafik Primitiv-Display
'# Datum: 1.1.87
'# Autor: tob
'# Version: 1.0
'#
'########################################

' Demonstriert das Entstehen eines Grafik-Displays auf
' dem Amiga mit Hilfe der "Graphic Primitives", den
' Grundbefehlen der Grafik-Bibliothek. Es wird ein HiRes
' (High Resolution) Screen mit einer Bitplane erzeugt
' (Tiefe=1), in den der Inhalt der ersten Bitplane dieses
' Screens kopiert wird.

PRINT "Suche die .bmap-Dateien..."

'GRAPHICS-Bibliothek
DECLARE FUNCTION AllocRaster& LIBRARY
DECLARE FUNCTION GetColorMap& LIBRARY
```

```
'FreeRaster()
'FreeColorMap()
'FreeVPortCopLists()
'FreeCprList()
'InitView()
'InitVPort()
'InitBitMap()
'LoadRGB4()
'MakeVPort()
'MrgCop()
'LoadView()

'EXEC-Bibliothek
DECLARE FUNCTION AllocMem& LIBRARY
'FreeMem()

'INTUITION-Bibliothek
DECLARE FUNCTION ViewAddress& LIBRARY

LIBRARY "exec.library"
LIBRARY "graphics.library"
LIBRARY "intuition.library"

init:     '* Screen-Parameter definieren
          weite%       = 640
          hoehe%       = 200
          tiefe%       = 1
          o.bitplane1& = PEEKL(PEEKL(WINDOW(8)+4)+8)

          '* Zeiger auf unseren eigenen View retten, damit wir auch
          '* mal wieder zurueck koennen
          oldview&     = ViewAddress&
          '* Speicherplatz fuer benoetigte Strukturen reservieren
          '* View - das Stammhirn des Displays
          view&        = 18
          GetMemory view&
          '* ViewPort - unser Screen in spe
          viewport&    = 40
          GetMemory viewport&
          '* BitMap - Verwalter der BitPlanes
          bitmap&      = 40
          GetMemory bitmap&
          '* RasInfo - Informationen fuer den ViewPort
          RasInfo&     = 12
          GetMemory RasInfo&

          '* View und ViewPort gebrauchsfertig machen
          CALL InitView(view&)
          CALL InitVPort(viewport&)

          '* Hires
          hires& = &H8000
```

```
POKEW viewport&+32,hires&

'* ViewPort in View einhaengen
POKEL view&,viewport&

'* Farbtabelle schaffen
colorMap&    = GetColorMap&(2)
IF colorMap& = 0 THEN ERROR 7

'* ViewPort mit unseren Parametern bestuecken
POKEW viewport&+24,weite%
POKEW viewport&+26,hoehe%

'* RasInfo in ViewPort einhaengen
POKEL viewport&+36,RasInfo&

'* Farbtabelle in den ViewPort einhaengen
POKEL viewport&+4,colorMap&

'* BitMap Struktur mit unseren Parametern fuellen
CALL InitBitMap(bitmap&,tiefe%,weite%,hoehe%)

'* eine BitPlane besorgen
plane&    = AllocRaster&(weite%,hoehe%)
IF plane& = 0 THEN ERROR 7

'* BitPlane in BitMap einhaengen
POKEL bitmap&+8,plane&

'* BitMap in RasInfo einhaengen
POKEL RasInfo&+4,bitmap&

'* Farben definieren
rot$       = CHR$(15)+CHR$(0)
schwarz$   = CHR$(0)+CHR$(0)
colortable$ = rot$+schwarz$

'* Farben in Farbtabelle laden
CALL LoadRGB4(viewport&,SADD(colortable$),2)

'* Copper Instruction List konstruieren
CALL MakeVPort(view&,viewport&)
CALL MrgCop(view&)

'* Neues Display in den Copper laden
CALL LoadView(view&)

'* Mit dem Display spielen
BEEP
size& = weite%*hoehe%/8

FOR loop& = 0 TO size&-1
```

```
        POKE plane&+loop&,PEEK(o.bitplane1&+loop&)
      NEXT loop&
      BEEP

     '* Unsere alten Copperlisten wieder zurueckladen
      CALL LoadView(oldview&)

     '* Aufraeumen: Speicher fuer BitPlane zurueckgeben
      CALL FreeRaster(plane&,weite%,hoehe%)
     '* Farbtabelle freigeben
      CALL FreeColorMap(colorMap&)
     '* Zwischenlisten des ViewPorts freigeben
      CALL FreeVPortCopLists(viewport&)
     '* Copper Instruction List freigeben
      copperlist& = PEEKL(view&+4)
      CALL FreeCprList(copperlist&)
     '* Struktur-Speicher freigeben
      FreeMemory view&
      FreeMemory viewport&
      FreeMemory RasInfo&
      FreeMemory bitmap&

     '* und das war's
      LIBRARY CLOSE
      END


SUB GetMemory(size&) STATIC
        opt&      = 2^0+2^1+2^16
        RealSize& = size&+4
        size&     = AllocMem&(RealSize&,opt&)
        IF size&  = 0 THEN ERROR 255
        POKEL size&,RealSize&
        size&     = size&+4
END SUB

SUB FreeMemory(add&) STATIC
        add&      = add&-4
        RealSize& = PEEKL(add&)
        CALL FreeMem(add&,RealSize&)
END SUB
```

Dokumentation:

Zunächst müssen wir uns überlegen, was wir erzeugen wollen. Ein Display soll's sein. Aber wie breit und wie hoch? Wir wählen einen Hi-Res-Bildschirm mit einer Standard-Ausdehung von 640*200 Pixel, eine Bitplane tief.

Um nach unseren Manipulationen wieder zurück zu unserem eigenen Display finden zu können, muß die Adresse unserer eigenen View-Struktur in einer Variablen gerettet werden. Die Intuition-Funktion ViewAddress liefert uns den benötigten Zeiger.

Jetzt geht's los: Für die Erzeugung unseres Displays benötigen wir diese Strukturen:

View (18 Bytes)
Viewport (40 Bytes)
Bitmap (40 Bytes)
RasInfo (12 Bytes)

Die View-Struktur bildet das Stammhirn unseres zukünftigen Displays. Es gibt nur einen einzigen aktiven View. Von diesem View zweigen beliebig viele Viewports ab.

View und Viewport müssen gebrauchsfertig gemacht werden. InitView füllt die View-Struktur mit den Standardwerten: Er wird automatisch darauf eingestellt, ca. 1,25 cm vom Rand des Monitors zu erscheinen. InitVPort tut dasselbe mit dem Viewport: Er wird standardmäßig auf LoRes geschaltet, der Zeiger auf den nächsten Viewport wird auf Null gesetzt, denn es folgen keine weiteren Viewports.

Jetzt muß eine Verbindung zwischen View und Viewport hergestellt werden. Dazu dient das erste Feld der View-Struktur. Dort wird die Adresse der ersten (und einzigen) Viewport-Struktur hinterlegt.

Nun muß eine Farbtabelle geschaffen werden, die später die Farbwerte unseres Screens aufnehmen wird. Diese Aufgabe erledigt GetColorMap.

Jetzt werden die Ausdehnungen unseres Viewports in denselben geschrieben. Der RasInfo-Block wird in den Viewport eingehängt.

Nun muß die Bitmap-Struktur gebrauchsfertig gemacht werden. InitBitMap() leistet die gröbste Arbeit. Die Adresse auf unsere eine Bitplane müssen wir allerdings selbst in die Bitmap-Struktur schreiben.

Die Adresse der Bitmap wird nun in die RasInfo-Struktur geschrieben. Die Farben werden mittels LoadRGB4 in den Viewport geladen.

Unser Display ist nun gebrauchsfertig, alle nötigen Daten sind verstaut. Aus diesen Informationen muß der Amiga nun Instruktionen für den Grafik-Prozessor erzeugen. Das geschieht schrittweise: Die Funktion MakeVPort bildet aus allen Daten des Viewports die entsprechenden Copper-Listen und schreibt die Zeiger auf diese Listen in den Viewport. Anschließend integriert die Funktion MrgCop die Instruktionen unseres Viewports mit denen des übrigen Displays (wir haben nur einen Viewport, also was soll's).

Die fertige Copper-Liste wird im View gespeichert. Aus den Daten für unser Display ist eine Liste mit Copper-Befehlen geworden. Diese Befehle müssen jetzt nur noch zum Copper gesendet werden, und unser neues Display erscheint. Diese Aufgabe erledigt LoadView. Sofort erscheint unser knallrotes Display.

Um zu zeigen, daß dies ein voll funktionsfähiges Display ist, wird nun die erste Bitplane des Workbench-Screens in unser Display kopiert. Das dauert ein Weilchen.

Alles hat geklappt, wir wollen wieder zurück. Die Adresse auf unseren alten View hatten wir zwischengespeichert, und so bereitet es keine Schwierigkeiten, in unser altes Display zurückzukehren: LoadView sendet die alten Copper-Listen zum Copper.

Und nun, obwohl die Demo fast zu Ende ist, kommt noch etwas ganz Wichtiges: das Aufräumen. Das Display hat eine Menge Speicher gefressen, den wir natürlich wieder zurückhaben wollen.

4.5 Copper-Programmierung: Der Coprozessor im Handgepäck

Gerade hat der Copper unter Beweis gestellt, wie mächtig er ist. Das werden wir gleich ausnutzen. Unser nächstes Programmierprojekt heißt: Double-Buffering.

4.5.1 Mit Double-Buffering blitzschnelle Grafik

Die Zeichengeschwindigkeit des Amiga läßt sich durch den Copper nicht beeinflussen, denn der arbeitet ohnehin auf Höchsttouren. Sie können aber dem Anwender Ihrer Programme glauben machen, Grafiken entstünden blitzschnell. Das Geheimnis heißt: Double-Buffering und funktioniert so: Sie zeigen dem Anwender ein Display, in dem sich gar nichts tut. Während der nun gelangweilt in dieses Display starrt, baut sich Ihre Grafik in einem zweiten, unsichtbaren Display in Ruhe auf. Ist die Grafik fertiggestellt, schalten Sie die Displays um, und - blitzartig erscheint Ihre Grafik auf dem Bildschirm.

Das Prinzip ist einfach: Die Zeiger auf die Copper-Listen des alten Displays werden aus dem View ausgelesen und gespeichert, die Zeiger im View werden gelöscht. Nun wird eine neue Bitmap mit neuen Bitplanes eingerichtet - das zweite Display. Für dieses Display werden mittels MakeVPort und MrgCop Copper-Listen generiert. Auch diese Copper-Listen werden gespeichert. Um von einem Display ins andere zu schalten, brauchen nun nur die entsprechenden Zeiger in die Viewstruktur geschrieben und LoadView aufgerufen zu werden.

Wieder haben wir ein kleines Programmpaket entwickelt. Es besteht aus diesen SUBs:

MakeDoubleBuffer
DoubleBufferOn
DoubleBufferOff
AbortDoubleBuffer
transmit

Es folgt das Listing:

```
'#####################################
'#
'# Programm: Double Buffered Display
'# Datum:
'# Autor: tob
'# Version: 1.0
'#
'#####################################

' Dieses Programm richtet einen zweiten Screen ein,
' der als Backup-Buffer fuer diesen Screen arbeitet.

PRINT "Suche .bmap-Dateien..."

'GRAPHICS-Bibliothek
DECLARE FUNCTION BltBitMap& LIBRARY
DECLARE FUNCTION AllocRaster& LIBRARY
'FreeRaster()
'MakeVPort()
'MrgCop()
'LoadView()
'FreeCprList()

'EXEC-Bibliothek
DECLARE FUNCTION AllocMem& LIBRARY
'FreeMem()
'CopyMem()

'INTUITION-Bibliothek
DECLARE FUNCTION ViewPortAddress& LIBRARY
DECLARE FUNCTION ViewAddress& LIBRARY

LIBRARY "intuition.library"
LIBRARY "graphics.library"
LIBRARY "exec.library"

init:       CLS
            PRINT "OHNE DOUBLE BUFFERING!"
            FOR t=1 TO 20
              PRINT STRING$(80,"+")
            NEXT t
            FOR t=1 TO 20
              x% = RND(1)*600
              y% = RND(1)*150
              r% = RND(1)*100
              CIRCLE (x%,y%),r%
            NEXT t
            CLS
            PRINT "UND NUN MIT DOUBLE BUFFERING!!!"
```

```
        MakeDoubleBuffer
        DoubleBufferOn
        FOR t=1 TO 20
          PRINT STRING$(80,"+")
        NEXT t
        transmit
        LOCATE 1,1
        FOR t=1 TO 20
          x% = RND(1)*600
          y% = RND(1)*150
          r% = RND(1)*100
          CIRCLE (x%,y%),r%
        NEXT t
        transmit
        LOCATE 5,10
        LINE (38,29)-(442,67),3,b
        PRINT "Auch das geht!! Gut, nicht? Fast wie UNDO!"
        PRINT TAB(10);"Double Buffering = Backup-Display"
        PRINT TAB(10);"Dies hier sind zwei separate Screens, die"
        PRINT TAB(10);"hin- und hergeschaltet werden!"
        FOR loop%=1 TO 15
          DoubleBufferOn
          FOR t=1 TO 1000:NEXT t
          DoubleBufferOff
          FOR t=1 TO 1000:NEXT t
        NEXT loop%
        PRINT
        PRINT "LINKE MAUSTASTE DRUECKEN!"
        SLEEP:SLEEP
        AbortDoubleBuffer
        LIBRARY CLOSE
        END


SUB MakeDoubleBuffer STATIC
     '* Ein zweites Display schaffen
      SHARED ZielBitmap&,rasInfo&,QuellBitMap&,view&
      SHARED bufferx%,buffery%,vp&
      SHARED home1&,home2&,guest1&,guest2&
      view&         = ViewAddress&
      vp&           = ViewPortAddress&(WINDOW(7))
      rasInfo&      = PEEKL(vp&+36)
      QuellBitMap&  = PEEKL(rasInfo&+4)
      opt&          = 2^0+2^1+2^16
      ZielBitmap&=AllocMem&(40,opt&)

     '* BitMaps kopieren
      IF  ZielBitmap& = 0 THEN ERROR 7
     '* ACHTUNG: NUR FUER KICKSTART VERSION 1.2
     '* FUER 1.0 UND 1.1 DIESE ZEILEN VERWENDEN:
     '*
     '* FOR loop&=0 to 40 STEP 4
```

```
      '*   POKEL ZielBitMap&+loop&,PEEKL(QuellBitMap&+loop&)
      '* NEXT loop&

       CALL CopyMem(QuellBitMap&,ZielBitmap&,40)

      '* Planes besorgen
       bufferx%  = PEEKW(QuellBitMap&)*8
       buffery%  = PEEKW(QuellBitMap&+2)
       tiefe%    = PEEK(QuellBitMap&+5)
       FOR loop% = 0 TO tiefe%-1
         plane&(loop%)    = AllocRaster&(bufferx%,buffery%)
         IF plane&(loop%) = 0 THEN ERROR 7
         POKEL ZielBitmap&+8+loop%*4,plane&(loop%)
       NEXT loop%

      '* aktives Display in Buffer kopieren
       plc% = BltBitMap&(QuellBitMap&,0,0,ZielBitmap&,0,0,bufferx%,
              buffery%,200,255,0)
       IF plc%<>tiefe% THEN ERROR 17

      '* Original-Copper-List speichern
       home1& = PEEKL(view&+4)
       home2& = PEEKL(view&+8)

      '* Zweite Copper List erzeugen
       POKEL view&+4,0
       POKEL view&+8,0
       POKEL rasInfo&+4,ZielBitmap&
       CALL  MakeVPort(view&,vp&)
       CALL  MrgCop(view&)
       CALL  LoadView(view&)
       guest1& = PEEKL(view&+4)
       guest2& = PEEKL(view&+8)

      '* Reset
       POKEL rasInfo&+4,QuellBitMap&
       POKEL view&+4,home1&
       POKEL view&+8,home2&
       CALL  LoadView(view&)

END SUB

SUB DoubleBufferOn STATIC
      '* Neue Copper List aktivieren
       SHARED view&,guest1&,guest2&
       SHARED rasInfo&,ZielBitmap&
       POKEL  view&+4,guest1&
       POKEL  view&+8,guest2&
       CALL   LoadView(view&)
END SUB

SUB DoubleBufferOff STATIC
```

```
     '* alte Copper List aktivieren
      SHARED view&,home1&,home2&
      SHARED rasInfo&,QuellBitMap&
      POKEL  view&+4,home1&
      POKEL  view&+8,home2&
      CALL   LoadView(view&)
END SUB

SUB transmit STATIC
     '* altes Display in den neuen Buffer kopieren
      SHARED QuellBitMap&,ZielBitmap&,bufferx%,buffery%
      plc% = BltBitMap&(QuellBitMap&,0,0,ZielBitmap&,0,0,bufferx%,
             buffery%,200,255,0)
END SUB

SUB AbortDoubleBuffer STATIC
      SHARED rasInfo&,view&,ZielBitmap&
      SHARED vp&,bufferx%,buffery%
      SHARED home1&,home2&,guest1&,guest2&

     '* altes Display und VPort-Lists herstellen
      POKEL view&+4,home1&
      POKEL view&+8,home2&
      CALL  MakeVPort(view&,vp&)
      CALL  MrgCop(view&)
      CALL  LoadView(view&)

     '* neue VPort-Copperlisten loeschen
      CALL  FreeCprList(guest1&)

     '* Zweites Set Copper Listen loeschen
      IF guest2&<>0 THEN CALL FreeCprList(guest2&)
      add& = ZielBitmap&+8
      pl&  = PEEKL(add&)

     '* BitPlanes und BitMap loeschen
      WHILE pl&<>0
        CALL FreeRaster(pl&,bufferx%,buffery%)
        add& = add&+4
        pl&  = PEEKL(add&)
      WEND
      CALL FreeMem(ZielBitmap&,40)
END SUB
```

Anwendung:

Sie schalten das Double-Buffer-System mit dem Befehl:

```
MakeDoubleBuffer
```

ein. Dadurch wird das zweite, unsichtbare Display geschaffen. Sie dürfen diesen Befehl nur ein einziges Mal aufrufen. Soll es losgehen mit Double-Buffering, dann benutzen Sie den Befehl:

```
DoubleBufferOn
```

Dadurch wird das versteckte Display aktiviert. Ihr altes Display, in das Sie zeichnen, wird unsichtbar. Sie können nun in Ruhe Ihre Grafik erzeugen, denn auf dem Bildschirm ist davon nichts zu sehen. Sobald Ihre Grafik fertiggestellt ist, genügt der Aufruf:

```
transmit
```

um den Inhalt des alten, unsichtbaren Displays - Ihre Grafik also - in das sichtbare Display zu senden. Sie können den transmit-Befehl beliebig oft gebrauchen.

Wollen Sie kurzfristig ein ungepuffertes Display, dann genügt der Aufruf:

```
DoubleBufferOff
```

Alle Zeichenbefehle und Prints erscheinen sofort und ungepuffert. Via "DoubleBufferOn" gelangen Sie wieder zurück ins gepufferte System.

Wollen Sie gänzlich raus aus dem System (weil Ihr Programm endet oder Sie die langwierigen Zeichnungen hinter sich gebracht haben), dann verwenden Sie:

```
AbortDoubleBuffer
```

Dadurch werden alle Speicherbereiche des Puffer-Displays ans System zurückgegeben.

4.5.2 Eigene Programmierung des Coppers

Bisher wurden die Copper Instruction Lists, die das Display erzeugen, vom Amiga selbst anhand der von uns gelieferten Daten erzeugt. Daneben gibt es aber die Möglichkeit, den Copper wirklich selbst zu programmieren.

Bevor wir das tun können, muß die Funktionsweise des Coppers erläutert werden: Der Copper lebt in enger Freundschaft zum Elektronenstrahl des Displays. Dieser Elektronenstrahl fegt alle 1/50 Sekunde von der linken oberen Display-Ecke bis zur rechten unteren und zeichnet dabei das sichtbare Bild.

Der Copper ist in der Lage, auf eine bestimmte Position dieses Elektronenstrahls zu warten. Das bewerkstelligt der WAIT-Befehl des Prozessors. Er verlangt eine Y- und eine X-Koordinate und veranlaßt den Copper, solange zu warten, bis der Elektronenstrahl diese Koordinate passiert hat. Erst danach werden weitere Instruktionen verarbeitet.

Durch den Befehl MOVE ist der Copper weiterhin in der Lage, die Hardware-Register der Special Purpose Chips zu adressieren. Sie finden die Belegung der Hardware-Register im Anhang. Der MOVE-Befehl verlangt den Offset des Hardware-Registers und den Wert, der in dieses Register geschrieben werden soll.

Die SKIP-Anweisung, der dritte und letzte Befehl des Coppers, wird dazu benutzt, bestimmte Anweisungen einer Copper-Liste zu überspringen.

Nun wäre es ein langwieriges Unterfangen, die Copper-Listen für ein gesamtes Display selbst zu schreiben. Das ist auch völlig unnötig, denn diese Arbeit erledigt die Funktion MakeVPort ja bereits ohne Probleme. Möchte man eigene Copper-Instruktionen in die Copper-Listen des Gesamt-Displays einbinden, geht man einen anderen Weg: In der Struktur eines jeden Viewports befindet sich der Zeiger auf eine sogenannte "User Copper List". Dieser Zeiger ist normalerweise =0. Möchte man eigene Instruktionen ins Display integrieren, dann erzeugt man eine

eigenständige Copper-Liste mit den gewünschten Befehlen. Anschließend hinterlegt man die Anfangsadresse dieser Liste als Zeiger im Viewport im Feld "User Copper List". Nun geht man wie gewohnt vor: MakeVPort bindet die User Liste in die Display Liste des Viewports ein, MrgCop bindet diese Liste in die Gesamtliste im View, und LoadView schließlich aktiviert die manipulierten Copper-Listen.

Jetzt stellt sich allerdings die Frage, wie die eigene Copper-Liste erzeugt wird. Dazu finden Sie im nächsten Programm vier SUBs:

InitCop
ActiCop
WaitC
MoveC

Zunächst muß eine Datenstruktur namens "UCopList" erzeugt werden. Diese Struktur benötigt einen freien Speicher von 12 Bytes. Diese Aufgabe erledigt "InitCop".

Nun läßt sich die User-Liste mit den Befehlen MoveC und WaitC programmieren (Skip ist für unsere Anwendungen uninteressant).

Der Aufruf des Wait-Befehls sieht so aus:

```
WaitC y%,x%
```

Es wird erst die Y-Bildschirmkoordinate verlangt, auf die der Copper warten soll. WaitC verlangt zuerst die Y-Koordinate, weil es dadurch MrgCop leichter fällt, die Inhalte der verschiedenen Copper-Listen der Reihe nach zusammenzufassen.

MoveC kann einen beliebigen 16-Bit-Wert in eines der Hardware-Register schreiben. Wir verwenden in diesem Kapitel nur eine sehr geringe Auswahl dieser Register. Eine vollständige Registerbeschreibung ist aber im Anhang dieses Buches wiedergegeben. Hier der Aufruf:

```
MoveC register%, wert%
```

register%:	**Offset des gewünschten Hardware-Registers**
wert%:	**16-Bit-Wert**

Hier eine Auswahl der für uns wichtigsten Hardware-Register:

Register	Bedeutung
384	**Farbregister 0 (Hintergrundfarbe)**
386	**Farbregister 1 (Zeichenfarbe)**
388	**Farbregister 2**
(...)	
444	**Farbregister 30**
446	**Farbregister 31**

Kommen wir nun wieder zu unserer User Copper List. Nach dem Aufruf "InitCop" können Sie beliebig viele MoveC's und WaitC's einbauen. Sie müssen jedoch darauf achten, daß Ihre WaitC's den Bildschirmkoordinaten entsprechend aufgerufen werden. Die obere linke Ecke des Displays ist Koodinate (0,0). Von dort wandert der Elektronenstrahl los. Ihre WaitC's müssen nun nach den Koordinaten, auf die sie warten sollen, nach steigenden X- und Y-Werten geordnet werden.

Ist Ihre User Copper List fertiggestellt, dann wird mittels ActiCop diese Liste in das bestehende Display eingebunden: Ihre Anweisungen werden vom Copper ausgeführt.

In unserem Beispielprogramm haben wir einen eigenen Screen geöffnet. Um den Speicherplatz, den unsere Copper Instructions belegt haben (inkl. der von uns reservierten User Liste) ans System zurückzugeben, genügt es, den Intuition Screen zu schließen. Intuition erledigt dann diese Aufgabe automatisch. Versuchen Sie daher nicht, User Instructions in den Workbench Screen einzubauen; zwar würden die Instructions ordnungsgemäß ausgeführt, Sie hätten aber keine Möglichkeit, das Normaldisplay wiederherzustellen und den belegten Speicher freizugeben.

Das folgende Programm zeigt Ihnen, wie eine einfache Programmierung des Coppers aussehen könnte:

```
'#######################################
'#
'# Programm: Copper Raster-Interrupt I
'# Datum: 15.12.87
'# Autor: tob
'# Version: 1.0
'#
'#######################################

' Demonstriert die Programmierung des Amiga Grafik Co-
' prozessors (Copper) von AmigaBASIC.

PRINT "Suche die .bmap-Dateien..."

'INTUITION-Bibliothek
DECLARE FUNCTION ViewAddress& LIBRARY
DECLARE FUNCTION ViewPortAddress& LIBRARY
'RethinkDisplay()

'EXEC-Bibliothek
DECLARE FUNCTION AllocMem& LIBRARY
'FreeMem()

'GRAPHICS-Bibliothek
'CWait()
'CMove()
'CBump()

LIBRARY "intuition.library"
LIBRARY "graphics.library"
LIBRARY "exec.library"

pre:        CLS
            SCREEN 1,640,255,2,2
            WINDOW 2,"COPPER!",(0,0)-(630,200),16,1

            PRINT "Raster-Interrupt durch Copper-Programmierung: Der
                  geteilte Bildschirm!"

init:       farbregister%=384
            rot%         = 15 '0...15
            gruen%       =  4 '0...15
            blau%        =  4 '0...15
            farbwert%    = rot%*2^8+gruen%*2^4+blau%
            yKoordinate% = 128
            xKoordinate% = 20

main:       InitCop
            waitC yKoordinate%,xKoordinate%
            moveC farbregister%,farbwert%
            ActiCop
```

```
            PRINT "Eine Taste druecken!"
            WHILE INKEY$="":WEND

            WINDOW CLOSE 2
            SCREEN CLOSE 1


ende:       LIBRARY CLOSE
            END


SUB InitCop STATIC
      SHARED UCopList&
      opt&       = 2^0+2^1+2^16
      UCopList& = AllocMem&(12,opt&)
      IF UCopList& = 0 THEN ERROR 7
END SUB

SUB ActiCop STATIC
      SHARED UCopList&
      waitC 10000,256
      viewport& = ViewPortAddress&(WINDOW(7))
      POKEL viewport&+20,UCopList&
      CALL  RethinkDisplay
END SUB

SUB waitC(y%,x%) STATIC
      SHARED UCopList&
      CALL CWait(UCopList&,y%,x%)
      CALL CBump(UCopList&)
END SUB

SUB moveC(reg%,wert%) STATIC
      SHARED UCopList&
      CALL CMove(UCopList&,reg%,wert%)
      CALL CBump(UCopList&)
END SUB
```

Programm-Beschreibung der SUBs:

InitCop: Die Exec-Funktion AllocMem beschafft einen 12 Bytes umfassenden Speicherbereich, der die UCop-List-Datenstruktur aufnehmen wird.

WaitC: Eine Wait-Instruktion wird in die User-Liste gefügt. Dazu wird der Grafik-Bibliotheks-Befehl CWait aufgerufen. CBump erhöht den internen Zeiger innerhalb der User-Liste.

MoveC: Die Funktion CMove der Grafik-Bibliothek wird aufgerufen. Sie fügt eine Move-Instruktion in die User-Liste ein. CBump() erhöht wiederum den Zeiger.

ActiCop: Ein letztes WaitC wird an die Liste gehängt. Dieses Wait wartet auf eine Bildschirmposition, die der Elektronenstrahl niemals erreichen kann. Diese Anweisung schließt die Liste ab und entspricht dem Macro CEND.

Anschließend wird die Adresse unserer User-Liste an die entsprechende Stelle im Viewport des gewünschten Screens geschrieben. Die Intuition-Funktion "RethinkDisplay" generiert die neuen Copper-Listen für den View und sendet sie zum Copper. Das neue Display erscheint.

Zum Schluß wird der Screen geschlossen. Dadurch werden die Copper-Listen aus der Haupt-Copper-Liste im View entfernt, aller belegte Speicher wird ans System zurückgegeben.

4.5.3 Mit Copper-Programmierung: 512 Farben gleichzeitig

Das Prinzip der Copper-Programmierung ist nun klar geworden. Das nächste Programm soll Ihnen eine kleine Kostprobe dieser machtvollen Technik zeigen. Unser Plan: Wir verändern mit einem WaitC für jede Bildschirmzeile die Hintergrundfarbe. Gleichzeitig wird in jeder Bildschirmzeile eine andere Zeichenfarbe aktiviert. Bei 256 Bildschirmzeilen pro Display kommen wir auf insgesamt 512 Farben, die gleichzeitig dargestellt werden. Die Farben 2 und 3 bleiben normal einfarbig. Achtung: Verwenden Sie nie mehr als ca. 1600 Copper-Instructions in einer Liste!

```
'########################################
'#
'# Programm: Copper Raster-Interrupt II
'# Datum: 11.4.87
```

```
'# Autor: tob
'# Version: 1.0
'#
'#####################################

' Copper-Programmierung erzeugt bei nur 2 Bitplanes
' anstatt der gewoehnlichen 4 Farben hier 512 verschie-
' dene Farbtoene im Hintergrund und als Zeichenfarbe

PRINT "Suche die .bmap-Dateien..."

'INTUITION-Bibliothek
DECLARE FUNCTION ViewAddress& LIBRARY
DECLARE FUNCTION ViewPortAddress& LIBRARY
'RethinkDisplay()

'EXEC-Bibliothek
DECLARE FUNCTION AllocMem& LIBRARY
'FreeMem()

'GRAPHICS-Bibliothek
'CWait()
'CMove()
'CBump()

LIBRARY "intuition.library"
LIBRARY "graphics.library"
LIBRARY "exec.library"

pre:        CLS
            SCREEN 1,640,255,2,2
            WINDOW 2,"COPPER!",(0,0)-(630,200),16,1

            PRINT "Individuelle Copperprogrammierung macht es moeglich."
            PRINT "256 Hintergrundfarben!"
            PRINT "Bitte etwas Geduld - Berechne Instruction Lists."

init:       farbregister1% = 384
            farbregister2% = 386
            xKoordinate%   = 20
            maxY%          = 256

main:       InitCop
            FOR loop%=1 TO maxY%
              waitC loop%,xKoordinate%
              moveC farbregister1%,loop%
              moveC farbregister2%,4096-loop%
            NEXT loop%
            ActiCop

           '* Text ausgeben fuer den lieben Effekt
            LOCATE 5,1
```

```
            PRINT "Die Hintergrundfarbe besteht aus 256 Einzel-"
            PRINT "Farben! Aber auch die Schriftfarbe ist nicht"
            PRINT "mehr eintoenig: 256 Gelbwerte, pro Raster-"
            PRINT "zeile einer. Hier koennte statt dieser niveau-"
            PRINT "losen Warteschleife ein Wahnsinnsprogramm"
            PRINT "die Arbeit aufnehmen...!"
            LOCATE 15,1
            PRINT "Bitte druecken Sie eine Taste, wenn"
            PRINT "Sie ferti gelesen haben."
            WHILE INKEY$=""
            WEND

            LOCATE 11,1
            PRINT "Tut es aber nicht."

            FOR t=1 TO 2000:NEXT t
            CLS
            PRINT "Es folgt eine Grafik-Demo!"

            LINE (0,100)-(630,190),2,bf
            FOR loop%=0 TO 630 STEP 30
              LINE (loop%*1.5,190)-(loop%,100),1
            NEXT loop%
            FOR loop%=100 TO 190 STEP 20
              LINE (0,loop%)-(630,loop%),1
            NEXT loop%

            CIRCLE (300,80),120,3
            PAINT (300,80),3,3

            CIRCLE (300,80),100,1
            PAINT (300,80),1,1

            CIRCLE (300,146),180,3,,,1/15
            PAINT (300,146),1,3

            LOCATE 1,1
            PRINT "Taste druecken!"+SPACE$(40)

            WHILE INKEY$="":WEND

            WINDOW CLOSE 2
            SCREEN CLOSE 1


ende:       LIBRARY CLOSE
            END


SUB InitCop STATIC
      SHARED UCopList&
      opt&      = 2^0+2^1+2^16
```

```
      UCopList& = AllocMem&(12,opt&)
      IF UCopList&=0 THEN ERROR 7
END SUB

SUB ActiCop STATIC
      SHARED UCopList&
      waitC 10000,256
      viewport& = ViewPortAddress&(WINDOW(7))
      POKEL viewport&+20,UCopList&
      CALL RethinkDisplay
END SUB

SUB waitC(y%,x%) STATIC
      SHARED UCopList&
      CALL CWait(UCopList&,y%,x%)
      CALL CBump(UCopList&)
END SUB

SUB moveC(reg%,wert%) STATIC
      SHARED UCopList&
      CALL CMove(UCopList&,reg%,wert%)
      CALL CBump(UCopList&)
END SUB
```

Zunächst erscheint ein Text. Er macht die veränderten Umstände des Grafik-Displays deutlich. Imposant wird es anschließend. Eine sehr simple Grafik erscheint, die aber ob der Copper-Programmierung sehr faszinierend wirkt: Sie besteht aus mehr als 500 Farben, bei nur zwei Bitplanes. Schlagartig werden die Möglichkeiten des Coppers deutlich. Per Tastendruck gelangen Sie wieder in den Normalmodus zurück. Gleichzeitig verliert die Grafik ihre unbeschreibliche Ausstrahlung und entpuppt sich als eine Ansammlung von Füllobjekten.

4.6 Die Layers: Seele der Fenster

Wir wollen unser Bild des Grafiksystems weiter präzisieren. In der Rastport-Struktur (siehe Kapitel 3.6) fand sich ein Zeiger auf sogenannte "Layers". Bei diesen Layers handelt es sich um eine eigenständige Systemkomponente des Betriebssystems, die durch die Layers-Bibliothek repräsentiert wird.

Bleibt die Frage, was Layers sind. Schauen Sie einmal ganz genau auf Ihren Amiga-Monitor. Was sehen Sie? Nichts? Schalten

Sie ihn ein. Und jetzt? Noch immer keine Spur von Layers? Sie sehen wahrscheinlich die Layers vor lauter Fenstern nicht: Jedes Fenster ist im Grunde nichts weiter als ein Layer.

Genauso wie ein Screen nichts weiter ist als ein erweiterter Viewport, ist ein Window (Fenster) nichts weiter als ein erweitertes Layer. Die Layers erledigen den Großteil der Arbeit, die bei Windowing entsteht. Wenn ein Computer mit Fenstern arbeitet, entsteht immer ein Problem: Alles, was sich Ihnen auf dem Bildschirm präsentiert, ist in den Bitplanes der Bitmap gespeichert. Dazu zählt der Screen-Hintergrund sowie die Fenster. Im Idealfall. Normalerweise enthält das Display den Screen-Hintergrund sowie zahlreiche Bruchstücke verschiedener Fenster. Fenster überlappen sich, werden gänzlich von anderen verdeckt oder können sich frei entfalten. Sobald ein Fenster ein anderes überlappt, muß diese Tatsache registriert werden, denn an der überlappten Stelle teilen sich zwei Fensterteile dieselbe Bitmap. Die Layers sorgen dafür, daß der Teil des verdeckten Fensters an anderer Stelle im Speicher gespeichert wird. Erst, wenn das verdeckte Fenster (oder ein Teil davon) wieder ans Tageslicht kommt, kopiert das Layer den ehemals verdeckten Teil zurück in die Screen-Bitmap.

Bevor wir uns weiter mit dieser Theorie beschäftigen, fangen wir für Sie ein Layer ein und zeigen es Ihnen! Diese Aufgabe erledigt das folgende kleine Programm:

```
'########################################
'#
'# Programm: Ein Layer
'# Datum: 5.1.87
'# Autor: tob
'# Version: 1.0
'#
'########################################

' Ein einfaches Layer - die Grundlage eines jeden
' Fensters - wird erzeugt.

PRINT "Suche die .bmap-Dateien..."

'LAYERS-Bibliothek
DECLARE FUNCTION CreateUpFrontLayer& LIBRARY
```

```
'DeleteLayer()
'MoveLayer()

'GRAPHICS-Bibliothek
'Text()
'Move()

LIBRARY "graphics.library"
LIBRARY "layers.library"

initPars:     CLS
              scrAdd&          = PEEKL(WINDOW(7)+46)
              screenLayerInfo& = scrAdd&+224
              screenBitMap&    = scrAdd&+184
              x0%              = 10
              y0%              = 20
              x1%              = 400
              y1%              = 80
              yp%              = 1

damitMans:    'auch sehen kann
              CLS
              LINE (1,1)-(600,180),2,bf

LayerHer:     layer& = CreateUpFrontLayer&(screenLayerInfo&, screen
                       BitMap&,x0%,y0%,x1%,y1%,typ%,0)

wasDamitTun:  layerRast& = PEEKL(layer&+12)
              text$      = "Dies ist das Herz eines Fensters: Ein Layer!"
              CALL Move(layerRast&,3,8)
              CALL text(layerRast&,SADD(text$),LEN(text$))

bewegen:      dx% = 2
              dy% = 1
              FOR loop1%=1 TO 30
                CALL MoveLayer(screenLayerInfo&,layer&,dx%,dy%)
              NEXT loop1%

warten:       LOCATE 1,1
              PRINT "Beliebige Taste = ende!"
              WHILE in$=""
                in$=INKEY$
              WEND

wegDamit:     CALL DeleteLayer(screenLayerInfo&,layer&)

dasWars:      LIBRARY CLOSE
              END
```

Sehen Sie? Unser kleines Layer benimmt sich bereits fast wie ein "großes" Fenster: Wenn Sie mit der Maus auf das Layer fah-

ren und die linke Maustaste drücken, wird das Layer aktiviert, Ihr eigenes Fenster riffelt sich. Zu einem richtigen Fenster fehlt unserem Layer nur der Rahmen, die Gadgets und ein Menü.

Das Layer verschwindet restlos, sobald Sie eine beliebige Taste drücken.

Zur Realisierung des obigen Programms verwendeten wir Funktionen der Grafik- und der Layers-Bibliothek, wobei die zweite zweifelsohne die wichtigere ist. Die Layers-Funktion "CreateUpfrontLayer" generiert unser Layer. Sie fordert acht Argumente und liefert die Anfangsadresse des Layers-Datenblocks an das BASIC-Programm zurück:

```
layer&=CreateUpfrontLayer&(layerInfo&,bitmap&,x0%,yo%,x1%,y1%,typ%
,sbitmap&)
```

layer&:	Die Adresse unseres neuen Layerdatenblocks
layerInfo&:	Die Adresse der Struktur LayerInfo
bitmap&:	Die Adresse der Bitmap, in die das neue Layer projiziert werden soll
x0%,y0%:	Koordinaten der oberen linken Ecke des Layers
x1%,y1%:	Koordinaten der unteren rechten Ecke

Die Adresse der LayerInfo-Struktur findet sich in der uns bekannten Screen-Struktur (siehe Kapitel 3.4). Dasselbe gilt für die Adresse der Bitmap-Struktur.

Zurück zum Programm: Durch obige Funktion öffnet es ein Layer. Nun soll Text innerhalb des Layers erscheinen. Auch ein Layer besitzt einen Rastport (siehe Kapitel 3.6). Durch die Funktionen Text und Move der Grafik-Bibliothek (siehe Kapitel 3.6.1) wird Text in diesen Rastport ausgegeben.

Nachdem Sie das Layer lang genug bewundert haben, schließt es die Layers-Funktion "DeleteLayer" wieder.

Kommen wir nachträglich zur Layers-Datenstruktur. Wie jedes Fenster besitzt auch ein jedes Layer eine solche Struktur. Sie ist folgendermaßen aufgebaut:

Datenstruktur "Layer"/layers/ 192 Bytes

Offset	Typ	Bezeichnung
+ 000	Long	Zeiger auf Layer im Vordergrund
+ 004	Long	Zeiger auf Layer im Hintergrund
+ 008	Long	Zeiger auf erstes ClipRect
+ 012	Long	Zeiger auf den Rastport des Layers
+ 016	----	Rectangle-Struktur, die Grenzen des Layer
		+ 16 Word MinX
		+ 18 Word MinY
		+ 20 Word MaxX
		+ 22 Word MaxY
+ 024	Byte	Lock
+ 025	Byte	LockCount
+ 026	Byte	LayerLockCount
+ 027	Byte	reserviert
+ 028	Word	reserviert
+ 030	Word	Layer-Flags
+ 032	Long	Zeiger auf Superbitmap, falls vorhanden
+ 036	Long	SuperClipRect
+ 040	Long	Zeiger auf Fenster
+ 044	Word	ScrollX
+ 046	Word	ScrollY
+ 048	----	Message Port "LockPort"
+ 082	----	Message "LockMessage"
+ 102	----	Message Port "ReplyPort"
+ 136	----	Message "1 LockMessage"
+ 156	Long	Zeiger auf erstes Rectangle der Damagelist
+ 160	Long	Zeiger auf ClipRects
+ 164	Long	Zeiger auf LayerInfo-Struktur
+ 168	Long	Zeiger auf Task mit aktuellem Lock
+ 172	Long	Zeiger auf SuperSaveClipRects
+ 176	Long	Zeiger auf CR ClipRects
+ 180	Long	Zeiger auf CR2 ClipRects
+ 184	Long	Zeiger auf CRNEW ClipRects
+ 188	Long	System-Use

4.6.1 Kommentierte Datenstruktur

Die eben präsentierte Datenstruktur "Layer" bedarf unbedingt der weiteren Erläuterung. In gewohnter Weise gehen wir Feld für Feld mit Ihnen durch.

Die Anfangsadresse des Layers Ihres aktuellen Ausgabefensters ermitteln Sie so:

```
layer&=PEEKL(WINDOW(8))
```

GFA:

```
OPENW 0
layer%=LPEEK(LPEEK(WINDOW(0)+50))
```

Die Anfangsadresse auf diese Datenstruktur bei einem selbsterzeugten Layer wird Ihnen von den entsprechenden Layer-Funktionen automatisch zurückgeliefert. Darauf kommen wir im späteren Verlauf aber noch zurück.

Offset 0 und 4: Zeiger auf andere Layers

Hier finden Sie die Anfangsadressen der Layer-Datenblöcke der Layer, die vor oder hinter Ihrem eigenen Layer liegen. Ganz analog zu den Fenstern können Sie auch von einem beliebigen Layer zu allen anderen im System gelangen.

Offset 8: Erstes ClipRect

ClipRect ist eine weitere Datenstruktur. Ein ClipRect beschreibt jeweils einen rechteckigen Ausschnitt eines Layers. Hier findet sich die Adresse auf die erste ClipRect-Struktur dieses Layers. Von dort findet sich das nächste ClipRect und so fort. Diese Kette von ClipRects beschreibt den sichtbaren Teil dieses Layers.

Offset 12: Der Rastport

Hier findet sich die Anfangsadresse des Rastports für dieses Layer. Die meisten Funktionen der Grafik-Bibliothek verlangen die Adresse des Rastports, in dem sie ausgeführt werden sollen.

Da jedes Intuition-Fenster ein Layer besitzt, hat es auch einen eigenen Layer-Rastport. Dieser Rastport ist identisch mit dem der Fenster-Datenstruktur:

für AmigaBASIC:

```
rastport1&=PEEKL(WINDOW(7)+50)
rastport2&=WINDOW(8)
layer&=PEEKL(WINDOW(8))
rastport3&=PEEKL(layer&+12)
PRINT rastport1&
PRINT rastport2&
PRINT rastport3&
```

Die gelieferten Anfangsadressen der Rastports sind identisch.

Offset 16, 18, 20, 22: Bounds

"Bound" ist ein englisches Wort und steht für "Grenze". Die hier gelagerten X- und Y-Werte legen die Grenzen dieses Layers fest. Wann immer eine Zeichenfunktion mit Koordinaten arbeitet, die außerhalb dieser Grenzen liegen, wird die Zeichnung abgeschnitten, sobald die Grenz-Koordinaten überschritten werden.

Offset 24, 25 und 26: Lock-Felder

Der Amiga ist ein Multi-Tasking-Computer. Das bedeutet, mehrere Programme können quasi gleichzeitig ablaufen. Daher kann es vorkommen, daß mehrere Programme gleichzeitig auf ein Layer zugreifen wollen und sich dabei unweigerlich ins Gehege kommen würden. Deshalb gibt es das Lock. Mit Hilfe der Layer-Funktionen "LockLayer" und "UnlockLayer" können sich Tasks uneingeschränkten Zugang zu Layers verschaffen. Solange ein Task das Lock innehat, kann kein anderer Task den Inhalt der Layer-Datenstruktur verändern.

Diese Felder verwalten die Lock-Technik. Das erste Feld gibt Auskunft, ob dieses Layer gerade "gelockt" ist, das zweite ist ein

Zähler für das besitzergreifende Programm, das dritte zählt die Interessenten, die sich der Reihe nach angemeldet haben, exklusive Zugriffsrechte zu bekommen.

Offset 30: Flags

Es gibt verschiedene Layer-Typen, die wir gleich eingehend behandeln werden. Dieses Feld enthält das Erkennungsflag dieses Layers:

Bit 0: 1=Layersimple
Bit 1: 1=Layersmart
Bit 2: 1=Layersuper
Bit 6: 1=Layerbackdrop
Bit 7: 1=Layerrefresh

Offset 32: Superbitmap

Im Falle des Layer-Modus' "Layersuper" besitzt das Layer eine gänzlich eigene Zeichenfläche, eine eigene Bitmap. Der Zeiger auf diese findet sich hier. Wir werden das gleich ausführlich behandeln.

Offset 36: SuperClipRect

Hier finden sich die ClipRects für die Superbitmap, falls eine vorhanden ist (siehe Offset 8).

Offset 40: Fenster

Normalerweise treten Layers in Verbindung mit Intuition-Fenstern in Erscheinung. Ist dies der Fall, dann findet sich hier die Adresse der entsprechenden Fenster-Datenstruktur.

Dieses Feld ist von unglaublicher Wichtigkeit, wenn man Layers in bestehende Fenster integrieren will. Wir werden darauf aber gleich zu sprechen kommen.

Offset 44 und 46: Scrolling

Im Falle eines Layers des Typs "Layersuper" kann die Zeichenfläche, die das Layer repräsentiert, viel größer sein als die Abmessungen des Layers. Man kann dann das Layer quasi als Auge benutzen, mit dem man über eine riesige Grafik fährt. Mehr darüber gleich.

Offset 48 - 136: Messages und Message Ports

Messages und Message Ports werden von der Exec-Bibliothek gehandhabt. Wir werden nicht weiter darauf eingehen, denn diese Komponenten haben nichts mit Grafik zu tun. Es sei nur soviel verraten: Mit Hilfe von Messages und Message Ports können verschiedene Tasks miteinander kommunizieren. Dabei ist ein Message Port eine Art Briefkasten und Sendestation, Messages sind die versandten Briefe. Der Reply Port ist der Empfangsbriefkasten, der andere Message Port versendet Messages.

Offset 156: Damage List

Wir erwähnten bereits, daß es die Aufgabe der Layers ist, verdeckte Fensterbereiche wiederherzustellen, sobald sie nicht mehr von anderen Fenstern verdeckt sind. Dazu gibt es eine sogenannte Damage List. Sie besteht aus einer Kette von Datenstrukturen namens "Region". Regionen beschreiben rechteckige Teilbereiche des Layers. Die Damagelist enthält die durch andere Fenster (oder Layers) beschädigten Teile des eigenen Fensters (bzw. Layers).

Die übrigen Offset-Felder enthalten System-Informationen, mit denen BASIC nichts anfangen kann.

4.7 Die verschiedenen Layer-Typen

Insgesamt kennt der Amiga vier verschiedene Layertypen:

Layersimple
Layersmart
Layersuper
Layerbackdrop

Diese Modi charakterisieren die Art und Weise, wie zeitweise verdeckte Teile eines Layers behandelt werden sollen:

Simple Refresh (Layersimple)

Jedesmal, wenn ein Teil dieses Layers sichtbar wird (also im Vordergrund liegende Fenster verschwunden sind oder verschoben wurden), wird das Programm, das dieses Layer aufrief, damit beauftragt, die sichtbar gewordene Region des Layers erneut zu zeichnen. Ein Layer dieses Typs speichert also nicht automatisch verdeckte Teile ab, um ein "beschädigtes" Layer später wieder zu reparieren. Das ist Aufgabe des Programms, in diesem Fall also Ihre Aufgabe.

Layers dieses Typs sind schnell und benötigen wenig Speicher. Sie sind aber arbeitsintensiv, denn ihr Inhalt muß jedesmal von neuem gezeichnet werden, wenn ein anderes Layer dieses überdeckt hatte.

Smart Refresh (Layersmart)

Werden Teile dieses Layers verdeckt, dann richtet das System automatisch einen Zwischenspeicher ein, in dem die verdeckten Teile zwischengespeichert werden. Wird das Layer wieder freigelegt, dann werden diese zwischengespeicherten Teile automatisch wieder an ihren angestammten Platz zurücktransferiert.

Superbitmap (Layersuper)

Das Layer ist mit eigenen Bitplanes ausgerüstet, in denen zu jeder Zeit der gesamte Inhalt des Layers abgelegt ist. Der Teil des Layers, der momentan auf dem Screen sichtbar ist, wird in die allgemeine Screen-Bitmap kopiert.

Es ist möglich, eine Layer-Bitmap einzurichten, die (viel) größer ist als das Layer selbst. Sie kann bis zu 1024 x 1024 Punkte groß sein. Mit dieser Riesenfläche kann dann problemlos gescrollt werden.

Backdrop (Layerbackdrop)

Ein Backdrop-Layer liegt immer hinter allen anderen existierenden Layern.

Wir werden Ihnen nun einen Einblick in die Welt der Layers geben und Ihnen zeigen, was sich mit ihrer Hilfe bewerkstelligen läßt:

4.7.1 Simple Layers: Die Eigenbau-Requester

Simple Layers eignen sich ganz hervorragend für die folgende Aufgabe: Requester. Mit Hilfe von Requestern will man den Anwender auf besondere Ereignisse aufmerksam machen: Eine Diskette soll ins Laufwerk eingelegt werden, eine Grafik wird geladen etc. Das folgende Programm bastelt mit Hilfe von Simple-Layers Eigenbau-Requester. Sie rufen einen Request auf mittels:

```
Request nr%,x%,y%,text$
```

nr%:	Nummer des Requests (0-10)
x%:	X-Koordinate der linken Ecke des Requesters
y%:	Y-Koordinate der obern Ecke des Requesters
text$:	Text für den Requester

```
'#####################################
'#
'# Programm: Ein Layer - eigener Requester
'# Datum: 5.1.87
'# Autor: tob
'# Version: 1.0
'#
'#####################################

PRINT "Suche die .bmap-Dateien..."

' Demonstriert die Anwendung von Layers

'LAYERS-Bibliothek
DECLARE FUNCTION CreateUpFrontLayer& LIBRARY
'DeleteLayer()

'GRAPHICS-Bibliothek
'Draw()
'Move()
'Text()

LIBRARY "graphics.library"
LIBRARY "layers.library"

variablen:   DIM SHARED layer&(10)

init:       'Hintergrund
             CLS
             FOR loop%=1 TO 15
               PRINT STRING$(80,"#")
             NEXT loop%

main:        Request 1,80,40,"Request Nr. 1"
             Request 2,50,50,"Request 2: Dies sind Layers!"
             FOR t%=1 TO 30000:NEXT t%
             CloseRequest 1
             Request 1,30,30,"Beliebig positionierbar"
             FOR t%=1 TO 30000:NEXT t%
             CloseRequest 2
             CloseRequest 1
             Request 1,200,100,"Das war's."
             FOR t%=1 TO 2000:NEXT t%
             CloseRequest 1

dasWars:     LIBRARY CLOSE
             END

SUB Request(nr%,x0%,y0%,text$) STATIC
        SHARED screenLayerInfo&
        IF layer&(nr%)<>0 THEN EXIT SUB
        scrAdd&          = PEEKL(WINDOW(7)+46)
```

```
        screenLayerInfo& = scrAdd&+224
        screenBitMap&    = scrAdd&+184
        x1%              = (LEN(text$)+2)*8-8
        y1%              = 12
        layer&(nr%) = CreateUpFrontLayer&(screenLayerInfo&,
                        screenBitMap&,x0%,y0%,x0%+x1%,y0%+y1%,typ%,0)
        layerRast&  = PEEKL(layer&(nr%)+12)
        CALL Draw(layerRast&,x1%,0)
        CALL Draw(layerRast&,x1%,y1%)
        CALL Draw(layerRast&,0,y1%)
        CALL Draw(layerRast&,0,0)
        CALL Move(layerRast&,3,9)
        CALL text(layerRast&,SADD(text$),LEN(text$))
END SUB
SUB CloseRequest(nr%) STATIC
        SHARED screenLayerInfo&
        IF layer&(nr%) = 0 THEN EXIT SUB
        CALL DeleteLayer(screenLayerInfo&,layer&(nr%))
        layer&(nr%) = 0
END SUB
```

Sie können nun bis zu 11 Requester gleichzeitig öffnen (leicht läßt sich diese Zahl erhöhen, aber sagen Sie selbst: Sind elf Requester gleichzeitig nicht genug?). Die X- und Y-Koordinaten der linken oberen Ecke eines jeden Requesters verstehen sich relativ zur linken oberen Ecke des Screens, nicht Ihres Fensters. Ihre Requester können demnach überall auftauchen, nicht bloß innerhalb Ihres Fensters. Außerhalb können sie jedoch kurzzeitig kleine Schäden anrichten, denn dort wird die Damagelist nicht aktiviert.

Der Befehl "CloseRequest" schließt den Requester (und damit das Layer) wieder.

GFA-BASIC macht es seinen Benutzern wieder leicht: Hier benötigt man gar keine eigens kreierten Layer, um Requester erscheinen zu lassen. Eine viel einfacherere und viel komfortablere Programmierung ermöglicht der GFA-Befehl ALERT, mit dem in nur einer einzigen Programmzeile ein kompletter Requester erschaffen und verwaltet werden kann:

```
ALERT 0,"Ein einziger|Befehl machts möglich!",0," Toll ",a%
```

4.7.2 Mit dem Superlayer 1024 x 1024 Punkte!

Kommen wir nun zu einem ganz besonderen Layer-Typ: dem Superlayer. Anders als alle anderen Typen ist dieses Layer mit einem völlig eigenen Grafikspeicher ausgerüstet. Dieser Grafikspeicher kann zudem größer sein als der tatsächlich auf dem Bildschirm erscheinende Teil. Insgesamt kann ein solches Layer eine bis zu 1024 x 1024 Punkte große Zeichenfläche verwalten.

Es ist gar kein Problem, ein solches Layer zu generieren: Aus dem vorangegangenen Beispiel kennen Sie den "CreateUpfront-Layer"-Befehl der Layer-Bibliothek. Ein viel größeres Problem ist es, wie wir das Layer für uns nutzbar machen.

Da wäre zunächst einmal die Positionierung des neuen Layers. Die beste Methode ist hier, das Layer in ein bereits bestehendes Fenster "einzupflanzen". Dazu wählt man ein Layer, dessen Abmessungen denen des Fensters entsprechen und legt anschließend das Layer genau auf das Fenster. So bemerkt niemand etwas von dem Trick. Wir werden das ausprobieren. Hier das Programm:

```
'########################################
'#
'# Programm: Superbitmap
'# Datum: 4.1.87
'# Autor: tob
'# Version: 1.0
'#
'########################################

' Zeigt, wie bis zu 1024x1024-Punkte grosse Layers
' erzeugt, programmiert und gescrollt werden. Erste
' Demo.

'LAYERS-Bibliothek
DECLARE FUNCTION CreateUpFrontLayer& LIBRARY
'DeleteLayer()
'ScrollLayer()

'GRAPHICS-Bibliothek
DECLARE FUNCTION AllocRaster& LIBRARY
'FreeRaster()
'SetRast()
'Move()
'Draw()
```

```
'WaitTOF()
'Text()

'EXEC-Bibliothek
DECLARE FUNCTION AllocMem& LIBRARY
'FreeMem()

'INTUITION-Bibliothek
'SetWindowTitles()

PRINT "Suche die .bmap-Dateien... ";

LIBRARY "layers.library"
LIBRARY "graphics.library"
LIBRARY "exec.library"
LIBRARY "intuition.library"

PRINT "...gefunden. Es geht los."

initPar:    '* Screen Parameter
            scrWeite% = 320
            scrHoehe% = 256
            scrTiefe% = 1
            scrMode%  = 1
            scrNr%    = 1

           '* Fenster Parameter
            windWeite% = scrWeite%-9
            windHoehe% = scrHoehe%-26
            windNr%    = 1
            windTitle$ = "Arbeitsflaeche"
            windMode%  = 0

           '* Super Bitmap
            superWeite% = 800
            superHoehe% = 400
            superFlag%  = 4

initDisp:  '* Screen und Fenster oeffnen
            SCREEN scrNr%,scrWeite%,scrHoehe%,scrMode%,scrTiefe%
            WINDOW windNr%,windTitle$,(0,0)(windWeite%,windHoehe%),
                   windMode%,scrNr%
            WINDOW OUTPUT windNr%
            PALETTE 1,0,0,0
            PALETTE 0,1,1,1

           '* Layer Groesse
            windLayer& = PEEKL(WINDOW(8))
            LayMinX%   = PEEKW(windLayer&+16)
            layMinY%   = PEEKW(windLayer&+18)
            layMaxX%   = PEEKW(windLayer&+20)
            layMaxY%   = PEEKW(windLayer&+22)
```

```
initSys:    '* System-Parameter lesen
            windAdd&       = WINDOW(7)
            scrAdd&        = PEEKL(windAdd&+46)
            scrBitMap&     = scrAdd&+184
            scrLayerInfo&  = scrAdd&+224

initSBMap:  '* Superbitmap schaffen
            opt&          = 2^0+2^1+2^16
            superBitMap&  = AllocMem&(40,opt&)
            IF superBitMap& = 0 THEN
              PRINT "Hm. Nicht mal 40 Bytes, nein?"
              ERROR 7
            END IF

            '* ...und in Betrieb nehmen
            CALL InitBitMap(superBitMap&,scrTiefe%,superWeite%,super
                 Hoehe%)
            superPlane&    = AllocRaster&(superWeite%,superHoehe%)
            IF superPlane& = 0 THEN
              PRINT "Kein Plaaaaatz!"
              CALL FreeMem(superBitMap&,40)
              ERROR 7
            END IF
            POKEL superBitMap&+8,superPlane&

            '* Superbitmap-Layer oeffnen
            superLayer&=CreateUpFrontLayer&(scrLayerInfo&, scrBitMap&,
            LayMinX%,layMinY%,layMaxX%,layMaxY%,superFlag%,superBitMap&)
            IF superLayer&=0 THEN
              PRINT "Heute keine Layer!"
              CALL FreeRaster(superPlane&,superWeite%,superHoehe%)
              CALL FreeMem(superBitMap&,40)
              ERROR 7
            END IF

            '* naechste Zeile vorerst nicht beachten!
            '**********HIER ERWEITERUNG EINFUEGEN**********

            '* neuer RastPort          run
            superRast& = PEEKL(superLayer&+12)

prepare:    '* Zeichenflaeche vorbereiten
            CALL SetRast(superRast&,0)

            CALL Move(superRast&,0,0)
            CALL Draw(superRast&,superWeite%,superHoehe%)

            CALL Move(superRast&,0,10)
            text1$="Cursortasten = Scrolling"
            CALL Text(superRast&,SADD(text1$),LEN(text1$))
```

```
            CALL Move(superRast&,0,30)
            text2$="'S'-Taste = Abbruch"
            CALL Text(superRast&,SADD(text2$),LEN(text2$))

           '* Koordinaten
            POKEW superRast&+34,&HAAAA
            FOR loop%=0 TO superWeite% STEP 50
              CALL Move(superRast&,loop%,0)
              CALL Draw(superRast&,loop%,superHoehe%)
            NEXT loop%
            FOR loop%=0 TO superHoehe% STEP 50
              CALL Move(superRast&,0,loop%)
              CALL Draw(superRast&,superWeite%,loop%)
            NEXT loop%
            POKEW superRast&+34,&HFFFF

doScroll:   '* Kontrolliere Scrolling
            WHILE in$<>"S"
              in$ = UCASE$(INKEY$)
              y%  = 0
              x%  = 0
              IF in$ = CHR$(30) THEN '<-
                IF ox%<(superWeite%-layMaxX%+LayMinX%-1) THEN
                  x%  = 1
                  ox% = ox%+1
                END IF
              ELSEIF in$=CHR$(31) THEN '->
                IF ox%>0 THEN
                  x%  = -1
                  ox% = ox%-1
                END IF
              ELSEIF in$=CHR$(29) THEN 'up
                IF oy%<(superHoehe%-layMaxY%+layMinY%-1) THEN
                  y%  = 1
                  oy% = oy%+1
                END IF
              ELSEIF in$=CHR$(28) THEN 'down
                IF oy%>0 THEN
                  y%  = -1
                  oy% = oy%-1
                END IF
              END IF
              IF in$<>"" THEN
                CALL ScrollLayer(scrLayerInfo&,superLayer&,x%,y%)
                actu$ = windTitle$+" [X]="+STR$(ox%)+" [Y]="+STR$
                        (oy%)+CHR$(0)
                CALL WaitTOF
                CALL SetWindowTitles(windAdd&,SADD(actu$),0)
              END IF
            WEND

deleteSys:  '* System entfernen
```

```
CALL DeleteLayer(scrLayerInfo&,superLayer&)
CALL FreeRaster(superPlane&,superWeite%,superHoehe%)
CALL FreeMem(superBitMap&,40)
SCREEN CLOSE scrNr%
WINDOW windNr%,"hi!",,,-1
LIBRARY CLOSE
END
```

Gleich nach dem Start sehen Sie ein Fenster namens "Arbeitsfläche". Es enthält ein Raster sowie eine nach unten gerichtete Diagonale. Was Sie da sehen, ist ein Superbitmap-Layer, das es sich in unserem Fenster bequem gemacht hat. Diese Behauptung ist leicht zu beweisen: Fahren Sie einmal mit der Maus auf die Rasterfläche, und drücken Sie die linke Maustaste. Sofort riffelt sich die Kopfleiste unseres Fensters: Sie haben mit diesem Mausdruck das unscheinbar vor unserem Fenster liegende Layer aktiviert. Ein Mausdruck auf die Kopfleiste unseres Fensters, und alles ist wieder beim alten.

Wir hatten bereits mehrmals erwähnt, daß Superbitmap-Layers einen viel größeren Bereich kontrollieren können als auf den Bildschirm paßt. Das haben wir in unserer Demo ausgenutzt. Drücken Sie einmal eine der Cursor-(Pfeil-)-Tasten links neben dem Zahlenblock der Tastatur. Mit ihrer Hilfe können Sie die Position unseres Layers verschieben und so einen anderen Teil der durch das Layer kontrollierten Zeichnung sehen.

Wenn Sie von diesem Programm genug gesehen haben, drücken Sie bitte auf die "S"-Taste (für Stop). Sofort gelangen Sie zu Ihrem alten Display zurück (Drücken Sie niemals CTRL-C, also BREAK, weil dann das Superbitmap-Layer nicht verschwinden würde).

Kommen wir zur Realisierung dieses Projektes. Wir benutzen in diesem Programm die Funktionen der Layers-, Grafik-, Exec- und Intuition-Bibliothek. Von besonderer Wichtigkeit sind diese Funktionen:

CreateUpfrontLayer()
AllocRaster()

AllocMem()
((s1p10v0b5TScrollLayer()

Des weiteren benutzen wir:

InitBitMap()
SetRast()
Move()
Draw()
WaitTOF()
SetWindowTitles()

und natürlich:

DeleteLayer()
FreeRaster()
FreeMem()

Nun zum Programm: Es wird zunächst ein Screen der Tiefe 1 geöffnet. Tiefe 1 entspricht einer Bitplane, also maximal zwei Farben. Wir wählen diese Tiefe, weil unser Superbitmap-Layer ebenso viele Speicherebenen benötigt wie der Screen, in dem es auftaucht, tief ist. Da die Speicherebenen des Superbitmaps sehr speicherintensiv sind, können wir es uns nur leisten, eine einzige Ebene einzurichten.

Unsere Superbitmap soll 800 Punkte weit und 400 Punkte hoch werden.

Nachdem Fenster und Screen geöffnet sind, wird die Größe des Layers festgelegt. Bei dieser Größe handelt es sich um die Größe des Layers auf dem Bildschirm, nicht aber um die Größe der Zeichenebene des Layers. Da das Layer den gesamten Fensterinhalt ausfüllen soll, erfragen wir die entsprechenden Parameter aus dem bereits existierenden Layer unseres Fensters (siehe Kapitel 4.5, Offsets 16 - 22).

Bevor wir nun endlich mittels "CreateUpfrontLayer" das Layer zum Leben erwecken können, müssen wir die private Bitmap

unseres Layers beschaffen. Das ist nur bei Layers des Typs "Layersuper" notwendig; alle anderen Layer bekommen automatisch ihren Speicher. Wir gehen dazu ganz analog zu unserem Primitiv-Display aus Kapitel 4.4 vor: Eine 40 Bytes große Bitmap-Struktur wird eingerichtet und mittels "InitBitMap" gebrauchsfertig gemacht. Anschließend besorgen wir uns mit Hilfe der Grafik-Funktion "AllocRaster" eine zusätzliche Bitplane. Diese Funktion verlangt die X- und Y-Dimension der Bitplane in Pixel und liefert einen Zeiger auf die Anfangsadresse der neuen Plane, wenn soviel Speicherplatz vorhanden ist.

Nachdem die Bitplane in unsere neue Bitmap-Struktur eingebunden wurde, kann endlich der Aufruf "CreateUpfrontLayer" erfolgen. Die Variable superflag% beinhaltet den Wert 4 (=Superlayer), außerdem wird erstmals die Adresse einer (unserer neuen) Bitmap-Struktur mitgeliefert.

Sobald sich das Layer erfolgreich geöffnet hat, soll etwas in seinem Inneren erscheinen. Dazu löschen wir mittels der Grafik-Funktion SetRast seinen Inhalt und zeichnen eine Diagonale mit Hilfe des Draw-Befehls der Grafik-Bibliothek.

Der Programmteil "doScroll" verwaltet das Scrolling (das Verschieben) der Superbitmap auf Druck der Cursortasten. Dazu dient die Layer-Funktion ScrollLayer). Sie verlangt vier Parameter:

```
ScrollLayer(layerinfo&,layer&,x%,y%)
```

layerinfo&:	Adresse der Layerinfo-Struktur (siehe "Screen")
layer&:	Adresse auf unser neues Superlayer
x%,y%:	Anzahl der Pixel, um die der Inhalt des Layers gescrollt werden soll (negative Werte = umgekehrte Richtung)

Nach jedem Scrolling wird via Intuition-Funktion SetWindowTitles die augenblickliche X- und Y-Position in der Kopfzeile des Fensters ausgegeben. Die Funktion WaitTOF entstammt der Grafik-Bibliothek. "TOF" steht für "Top Of Frame". Diese Funktion wartet darauf, daß der Elektronenstrahl die oberste Display-Zeile erreicht. Dadurch wird verhindert, daß die

Fenster-Kopfzeile verändert wird, während der Elektronenstrahl über sie hinwegsaust - denn das hätte ein unschönes Flackern zur Folge.

Wurde die "S"-Taste gedrückt, dann wird zunächst das Superlayer dicht gemacht. Anschließend wird die von uns erzeugte Bitplane und danach die Bitmap-Struktur ans System zurückgegeben.

Als erster Test hat sich das Programm bewährt. Aber unsere Programmiertechnik ist noch unzureichend, denn es gibt ein paar schwerwiegende Probleme:

a) Wenn der Anwender zufällig mit seiner Maus das Layer anklickt, wird dieses aktiv, das eigene Fenster wird deaktiviert. Das hat zur Folge, daß das eigene Programm keine Tastatur- oder Mauseingaben mehr registrieren kann.

b) Dadurch, daß wir das Superlayer direkt aus dem System generieren, haben wir keine Möglichkeit, mit Hilfe der BASIC-Zeichenbefehle etwas in das Superlayer zu zeichnen. Statt dessen müssen wir umständliche Funktionen der Grafik-Bibliothek bemühen.

Damit sich Superbitmap-Layer nutzen lassen, müssen diese Probleme gelöst werden. Das soll nun geschehen.

4.7.3 Permanente Deaktivierung des Layers

Oder fällt Ihnen eine bessere Überschrift ein? Wir werden uns hier des Problems a) annehmen. Es muß verhindert werden, daß sich das Layer mit der Maus anklicken und aktivieren läßt. Unser Ziel ist, daß sich das Layer von der Maus nicht beeinflussen läßt, unser eigenes Fenster also permanent aktiv bleibt.

Ein Blick in das Grafiksystem des Amiga, und die Antwort ist gefunden: In jeder Layer-Struktur existiert ab Offset 40 ein Feld namens "Zeiger auf Fenster". Bei einfachen Layern ist dieses

Feld Null. Anders bei Layern, die von einem Intuition-Fenster benutzt werden. Dort enthält dieses Feld einen Zeiger auf die Fenster-Datenstruktur des Fensters, das dieses Layer benutzt. Einziger Zweck dieses Zeigers ist es, Intuition mitzuteilen, wenn der Benutzer per Mausklick dieses Layer aktiviert hat.

Wir wollen verhindern, daß Intuition unser eigenes Fenster deaktiviert, sobald das Layer aktiv wird. Also müssen wir in das Datenfeld unseres Layers die Adresse der Fenster-Struktur schreiben, die aktiv bleiben soll. Das geschieht durch folgende Zeile:

```
POKEL layer&+40,WINDOW(7)
```

Sie können diese Technik gleich an unserem Demoprogramm aus Kapitel 4.7.2 ausprobieren. Fügen Sie dazu die folgende Zeile an die mit "HIER ERWEITERUNG EINFÜGEN" markierte Stelle des Listings ein:

```
POKEL superLayer&+40,WINDOW(7)
```

Nach dem Start können Sie mit der Maus beliebig auf dem Layer herumfahren und die linke Maustaste drücken - unser Fenster bleibt aktiviert.

Damit wäre das erste Problem gelöst. Kommen wir zur Lösung des zweiten:

4.7.4 Verwendung der BASIC-Befehle innerhalb eines Layers

Analysieren wir zunächst das Problem: AmigaBASIC-Grafikbefehle wie LINE, CIRCLE oder auch PRINT können nicht in unser Layer zeichnen, denn es gibt keine Möglichkeit, die Ausgabe dort hinzuleiten. Wir müssen also einen kleinen Systemeingriff vornehmen.

Es ist durch geschickte Zeigermanipulation möglich, die Grafikausgabe vom eigenen Fenster in ein Layer zu transferieren.

Dabei ist es jedoch von großer Wichtigkeit, daß die alten Zustände wiederhergestellt werden, bevor das Layer geschlossen wird. Andernfalls kommt das System ins Schleudern und hängt sich auf bzw. holt den Guru.

In der Praxis sieht die Technik so aus:

(nachdem das Layer geöffnet ist...)

```
backupRast&=PEEKL(layer&+12)
'* Rastport des Layers retten

backupLayer&=PEEKL(WINDOW(8))
'* Layer des Fensters retten

POKEL WINDOW(8),layer&
POKEL layer&+12,WINDOW(8)
```

Jetzt werden alle Grafikbefehle des AmigaBASIC innerhalb des Layers ausgeführt. Achtung: Sie können getrost alle BASIC-Befehle verwenden, mit Ausnahme jeglicher Fill-Kommandos (wie z.B. PAINT, LINE ()-(),,bf). Der Grund dafür liegt in einer Datenstruktur namens "TmpRas", die im Rastport zu finden ist. Für Fill-Befehle muß sie auf einen Speicherbereich zeigen, der mindestens so groß ist wie eine Bitplane des Layers. Es ist kein Problem, diese TmpRas-Struktur mit mehr Speicherplatz auszurüsten, um danach mit den Fill-Befehlen arbeiten zu können. Das würde aber soviel Speicher kosten, daß es sich nicht mehr lohnt. In unserem Fall kämen 40.000 Bytes zusammen. Für die Leser unter Ihnen, die über genügend Speicherplatz verfügen, folgt aber trotz allem an späterer Stelle eine Möglichkeit, die TmpRas-Struktur umzumodeln.

```
Die Ausgabe wird durch diese Zeilen zurück zum eigenen Fenster
geleitet:
POKEL WINDOW(8),backupLayer&
POKEL layer&+12,backupRast&
```

Wir werden das erlangte Wissen gleich einsetzen, und zwar in unserem bereits bekannten Demo-Programm:

```
'#####################################
'#
'# Programm: Superbitmap mit BASIC Grafik
'#           Befehlen
'# Datum: 12.4.87
'# Autor: tob
'# Version: 1.0
'#
'#####################################

' Ermoeglicht, die AmigaBASIC Grafik-Befehle auch
' im SuperBitmap-Layer anwenden zu koennen.

PRINT "Suche die .bmap-Dateien..."

'LAYERS-Bibliothek
DECLARE FUNCTION CreateUpFrontLayer& LIBRARY
'DeleteLayer()
'ScrollLayer()

'GRAPHICS-Bibliothek
DECLARE FUNCTION AllocRaster& LIBRARY
'FreeRaster()
'SetRast()
'Move()
'Draw()
'WaitTOF()

'EXEC-Bibliothek
DECLARE FUNCTION AllocMem& LIBRARY
'FreeMem()

'INTUITION-Bibliothek
'SetWindowTitles()

LIBRARY "layers.library"
LIBRARY "graphics.library"
LIBRARY "exec.library"
LIBRARY "intuition.library"


initPar:    '* Screen Parameter
            scrWeite% = 320
            scrHoehe% = 256
            scrTiefe% = 1
            scrMode%  = 1
            scrNr%    = 1

           '* Fenster Parameter
            windWeite% = scrWeite%-9
            windHoehe% = scrHoehe%-26
            windNr%    = 1
```

```
            windTitle$ = "Arbeitsflaeche"
            windMode%  = 0

           '* Super Bitmap
            superWeite% = 800
            superHoehe% = 400
            superFlag%  = 4

initDisp:  '* Screen und Fenster oeffnen
            SCREEN scrNr%,scrWeite%,scrHoehe%,scrMode%,scrTiefe%
            WINDOW windNr%,windTitle$,(0,0)(windWeite%,windHoehe%),
                   windMode%,scrNr%
            WINDOW OUTPUT windNr%
            PALETTE 1,0,0,0
            PALETTE 0,1,1,1

           '* Layer Groesse
            windLayer& = PEEKL(WINDOW(8))
            LayMinX%   = PEEKW(windLayer&+16)
            layMinY%   = PEEKW(windLayer&+18)
            layMaxX%   = PEEKW(windLayer&+20)
            layMaxY%   = PEEKW(windLayer&+22)

initSys:   '* System-Parameter lesen
            windAdd&      = WINDOW(7)
            scrAdd&       = PEEKL(windAdd&+46)
            scrBitMap&    = scrAdd&+184
            scrLayerInfo& = scrAdd&+224

initSBMap: '* Superbitmap schaffen
            opt&         = 2^0+2^1+2^16
            superBitMap& = AllocMem&(40,opt&)
            IF superBitMap& = 0 THEN
              PRINT "Hm. Nicht mal 40 Bytes, nein?"
              ERROR 7
            END IF

           '* ...und in Betrieb nehmen
            CALL InitBitMap(superBitMap&,scrTiefe%,superWeite%,super-
Hoehe%)
            superPlane&    = AllocRaster&(superWeite%,superHoehe%)
            IF superPlane& = 0 THEN
              PRINT "Kein Plaaaaatz!"
              CALL FreeMem(superBitMap&,40)
              ERROR 7
            END IF
            POKEL superBitMap&+8,superPlane&

           '* Superbitmap-Layer oeffnen
           superLayer&=CreateUpFrontLayer&(scrLayerInfo&,scrBitMap&,
            LayMinX%,layMinY%,layMaxX%,layMaxY%,superFlag%,superBitMap&)
            IF superLayer&=0 THEN
```

```
            PRINT "Heute keine Layer!"
            CALL FreeRaster(superPlane&,superWeite%,superHoehe%)
            CALL FreeMem(superBitMap&,40)
            ERROR 7
          END IF

          '* naechste Zeile vorerst nicht beachten!
          '**********HIER ERWEITERUNG EINFUEGEN**********

          '* neuer RastPort
          superRast& = PEEKL(superLayer&+12)

prepare:  '* Zeichenflaeche vorbereiten
          CALL SetRast(superRast&,0)

          '* Layer aktivieren
          POKEL superLayer&+40,WINDOW(7)
          backup.rast&  = PEEKL(superLayer&+12)
          backup.layer& = PEEKL(WINDOW(8))
          POKEL superLayer&+12,WINDOW(8)
          POKEL WINDOW(8),superLayer&

          '* Koordinaten
          POKEW superRast&+34,&HAAAA
          FOR loop%=0 TO superWeite% STEP 50
            LINE (loop%,0)-(loop%,superHoehe%)
          NEXT loop%
          FOR loop%=0 TO superHoehe% STEP 50
            LINE (0,loop%)-(superWeite%,loop%)
          NEXT loop%
          POKEW superRast&+34,&HFFFF

zeichne:  '* Hier kommen die AmigaBasic-Befehle zum Zuge
          CIRCLE (400,200),250
          CIRCLE (400,200),300
          LINE   (200,100)-(600,300),1,bf

scrollD:  '* scroll Display
          FOR loop%=0 TO 150
            y% = 1
            GOSUB scrollIt
          NEXT loop%

          FOR loop%=0 TO 500
            y% = 0
            x% = 1
            GOSUB scrollIt
          NEXT loop%

          FOR loop%=0 TO 150
            y% = -1
            x% = -1
```

```
            GOSUB scrollIt
          NEXT loop%

          FOR loop%=0 TO 350
            y% = 0
            x% =-1
            GOSUB scrollIt
          NEXT loop%

deleteSys: '* System entfernen
          POKEL WINDOW(8),backup.layer&
          POKEL superLayer&+12,backup.rast&
          POKEL superLayer&+40,0

          CALL DeleteLayer(scrLayerInfo&,superLayer&)
          CALL FreeRaster(superPlane&,superWeite%,superHoehe%)
          CALL FreeMem(superBitMap&,40)
          SCREEN CLOSE scrNr%
          WINDOW windNr%,"hi!",,,-1
          LIBRARY CLOSE
          END


scrollIt:  '* ScrollFunktion
          CALL ScrollLayer(scrLayerInfo&,superLayer&,x%,y%)
          RETURN
```

Eine durch AmigaBASIC-Grafikbefehle geschaffene Supergrafik wird auf dem Bildschirm hin- und hergescrollt. Für einen Test reicht das aus. Am Ende dieses Buches finden Sie ein voll ausgebautes Grafik-Zeichenprogramm, das die Ihnen hier gezeigte Layer-Technik benutzt und weitere Anregungen bieten wird. Als Abschluß dieses Kapitels sei noch ein kleiner Tip gegeben: Mit den hier gezeigten Programmteilen lassen sich wirklich alle Grafikbefehle des AmigaBASIC in einem Layer verwenden. Bei zwei Befehlen ist allerdings Vorsicht geboten: Der Befehl "CLS" löscht lediglich die einem Fensterinhalt entsprechende linke obere Ecke des Layers. Wollen Sie wirklich das gesamte Layer löschen, funktioniert dies über den Grafikbefehl "SetRast". Sie rufen diesen Befehl mit folgender Syntax auf:

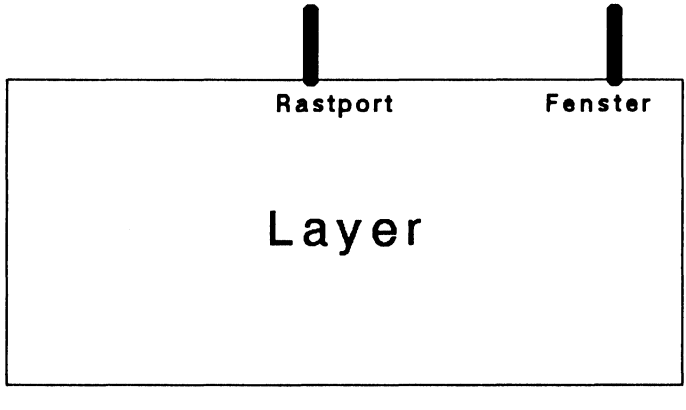
```
CALL SetRast(rastport&,farbe%)
```

rastport&:	Adresse des Rastports Ihres Layers/Fensters
farbe%:	Die Farbe, mit der Ihr Rastport ausgefüllt werden soll. Zum Löschen normalerweise =0.

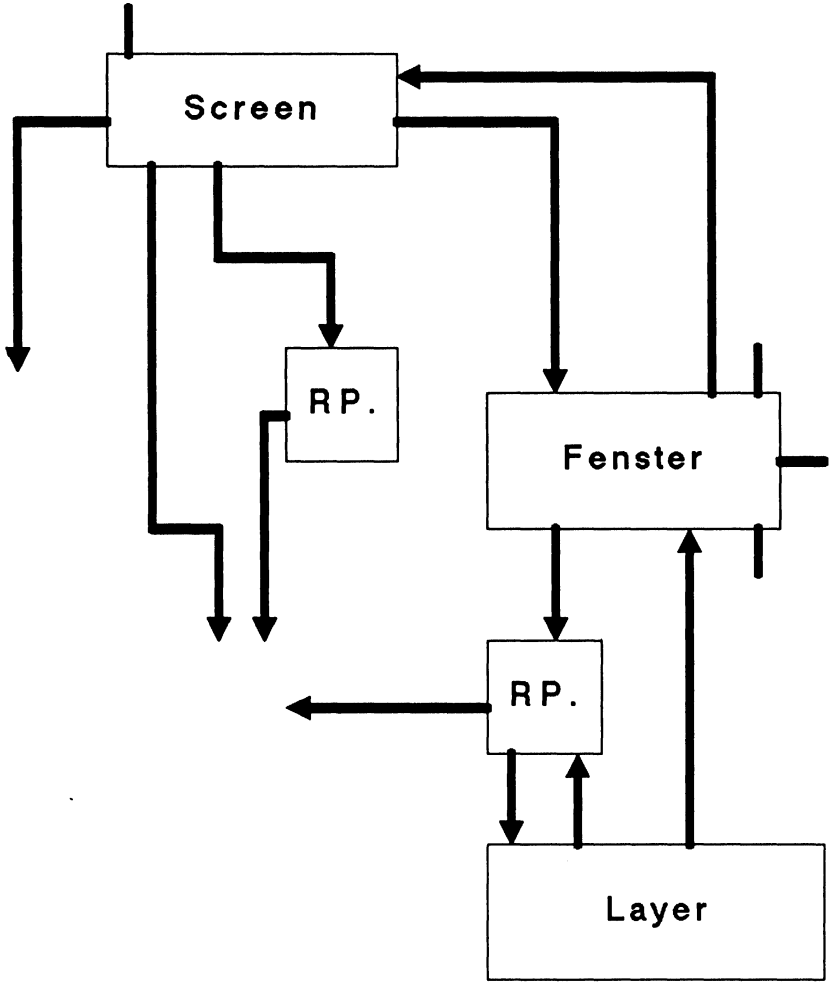
Wenn Sie Text in einen Teil des Layers drucken (LOCATE-Befehl), der unterhalb der unteren Kante Ihres Fensters liegt, dann scrollt AmigaBASIC unglücklicherweise das ehemalige Ausgabefenster, also den linken oberen Teil des Layers, um eine Zeile nach oben. Man umgeht dieses Problem, indem die in Kapitel 2 beschriebene Grafik-Funktion "Text" an Stelle von PRINT benutzt wird.

4.8 Layer im System

Erinnern Sie sich an unseren Versuch aus Kapitel 4.3, das System mit seinen Datenstrukturen zeichnerisch darzustellen? Mittlerweile sollten auch die "Layers" für Sie keine Unbekannten mehr sein. Mit dem Wissen über sie läßt sich das System nun noch genauer darstellen. Zunächst unser Layer...



...und hier das gesamte System:



An dieser Stelle haben wir alle wichtigen Systemkomponenten abgeschlossen. Das Grafiksystem in seinen Grundzügen steht Ihnen nun offen. Kommen wir jetzt zu den untergeordneten Datenstrukturen, die zum Teil nicht minder wichtig sind.

5. Die Zeichensätze des Amiga

Der Amiga kennt verschiedene Schrifttypen. Wie bei jedem anderen Computer auch, gibt es beim Amiga einen Speicherbereich, in dem das Äußere des gerade aktiven Zeichensatzes abgespeichert ist. Anders als bei den meisten Computern besitzt der Amiga zwei dieser Zeichensätze werksmäßig eingebaut. Diese beiden Schrifttypen wurden auf die Namen:

topaz 8

und

topaz 9

getauft. Welcher der beiden Typen beim Einschalten Ihres Rechners aktiv ist, hängt von Ihren Einstellungen in den Workbench-"Preferences" ab (60- oder 80-spaltiger Text).

Anders als die meisten anderen Computer hat Ihr Amiga zudem die Möglichkeit, andere Zeichensätze vom Disk-Drive nachzuladen. So steht Ihnen eine praktisch unbegrenzte Anzahl der verschiedensten Schrifttypen zur Gestaltung Ihrer Projekte zur Verfügung.

Schließlich gibt es die Möglichkeit, völlig eigene Schrifttypen zu entwickeln, wobei Ihrer Phantasie keine Grenzen gesetzt sind.

AmigaBASIC-Programmierern stehen alle drei Wege offen. Wir werden Ihnen nacheinander in diesem Kapitel diese drei Methoden vorstellen und anhand vieler Beispielprogramme zeigen, wie die Programmierung funktioniert.

5.1 Erster Kontakt zum Amiga-Zeichengenerator

Bevor wir überhaupt an die Verwirklichung eines der geplanten Projekte gehen können, benötigen wir einen Anfangspunkt, an

dem wir ins Zeichensatz-System des Amiga eindringen können. Diese Hintertür ins System findet sich im Rastport Ihres Fensters. Die Adresse des Rastports liegt, wie immer, in der Variablen WINDOW(8) (siehe Kapitel 3.6). Dort findet sich ab Offset 52 die Anfangsadresse des momentan aktiven Zeichengenerators. Genauer gesagt handelt es sich um die Anfangsadresse auf eine Datenstruktur namens "TextFont" (engl. "Font" = "Schriftart", "Zeichensatz"). Hier ihr interner Aufbau:

```
textFont&=PEEKL(WINDOW(8)+52)
```

GFA:

```
OPENW 0
textFont%=LPEEK(LPEEK(WINDOW(0)+50)+52)
```

Datenstruktur "TextFont"/graphics/ 52 Bytes

Offset	Typ	Bezeichnung		
+ 000	----	Message Struktur		
+ 010	Long	Zeiger auf Namensstring		
+ 020	Word	Höhe des Zeichensatzes		
+ 022	Byte	Stil des Zeichensatzes		
		normal	=0	
		unterstrichen	= 1	= Bit 0 = 1
		fett	= 2	= Bit 1 = 1
		kursiv	= 4	= Bit 2 = 1
		extended	= 8	= Bit 3 = 1
+ 023	Byte	Preferences und Flags		
		ROM-Zeichensatz	= 1	= Bit 0 = 1
		Disk-Zeichensatz	= 2	= Bit 1 = 1
		Rev-Path	= 4	= Bit 2 = 1
		Talldot	= 8	= Bit 3 = 1
		Widedot	= 16	= Bit 4 = 1
		Proportional	= 32	= Bit 5 = 1
		Designed	= 64	= Bit 6 = 1
		Removed	= 128	= Bit 7 = 1
+ 024	Word	Breite des Zeichensatzes (Durchschnitt)		
+ 026	Word	Höhe der Zeichen ohne Unterlängen		
+ 028	Word	Smear-Effekt für Fettdruck		
+ 030	Word	Zugriffszähler		
+ 032	Byte	ASCII-Code des ersten Zeichens		
+ 033	Byte	ASCII-Code des letzten Zeichens		

Offset	Typ	Bezeichnung
+ 034	Long	Zeiger auf Zeichendaten
+ 038	Word	Bytes pro Zeichensatz-Zeile (Modulo)
+ 040	Long	Zeiger auf Offset-Daten für Zeichen-Decodierung
+ 044	Long	Zeiger auf Breiten-Tabelle der Zeichen
+ 048	Long	Zeiger auf Zeichen-Kern-Tabelle

In dieser Datenstruktur finden sich alle Parameter, die der Amiga benötigt, um einen Zeichensatz darzustellen. Es folgt nun die detaillierte Beschreibung der Datenfelder. Sie können diesen Teil im Moment getrost überspringen, denn wir werden uns erst sehr viel später wieder auf ihn beziehen.

Detaillierte Beschreibung:

Offset 0: Message

Da ein Zeichensatz unabhängig von anderen laufenden Tasks quasi als eigenständiges Programm operiert, bedarf es der Message-Technik, um Mitteilungen an ihn zu übermitteln. Diese Message-Struktur dient der Aufnahme des Signals für die Entfernung dieses Zeichensatzes aus dem System.

Offset 10: Zeiger auf Namensstring

Dieses Feld liegt innerhalb der Message-Struktur. Hier findet sich ein Zeiger auf den Namensstring dieses Zeichensatzes. Das Ende des Namens ist wie immer mit einem Null-Byte gekennzeichnet. Den Namen Ihres augenblicklichen Zeichensatzes können Sie also mit Hilfe der folgenden Zeilen bestimmen:

```
fenster.rast&=WINDOW(8)
font.add&=PEEKL(fenster.rast&+52)
font.name&=PEEKL(font.add&+10)
gefunden%=PEEK(font.name&)
WHILE gefunden%)0
  font.name&=font.name&+1
  font.name$=font.name$+CHR$(gefunden%)
  gefunden%=PEEK(font.name&)
```

```
WEND

PRINT "Name des Zeichensatzes: ";font.name$
```

Offset 20 und 22: Zeichensatz-Attribute

Hier finden sich die Charakteristika des Zeichensatzes: seine Höhe in Pixel und seine Stil-Bits.

Offset 23: Preferences und Flags

Die Bits dieses Bytes reflektieren den augenblicklichen Status dieses Zeichensatzes. Bei der Suche nach einem Zeichensatz können Bits analog zu dieser Belegung gesetzt und als Preferences eingesetzt werden. Das heißt, der gefundene Zeichensatz muß nicht unbedingt diesen Bits entsprechen, tut es aber im Rahmen der Möglichkeiten.

Offset 24 und 26: Weitere Dimensionen des Zeichensatzes

Offset 28: Zähler für Fettdruck

Wenn der Amiga Fettdruck ausgibt, wird der Text dazu normalerweise um ein Pixel nach rechts verschoben nochmals ausgegeben. Hier findet sich dieser Zähler. Durch andere Werte kann der Text aber auch um größere Abstände verschoben werden:

```
font&=PEEKL(WINDOW(8)+52)
POKE  WINDOW(8)+56,2 'Fettdruck ein
POKEW font&+28,3 '3 Pixel nach rechts verschieben
PRINT "Demo-Text"
POKE  WINDOW(8)+56,0 'normal
```

Offset 30: Zugriffszähler

Sobald ein Zeichensatz geöffnet wird, steht er dem gesamten System zur Verfügung. Mehrere Programme (Tasks) können also gleichzeitig auf einen Zeichensatz zugreifen. Jeder Task, der einen Zeichensatz für seinen eigenen Gebrauch öffnet, ist verpflichtet, ihn nach erfolgreicher Benutzung wieder zu schließen.

Öffnet ein Task nun einen Zeichensatz, der bereits von einem anderen Task geöffnet wurde, wird keine neue (und speicherintensive) Datenstruktur eingerichtet. Statt dessen erhält der zweite Task Zugriff auf dieselbe Datenstruktur, die Task 1 geöffnet hat. Gleichzeitig erhöht sich der Zugriffszähler von 1 auf 2. Gibt nun ein Task die Anweisung, diesen Zeichensatz zu schließen, wird der Zugriffszähler um eins vermindert. Erst wenn der Zähler den Wert 0 erreicht, wird die Datenstruktur tatsächlich aus dem Speicher entfernt. Durch diese Technik wird verhindert, daß der Task, der diesen Zeichensatz ursprünglich geöffnet hat, ihn schließt, während andere Programme mittlerweile auch auf diesen Zeichensatz zugreifen und ihn noch benötigen.

Offset 32 und 33: ASCII-Codes

Wie Sie sicher wissen, ist der Amiga in der Lage, bis zu 256 verschiedene Zeichen in einem Zeichensatz unterzubringen. Nicht immer ist es jedoch sinnvoll, so viele Zeichen zu definieren. Das Alphabet beispielsweise besteht aus lediglich 26 Zeichen. Die meisten Zeichensätze schöpfen daher die Palette der 256 Zeichen nicht aus, sondern bestimmen eine untere und eine obere Grenze, zwischen denen die im Zeichensatz definierten Zeichen liegen. Diese Grenzen werden in diesen beiden Feldern hinterlegt.

Offset 34: Die Zeichendaten

Hier findet sich der Zeiger auf die Definition der Zeichen in diesem Zeichensatz. Wie dieser Datenblock aufgebaut ist, wird später in Kapitel 5 behandelt.

Offset 38: Modulo

Als "Modulo" bezeichnet man die Anzahl der Bytes pro Zeile eines Datenblockes. Der Amiga speichert den Zeichensatz zeilen-

weise ab, d.h. jeweils eine Zeile aller Zeichenzeilen. Mit Hilfe des Modulos gelangt man an den Anfang der nächsten Zeile.

Offset 40: Daten-Decodierung

Mit Hilfe der Daten-Decodierung ist es möglich, aus den jeweiligen Datenzeilen die Stücke herauszufischen, die zu dem gewüschten Zeichen gehören. Einzelheiten folgen.

Offset 44: Breiten-Tabelle

Die Zeichen eines Zeichensatzes müssen nicht eine konstante Breite besitzen. Sogenannte "Proportionalschrift" definiert für jedes Zeichen eine individuelle Breite. Ein "i" ist damit schmaler als beispielsweise ein "W". Hier findet sich ein Zeiger auf die Breiten-Tabelle. Auch hierzu folgen weitere Einzelheiten.

Offset 48: Zeichen-Kern

Einzelheiten folgen.

5.2 Öffnen des ersten Zeichensatzes

Sie haben gerade die innerste Datenstruktur eines Zeichensatzes kennengelernt. Bevor wir mit ihr weiterarbeiten, stellen wir Ihnen eine weitere, kürzere Datenstruktur namens "TextAttr" vor:

Datenstruktur "TextAttr"/graphics/8 Bytes

Offset	Typ	Bezeichnung
+ 000	Long	Zeiger auf nullterminierten Namensstring
+ 004	Word	Höhe des Zeichensatzes
+ 006	Byte	Stil-Bits
+ 007	Byte	Preferences

(Definition der Felder: Siehe Kapitel 5.1)

Mit Hilfe dieser Struktur können Sie Zeichensätze beschreiben, oder, wenn Sie so wollen, "zur Fahndung ausschreiben": Die Grafik-Routine "OpenFont" (= Öffne Zeichensatz) sucht mit Hilfe der dort gelagerten Daten nach einem entsprechenden Zeichensatz. Wir werden das gleich ausprobieren. Zunächst die Syntax der Funktion "OpenFont":

```
newFont&=OpenFont&(textAttr&)
```

textAttr&:	Anfangsadresse der korrekt ausgefüllten "TextAttr"-Datenstruktur (siehe oben!).
newFont&:	Wenn der Zeichensatz erfolgreich geöffnet wurde, liegt hier die Anfangsadresse der "TextFont"-Datenstruktur des neuen Zeichensatzes (siehe Kapitel 5.1!).

```
'########################################
'#
'# Programm: Zeichensatz laden
'# Datum: 10.4.87
'# Autor: tob
'# Version: 1.0
'#
'########################################

' Laedt die beiden ROM-Zeichensaetze "topaz 8" und
' "topaz 9"

PRINT "Suche die .bmap-Dateien..."

'GRAPHICS-Bibliothek
DECLARE FUNCTION OpenFont& LIBRARY
'CloseFont()
'SetFont()

LIBRARY "graphics.library"

demo:     '* Demonstriert die beiden ROM-Fonts!
          demo.1$ = "TOPAZ 9 *** topaz 9"
          demo.2$ = "TOPAZ 8 *** topaz 8"
          CLS

          FOR demo%=1 TO 10
            OeffneZeichensatz 9
            FOR loop%=1 TO 10
              PRINT demo.1$;
            NEXT loop%
```

```
            PRINT

            OeffneZeichensatz 8
            FOR loop%=1 TO 10
              PRINT demo.2$;
            NEXT loop%
            PRINT
          NEXT demo%

          LIBRARY CLOSE
          END

SUB OeffneZeichensatz(hoehe%) STATIC
      font.name$  = "topaz.font"+CHR$(0)
      font.hoehe% = hoehe%
      font.stil%  = 0
      font.prefs% = 0
      font.alt&   = PEEKL(WINDOW(8)+52)

     '* TextAttr-Struktur ausfuellen
      textAttr&(0) = SADD(font.name$)
      textAttr&(1) = font.hoehe%*2^16+font.stil%*2^4+font.prefs%

     '* neuen Zeichensatz oeffnen
      font.neu& = OpenFont&(VARPTR(textAttr&(0)))
      IF font.neu&<>0 THEN
        CALL CloseFont(font.alt&)
        CALL SetFont(WINDOW(8),font.neu&)
      END IF
END SUB
```

Mit diesem Programm lassen sich die beiden ROM-Zeichensätze "topaz8" und "topaz9" öffnen und benutzen. Sollten Sie versuchen, für die Zeichensatzhöhe Werte kleiner als 8 oder größer als 9 einzugeben, dann passiert scheinbar "gar nichts", einer der beiden ROM-Zeichensätze erscheint.

Das Programm verwendet zwei weitere Bibliotheksroutinen:

CloseFont()

und

SetFont()

Sobald Sie einen neuen Zeichensatz öffnen, muß der alte von Ihnen geschlossen werden, damit der Zugriffszähler aus der

"TextFont"-Struktur (siehe Kapitel 5.1) korrekte Werte enthält. Dazu verwenden Sie die Routine "CloseFont". Als Argument dient die Adresse der "TextFont"-Struktur des alten Zeichensatzes. Diese findet sich in Ihrer Rastport-Struktur.

Mit dem Öffnen des Zeichensatzes allein ist es noch nicht getan. Vielmehr müssen Sie im Anschluß daran die Informationen über den neuen Zeichensatz Ihrem Rastport zukommen lassen. Diese Aufgabe übernimmt die Routine "SetFont". Sie verlangt zwei Argumente: Die Adresse des Rastports sowie die Adresse der "TextFont"-Struktur eines korrekt geöffneten Zeichensatzes. Dieser Wert wird von der "OpenFont"-Funktion geliefert.

5.3 Zugriff auf die Disk-Fonts

Nach ein paar Minuten des Nachdenkens werden Sie das eben Gesagte sicherlich verinnerlichen. Das Programmbeispiel hat gezeigt: Es ist gar nicht so schwer, einen alternativen Zeichensatz zu aktivieren. Zwischen den beiden ROM-Zeichensätzen konnte beliebige hin- und hergeschaltet werden.

Diese beiden Zeichensätze sind zugegebenermaßen nicht sehr abwechslungsreich. Sie wurden mit einem praktischen Hintergedanken konzipiert, denn sie sollten 60- bzw. 80-spaltigen Text darstellen können. Um an wirklich abwechslungsreiche Zeichensätze zu kommen, muß ein anderer Weg beschritten werden. Auf jeder Workbench-Diskette befinden sich serienmäßig mehrere Zeichensätze gespeichert. Sie befinden sich im Unterdirectory "Fonts". Sofern Sie die Diskette noch nicht einem ausgedehnten "Ausforsten" unterzogen haben, liegen dort (Version 1.2) folgende Schrifttypen:

Nr.	Höhe	Name
01	08	ruby.font
02	12	ruby.font
03	15	ruby.font
04	12	diamond.font
05	20	diamond.font

Nr.	Höhe	Name
06	09	opal.font
07	12	opal.font
08	17	emerald.font
09	20	emerald.font
10	11	topaz.font
11	09	garnet.font
12	16	garnet.font
13	14	sapphire.font
14	19	sapphire.font

Die beiden Topaz-Typen aus dem vorangegangenen Beispiel finden sich hier natürlich nicht, denn sie wurden mit der Kickstart-Diskette bereits in den Rechner geladen oder befinden sich bereits im ROM.

Disk-Zeichensätze lassen sich nicht mit Hilfe des "OpenFont"-Befehls aktivieren, denn dieser Befehl funktioniert nur mit Zeichensätzen, die bereits im Speicher des Amiga liegen. Statt dessen wird der Befehl "OpenDiskFont" der Diskfont-Bibliothek benutzt. Er wird analog zu "OpenFont" aufgerufen, benötigt also ebenfalls eine "TextAttr"-Datenstruktur.

Das folgende Programm macht es möglich, Disk-Fonts zu benutzen. Es ist als erster Test gedacht und entsprechend simpel gehalten. Damit dieses Programm laufen kann, bedarf es der Workbench-Diskette beziehungsweise der Zeichensätze darauf. Die Routine "OpenDiskFont" sucht den gewünschten Zeichensatz zunächst im eigenen Directory, danach im System-Directory FONTS:. Sollten die Zeichensätze, die das Demo-Programm benutzt, nicht auf der Workbench-Diskette befinden, verändert sich der Zeichensatz nicht.

```
'#######################################
'#
'# Programm: Disk-Zeichensatz laden
'# Datum: 10.4.87
'# Autor: tob
'# Version: 1.0
'#
'#######################################
```

```
' Laedt einen beliebigen Disk-Zeichensatz (sog. Disk-Font)

PRINT "Suche die .bmap-Dateien..."

'DISKFONT-Bibliothek
DECLARE FUNCTION OpenDiskFont& LIBRARY

'GRAPHICS-Bibliothek
'CloseFont()
'SetFont()

LIBRARY "diskfont.library"
LIBRARY "graphics.library"

demo:     '* Demonstriert Disk-Fonts!
          demo.1$   = "DIAMOND 20 *** diamond 20 "
          demo.2$   = "SAPPHIRE 14 *** sapphire 14 "
          font.alt& = PEEKL(WINDOW(8)+52)
          CLS

          FOR demo%=1 TO 5
            OeffneDiskZeichensatz "diamond",20
            FOR loop%=1 TO 5
              PRINT demo.1$;
            NEXT loop%
            PRINT

            OeffneDiskZeichensatz "Sapphire",14
            FOR loop%=1 TO 5
              PRINT demo.2$;
            NEXT loop%
            PRINT
          NEXT demo%

          '* normalen Zeichensatz aktivieren
          font.neu& = PEEKL(WINDOW(8)+52)
          CALL CloseFont(font.neu&)
          CALL SetFont(WINDOW(8),font.alt&)

          LIBRARY CLOSE
          END

SUB OeffneDiskZeichensatz(n$,hoehe%) STATIC
      font.name$  = n$+".font"+CHR$(0)
      font.hoehe% = hoehe%
      font.stil%  = 0
      font.prefs% = 0
      font.alt&   = PEEKL(WINDOW(8)+52)

     '* TextAttr-Struktur ausfuellen
      textAttr&(0) = SADD(font.name$)
      textAttr&(1) = font.hoehe%*2^16+font.stil%*2^4+font.prefs%
```

```
      '* neuen Zeichensatz oeffnen
       font.neu& = OpenDiskFont&(VARPTR(textAttr&(0)))
       IF font.neu&<>0 THEN
         CALL CloseFont(font.alt&)
         CALL SetFont(WINDOW(8),font.neu&)
       END IF
END SUB
```

Eines fällt auf: Nach jeder Textzeile der Demo fängt der Amiga von neuem an zu laden. Da jedesmal beim Umschalten des Zeichensatzes der alte Schrifttyp aus dem RAM gelöscht wird, ist das nur logisch. Praktisch ist es jedoch nicht. Man kann sich helfen, indem eine Auswahl der häufig benutzten Zeichensätze geöffnet bleibt und erst am Schluß des Programms zusammen geschlossen wird. Zwei Dinge sind dabei wichtig: Alle von unserem Programm geöffneten Typen müssen geschlossen werden. Des weiteren dürfen Schrifttypen nur einmal ins RAM geladen werden (sonst verschwenden wir kostbaren Speicherplatz). Nach diesem System arbeitet das folgende Programm. Es ist in der Lage, sowohl Disk- als auch ROM-Typen zu aktivieren und lädt Disk-Zeichensätze nur ein einziges Mal ein. Das beschleunigt die Programmabarbeitung ganz erheblich. Hier das Listing:

```
'########################################
'#
'# Programm: Zeichensatz laden&halten
'# Datum: 10.4.87
'# Autor: tob
'# Version: 1.0
'#
'########################################

' laedt Disk- und RAM/ROM-Zeichensaetze. Sobald ein neuer
' Zeichensatz geladen wird, schliesst das Programm den
' alten Zeichensatz NICHT, sondern nimmt ihn in eine Liste
' auf. Das naechste Mal, wenn dieser Zeichensatz geoeffnet
' werden soll, befindet er sich bereits im RAM-Speicher und
' erscheint sofort. Am Programmende werden dann alle Zeichen-
' saetze gleichzeitig geschlossen.

PRINT "Suche die .bmap-Dateien..."

'DISKFONT-Bibliothek
DECLARE FUNCTION OpenDiskFont& LIBRARY

'GRAPHICS-Bibliothek
```

```
DECLARE FUNCTION OpenFont& LIBRARY
'CloseFont()
'SetFont()

LIBRARY "diskfont.library"
LIBRARY "graphics.library"

init:     '* Speicherfeld dimensionieren
          DIM SHARED storage&(30) 'max. 30 versch. Typen
          CLS

demo:     '* Hier geht es los:
          LOCATE 3,1
          OeffneZeichensatz "Opal",12
          PRINT "Arbeiten mit Amigas Zeichensaetzen!"
          OeffneZeichensatz "Diamond",12

          WHILE z$<>"ende"
            LINE INPUT "Name des Zeichensatzes: ";z$
            IF z$<>"ende" THEN
              INPUT "Hoehe";h%
              OeffneZeichensatz z$,h%
              PRINT "Dies ist ";z$;", ";h%;" Punkte hoch."
              PRINT "Mit 'ende' beenden!"
              PRINT "Geoeffnete Schrifttypen: ";zaehler%
              OeffneZeichensatz "opal",12
            END IF
          WEND

          '* normalen Zeichensatz aktivieren
          '* alle anderen aus RAM entfernen
          SchliesseZeichensaetze

          LIBRARY CLOSE
          END

SUB OeffneZeichensatz(n$,hoehe%) STATIC
      SHARED zaehler%,modus%,font.original&

      IF modus% = 0 THEN
        modus%         = 1
        font.original& = PEEKL(WINDOW(8)+52)
      END IF

      font.name$  = n$+".font"+CHR$(0)
      teil2$      = RIGHT$(font.name$,LEN(font.name$)-1)
      teil1%      = ASC(LEFT$(font.name$,1))
      teil1%      = teil1% OR 32

      font.name$  = CHR$(teil1%)+teil2$
      font.hoehe% = hoehe%
      font.stil%  = 0
```

```
      font.prefs%  = 0

     '* TextAttr-Struktur ausfuellen
      textAttr&(0) = SADD(font.name$)
      textAttr&(1) = font.hoehe%*2^16+font.stil%*2^4+font.prefs%

     '* neuer Zeichensatz im RAM?
      font.neu&    = OpenFont&(VARPTR(textAttr&(0)))
      IF font.neu&<>0 THEN
       '* ja, es ist ein Zeichensatz dieses Namens im RAM
        test.hoehe%=PEEKW(font.neu&+20)
        CALL CloseFont(font.neu&)
        IF test.hoehe%<>font.hoehe% THEN
         '* aber er ist nicht so hoch wie der gesuchte,
         '* also neuen suchen
          font.neu&=0
        END IF
      END IF

     '* neuen Zeichensatz oeffnen
      IF font.neu& = 0 THEN
       '* auf Disk nachschauen (letzte Chance...)
        font.neu&  = OpenDiskFont&(VARPTR(textAttr&(0)))
        IF font.neu&<>0 THEN
         '* gefunden!
          zaehler%           = zaehler%+1
          storage&(zaehler%) = font.neu&
        END IF
      END IF
      IF font.neu&<>0 THEN
       '* ein neuer Zeichensatz soll aktiviert werden
        CALL SetFont(WINDOW(8),font.neu&)
      END IF
END SUB

SUB SchliesseZeichensaetze STATIC
      SHARED zaehler%,font.original&

      FOR loop%=1 TO zaehler%
        IF storage&(loop%)<>0 THEN
          CALL CloseFont(storage&(loop%))
        ELSE
          ERROR 255
        END IF
        storage&(loop%) = NULL
      NEXT loop%

      CALL SetFont(WINDOW(8),font.original&)
END SUB
```

Die Variablen zaehler% und modus% sind für die SUBs reserviert und dürfen an keiner anderen Stelle im Programm verändert oder gelöscht werden.

Mit Hilfe des SUBs "OeffneZeichensatz" läßt sich ein beliebiger Schrifttyp suchen:

```
OeffneZeichensatz name$,hoehe%
```

name$:	**Name des Zeichensatzes**
hoehe%:	**Höhe des Zeichensatzes**

Zunächst richtet das SUB eine Variable namens modus% ein. Ist modus%=0, dann bedeutet dies, daß noch kein alternativer Zeichensatz geladen wurde. In diesem Fall initialisiert das Programm den Zeiger font.original&, der auf den Original-Zeichensatz deutet. Mit Hilfe dieses Zeigers gelangt man nach eigenen Experimenten immer wieder zurück zum alten Schrifttyp.

Das SUB bereitet anschließend die TextAttr-Struktur vor. Mit Hilfe des Befehls UCASE$ wird der Name des gesuchten Zeichensatzes ausschließlich in Großbuchstaben ausgedrückt. Die Routine OpenFont behandelt Klein- und Großschrift nicht gleich, sondern unterscheidet dazwischen. Der im RAM befindliche Zeichensatz "Diamond" könnte unter dem Namen "diamond" nicht gefunden werden. Aus diesem Grund werden die Namen der Zeichensätze grundsätzlich einheitlich geschrieben.

Anschließend wird die Struktur initialisiert. Die Variable textAttr& dient dabei als Speicherplatz.

Jetzt wird geprüft, ob sich der gewünschte Zeichensatz bereits im RAM befindet. Dann müßte er nicht mehr von Diskette geladen werden. Falls die Routine "OpenFont" einen Zeiger zurückliefert, dann gibt es einen Zeichensatz mit dem von uns angegebenen Namen im Speicher. Es ist aber noch nichts über seine Höhe ausgesagt. Deshalb wird in der Variablen test.hoehe% die Höhe des RAM-Zeichensatzes zum späteren Vergleich zwischengespeichert. Anschließend wird der RAM-Zeichensatz wie-

der via CloseFont geschlossen. Das ist wichtig und nötig aus folgender Erwägung: Gibt es den gewünschten Zeichensatz bereits, dann haben wir ihn schon einmal mittels "OpenDiskFont" geöffnet. Damit der Zugriffszähler nicht weiter erhöht wird, weil wir "OpenFont" benutzt haben, schließen wir diesen Zeichensatz umgehend wieder. Dadurch wird der Zeichensatz nicht wirklich geschlossen, sondern lediglich unsere Zugriffseintragung für den "OpenFont"-Befehl eliminiert. Sollte der RAM-Zeichensatz hingegen nicht der gesuchte sein, muß er ohnehin geschlossen werden.

Nun wird die Höhe des gefundenen Zeichensatzes mit unserer Wunschhöhe verglichen. Ist sie identisch, dann bleibt der Zeiger auf den RAM-Zeichensatz in font.neu& erhalten, ansonsten wird er gelöscht.

Wurde er gelöscht, dann wird nun auf Diskette nach dem Schrifttyp gesucht. Wird er dort gefunden, dann lädt ihn "OpenDiskFont" ins RAM. Da ein gänzlich neuer Zeichensatz geladen wurde, muß er von uns am Programmende wieder gelöscht werden. Dazu wird die Adresse auf diesen Schriftsatz im Feld storage& gespeichert und der Zeiger erhöht.

Zum Schluß wird der neue Zeichensatz aktiviert. Das geschieht nur, wenn font.neu& nicht 0 ist, was passiert, wenn der angegebene Zeichensatz weder im RAM noch im ROM noch auf Disk zu finden war.

Am Schluß Ihres Programms muß der Aufruf "SchliesseZeichensaetze" stehen. Er durchläuft das Feld storage& und ruft für alle dort eingetragenen Schriftsätze die Funktion "CloseFont" auf. Dadurch wird dem System kostbarer Speicherplatz zurückgegeben.

5.4 Das Zeichensatz-Menü

Mit dem Wissen und den Programmen aus den vorangegangenen Kapiteln dürften Sie die Mittel besitzen, um mit den Amiga-

Zeichensätzen arbeiten zu können. Kommen wir deshalb zu einer lebensnahen Anwendung, die gleichzeitig einen gravierenden Nachteil beseitigt: Sie konnten bislang zwar Zeichensätze laden und benutzen, aber das funktionierte nur unter der Voraussetzung, daß Sie über Namen und Höhe der gewünschten Schriftart Bescheid wußten. Unser nächstes Projekt: verschiedene Zeichensätze per Menüwahl.

Nun könnte man zwar die Namen aller vorhandenen Zeichensätze als DATAs in einem Programm ablegen, aber das würde wohl kaum sinnvoll sein: Es können immer Zeichensätze auf Diskette hinzukommen oder gelöscht werden. Aus diesem Grund gibt es die Funktion "AvailFonts" der Diskfont-Bibliothek. Diese Routine (AvailFonts = Available Fonts = verfügbare Zeichensätze) sammelt für Sie eine Liste der momentan verfügbaren Zeichensätze zusammen. Der Aufruf der Routine sieht so aus:

```
status%=AvailFonts%(buffer&,buflen&,modus%)
```

buffer&:	Adresse auf einen freien Speicherpuffer
buflen&:	Größe dieses Puffers
modus%:	1=RAM/ROM
	2=DISK
	3=egal woher
status%:	0=alles ok
	ansonsten Anzahl der Bytes, um die der Puffer zu klein war

In Abhängigkeit von der modus-Variablen füllt AvailFonts den Puffer folgendermaßen:

Offset	Typ	Bezeichnung
+ 000	Word	Anzahl der nun folgenden Einträge
(Die nächsten fünf Einträge wiederholen sich Anzahl-mal)		
+ 002	Word	ID (1=RAM/ROM, 2=Disk)
+ 004	Long	Zeiger auf Namensstring
+ 008	Word	Höhe des Zeichensatzes
+ 010	Byte	Stil-Bits
+ 011	Byte	Preferences

Damit liefert die AvailFonts-Routine also die TextAttr-Strukturen für sämtliche gefundenen Zeichensätze bereits mit.

Das folgende Programm liefert das SUB "GeneriereMenue".

```
'#####################################
'#
'# Programm: Menuegesteuerter Zeichensatz
'# Datum: 10.4.87
'# Autor: tob
'# Version: 1.0
'#
'#####################################

' baut automatisch ein Menue aus allen verfuegbaren
' Zeichensaetzen und kontrolliert dann das Menue.

PRINT "Suche die .bmap-Dateien..."

'EXEC-Bibliothek
DECLARE FUNCTION AllocMem& LIBRARY
'FreeMem()

'DISKFONT-Bibliothek
DECLARE FUNCTION OpenDiskFont& LIBRARY
DECLARE FUNCTION AvailFonts% LIBRARY

'GRAPHICS-Bibliothek
DECLARE FUNCTION OpenFont& LIBRARY
'CloseFont()
'SetFont()

LIBRARY "diskfont.library"
LIBRARY "graphics.library"
LIBRARY "exec.library"

init:     '* Speicherfeld dimensionieren
          DIM SHARED storage&(30) 'max. 30 versch. Typen
          DIM SHARED font.titel$(19)
          DIM SHARED font.hoehe%(19)
          CLS

demo:     '* Hier geht es los:
          MENU 5,0,1,"Zeichensaetze"
          MENU 5,1,1,"Laden"
          MENU 6,0,1,"Service"
          MENU 6,1,1,"Quit"

          menu.alt% = 1
```

```
            ON MENU GOSUB menucheck
            MENU ON

            PRINT "Das Menue steht nun zur Verfuegung."
            PRINT "Waehlen Sie einen Zeichensatz Ihrer Wahl,"
            PRINT "bzw. 'LADEN', um das Menue zu erstellen."


            WHILE forever=forever
            WEND

menucheck:'* Menue-Handling

            menuId   = MENU(0)
            menuItem = MENU(1)

            IF menuId=5 THEN
              IF menu.alt%=1 THEN
                menu.alt% = 0
                menu.nr%  = 5
                modeALL%  = 3
                GeneriereMenue menu.nr%,modeALL%
                PRINT "Menue ist bereit!"
              ELSE
                ft$ = font.titel$(menuItem-1)
                fh% = font.hoehe%(menuItem-1)

                OeffneZeichensatz ft$,fh%
              END IF
            ELSEIF menuId=6 THEN
              GOTO ende
            END IF

            GOSUB ShowText

            RETURN

ShowText: '* Text ausgeben
            PRINT ft$;fh%;" Punkt - TEXTBEISPIEL *** textbeispiel"
            RETURN


ende:      '* normalen Zeichensatz aktivieren
           '* alle anderen aus RAM entfernen
            SchliesseZeichensaetze

            LIBRARY CLOSE
            END

SUB GeneriereMenue(menu.nr%,modus%) STATIC
      mem.opt&        = 2^0+2^16
      buffer.groesse& = 3000
```

```
      buffer.add&     = AllocMem&(buffer.groesse&,mem.opt&)
      IF buffer.add&<>0 THEN
        status% = AvailFonts%(buffer.add&,buffer.groesse&,modus%)
        IF status% = 0 THEN
          eintrag% = PEEKW(buffer.add&)
          IF eintrag%>19 THEN eintrag%=19
          FOR loop% = 0 TO eintrag%-1
            counter%    = loop%*10
            font.name&  = PEEKL(buffer.add&+4+counter%)
            font.hoehe% = PEEKW(buffer.add&+8+counter%)
            font.name$  = ""
            check%      = PEEK(font.name&)
            WHILE check%<>ASC(".")
              font.name$ = font.name$+CHR$(check%)
              font.name& = font.name&+1
              check%     = PEEK(font.name&)
            WEND
            font.titel$(loop%) = font.name$
            font.hoehe%(loop%) = font.hoehe%
            menu.name$         = UCASE$(font.name$+STR$(font.hoehe%))
            MENU CSNG(menu.nr%),CSNG(loop%+1),1,menu.name$
          NEXT loop%
          CALL FreeMem(buffer.add&,buffer.groesse&)
        END IF
      ELSE
        BEEP
      END IF
END SUB

SUB OeffneZeichensatz(n$,hoehe%) STATIC
      SHARED zaehler%,modus%,font.original&

      IF modus%=0 THEN
        modus%         = 1
        font.original& = PEEKL(WINDOW(8)+52)
      END IF

      font.name$  = n$+".font"+CHR$(0)
      teil2$      = RIGHT$(font.name$,LEN(font.name$)-1)
      teil1%      = ASC(LEFT$(font.name$,1))
      teil1%      = teil1% OR 32

      font.hoehe% = hoehe%
      font.stil%  = 0
      font.prefs% = 0

     '* TextAttr-Struktur ausfuellen
      textAttr&(0) = SADD(font.name$)
      textAttr&(1) = font.hoehe%*2^16+font.stil%*2^4+font.prefs%

     '* neuer Zeichensatz im RAM?
      font.neu& = OpenFont&(VARPTR(textAttr&(0)))
```

```
      IF font.neu&<>0 THEN
       '* ja, es ist ein Zeichensatz dieses Namens im RAM
        test.hoehe% = PEEKW(font.neu&+20)
        CALL CloseFont(font.neu&)
        IF test.hoehe%<>font.hoehe% THEN
         '* aber er ist nicht so hoch wie der gesuchte,
         '* also neuen suchen
          font.neu& = 0
        END IF
      END IF

     '* neuen Zeichensatz oeffnen
      IF font.neu& = 0 THEN
       '* auf Disk nachschauen (letzte Chance...)
        font.neu&  = OpenDiskFont&(VARPTR(textAttr&(0)))
        IF font.neu&<>0 THEN
         '* gefunden!
          zaehler%           = zaehler%+1
          storage&(zaehler%) = font.neu&
        END IF
      END IF
      IF font.neu&<>0 THEN
       '* ein neuer Zeichensatz soll aktiviert werden
        CALL SetFont(WINDOW(8),font.neu&)
      END IF
END SUB

SUB SchliesseZeichensaetze STATIC
      SHARED zaehler%,font.original&

      FOR loop%=1 TO zaehler%
        IF storage&(loop%)<>0 THEN
          CALL CloseFont(storage&(loop%))
        ELSE
          ERROR 255
        END IF
        storage&(loop%) = NULL
      NEXT loop%

      IF font.original&<>0 THEN
        CALL SetFont(WINDOW(8),font.original&)
      END IF
END SUB
```

Der Aufruf dieses Unterprogramms sieht so aus:

```
GeneriereMenue menue.nr%,modus%
```

menue.nr%: Nummer des zu generierenden Menues
modus%: 1 = RAM/ROM Zeichensätze

```
2 = Disk Zeichensätze
3 = Alle Zeichensätze
```

Nach dem Aufruf dieses SUBs passieren zwei Dinge:

a) Ein Menü wird eingerichtet. In ihm finden sich die Namen und die Höhen aller - im Rahmen der von modus% gegebenen Grenzen - verfügbaren Zeichensätze inclusive ihrer Y-Abmessungen.

b) Zwei Datenfelder werden initialisiert: font.title$ enthält die Namen der Zeichensätze, font.hoehe% ihre Y-Abmessungen.

Über das Menü kann der Anwender nun einen der verfügbaren Zeichensätze auswählen. Er braucht nun nicht mehr genau darüber Bescheid zu wissen, welche Schrifttypen sich auf der Diskette befinden. Hat der Anwender seine Wahl getroffen, dann können Name und Höhe des ausgewählten Zeichensatzes direkt dem Variablenfeld entnommen und unserer altbekannten Routine "OeffneZeichensatz" übergeben werden. Diese regelt dann alles Weitere.

5.5 Der selbstdefinierte Zeichensatz

Sie haben nun zur Genüge mit den Amiga-eigenen Zeichensätzen Vorlieb nehmen müssen. Wir wollen uns jetzt dem letzten (und schwierigsten) Teil zuwenden: der Definition einer völlig individuellen Schriftart.

Dazu ist es nötig, daß Sie genau über den Aufbau eines Zeichensatzes Bescheid wissen. Am Anfang dieses Kapitels hatten Sie bereits das Vergnügen mit einer Datenstruktur namens "TextFont". Bei ihr handelt es sich um das Herzstück eines jeden Zeichensatzes. Bevor wir uns näher mit dieser Struktur beschäftigen, hier einige generelle Besonderheiten eines Amiga-Zeichensatzes.

Es gibt zwei grundsätzlich verschiedene Zeichensatzarten auf dem Amiga:

a) Normalschrift-Zeichensatz

b) Proportionalschrift-Zeichensatz

Während die Zeichen in einem Normalschrift-Zeichensatz (NZ) über sowohl einheitliche Höhe als auch Breite verfügen, können die Zeichen in einem Proportionalschrift-Zeichensatz (PZ) bei einheitlicher Höhe eine völlig individuelle Breite aufweisen.

Zur Definition eines Zeichensatzes sind demnach maximal vier Speicherblöcke notwendig:

charData
charLoc
charSpace
charKern

"charData" enthält die eigentliche Definition der Zeichen des Zeichensatzes. Dabei handelt es sich um sogenannte "bit-packed" Zeicheninformationen: Da die Zeichen eines Amiga eine von Ihnen gewählte Anzahl von Punkten breit sein können, wäre es unsinnig, die Daten eines jeden Zeichens in mehr oder weniger vielen Bytes abzuspeichern. Vielmehr speichert der Amiga die Zeichendaten folgendermaßen ab:

Gehen wir von diesen beiden Zeichen aus, wobei ein "." einen ungesetzten und ein "*" einen gesetzten Punkt repräsentiert:

```
....*....     ****.
...*.*...     *...*
..*...*..     *...*
.*.....*.     ****.
*********     *...*
*.......*     *...*
*.......*     ****
```

Diese beiden Zeichen weisen eine unterschiedliche Breite auf und könnten also einem PZ entsprechen. Der Amiga reiht nun

jeweils eine Bit-Zeile aller Zeichen des Zeichensatzes aneinander. Würde unser Zeichensatz nur die beiden Zeichen beinhalten, sähe "CharData" so aus:

```
1. Zeile:    ....*....****.
2. Zeile:    ...*.*...*...*
3. Zeile:    ..*...*..*...*
4. Zeile:    .*.....*.****.
5. Zeile:    **********...*
6. Zeile:    *.......**...*
7. Zeile:    *.......*****.
```

Die einzelnen Zeilen werden dann selbstverständlich direkt hintereinander in den Speicherblock geschrieben:

```
charData: ....*....****....*.*...*...*..*...*..*...* etc.
```

Diese Speichermethode ist zwar recht effizient, aber hat ein Problem: Wie bekommt man aus den Bits wieder Zeichen? Dazu gibt es den Speicherblock namens "charLoc". Für jedes Zeichen des Zeichensatzes finden sich dort zwei Words (also zwei Zwei-Byte-Felder). Das erste enthält die Anzahl der Bits vom Anfang einer Datenzeile bis zu den Bit-Informationen für das Zeichen, das zweite enthält die Anzahl der Bits für das Zeichen. Für die zwei Zeichen unseres Beispielzeichensatzes sieht das so aus:

```
charLoc: 0,9, 9,5
```

Das erste Zeichen beginnt 0 Bits vom Anfang einer Datenzeile und ist 9 Bits (entspricht Punkten) breit. Das zweite Zeichen beginnt 9 Bits nach dem Anfang der Datenzeile und ist 5 Bits breit. Diese Definition gilt für alle sieben Zeilen unserer Zeichen.

Ein weiteres Problem: Wie kommt man von einer charData-Datenzeile zur nächsten? Dazu gibt es das Feld "Modulo" innerhalb der "TextFont"-Struktur. Hier finden Sie die Anzahl der Bytes, die eine Datenzeile lang ist. Indem Sie diesen Wert zur Adresse der augenblicklichen Datenzeile addieren, kommen Sie zur nächsten.

Das Datenfeld charData enthält lediglich die blanken Informationen eines jeweiligen Zeichens. So sollen die Zeichen aber meist nicht auf den Bildschirm ausgegeben werden, sondern mit einem Abstand von einem oder einigen Punkten zum nächsten Zeichen. Deshalb enthält das Feld "charSpace" in einem Word für jedes Zeichen die tatsächliche Breite in Punkten. Wieder unser Beispiel:

```
charSpace: 11,7
```

Das erste Zeichen soll 11 Punkte breit sein, das zweite 7.

Nun bleibt ein letztes Problem: charSpace setzt die Breite eines Zeichens fest. Für das erste Zeichen sind das in unserem Beispiel 11 Punkte:

```
...........
...........
...........
...........
...........
...........
...........
```

Das Zeichen selbst ist aber lediglich neun Punkte breit. Es ist also (noch) offen, an welcher Position dieses Feldes das eigentliche Zeichen beginnen soll. Dazu gibt es den Block "charKern". Er enthält für jedes Zeichen im Zeichensatz ein Word mit der Anzahl der Bits, die vom linken Rand des obigen Feldes vergehen sollen, bis das eigentliche Zeichen ausgegeben wird. Wieder für unser Beispiel:

```
charKern: 1,2
```

Damit sehen unsere Zeichen auf dem Bildschirm so aus:

```
.....*.....      ..****.
....*.*....      ..*...*
...*...*...      ..*...*
..*.....*..      ..****.
.*********.      ..*...*
.*.......*.      ..*...*
.*.......*.      ..****.
```

Space=11	Space=7
Kern=1	Kern=2
Breite=9	Breite=5

5.5.1 Auslesen des Zeichengenerators

Mit diesem Wissen können wir bereits einen bestehenden Zeichensatz "auslesen". Darunter versteht man das Ausfiltern der Daten für ein bestimmtes Zeichen, das dann auf den Bildschirm gebracht werden kann.

Die Adresse auf die "TextFont"-Struktur des augenblicklich geöffneten Zeichensatzes findet sich im Rastport:

```
font&=PEEKL(WINDOW(8)+52)
```

Dort finden sich die gesuchten Zeiger auf die entsprechenden Speicherblöcke (siehe Kapitel 5.1). Hier zunächst das Ausleseprogramm:

```
'########################################
'#
'# Programm: Zeichensatz auslesen
'# Datum: 11.4.87
'# Autor: tob
'# Version: 1.0
'#
'########################################

' Liest den gerade aktiven Zeichensatz aus und stellt
' die decodierten Informationen in verschiedenen Ver-
' groesserungsstufen dar.

PRINT "Suche die .bmap-Dateien..."

'GRAPHICS-Bibliothek
DECLARE FUNCTION OpenFont& LIBRARY
'SetFont()
'CloseFont()

LIBRARY "graphics.library"

init:       '* Variable
            DIM SHARED zeichen$(256)
```

```
            g% = 12  'Kaestchengroesse
            CLS

            FOR demo%=32 TO 255
              Matrix demo%
              CLS
              TopazEIN
              PRINT "Zeichen: ASCII ";demo%;" = ";CHR$(demo%)
              FOR show% = 1 TO hoehe%
                LOCATE show%+2,1
                PRINT zeichen$(show%)
                FOR ex% = 1 TO LEN(zeichen$(show%))
                  z$ = MID$(zeichen$(show%),ex%,1)
                  IF z$ = "*" THEN
                    farbe% = 2
                  ELSEIF z$ = "." THEN
                    farbe% = 1
                  ELSEIF z$ = "," THEN
                    farbe% = 3
                  END IF
                  LINE (300+ex%*g%,show%*g%)(300+ex%*g%+g%,show%*g%+g%),
                       farbe%,bf
                  LINE (500+ex%*2,show%*2)(500+ex%*2+2,show%*2+2),
                       farbe%,bf
                NEXT ex%
              NEXT show%
              TopazAUS
            NEXT demo%

            END

SUB Matrix(code%) STATIC
            SHARED hoehe%

            f.1%       = 0
            f.2%       = 0
            font&      = PEEKL(WINDOW(8)+52)
            charData&  = PEEKL(font&+34)
            charLoc&   = PEEKL(font&+40)
            charSpace& = PEEKL(font&+44)
            charKern&  = PEEKL(font&+48)
            modulo%    = PEEKW(font&+38)

            IF charSpace& = 0 THEN f.1%=1
            IF charKern&  = 0 THEN f.2%=1

            hoehe%   = PEEKW(font&+20)

            loASCII% = PEEK(font&+32)
            hiASCII% = PEEK(font&+33)

            IF code%<loASCII% OR code%>hiASCII% THEN
```

```
    PRINT "ASCII-Code";code%;" nicht im Zeichensatz"
  END IF

'* Decodierungsinformationen
  offset%         = code%-loASCII%
  offset.bit&     = PEEKW(charLoc&+4*offset%)
  offset.byte%    = INT(offset.bit&/8)
  offset.bit%     = offset.bit&-(8*offset.byte%)
  zeichen.breite% = PEEKW(charLoc&+4*offset%+2)
  IF f.1% = 0 THEN
    zeichen.space%=PEEKW(charSpace&+2*offset%)
  END IF
  IF f.2% = 0 THEN
    zeichen.kern% = PEEKW(charKern&+2*offset%)
  END IF

'* Auslesen
  FOR loop1% = 1 TO hoehe%
    zeichen$(loop1%) = ""
    IF f.2% = 0 THEN
      IF zeichen.kern%>0 THEN
        zeichen$(loop1%)=STRING$(zeichen.kern%,",")
      END IF
    END IF
    linedata&  = PEEK(charData&+offset.byte%)
    zaehler%   = 7-offset.bit%
    FOR loop2% = 1 TO zeichen.breite%
      IF (linedata& AND 2^zaehler%)<>0 THEN
        linedata&        = linedata&-2^zaehler%
        zeichen$(loop1%) = zeichen$(loop1%)+"*"
      ELSE
        zeichen$(loop1%) = zeichen$(loop1%)+"."
      END IF
      zaehler%           = zaehler%-1
      IF zaehler%<0 THEN
        offset.long% = offset.long%+1
        linedata&    = PEEK(charData&+offset.byte%+
                       offset. long%)
        zaehler%     = 7
      END IF
    NEXT loop2%
    offset.long% = 0
    charData&    = charData&+modulo%
    IF f.2%=0 THEN
      zeichen.diff% = zeichen.space%-zeichen.breite%zeichen.
                      kern%
    ELSEIF f.2%=0 THEN
      zeichen.diff% = zeichen.space%-zeichen.breite%
    END IF
    IF zeichen.diff%>0 THEN
      zeichen$(loop1%)=zeichen$(loop1%)+STRING$(zeichen.
               diff%,",")
```

```
            END IF
          NEXT loop1%
END SUB

SUB TopazEIN STATIC
      SHARED font&,font.alt&
      font$         = "topaz.font"+CHR$(0)
      textAttr&(0) = SADD(font$)
      font.alt&     = PEEKL(WINDOW(8)+52)
      font&         = OpenFont&(VARPTR(textAttr&(0)))
      CALL SetFont(WINDOW(8),font&)
END SUB

SUB TopazAUS STATIC
      SHARED font&,font.alt&
      CALL CloseFont(font&)
      CALL SetFont(WINDOW(8),font.alt&)
END SUB
```

Das Programm liest den augenblicklich aktiven Zeichensatz aus. Das wird in den meisten Fällen einer der beiden ROM-Schrifttypen sein. Wenn Sie wirklich etwas Interessantes beobachten wollen, sollten Sie zuvor einen der Disk-Schrifttypen, zum Beispiel "sapphire", laden.

Alle erreichbaren Zeichen des Zeichensatzes werden nun dreifach auf dem Bildschirm dargestellt: In normaler Bildschirmdarstellung ("*" und ".") sowie als große und kleine Grafik.

Die Grafiken sind mehrfarbig, es kommen insgesamt drei Farben zum Einsatz: Die durch die in charData liegenden Daten definierte Fläche ist weiß, alle gesetzten Punkte sind schwarz. Die durch charSpace und charKern zusätzlich definierte Fläche erscheint orangefarben. Bei NZs wird diese Farbe allerdings nicht auftreten, denn es gibt charSpace und charKern bei ihnen nicht.

Zum Programm:

Herzstück ist das SUB "Matrix". Es erledigt die schwierige Aufgabe des Auslesens und kann von Ihnen natürlich auch für andere Zwecke übernommen werden. Hier der Aufruf:

```
Matrix ascii.code%

ascii.code%:   ASCII-Code des Zeichens (0-255)
```

bzw.

```
Matrix CINT(ASC(z$))

z$:            das gewünschte Zeichen
```

Außerdem finden Sie die beiden SUBs "TopazEIN" und "TopazAUS". Sie schalten jeweils den Systemzeichensatz bzw. den ehemals aktiven Zeichensatz ein. Dadurch können während des Auslesevorgangs Kommentare auf den Bildschirm gedruckt werden, die unabhängig vom gerade aktiven (und ausgelesenen) Zeichensatz lesbar bleiben.

Die Funktionsweise des Matrix-SUBs bedarf vermutlich nicht der weiteren Erklärung. Die Grundlagen finden Sie in Kapitel 5.5.

5.5.2 BigText: Text vergrößern!

Wie vielseitig die Ausleseroutine "Matrix" des vorangegangenen Beispiels ist, zeigt diese Anwendung. Mit ihrer Hilfe funktioniert das SUB "BigText", mit dem Sie Text in einer beliebigen Vergrößerung auf den Bildschirm zeichnen können.

Hier der Aufruf:

```
BigText text$,groesse%,farbe%
text$:         Auszugebender Text
groesse%:      Vergrößerungsfaktor (1-...)
farbe%:        Textfarbe
```

```
'#######################################
'#
'# Programm: Text vergroessern
'# Datum: 11.4.87
```

```
'# Autor: tob
'# Version: 1.0
'#
'#####################################

' vergroessert jeden beliebigen Text. Der Text wird in der
' Schriftart des augenblicklich aktiven Zeichensatzes ver-
' groessert.

init:       '* Variable
            DIM SHARED zeichen$(256)
            BigText "Hallo",15,2
            BigText "Commodore AMIGA",4,3
            LOCATE 3,1
            BigText "klein",1,1
            BigText "groesser",2,1
            BigText "noch groesser!",3,1
            BigText "GIGANTISCH!",8,3
            END

SUB BigText(text$,groesse%,farbe%) STATIC
            SHARED hoehe%,zeichen.kern%,zeichen.breite%
            SHARED zeichen.space%

            o.xo% = 0
            z.x%  = PEEKW(WINDOW(8)+58)
            z.y%  = PEEKW(WINDOW(8)+58)
            y%    = CSRLIN*z.y%
            x%    = POS(0)*z.x%

            FOR loop1%=1 TO LEN(text$)
              z$    = MID$(text$,loop1%,1)
              Matrix CINT(ASC(z$))
              o.xo% = o.xo%+zeichen.kern%*groesse%
              FOR loop2%=1 TO hoehe%
                FOR loop3%=1 TO LEN(zeichen$(loop2%))
                  m$=MID$(zeichen$(loop2%),loop3%,1)
                  IF m$="*" THEN
                    o.x% = x%+o.xo%+loop3%*groesse%
                    o.y% = y%+loop2%*groesse%
                    LINE (o.x%,o.y%)(o.x%+groesse%,o.y%+groesse%),
                          farbe%,bf
                  END IF
                NEXT loop3%
              NEXT loop2%
              rest% = zeichen.space%-zeichen.breite%-zeichen.kern%
              IF rest%<0 THEN rest%=0
              o.xo% = o.xo%+zeichen.breite%*groesse%+rest%*groesse%
            NEXT loop1%
            PRINT
END SUB
```

```
SUB Matrix(code%) STATIC
            SHARED hoehe%,zeichen.kern%,zeichen.breite%
            SHARED zeichen.space%
            f.1%       = 0
            f.2%       = 0
            font&      = PEEKL(WINDOW(8)+52)
            charData&  = PEEKL(font&+34)
            charLoc&   = PEEKL(font&+40)
            charSpace& = PEEKL(font&+44)
            charKern&  = PEEKL(font&+48)
            modulo%    = PEEKW(font&+38)

            IF charSpace& = 0 THEN f.1%=1
            IF charKern&  = 0 THEN f.2%=1

            hoehe%   = PEEKW(font&+20)

            loASCII% = PEEK(font&+32)
            hiASCII% = PEEK(font&+33)

            IF code%<loASCII% OR code%>hiASCII% THEN
              PRINT "ASCII-Code";code%;" nicht im Zeichensatz"
            END IF

           '* Decodierungsinformationen
            offset%        = code%-loASCII%
            offset.bit&    = PEEKW(charLoc&+4*offset%)
            offset.byte%   = INT(offset.bit&/8)
            offset.bit%    = offset.bit&-(8*offset.byte%)
            zeichen.breite%=PEEKW(charLoc&+4*offset%+2)
            z.b%           = zeichen.breite%
            IF f.1% = 0 THEN
              zeichen.space% = PEEKW(charSpace&+2*offset%)
              z.b%           = zeichen.space%
            END IF
            IF f.2% = 0 THEN
              zeichen.kern%  = PEEKW(charKern&+2*offset%)
            END IF

           '* Auslesen
            FOR loop1% = 1 TO hoehe%
              zeichen$(loop1%) = ""
              IF f.2% = 0 THEN
                IF zeichen.kern%>0 THEN
                  zeichen$(loop1%)=STRING$(zeichen.kern%,",")
                END IF
              END IF
              linedata&  = PEEK(charData&+offset.byte%)
              zaehler%   = 7-offset.bit%
              FOR loop2% = 1 TO zeichen.breite%
                IF (linedata& AND 2^zaehler%)<>0 THEN
                  linedata&        = linedata&-2^zaehler%
```

```
                zeichen$(loop1%) = zeichen$(loop1%)+"*"
              ELSE
                zeichen$(loop1%) = zeichen$(loop1%)+"."
              END IF
              zaehler% = zaehler%-1
              IF zaehler%<0 THEN
                offset.long% = offset.long%+1
                linedata&    = PEEK(charData&+offset.byte%+offset.
long%)
                zaehler%     = 7
              END IF
            NEXT loop2%
            offset.long% = 0
            charData&    = charData&+modulo%
            IF f.2% = 0 THEN
              zeichen.diff% = zeichen.space%-zeichen.breite%zeichen.
kern%
            ELSEIF f.2%=0 THEN
              zeichen.diff% = zeichen.space%-zeichen.breite%
            END IF
            IF zeichen.diff%>0 THEN
              zeichen$(loop1%) = zeichen$(loop1%)+STRING$ (zeichen.
diff%,",")
            END IF
          NEXT loop1%
END SUB
```

Die Matrix-Routine muß übrigens leicht abgeändert werden: Die SHARED-Anweisung zu Anfang des SUBs wird um einige Parameter erweitert, die das SUB "BigText" unbedingt benötigt, um den Text richtig zu plazieren.

5.5.3 Ein Fixed-Width-Zeichengenerator

Nachdem Sie sich mit den Zeigern und den Inhalten eines Zeichensatzes vertraut gemacht haben, ist es nun an der Zeit, unser Hauptprojekt in Angriff zu nehmen: der eigene Zeichengenerator.

Sie haben inzwischen gesehen, welche Mühe es macht, einen Proportional-Zeichensatz zu definieren und auch zu handhaben. Deshalb ist unser erster Zeichengenerator ein "Fixed-Width"-Generator, er generiert also einen NZ mit Zeichen einheitlicher Breite.

Wir gehen in der Vereinfachung sogar noch einen Schritt weiter: In Anlehnung an den ROM-Schriftsatz "topaz 8" werden auch unsere Zeichen eine festgelegte Größe von 8x8 Punkten haben. Dadurch lassen sie sich leicht speichern und handhaben, denn bei dieser Größe liegen die Zeichendaten in charData auf Byte-offsets.

Bevor wir ins Detail gehen, zunächst das Programmlisting:

```
'#######################################
'#
'# Programm: Fixed-Width Zeicheneenerator
'# Datum: 12.4.87
'# Autor: tob
'# Version: 1.0
'#
'#######################################

' Dieses Programm ermoeglicht die Erstellung beliebig
' vieler verschiedener Zeichensaetze. Jedes Zeichen besitzt
' eine feste Groesse von 8x8 Punkten. Jedes Zeichen kann
' nach Belieben definiert werden. Alle undefinierten Zeichen
' entstammen dem ROM-Standard-Zeichensatz "topaz 8". Alle
' nicht im Zeichensatz enthaltenen Zeichen werden durch das
' sogenannte "unprintable Character"-Symbol dargestellt; hier
' ein "TW".

'GRAPHICS-Bibliothek
DECLARE FUNCTION OpenFont& LIBRARY
'CloseFont()
'SetFont()
'AddFont()

'EXEC-Bibliothek
DECLARE FUNCTION AllocMem& LIBRARY
'FreeMem()
'CopyMem()

LIBRARY "graphics.library"
LIBRARY "exec.library"

init:    '* Zeichensatz generieren
         '* Aufruf:
         '* SchaffeZeichensatz "name",asciiLo%,asciiHi%

          SchaffeZeichensatz "tobi",22,201
          SchaffeZeichensatz "ralfi",60,122

         '* Aufruf:
```

```
        '* AktiviereZeichensatz "name"

         AktiviereZeichensatz "tobi"

        '* Neues Zeichen definieren
        '* Aufruf:
        '* NeuD "zeichen",zeile%,"definition"
        '* zeile%: 0...7 definition: *=gesetzter Punkt

         NeuD "A",0,".....*."
         NeuD "A",1,"....**."
         NeuD "A",2,"...***."
         NeuD "A",3,"..*.**."
         NeuD "A",4,".*****."
         NeuD "A",5,".*...*."
         NeuD "A",6,"***..***"
         NeuD "A",7,""

         AktiviereZeichensatz "ralfi"
        '* zweites Zeichen nach Byte-Methode (schneller)
         NeuB "@",0,126
         NeuB "@",1,129
         NeuB "@",2,157
         NeuB "@",3,161
         NeuB "@",4,161
         NeuB "@",5,157
         NeuB "@",6,129
         NeuB "@",7,126

        '* Beispieltext
         AktiviereZeichensatz "tobi"
         PRINT "@ 1987 by Data Becker's Amiga Grafik-Buch"
         PRINT TAB(25) "^"
         AktiviereZeichensatz "ralfi"
         PRINT "@ 1987 by Data Becker's Amiga Grafik-Buch"
         PRINT "^"

        '* Zeichensatz loeschen
        '* Aufruf:
        '* LoeschZeichensatz "name"

         LoeschZeichensatz "tobi"
         LoeschZeichensatz "ralfi"
         END

SUB AktiviereZeichensatz(z.n$) STATIC
      z.name$   = UCASE$(z.n$+".font"+CHR$(0))
      t&(0)     = SADD(z.name$)
      t&(1)     = 8*2^16
      font&     = OpenFont&(VARPTR(t&(0)))
      IF font&  = 0 THEN BEEP:EXIT SUB
      CALL CloseFont(font&)
```

```
      CALL SetFont(WINDOW(8),font&)
END SUB

SUB NeuB(zeichen$,zeile%,wert%) STATIC
      n.font&  = PEEKL(WINDOW(8)+52)
      n.data&  = PEEKL(n.font&+34)
      n.ascii% = ASC(zeichen$)
      n.lo%    = PEEK(n.font&+32)
      n.hi%    = PEEK(n.font&+33)
      n.modulo% = PEEKW(n.font&+38)
      n.offset% = (n.ascii%-n.lo%)+zeile%*n.modulo%
      n.data%  = 0

      IF n.ascii%<n.lo% OR n.ascii%>n.hi% THEN
        PRINT "Zeichen nicht im Zeichensatz!"
        ERROR 255
      END IF

      POKE n.data&+n.offset%,wert%
END SUB

SUB NeuD(zeichen$,zeile%,bit$) STATIC
      n.font&  = PEEKL(WINDOW(8)+52)
      n.data&  = PEEKL(n.font&+34)
      n.ascii% = ASC(zeichen$)
      n.lo%    = PEEK(n.font&+32)
      n.hi%    = PEEK(n.font&+33)
      n.modulo% = PEEKW(n.font&+38)
      n.offset% = (n.ascii%-n.lo%)+zeile%*n.modulo%
      n.data%  = 0

      IF n.ascii%<n.lo% OR n.ascii%>n.hi% THEN
        PRINT "Zeichen nicht im Zeichensatz!"
        ERROR 255
      END IF

     '* 8 Bit Alignment
      IF LEN(bit$)<>8 THEN
        IF LEN(bit$)>8 THEN bit$ = LEFT$(bit$,8)
        IF LEN(bit$)<8 THEN bit$ = bit$+soace$(8-LEN(bit$))
      END IF

     '* Daten in charData schreiben
      FOR loop1%=7 TO 0 STEP -1
        n.check$  = MID$(bit$,8-loop1%,1)
        IF n.check$="*" THEN
          n.data% = n.data%+2^loop1%
        END IF
      NEXT loop1%
      POKE n.data&+n.offset%,n.data%
END SUB
```

```
SUB LoeschZeichensatz(z.n$) STATIC
      z.name$ = UCASE$(z.n$+".font"+CHR$(0))
      t&(0)   = SADD(z.name$)
      t&(1)   = 8*2^16
      font&   = OpenFont&(VARPTR(t&(0)))
      IF font&=0 THEN ERROR 255

      z.groesse& = PEEKL(font&-4)
      IF z.groesse&<100 OR z.groesse&>4000 THEN ERROR 255

     '* aus System-Liste entfernen
      z.1& = PEEKL(font&)
      z.2& = PEEKL(font&+4)
      POKEL z.1&+4,z.2&
      POKEL z.2&,z.1&

     '* RAM freigeben
      font& = font&-4
      CALL FreeMem(font&,z.groesse&)

     '* Standard Zeichensatz laden
      standard$ = "topaz.font"+CHR$(0)
      t&(0)     = SADD(standard$)
      font&     = OpenFont&(VARPTR(t&(0)))
      IF font&  = 0 THEN ERROR 255
      CALL SetFont(WINDOW(8),font&)
END SUB

SUB SchaffeZeichensatz(z.n$,ascii.lo%,ascii.hi%) STATIC
      z.name$    = UCASE$(z.n$+".font"+CHR$(0))
      z.anzahl%  = ascii.hi%-ascii.lo%+2
      z.modulo%  = z.anzahl%
      IF (z.modulo% MOD 2)<>0 THEN
        z.modulo%=z.modulo%+1
      END IF
      z.groesse& = z.modulo%*8+z.anzahl%*4+110
      z.offset%  = ascii.lo%-32
      z.begin%   = 0

      mem.opt&   = 2^0+2^16
      z.add&     = AllocMem&(z.groesse&,mem.opt&)
      IF z.add&  = 0 THEN ERROR 7
      POKEL z.add&,z.groesse&
      z.add&     = z.add&+4

      z.data&    = z.add&+100
      z.loc&     = z.data&+z.anzahl%*8
      z.name&    = z.add&+65

      POKEL z.add&+10,z.name&
      POKEW z.add&+18,z.groesse&-4
      POKEW z.add&+20,8
```

```
POKE  z.add&+23,64
POKEW z.add&+24,8
POKEW z.add&+26,6
POKE  z.add&+32,ascii.lo%
POKE  z.add&+33,ascii.hi%
POKEL z.add&+34,z.data&
POKEW z.add&+38,z.modulo%
POKEL z.add&+40,z.loc&

'* Namensfeld ausfuellen
FOR loop1%=1 TO LEN(z.name$)
  POKE z.name&+loop1%-1,ASC(MID$(z.name$,loop1%,1))
NEXT loop1%

'* charLoc Feld
FOR loop1%=0 TO z.anzahl%-1
  POKEW z.loc&+(4*loop1%)+0,loop1%*8
  POKEW z.loc&+(4*loop1%)+2,8
NEXT loop1%

'* charData Feld
sample$ = "topaz.font"+CHR$(0)
t&(0)   = SADD(sample$)
t&(1)   = 8*2^16
sample& = OpenFont&(VARPTR(t&(0)))
IF sample&=0 THEN
  PRINT "ROM-Fonts weg???!"
  ERROR 255
END IF
s.char&   = PEEKL(sample&+34)
s.modulo% = PEEKW(sample&+38)
CALL CloseFont(sample&)

IF z.offset%<0 THEN
  z.anzahl% = z.anzahl%+z.offset%
  z.begin%  = ABS(z.offset%)
  z.offset% = 0
END IF

FOR loop1%=0 TO 7
  CALL CopyMem(s.char&+z.offset%+loop1%*s.modulo%,z.data&+z.
       begin%+loop1%*z.modulo%,z.anzahl%-1)
NEXT loop1%

'* unprintable Character
POKE z.data&+z.modulo%-1+0*z.modulo%,224
POKE z.data&+z.modulo%-1+1*z.modulo%,64
POKE z.data&+z.modulo%-1+2*z.modulo%,64
POKE z.data&+z.modulo%-1+3*z.modulo%,64
POKE z.data&+z.modulo%-1+4*z.modulo%,73
POKE z.data&+z.modulo%-1+5*z.modulo%,73
POKE z.data&+z.modulo%-1+6*z.modulo%,77
```

```
    POKE z.data&+z.modulo%-1+7*z.modulo%,74

   '* einbinden
    CALL AddFont(z.add&)
    t&(0)        = SADD(z.name$)
    font.neu&    = OpenFont&(VARPTR(t&(0)))
    IF font.neu& = 0 THEN ERROR 255

    CALL SetFont(WINDOW(8),font.neu&)
END SUB
```

Das Programm:

Es liefert Ihnen insgesamt fünf SUB-Programme:

- SchaffeZeichensatz
- LoeschZeichensatz
- NeuD
- NeuB
- AktiviereZeichensatz

1. SchaffeZeichensatz

Mit diesem Befehl kreieren Sie einen völlig neuen Zeichensatz. Er trägt den von Ihnen gewünschten Namen. Hier der Aufruf:

```
SchaffeZeichensatz name$,lo%,hi%
```

name$:	Name des neuen Zeichensatzes
lo%:	ASCII-Wert des ersten Zeichens
hi%:	ASCII-Wert des letzten Zeichens

Es ist Ihnen freigestellt, wie viele Zeichen Sie in Ihrem Zeichensatz unterbringen wollen. Wählen Sie dazu die untere und obere ASCII-Grenze: Jedes Zeichen besitzt einen ASCII-Wert, der sich durch den Befehl ASC ermitteln läßt:

```
LINE INPUT "Zeichen: ";z$
PRINT ASC(z$)
```

Insgesamt stehen die Codes 0 bis 255 zur Verfügung.

Nachdem der Zeichensatz eingerichtet ist, enthält er natürlich keine Zeichendefinitionen. Er ist praktisch "leer", alle Zeichen bestehen aus "nichts". Deshalb wird der neue Zeichensatz mit den Daten des ROM-Schrifttyps "topaz 8" gefüllt.

Nach diesem Aufruf steht Ihnen der neue Zeichensatz zur Verfügung. Alle Zeichen innerhalb der von Ihnen angegebenen ASCII-Grenzen werden wie topaz-8-Zeichen dargestellt, alle außerhalb der Grenzen liegenden Zeichen werden als "unprintable Character" ausgegeben: ein kleines TW-Zeichen.

2. AktiviereZeichensatz

Wenn Sie nur mit einem einzigen eigenen Zeichensatz arbeiten wollen, ist dieser Befehl für Sie bedeutungslos. Anders ist es, wenn Sie mehrere Zeichensätze mit verschiedenen Namen geschaffen haben. Dann nämlich können Sie mit dieser Routine den Zeichensatz Ihrer Wahl aktivieren:

```
AktiviereZeichensatz name$
```

name$:	**der Name eines zuvor durch "SchaffeZeichensatz" generierten Zeichensatzes**

3. NeuD

Unser Ziel war die Definition eigener Zeichen. Diesen Zweck erfüllt NeuD. Jedes Zeichen unseres Zeichensatzes ist 8 Punkte breit und ebenfalls 8 Punkte hoch. Mit Hilfe von NeuD läßt sich jeweils eine der acht Zeilen eines beliebigen Zeichens umdefinieren:T

```
NeuD zeichen$,nr%,bit$
```

zeichen$:	**Das Zeichen, das Sie umdefinieren wollen**
nr%:	**Die Zeile des Zeichens (0-7)**
bit$:	**Die neue Datenzeile (*=gesetzer Punkt, "."=ungesetzer Punkt)**

Achtung: NeuD definiert das Zeichen aus dem zuletzt generierten bzw. mittels AktiviereZeichensatz bestimmten Zeichensatz. Das Zeichen muß logischerweise im Zeichensatz enthalten sein, um umdefiniert werden zu können.

4. NeuB

Dies ist eine Variante des Befehls NeuD. Dort wurden die Zeichendaten für die neue Zeichenzeile als Sterne und Punkte angegeben, also in binärer Darstellung. Diese muß jedoch vom Rechner erst in Dezimalzahlen umgewandelt werden. Wenn Sie sich ein bißchen mit Binär-Dezimal-Umwandlung auskennen, können Sie die dezimalen Zahlenwerte natürlich direkt verwenden. Dazu dient NeuB:

```
NeuB zeichen$,nr%,wert%
```

zeichen$,nr%: s.o.
wert%: Dezimalwert für Zeile (0-255)

5. LoeschZeichensatz

Wenn Sie einen der von Ihnen geöffneten Zeichensätze nicht mehr brauchen, ist es Ihre Pflicht, ihn wieder zu schließen. Das geschieht durch diesen Befehl:

```
LoeschZeichensatz name$
```

name$: Name des von Ihnen durch SchaffeZeichensatz geöffneten Zeichensatzes.

Spätestens am Ende Ihres Programms müssen alle durch "SchaffeZeichensatz" erzeugten Zeichensätze mit diesem Befehl wieder ans System zurückgegeben werden!

Wichtige Programmhinweise:

Wer etwas genauer über die Hintergründe eigener Zeichensätze Bescheid wissen möchte, findet auf den folgenden Seiten Hintergrundinformationen. Sie können diesen Teil aber getrost überspringen, wenn er Sie nicht interessiert.

SchaffeZeichensatz

Die aus Kapitel 5.1 bekannte "TextFont"-Datenstruktur wird mit allen nötigen Parametern ausgefüllt. Das Feld "charLoc" wird mit den nötigen Werten initialisiert. Da die Zeichen des Zeichensatzes eine einheitliche Breite von 8 Punkten haben, ist der Offset-Wert ein Vielfaches von 8, der Breite-Wert immer =8 (siehe Kapitel 5.5).

Das "charData"-Feld soll eigentlich vom Anwender mit den selbstdefinierten Zeichen ausgefüllt werden. Da man aber davon ausgehen kann, daß nicht alle Zeichen umdefiniert werden, soll "charData" zunächst mit den Daten des ROM-Schrifttyps "topaz 8" ausgefüllt werden. Dazu wird "Topaz 8" geöffnet und ein Zeiger auf das charData-Feld des topaz-Zeichensatzes in sample& gerettet. Auch das Modulo wird ausgelesen. Nun kann "topaz" getrost wieder geschlossen werden, denn die "charData"-Daten liegen im ROM (bzw. WOM) und werden daher nicht entfernt.

Als nächstes müssen zwei Variablen initialisiert werden: z.offset% und z.begin%. Nicht immer nämlich enthält Ihr Zeichensatz dieselben Zeichen wie der ROM-Typ. z.offset% enthält die Anzahl der Zeichen, die der neue Zeichensatz später beginnt: Der ROM-Zeichensatz beginnt immer bei ASCII-Code 32. Soll das erste Zeichen in Ihrem Zeichensatz aber ein "A" sein (Code=65), dann beträgt z.offset% 65-32=33. z.begin% hat die umgekehrte Aufgabe. Wenn der ASCII-Code des ersten Zeichens in Ihrem neuen Zeichensatz kleiner ist als 32, dann ist hier die Differenz gespeichert.

Jetzt können die ROM-Daten in den RAM-Speicher kopiert werden. Dazu dient eine Funktion der Exec-Bibliothek:

```
CALL CopyMem(o.data&,z.data&,bytes&)
```

o.data&:	Originaldaten
z.data&:	Zieldaten
bytes&:	Anzahl der zu kopierenden Bytes

Diese Routine gibt es erst ab Kickstart Version 1.2. Benutzer älterer Versionen müssen diesen Befehl durch PEEK und POKE umgehen oder die Schleife ganz herauslassen. Dann allerdings müssen alle Zeichen des neuen Zeichensatzes von Ihnen definiert werden, bevor Sie mit ihm arbeiten können.

Sind alle Zeichen kopiert, dann wird das Aussehen eines "unprintable Characters" definiert. Dieses Zeichen erscheint immer dann, wenn das vom Anwender angeforderte Zeichen nicht im Zeichensatz liegt. Wir definieren ein kleines "TW". Die Daten für diesen "unprintable Character" liegen jeweils hinter den Daten für alle übrigen Zeichen.

Jetzt ist der neue Zeichensatz voll funktionsfähig. Er wird nun mit Hilfe der Funktion "AddFont" ins System aufgenommen. Von diesem Zeitpunkt an können auch andere Programme Ihre Zeichengeneratoren mitbenutzen (z.B. das NOTEPAD). Anschließend wird der Zeichensatz mit Hilfe der "OpenFont&"-Funktion geöffnet. Die Adresse font.neu& muß dabei der Adresse z.add& entsprechen.

An dieser Stelle ist es nötig, sich den Anfang der "TextFont"-Struktur genauer anzusehen. Bei ihm handelt es sich um eine "Message"-Struktur. Sie ist folgendermaßen aufgebaut:

Datenstruktur "Message"/exec/20 Bytes

Offset	Typ	Bezeichnung
+ 000	Long	Zeiger auf nächsten Zeichensatz
+ 004	Long	Zeiger auf vorangegangenen Zeichensatz
+ 008	Byte	Node-Typ
+ 009	Byte	Priorität

Offset	Typ	Bezeichnung
+ 010	Long	Zeiger auf Namen des Zeichensatzes
+ 014	Long	Zeiger auf Replyport
+ 018	Word	Länge der Message

Vor Aufruf der AddFont-Routine muß lediglich das Namensfeld und die Länge der Message (= Länge des Zeichensatzes) eingesetzt werden. Nach AddFont ist der Rest initialisiert, d.h. die beiden ersten Zeiger zeigen auf zwei andere Zeichensätze.

Diese Tatsache wird noch sehr wichtig, wenn Sie versuchen, Ihren eigenen Zeichensatz wieder aus dem System zu entfernen. Mehr dazu bei "LoeschZeichensatz".

AktiviereZeichensatz:

Hier wird nach dem Zeichensatz mit dem angegebenen Namen gesucht. Sie dürfen mit diesem SUB nur nach Zeichensätzen suchen, die durch SchaffeZeichensatz erzeugt wurden, denn der Zeichensatz wird nur kurz geöffnet, um alle nötigen Zeiger zu bekommen. Sofort danach wird er wieder geschlossen. Wir brauchen ihn nicht offen zu halten, denn er ist bereits durch SchaffeZeichensatz geöffnet.

SetFont

aktiviert den Zeichensatz.

NeuD, NeuB

Alle nötigen Zeichensatz-Daten werden über den Rastport direkt ausgelesen. NeuD wandelt die Bit-Definition in eine Dezimalzahl um, NeuB arbeitet von vornherein damit und ist also schneller. Der Wert wird an die aus den Parametern bestimmte Stelle gepoked.

LoeschZeichensatz

Auch hier wird der Zeichensatz des angegebenen Namens geöffnet. Zeichensätze müssen immer von demjenigen entfernt werden, der sie geschaffen hat. ROM-Zeichensätze verschwinden daher nie. Disk-Zeichensätze werden von AmigaDOS geladen und auch gehandhabt. Bei den eigenen Zeichensätzen sind wir allein für die ordnungsgemäße Beseitigung zuständig. Schaffe-Zeichensatz hatte bei der Speicherfestlegung die Länge des Zeichensatzes in die letzten vier Bytes vor dem Zeichensatz geschrieben. Dieser Wert wird ausgelesen. Bevor der Zeichensatz mit FreeMem gelöscht werden kann, muß die Systemliste korrigiert werden. AddFont hatte unseren Zeichensatz in sie hineingebunden. Die entsprechenden Felder werden zurückgesetzt, der Zeichensatz stiehlt sich davon.

Jetzt kann das RAM des Zeichensatzes ans System zurückgegeben werden. Damit man nach diesem Schritt nicht ganz ohne Zeichensatz dasteht, wird der ROM-Zeichensatz aktiviert.

5.5.4 Ein Proportionalschrift-Zeichensatz

Kommen wir nun zu dem sehr komplexen Proportionalschrift-Zeichensatz. Es ist praktisch unmöglich, in ihm Zeichen nachträglich umzudefinieren, denn dann müßten jedesmal mehrere hundert Bytes zur Seite geshiftet werden, um für die variablen Dimensionen dieses neuen Zeichens Platz zu schaffen. Um dennoch sinnvoll mit einem Proportionalschrift-Zeichensatz umgehen zu können, haben wir folgende Methode verwandt: Sie geben die maximale verwendete Zeichenbreite ein. Danach reserviert das Programm für jedes Zeichen genügend Speicherplatz. Diese Methode ist zwar nicht gerade speichereffizient, aber die einzig praktikable in diesem Fall.

Wieder stand die Anwendungsfreundlichkeit des Zeichengenerators im Vordergrund. Zeichen sollen leicht und ohne komplexe Zahlen- und Parametereinstellungen umdefinierbar sein. Dazu dienen sechs SUBs:

- SchaffeZeichensatz
- LoeschZeichensatz
- NeuB
- NeuD
- AktiviereZeichensatz
- Set

Sicherlich werden Ihnen diese Namen bekannt vorkommen. Ganz analog zum Fixed-Width-Zeichengenerator des vorherigen Kapitels arbeiten auch diese SUBs:

Wieder können Sie beliebig viele Zeichensätze erstellen. Das besorgt der Befehl:

```
SchaffeZeichensatz name$,lo%,hi%,breite%,höhe%,base%
```

name$:	**Name des Zeichensatzes**
lo%:	**untere ASCII-Grenze (siehe Kapitel 5.5.3)**
hi%:	**obere ASCII-Grenze**
breite%:	**maximale Breite in Punkten**
höhe%:	**einheitliche Höhe in Punkten**
base%:	**Baseline (Höhe ohne Unterlängen)**

Achtung: **Baseline muß mindestens einen Punkt kleiner sein als höhe%, weil sonst bei algorithmisch gesteuerter Schrift (speziell kursiv) Systemspeicher überschrieben werden kann!**

Nach diesem Aufruf generiert der Amiga einen Zeichensatz, der den oben genannten Anforderungen genügt. Er ist (im Gegensatz zum Fixed-Width-Generator) "leer", enthält also noch keine Zeichendefinitionen. Es wird Ihre Aufgabe sein, jedes einzelne Zeichen in diesem Generator selbst zu definieren.

Bevor Sie mit der jeweiligen Zeichendefinition beginnen können, zu der Ihnen die bereits aus Kapitel 5.5.3 bekannten SUBs NeuD und NeuB zur Verfügung stehen, müssen Sie die individuelle Größe des Zeichens einstellen. Das erledigt der Set-Befehl:

```
Set zeichen$,spacing%,kerning%
```

zeichen$:	Das Zeichen, für das diese Werte gelten sollen.
spacing%:	Breite dieses Zeichens (darf die maximale Breite des Zeichengenerators nicht überschreiten).
kerning%:	Anzahl der Punkte, die vergehen sollen, bis das von Ihnen definierte Zeichen erscheinen soll.

Nähere Informationen hierzu finden Sie in Kapitel 5.5

Alle anderen SUBs entsprechen in ihrer Funktion denen des Fixed-Width-Generators (sind aber nicht identisch!).

```
'########################################
'#
'# Programm: Proportional-Zeichengenerator
'# Datum: 12.4.87
'# Autor: tob
'# Version: 1.0
'#
'########################################

' Dieses Programm ermoeglicht die Herstellung beliebig
' vieler und bei Namen unterscheidbarer "Proportional-
' schrift"-Zeichensaetze. Jeder Zeichensatz darf eine
' eigene individuelle Hoehe haben. Jedes Zeichen darf zu-
' dem eine individuelle Breite aufweisen. Alle undefinier-
' ten Zeichen sind ebendies, erscheinen also erst NACH
' erfolgter Definition.

'GRAPHICS-Bibliothek
DECLARE FUNCTION OpenFont& LIBRARY
'CloseFont()
'SetFont()
'AddFont()

'EXEC-Bibliothek
DECLARE FUNCTION AllocMem& LIBRARY
'FreeMem()

LIBRARY "graphics.library"
LIBRARY "exec.library"

init:    '* Zeichensatz generieren
         SchaffeZeichensatz "tobi",32,65,9,10,7

         'Set-Format:
         'Set "Zeichen",Space,Kern
```

```
        Set  "@",20,3
        NeuD "@",0,"...***..."
        NeuD "@",1,"..*...*.."
        NeuD "@",2,"......*.."
        NeuD "@",3,".....*..."
        NeuD "@",4,"....*...."
        NeuD "@",5,"....*...."
        NeuD "@",6,"........."
        NeuD "@",7,"*********"
        NeuD "@",8,".*******."
        NeuD "@",9,"..*****.."

       '* zweites Zeichen nach Byte-Methode (schneller)
        Set  "A",11,1
        NeuB "A",0,8,126
        NeuB "A",1,8,129
        NeuB "A",2,8,157
        NeuB "A",3,8,161
        NeuB "A",4,8,161
        NeuB "A",5,8,157
        NeuB "A",6,8,129
        NeuB "A",7,8,126

       '* Beispieltext
        PRINT STRING$(40,"A")
        PRINT STRING$(40,"@")

       '* Zeichensatz loeschen
        LoeschZeichensatz "tobi"

        END

SUB AktiviereZeichensatz(z.n$) STATIC
      z.name$  = UCASE$(z.n$+".font"+CHR$(0))
      t&(0)    = SADD(z.name$)
      t&(1)    = 8*2^16
      font&    = OpenFont&(VARPTR(t&(0)))
      IF font& = 0 THEN BEEP:EXIT SUB
      CALL CloseFont(font&)
      CALL SetFont(WINDOW(8),font&)
END SUB

SUB Set(zeichen$,spacing%,kerning%) STATIC
      n.font&    = PEEKL(WINDOW(8)+52)
      n.space&   = PEEKL(n.font&+44)
      n.kern&    = PEEKL(n.font&+48)
      n.ascii%   = ASC(zeichen$)
      n.lo%      = PEEK(n.font&+32)
      n.hi%      = PEEK(n.font&+33)
      n.anzahl%  = n.ascii%-n.lo%
      IF n.ascii%<n.lo% OR n.ascii%>n.hi% THEN
        EXIT SUB
```

```
      END IF
      POKEW n.space&+(2*n.anzahl%),spacing%
      POKEW n.kern&+(2*n.anzahl%),kerning%
END SUB

SUB NeuB(zeichen$,zeile%,bits%,wert%) STATIC
      n.byte%   = 0
      n.bit%    = 0
      n.font&   = PEEKL(WINDOW(8)+52)
      n.data&   = PEEKL(n.font&+34)
      n.loc&    = PEEKL(n.font&+40)
      n.ascii%  = ASC(zeichen$)
      n.lo%     = PEEK(n.font&+32)
      n.hi%     = PEEK(n.font&+33)
      n.modulo% = PEEKW(n.font&+38)
      n.offset% = zeile%*n.modulo%
      n.hoehe%  = PEEKW(n.font&+20)
      n.anzahl% = n.ascii%-n.lo%
      n.breite% = PEEKW(PEEKL(n.font&+40)+(4*n.anzahl%)+2)
      n.offset& = PEEKW(PEEKL(n.font&+40)+(4*n.anzahl%))
      n.byte%   = INT(n.offset&/8)
      n.bit%    = 7-(n.offset&-(n.byte%*8))

      IF n.ascii%<n.lo% OR n.ascii%>n.hi% THEN
        EXIT SUB
      END IF

      IF bits%>n.breite% THEN
        bits% = n.breite%
      END IF
      n.data&    = n.data&+n.offset%
      FOR loop1% = bits%-1 TO 0 STEP -1
        IF (wert% AND 2^loop1%)<>0 THEN
          POKE n.data&+n.byte%,PEEK(n.data&+n.byte%) OR (2^n.bit%)
        END IF
        n.bit% = n.bit%-1
        IF n.bit%<0 THEN
          n.bit%  = 7
          n.byte% = n.byte%+1
        END IF
      NEXT loop1%

      POKEW n.loc&+(4*n.anzahl%)+2,bits%
END SUB


SUB NeuD(zeichen$,zeile%,bits$) STATIC
      bits%     = LEN(bits$)
      n.byte%   = 0
      n.bit%    = 0
      n.font&   = PEEKL(WINDOW(8)+52)
      n.data&   = PEEKL(n.font&+34)
```

```
      n.loc&    = PEEKL(n.font&+40)
      n.ascii%  = ASC(zeichen$)
      n.lo%     = PEEK(n.font&+32)
      n.hi%     = PEEK(n.font&+33)
      n.modulo% = PEEKW(n.font&+38)
      n.offset% = zeile%*n.modulo%
      n.hoehe%  = PEEKW(n.font&+20)
      n.anzahl% = n.ascii%-n.lo%
      n.breite% = PEEKW(PEEKL(n.font&+40)+(4*n.anzahl%)+2)
      n.offset& = PEEKW(PEEKL(n.font&+40)+(4*n.anzahl%))
      n.byte%   = INT(n.offset&/8)
      n.bit%    = 7-(n.offset&-(n.byte%*8))

      IF n.ascii%<n.lo% OR n.ascii%>n.hi% THEN
        EXIT SUB
      END IF

      IF bits%>n.breite% THEN
        bits% = n.breite%
      END IF
      n.data& = n.data&+n.offset%
      FOR loop1%=bits%-1 TO 0 STEP -1
        c$ = MID$(bits$,bits%-loop1%,1)
        IF c$="*" THEN
          POKE n.data&+n.byte%,PEEK(n.data&+n.byte%) OR (2^n.bit%)
        END IF
        n.bit% = n.bit%-1
        IF n.bit%<0 THEN
          n.bit%  = 7
          n.byte% = n.byte%+1
        END IF
      NEXT loop1%

      POKEW n.loc&+(4*n.anzahl%)+2,bits%
END SUB

SUB LoeschZeichensatz(z.n$) STATIC
      z.name$  = UCASE$(z.n$+".font"+CHR$(0))
      t&(0)    = SADD(z.name$)
      t&(1)    = 8*2^16
      font&    = OpenFont&(VARPTR(t&(0)))
      IF font& = 0 THEN ERROR 255

      z.groesse& = PEEKL(font&-4)
      IF z.groesse&<100 OR z.groesse&>40000& THEN ERROR 255

     '* aus System-Liste entfernen
      z.1& = PEEKL(font&)
      z.2& = PEEKL(font&+4)
      POKEL z.1&+4,z.2&
      POKEL z.2&,z.1&
```

```
      '* RAM freigeben
       font& = font&-4
       CALL FreeMem(font&,z.groesse&)

      '* Standard Zeichensatz laden
       standard$ = "topaz.font"+CHR$(0)
       t&(0)     = SADD(standard$)
       font&     = OpenFont&(VARPTR(t&(0)))
       IF font&  = 0 THEN ERROR 255
       CALL SetFont(WINDOW(8),font&)
END SUB

SUB SchaffeZeichensatz(z.n$,ascii.lo%,ascii.hi%,z.maxX%,z.hoehe%,z.
    baseline%) STATIC

       z.name$    = UCASE$(z.n$+".font"+CHR$(0))
       z.anzahl%  = ascii.hi%-ascii.lo%+2
       z.modulo%  = (z.anzahl%*z.maxX%+4)/8
       IF (z.modulo% MOD 2)<>0 THEN
         z.modulo%=z.modulo%+1
       END IF

       z.groesse& = z.modulo%*z.hoehe%+z.anzahl%*8+110

       IF z.baseline%>=z.hoehe% THEN
         z.baseline% = z.hoehe%-1
       END IF

       mem.opt&  = 2^0+2^16
       z.add&    = AllocMem&(z.groesse&,mem.opt&)
       IF z.add& = 0 THEN ERROR 7
       POKEL z.add&,z.groesse&

       z.add&    = z.add&+4
       z.data&   = z.add&+100
       z.loc&    = z.data&+z.modulo%*z.hoehe%
       IF z.loc&/2<>INT(z.loc&/2) THEN
         z.loc&  = z.loc&+1
       END IF

       z.kern&   = z.loc&+4*z.anzahl%
       z.space&  = z.kern&+2*z.anzahl%

       z.name&   = z.add&+65

       POKEL z.add&+10,z.name&
       POKEW z.add&+18,z.groesse&-4
       POKEW z.add&+20,z.hoehe%
       POKE  z.add&+23,64+32
       POKEW z.add&+24,z.maxX%
       POKEW z.add&+26,z.baseline%
       POKE  z.add&+32,ascii.lo%
```

```
      POKE  z.add&+33,ascii.hi%
      POKEL z.add&+34,z.data&
      POKEW z.add&+38,z.modulo%
      POKEL z.add&+40,z.loc&
      POKEL z.add&+44,z.space&
      POKEL z.add&+48,z.kern&

     '* Namensfeld ausfuellen
      FOR loop1%=1 TO LEN(z.name$)
        POKE z.name&+loop1%-1,ASC(MID$(z.name$,loop1%,1))
      NEXT loop1%

     '* charLoc Feld
      FOR loop1%=0 TO z.anzahl%-1
        POKEW z.loc&+(4*loop1%)+0,loop1%*z.maxX%
        POKEW z.loc&+(4*loop1%)+2,z.maxX%
      NEXT loop1%

     '* einbinden
      CALL AddFont(z.add&)
      t&(0)       = SADD(z.name$)
      font.neu&   = OpenFont&(VARPTR(t&(0)))
      IF font.neu&=0 THEN ERROR 255

      CALL SetFont(WINDOW(8),font.neu&)
END SUB
```

6. Grafik-Hardcopy

Was nützen die schönsten Computergrafiken, wenn sie mit einem letzten Aufflackern verschwinden, sobald dem Computer der Strom abgedreht wird? Natürlich nichts. Was wir benötigen, ist eine Routine, die diese Grafiken auf einem Drucker ausdruckt.

Bei den meisten anderen Computern wären jetzt langwierige und umständliche Maschinenroutinen erforderlich. Nicht jedoch beim Amiga, der die Möglichkeit, "Hardcopies" zu produzieren, bereits in der Systemsoftware enthalten hat. Nachdem Sie Ihren Drucker in den Workbench-Preferences eingestellt haben (und er in der Lage ist, Grafiken auszudrucken), brauchen Sie sich um nichts mehr zu kümmern. Alle weitere Arbeit, also das Auslesen der Grafik, die Ansteuerung des Druckers, die Musterung für die Farben, übernimmt das sogenannte "printer.device", was nichts weiter heißt als "Drucker Gerät". Bei diesem "printer.device" handelt es sich natürlich nicht um den Drucker selbst, sondern um eine Komponente des Amiga-Betriebssystems, die den Drucker steuert.

Um Grafiken ausdrucken zu können, muß man einen Weg finden, um mit dem "printer.device" in Kontakt zu treten. Das funktioniert über die standardisierte I/O (Eingabe-Ausgabe) des Amiga. Die I/O wird von der Exec-Bibliothek verwaltet.

Speziell für unser Anliegen gibt es eine Datenstruktur mit dem abenteuerlichen Namen "IODRPReq" (I/O Dump Rastport Request = Aufforderung zum Ausdrucken eines Rastports). Sie muß mit den wichtigsten Daten bestückt an das "printer.device" geschickt werden. Hier ihr Aufbau:

Datenstruktur "IODRPReq"/printer/62 Bytes

Offset	Typ	Bezeichnung
+ 000	Long	folgende Datenstruktur
+ 004	Long	vorherige Datenstruktur
+ 008	Byte	Node-Typ (5=MESSAGE)
+ 009	Byte	Priorität (0=normal)
+ 010	Long	Zeiger auf Name
+ 014	Long	Zeiger auf Message Port (Reply Port)
+ 018	Word	Länge der Message
+ 020	Long	ioDevice
+ 024	Long	ioUnit
+ 028	Word	Befehl (11=DumpRastport())
+ 030	Byte	Flag (1=Quick I/O)
+ 031	Byte	Error-Nr.
		1 = Benutzer brach Druckvorgang ab
		2 = kein Grafikdrucker angeschlossen
		3 = HAM kann nicht invertiert werden
		4 = Druck-Koordinaten sind unzulässig
		5 = Druck-Dimensionen sind unzulässig
		6 = kein Speicher frei für interne Berechnungen
		7 = kein Speicher frei für Druckpuffer
+ 032	Long	Adresse des Rastports
+ 036	Long	Adresse der Colormap
+ 040	Long	Modi (des Viewports)
+ 044	Word	Rastport: X-Beginn (0=normal)
+ 046	Word	Rastport: Y-Beginn (0=normal)
+ 048	Word	Rastport: Breite
+ 050	Word	Rastport: Höhe
+ 052	Long	Drucker: Breite
+ 056	Long	Drucker: Höhe
+ 060	Word	Spezial-Modi
		1 = Bit 0 = 1: Druck-Breite in 1/1000 inch
		2 = Bit 1 = 1: Druck-Höhe in 1/1000 inch
		4 = Bit 2 = 1: Druck-Breite so groß wie möglich
		8 = Bit 3 = 1: Druck-Höhe so groß wie mögl.
		16 = Bit 4 = 1: Druck-Breite ist Teil von FULL-X
		32 = Bit 5 = 1: Druck-Höhe ist Teil von FULL-Y
		128 = Bit 7 = 1: X-Y-Verhältnis ausgleichen
		256 = Bit 8 = 1: geringe Auflösung
		512 = Bit 9 = 1: mittlere Auflösung
		768 = Bits 8+9 = 1: höhere Auflösung
		1024 = Bit 10 = 1: höchste Auflösung

Die meisten Datenfelder dieser Struktur müssen vor Gebrauch mit den korrekten Werten gefüllt werden. Dazu gehört auch ein

Zeiger auf einen sogenannten "ReplyPort". Das ist ein Exec-Message-Port (ein Sender/Empfänger für zwischentaskliche Beziehungen), wie immer eine Datenstruktur:

Datenstruktur "Message Port"/exec/34 Bytes

Offset	Typ	Bezeichnung
+ 000	Long	nächste Datenstruktur
+ 004	Long	vorherige Datenstruktur
+ 008	Byte	Typ (4=MESSAGE PORT)
+ 009	Byte	Priorität (0=normal)
+ 010	Long	Zeiger auf Name
+ 014	Byte	Flags
		PA SIGNAL = 1
		PA SOFTINT = 2
		PA IGNORE = 4
+ 015	Byte	Signal-Bit
+ 016	Long	Zeiger auf Task für Signal
+ 020	Long	erste Datenstruktur
+ 024	Long	letzte Datenstruktur
+ 028	Long	vorletzte Datenstruktur
+ 032	Byte	Typ
+ 033	Byte	unbenutzt

Sind beide Strukturen korrekt initialisiert, benötigen wir Zugriff auf den Drucker. Den besorgt die Funktion "OpenDevice":

```
status%=OpenDevice%(name$,unit%,io&,flags%)
```

name$:	Zeiger auf nullterminierten Namensstring, hier: "printer.device"+CHR$(0)
unit%:	Geräte-Einheit; hier unwichtig, also 0
io&:	Adresse des korrekt initialisierten I/O-Datenblocks, hier: IODRPReq.
flag%:	hier unwichtig, also 0

Diese Funktion liefert einen Statusreport zurück. Verlief alles wunschgemäß, ist das eine Null. Ansonsten konnte der Drucker nicht erreicht werden. Er war eventuell nicht angeschlossen, abgeschaltet oder noch im Besitz eines anderen gleichzeitig laufenden Tasks. Achtung: Wenn Ihr Programm LPRINTs benutzt, kann die Hardcopy-Routine den Drucker nicht bekommen!

Wurde der Drucker ordnungsgemäß geöffnet, dann sind die Felder ioDevice und ioUnit der IODRPReq-Struktur jetzt ausgefüllt. Der Exec-Befehl "DoIO" startet den Druckvorgang:

```
fehler%=DoIO%(io&)
```

io&:	Adresse des IODRPReq-Blockes
fehler%:	alles ok = 0 für Fehlerdecodierung siehe IODRPReq-Struktur (oben), Error-Feld

Kommen wir von der Theorie nun zur Praxis:

6.1 Eine einfache Hardcopy-Routine

Das folgende Programm ist das Grundgerüst einer Hardcopy-Routine. Der Befehl "Hardcopy" druckt den Inhalt des augenblicklichen Ausgabefensters (mit WINDOW OUTPUT bestimmbar!) aus.

```
'########################################
'#
'# Programm: Hardcopy I
'# Datum: 13.4.87
'# Autor: tob
'# Version: 1.0
'#
'########################################

' Druckt den Inhalt des augenblicklichen Ausgabe-Fensters
' als Grafik-Hardcopy auf einem grafikfaehigen Drucker
' aus.

PRINT "Suche die .bmap-Dateien..."

'EXEC-Bibliothek
DECLARE FUNCTION AllocMem& LIBRARY
DECLARE FUNCTION DoIO% LIBRARY
DECLARE FUNCTION OpenDevice% LIBRARY
DECLARE FUNCTION AllocSignal% LIBRARY
DECLARE FUNCTION FindTask& LIBRARY
'FreeMem()
'CloseDevice()
'FreeSignal()
'AddPort()
```

```
'RemPort()

LIBRARY "exec.library"

init:     '* etwas zeichnen
          CLS
          CIRCLE (300,100),100
          LINE   (10,10)-(200,100),2,bf

          Hardcopy

          END

SUB Hardcopy STATIC
     mem.opt& = 2^0+2^16
     p.io&    = AllocMem&(100,mem.opt&)
     p.port&  = p.io&+62
     IF p.io& = 0 THEN ERROR 7

     f.fenster&   = WINDOW(7)
     f.rastport&  = PEEKL(f.fenster&+50)
     f.breite%    = PEEKW(f.fenster&+112)
     f.hoehe%     = PEEKW(f.fenster&+114)
     f.screen&    = PEEKL(f.fenster&+46)
     f.viewport&  = f.screen&+44
     f.colormap&  = PEEKL(f.viewport&+4)
     f.vp.modi%   = PEEKW(f.viewport&+32)


     p.sigBit% = AllocSignal%(-1)
     IF p.sigBit% = -1 THEN
       PRINT "Kein Signalbit frei!"
       CALL FreeMem(p.io&,100)
       EXIT SUB
     END IF
     p.sigTask& = FindTask&(0)

     POKE  p.port&+8,4
     POKEL p.port&+10,p.port&+34
     POKE  p.port&+15,p.sigBit%
     POKEL p.port&+16,p.sigTask&
     POKEL p.port&+20,p.port&+24
     POKEL p.port&+28,p.port&+20
     POKE  p.port&+34,ASC("P")
     POKE  p.port&+35,ASC("R")
     POKE  p.port&+36,ASC("T")

     CALL AddPort(p.port&)

     POKE  p.io&+8,5
     POKEL p.io&+14,p.port&
     POKEW p.io&+28,11
```

```
      POKEL p.io&+32,f.rastport&
      POKEL p.io&+36,f.colormap&
      POKEL p.io&+40,f.vp.modi%
      POKEW p.io&+48,f.breite%
      POKEW p.io&+50,f.hoehe%
      POKEL p.io&+52,f.breite%
      POKEL p.io&+56,f.hoehe%
      POKEW p.io&+60,4

      d.name$ = "printer.device"+CHR$(0)
      status% = OpenDevice%(SADD(d.name$),0,p.io&,0)
      IF status%<>0 THEN
        PRINT "Drucker ist nicht frei."
        CALL  FreeMem(p.io&,100)
        CALL  FreeSignal(p.sigBit%)
        EXIT SUB
      END IF

      fehler% = DoIO%(p.io&)

      CALL CloseDevice(p.io&)
      CALL RemPort(p.port&)
      CALL FreeMem(p.io&,100)
      CALL FreeSignal(p.sigBit%)
      PRINT "Fehlercode: ";fehler%
END SUB
```

Zum Programm-Ablauf:

Wenn Sie mit einem Schwarz-Weiß-Drucker arbeiten, wandelt das "printer.device" alle Farben automatisch in Muster um. Jede Farbe hat ihr eigenes Muster, das die Farbhelligkeit simulieren soll. Für einen weißen Hintergrund müssen die Farben entsprechend gesetzt werden:

```
PALETTE 0,1,1,1
COLOR 1,0
```

Es kommt kein Grafikausdruck zustande? Hier eine Checkliste der möglichen Fehlerquellen:

Wird ein Fehlercode zurückgeliefert (kann bis zu 30 Sekunden dauern)?

Wenn ja:

Der Fehlercode gibt Ihnen Aufschluß über die Art des Fehlers. Folgende Codes sind möglich:

Code 1: Druckvorgang wurde abgebrochen

Sie haben den Druckvorgang willkürlich beendet, indem Sie beispielsweise bei einem erscheinenden Requester (dem Fragekästchen) wie "PRINTER TROUBLE" auf das "Cancel"-Feld gedrückt haben, anstatt den Fehler zu beheben.

Code 2: Kein Grafikdrucker

Der in den Preferences angemeldete Drucker kann keine Grafiken ausdrucken (Typenrad-Drucker etc.).

Code 3: Hold-And-Modify

Hold-And-Modify-Grafiken lassen sich nicht invertieren, denn ihre Farben kommen auf ganz besondere Weise zustande. Siehe Kapitel 4.2.2.

Code 4: Unzulässige Druck-Koordinaten

Bei korrekter Programmeingabe darf dieser Fehler nicht auftreten. Tut er es doch, liegt ein Abtippfehler vor. Die X- und Y-Anfangskoordinaten des Rastports liegen außerhalb desselben.

Code 5: Unzulässige Dimensionen

Siehe Code 4. Die Rastport-Breite bzw. -Höhe ist größer als der existierende Rastport.

Code 6 und 7: Out Of Memory

Der Systemspeicher reicht nicht aus.

Wenn nein:

- Trat die Fehlermeldung "Kein Signalbit frei!" auf? Dann ist das Multi-Tasking-System überlastet. Sie müssen sich gedulden, bis andere Programme ein Signalbit freigeben.

- Trat die Fehlermeldung "Drucker nicht frei!" auf? Dann wird der Drucker augenblicklich von einem anderen Programm benutzt (LPRINTs in Ihrem eigenen Programm zum Beispiel), oder der Drucker wurde geöffnet und nicht wieder geschlossen. Dann allerdings hilft nur noch ein System-Boot (Reset).

Alle anderen Fehler beruhen auf Abtippfehlern.

6.2 Hardcopies: Vergrößern und Verkleinern!

Obiges Programm zaubert zwar schöne Hardcopies hervor, die für die allermeisten Anwendungen ausreichen, die Möglichkeiten des "printer.device" werden aber nicht annähernd ausgenutzt. Die folgende Hardcopy-Routine ermöglicht Ihnen den Ausdruck eines Ausschnittes eines Fensters. Dieser Ausschnitt, der natürlich auch das gesamte Fenster umfassen kann, läßt sich zudem verkleinert oder vergrößert auf den Drucker ausgeben.

```
'########################################
'#
'# Programm: Hardcopy II
'# Datum: 13.4.87
'# Autor: tob
'# Version: 1.0
'#
'########################################

' Erlaubt ausschnittsweises Ausdrucken, Vergroessern und
' Verkleinern des Ausdrucks.

PRINT "Suche die .bmap-Dateien..."

'EXEC-Bibliothek
DECLARE FUNCTION AllocMem& LIBRARY
DECLARE FUNCTION DoIO% LIBRARY
DECLARE FUNCTION OpenDevice% LIBRARY
DECLARE FUNCTION AllocSignal% LIBRARY
DECLARE FUNCTION FindTask& LIBRARY
'FreeMem()
'CloseDevice()
'FreeSignal()
'AddPort()
'RemPort()

LIBRARY "exec.library"
```

```
init:     '* etwas zeichnen
          CLS
          CIRCLE (300,100),100
          LINE   (10,10)-(200,100),2,bf

          '* Farben; schwarz-weiss-Kontrast
          PALETTE 0,1,1,1
          PALETTE 1,0,0,0

          ParameterHardcopy 200,10,200,100,1.2,.5

          END

SUB ParameterHardcopy(x%,y%,breite%,hoehe%,f1,f2) STATIC
      mem.opt& = 2^0+2^16
      p.io&    = AllocMem&(100,mem.opt&)
      p.port&  = p.io&+62
      IF p.io& = 0 THEN ERROR 7

      f.fenster&   = WINDOW(7)
      f.rastport&  = PEEKL(f.fenster&+50)
      f.breite%    = PEEKW(f.fenster&+112)
      f.hoehe%     = PEEKW(f.fenster&+114)
      f.screen&    = PEEKL(f.fenster&+46)
      f.viewport&  = f.screen&+44
      f.colormap&  = PEEKL(f.viewport&+4)
      f.vp.modi%   = PEEKW(f.viewport&+32)


      p.sigBit%    = AllocSignal%(-1)
      IF p.sigBit% = -1 THEN
        PRINT "Kein Signalbit frei!"
        CALL FreeMem(p.io&,100)
        EXIT SUB
      END IF
      p.sigTask&   = FindTask&(0)

      POKE  p.port&+8,4
      POKEL p.port&+10,p.port&+34
      POKE  p.port&+15,p.sigBit%
      POKEL p.port&+16,p.sigTask&
      POKEL p.port&+20,p.port&+24
      POKEL p.port&+28,p.port&+20
      POKE  p.port&+34,ASC("P")
      POKE  p.port&+35,ASC("R")
      POKE  p.port&+36,ASC("T")

      CALL AddPort(p.port&)

      POKE  p.io&+8,5
      POKEL p.io&+14,p.port&
```

```
      POKEW p.io&+28,11
      POKEL p.io&+32,f.rastport&
      POKEL p.io&+36,f.colormap&
      POKEL p.io&+40,f.vp.modi%
      POKEW p.io&+44,x%
      POKEW p.io&+46,y%
      POKEW p.io&+48,breite%
      POKEW p.io&+50,hoehe%
      POKEL p.io&+52,f.breite%*f1
      POKEL p.io&+56,f.hoehe%*f2

      d.name$ = "printer.device"+CHR$(0)
      status% = OpenDevice%(SADD(d.name$),0,p.io&,0)
      IF status%<>0 THEN
        PRINT "Drucker ist nicht frei."
        CALL FreeMem(p.io&,100)
        CALL FreeSignal(p.sigBit%)
        EXIT SUB
      END IF

      fehler% = DoIO%(p.io&)

      CALL CloseDevice(p.io&)
      CALL RemPort(p.port&)
      CALL FreeMem(p.io&,100)
      CALL FreeSignal(p.sigBit%)
      PRINT "Fehlercode: ";fehler%
END SUB
```

Aufruf:

```
ParameterHardcopy x%,y%,breite%,höhe%,f1,f2
```

x%,y%:	Koordinaten der linken oberen Ecke des Ausschnitts des Fensters, das Sie drucken möchten
breite%:	Breite des Ausschnitts in Punkten
höhe%:	Höhe des Ausschnitts in Punkten
f1:	Vergrößerungsfaktor horizontal
f2:	Vergrößerungsfaktor vertikal

6.3 Ausdrucken beliebiger Fenster

Bisher war es nur möglich, Fenster des AmigaBASIC auszudrucken. Das "printer.device" kann natürlich den Inhalt eines jeden Fensters drucken (z.B. den der Preferences). In Kapitel 3.1 hatten wir Ihnen bereits einen Weg gezeigt, wie Sie alle Fenster

des Systems erreichen können. Leicht abgewandelt kommt dieses Wissen dem nächsten Programm zugute. Die folgende Routine druckt ein beliebiges Fenster aus. Dazu benötigt die Routine lediglich den Namen des Fensters (Achtung: Das Preference-Fenster nennt sich " Preferences"):

```
'#####################################
'#
'# Programm: Hardcopy III
'# Datum: 13.4.87
'# Autor: tob
'# Version: 1.0
'#
'#####################################

' Druckt den Inhalt eines beliebigen System-Fensters, von
' dem der Name bekannt ist (incl. vergroessern, -kleinern,
' Ausschnitt).

PRINT "Suche die .bmap-Dateien..."

'EXEC-Bibliothek
DECLARE FUNCTION AllocMem& LIBRARY
DECLARE FUNCTION DoIO% LIBRARY
DECLARE FUNCTION OpenDevice% LIBRARY
DECLARE FUNCTION AllocSignal% LIBRARY
DECLARE FUNCTION FindTask& LIBRARY
'FreeMem()
'CloseDevice()
'FreeSignal()
'AddPort()
'RemPort()

LIBRARY "exec.library"

init:     '* es geht los
          CLS
          PALETTE 0,1,1,1
          PALETTE 1,0,0,0

          LIST
          UniversalHardcopy "LIST",0,0,200,100,.8,.5

          END

SUB UniversalHardcopy(namen$,x%,y%,breite%,hoehe%,f1,f2) STATIC
     f.fenster& = WINDOW(7)
     f.reg&     = PEEKL(f.fenster&+66)
     WHILE f.reg&>0
       f.fenster& = f.reg&
```

```
  f.reg&      = PEEKL(f.fenster&+66)
WEND

finder:
f.titel&      = PEEKL(f.fenster&+32)
check%        = PEEK(f.titel&+count%)
WHILE check%>0
  check$      = check$+CHR$(check%)
  count%      = count%+1
  check%      = PEEK(f.titel&+count%)
WEND
gefunden$     = check$:check$="":count%=0
IF UCASE$(gefunden$)<>UCASE$(namen$) THEN
  f.fenster&  = PEEKL(f.fenster&+70)
  IF f.fenster&>0 THEN
    GOTO finder
  ELSE
    PRINT "Fenster gibt es nicht!"
    EXIT SUB
  END IF
END IF

mem.opt&    = 2^0+2^16
p.io&       = AllocMem&(100,mem.opt&)
p.port&     = p.io&+62
IF p.io&    = 0 THEN ERROR 7

f.rastport& = PEEKL(f.fenster&+50)
f.breite%   = PEEKW(f.fenster&+112)
f.hoehe%    = PEEKW(f.fenster&+114)
f.screen&   = PEEKL(f.fenster&+46)
f.viewport& = f.screen&+44
f.colormap& = PEEKL(f.viewport&+4)
f.vp.modi%  = PEEKW(f.viewport&+32)


p.sigBit% = AllocSignal%(-1)
IF p.sigBit% = -1 THEN
  PRINT "Kein Signalbit frei!"
  CALL FreeMem(p.io&,100)
  EXIT SUB
END IF
p.sigTask&=FindTask&(0)

POKE  p.port&+8,4
POKEL p.port&+10,p.port&+34
POKE  p.port&+15,p.sigBit%
POKEL p.port&+16,p.sigTask&
POKEL p.port&+20,p.port&+24
POKEL p.port&+28,p.port&+20
POKE  p.port&+34,ASC("P")
POKE  p.port&+35,ASC("R")
```

```
      POKE  p.port&+36,ASC("T")

      CALL AddPort(p.port&)

      POKE  p.io&+8,5
      POKEL p.io&+14,p.port&
      POKEW p.io&+28,11
      POKEL p.io&+32,f.rastport&
      POKEL p.io&+36,f.colormap&
      POKEL p.io&+40,f.vp.modi%
      POKEW p.io&+44,x%
      POKEW p.io&+46,y%
      POKEW p.io&+48,breite%
      POKEW p.io&+50,hoehe%
      POKEL p.io&+52,f.breite%*f1
      POKEL p.io&+56,f.hoehe%*f2

      d.name$ = "printer.device"+CHR$(0)
      status% = OpenDevice%(SADD(d.name$),0,p.io&,0)
      IF status%<>0 THEN
        PRINT "Drucker ist nicht frei."
        CALL FreeMem(p.io&,100)
        CALL FreeSignal(p.sigBit%)
        EXIT SUB
      END IF

      fehler% = DoIO%(p.io&)

      CALL CloseDevice(p.io&)
      CALL RemPort(p.port&)
      CALL FreeMem(p.io&,100)
      CALL FreeSignal(p.sigBit%)
      PRINT "Fehlercode: ";fehler%
END SUB
```

6.4 ScreenDump - einen ganzen Screen

Da ein Screen einen eigenen Rastport besitzt, läßt er sich auch ausdrucken. Hier das Programm: Es entspricht dem vorangegangenen, lediglich verlangt es an Stelle des Fensternamens die Nummer des Screens (0=Workbench-Screen). Dieses Programm funktioniert nur dann, wenn das Ausgabefenster im Workbench-Screen liegt.

```
'#######################################
'#
'# Programm: Hardcopy IV
'# Datum: 13.4.87
'# Autor: tob
'# Version: 1.0
'#
'#######################################

' Druckt einen beliebigen Screeninhalt aus.

PRINT "Suche die .bmap-Dateien..."

'EXEC-Bibliothek
DECLARE FUNCTION AllocMem& LIBRARY
DECLARE FUNCTION DoIO% LIBRARY
DECLARE FUNCTION OpenDevice% LIBRARY
DECLARE FUNCTION AllocSignal% LIBRARY
DECLARE FUNCTION FindTask& LIBRARY
'FreeMem()
'CloseDevice()
'FreeSignal()
'AddPort()
'RemPort()

LIBRARY "exec.library"

init:     '* etwas zeichnen
          CLS
          CIRCLE (300,100),100
          LINE   (10,10)-(200,100),2,bf
          PALETTE 0,1,1,1
          PALETTE 1,0,0,0

          ScreenHardcopy 0,0,0,640,256,1.1,2!

          END

SUB ScreenHardcopy(nr%,x%,y%,breite%,hoehe%,f1,f2) STATIC
      f.fenster& = WINDOW(7)
      f.screen&  = f.fenster&+46

      FOR loop1% = 0 TO nr%
        f.screen&=PEEKL(f.screen&)
        IF f.screen&=0 THEN
          PRINT "Screen-Nummer gibt es nicht!"
          EXIT SUB
        END IF
      NEXT loop1%

      mem.opt&  = 2^0+2^16
      p.io&     = AllocMem&(100,mem.opt&)
```

```
p.port&   = p.io&+62
IF p.io&  = 0 THEN ERROR 7


f.breite%   = PEEKW(f.screen&+12)
f.hoehe%    = PEEKW(f.screen&+14)
f.rastport& = f.screen&+84
f.viewport& = f.screen&+44
f.colormap& = PEEKL(f.viewport&+4)
f.vp.modi%  = PEEKW(f.viewport&+32)


p.sigBit% = AllocSignal%(-1)
IF p.sigBit%=-1 THEN
  PRINT "Kein Signalbit frei!"
  CALL FreeMem(p.io&,100)
  EXIT SUB
END IF
p.sigTask& = FindTask&(0)

POKE  p.port&+8,4
POKEL p.port&+10,p.port&+34
POKE  p.port&  +15,p.sigBit%
POKEL p.port&+16,p.sigTask&
POKEL p.port&+20,p.port&+24
POKEL p.port&+28,p.port&+20
POKE  p.port&+34,ASC("P")
POKE  p.port&+35,ASC("R")
POKE  p.port&+36,ASC("T")

CALL AddPort(p.port&)

POKE  p.io&+8,5
POKEL p.io&+14,p.port&
POKEW p.io&+28,11
POKEL p.io&+32,f.rastport&
POKEL p.io&+36,f.colormap&
POKEL p.io&+40,f.vp.modi%
POKEW p.io&+44,x%
POKEW p.io&+46,y%
POKEW p.io&+48,breite%
POKEW p.io&+50,hoehe%
POKEL p.io&+52,f.breite%*f1
POKEL p.io&+56,f.hoehe%*f2

d.name$ = "printer.device"+CHR$(0)
status% = OpenDevice%(SADD(d.name$),0,p.io&,0)
IF status%<>0 THEN
  PRINT "Drucker ist nicht frei."
  CALL FreeMem(p.io&,100)
  CALL FreeSignal(p.sigBit%)
  EXIT SUB
```

```
    END IF

    fehler% = DoIO%(p.io&)

    CALL CloseDevice(p.io&)
    CALL RemPort(p.port&)
    CALL FreeMem(p.io&,100)
    CALL FreeSignal(p.sigBit%)
    PRINT "Fehlercode: ";fehler%
END SUB
```

6.5 Multi-Tasking-Hardcopy

Der Amiga kennt zwei verschiedene Arten, I/O zu betreiben: synchron und asynchron. Bisher arbeiteten wir stets mit synchroner I/O: Das Programm mußte warten, bis die Hardcopy fertiggestellt war. Das hat Vor- wie auch Nachteile: Während des Drucks kann das Programm die Zeichnung nicht verändern und dadurch zerstören. Andererseits bedeutet das Warten eine lästige Zeiteinbuße.

Abhilfe kann hier die ansynchrone I/O schaffen. Sie sendet die Druckdaten zum "printer.device" und gibt die Kontrolle unmittelbar danach wieder an den BASIC-Interpreter zurück. Das Programm muß also nicht auf den Drucker warten. Allerdings hat diese Methode neben diesem Vorteil einen schwerwiegenden Nachteil: äußerst komplexe Programmierung.

Zur Demonstration dieser eindrucksvollen Programmierung haben wir für Sie das Hardcopy-Programm aus Kapitel 6.1 auf asynchrone I/O umgeschrieben. Wie bisher haben Sie den "Hardcopy"-Befehl zur Hand, der den Inhalt des Ausgabefensters zum Drucker schickt. Das Besondere: Ihr BASIC-Programm wartet nicht mehr auf den Drucker, sondern setzt seine Arbeit unmittelbar fort!

Am Ende Ihres eigenen Programms muß der Aufruf "Loesch" erfolgen, mit dem der noch laufende I/O-Request befriedigt und der Systemspeicher zurückgegeben wird. Auch der Drucker wird erst hier geschlossen.

```
'#####################################
'#
'# Programm: Hardcopy V (Multitasking)
'# Datum: 13.4.87
'# Autor: tob
'# Version: 1.0
'#
'#####################################

' Dies ist eine Multi-Tasking Hardcopy Routine, die die
' gesamten Faehigkeiten des Amigas unter Beweis stellt:
' Das Programm braucht nicht auf den Ausdruck zu warten,
' sondern kann unmittelbar nach dem Aufruf weiterarbeiten,
' waehrend ein unabhaengiger Task das Ausdrucken uebernimmt.
' ACHTUNG: Der gesamte auszudruckende Bereich muss durch die
' Routine zwischengespeichert werden. Das kostet Speicher-
' platz!

PRINT "Suche die .bmap-Dateien..."

'GRAPHICS-Bibliothek
DECLARE FUNCTION BltBitMap% LIBRARY
DECLARE FUNCTION AllocRaster& LIBRARY
'FreeRaster()

'EXEC-Bibliothek
DECLARE FUNCTION AllocMem& LIBRARY
DECLARE FUNCTION OpenDevice% LIBRARY
DECLARE FUNCTION AllocSignal% LIBRARY
DECLARE FUNCTION FindTask& LIBRARY
DECLARE FUNCTION WaitIO% LIBRARY
DECLARE FUNCTION CheckIO% LIBRARY
'FreeMem()
'CloseDevice()
'FreeSignal()
'AddPort()
'RemPort()
'SendIO()

LIBRARY "exec.library"
LIBRARY "graphics.library"

init:     '* etwas zeichnen
          CLS
          CIRCLE (300,100),100
          LINE   (10,10)-(200,100),2,bf

          Hardcopy

demo:     '* Hier koennte Ihr Hauptprogramm stehen!
          WHILE INKEY$=""
            PRINT "Statt dieser Warteschleife koennte"
```

```
            PRINT "hier ein Grafikprogramm stehen, das"
            PRINT "nicht mehr auf den Drucker zu warten"
            PRINT "braucht!"
            PRINT
            PRINT "Beliebige Taste = Abbruch"
          WEND

          '* asynchrone I/O versorgen
          loesch
          END

SUB Hardcopy STATIC
      SHARED p.io&,p.sigBit%,f.rastport&
      SHARED p.port&
      mem.opt& = 2^0+2^16
      p.io&    = AllocMem&(240,mem.opt&)
      p.port&  = p.io&+62
      p.rast&  = p.io&+100
      p.bmap&  = p.io&+200
      IF p.io&=0 THEN ERROR 7

      f.fenster&  = WINDOW(7)
      f.rastport& = PEEKL(f.fenster&+50)
      f.bitmap&   = PEEKL(f.rastport&+4)
      f.breite%   = PEEKW(f.fenster&+112)
      f.hoehe%    = PEEKW(f.fenster&+114)
      f.screen&   = PEEKL(f.fenster&+46)
      f.viewport& = f.screen&+44
      f.colormap& = PEEKL(f.viewport&+4)
      f.vp.modi%  = PEEKW(f.viewport&+32)
      f.x%        = PEEKW(f.bitmap&)*8
      f.y%        = PEEKW(f.bitmap&+2)
      f.tiefe%    = PEEK(f.bitmap&+5)

      CALL CopyMem(f.rastport&,p.rast&,100)
      CALL CopyMem(f.bitmap&,p.bmap&,40)

      FOR loop1%=0 TO f.tiefe%-1
        p.plane&(loop1%) = AllocRaster&(f.x%,f.y%)
        IF p.plane&(loop1%)=0 THEN
          FOR loop2%=0 TO loop1%-1
            CALL FreeRaster(p.plane&(loop2%),f.x%,f.y%)
          NEXT loop2%
          CALL FreeMem(p.io&,240)
          PRINT "Out Of Memory!"
          EXIT SUB
        END IF
        POKEL p.bmap&+8+loop1%*4,p.plane&(loop1%)
      NEXT loop1%
      tempA$ = SPACE$(f.x%/8)
      pc%    = BltBitMap%(f.bitmap&,0,0,p.bmap&,0,0,f.x%,f.y%,200,255,
               SADD(tempA$))
```

```
      IF pc%<>f.tiefe% THEN ERROR 255

      POKEL p.rast&+4,p.bmap&
      f.rastport& = p.rast&

      p.sigBit% = AllocSignal%(-1)
      IF p.sigBit%=-1 THEN
        PRINT "Kein Signalbit frei!"
        CALL FreeMem(p.io&,240)
        EXIT SUB
      END IF
      p.sigTask& = FindTask&(0)

      POKE  p.port&+8,4
      POKEL p.port&+10,p.port&+34
      POKE  p.port&+15,p.sigBit%
      POKEL p.port&+16,p.sigTask&
      POKEL p.port&+20,p.port&+24
      POKEL p.port&+28,p.port&+20
      POKE  p.port&+34,ASC("P")
      POKE  p.port&+35,ASC("R")
      POKE  p.port&+36,ASC("T")

      CALL AddPort(p.port&)

      POKE  p.io&+8,5
      POKEL p.io&+14,p.port&
      POKEW p.io&+28,11
      POKEL p.io&+32,f.rastport&
      POKEL p.io&+36,f.colormap&
      POKEL p.io&+40,f.vp.modi%
      POKEW p.io&+48,f.breite%
      POKEW p.io&+50,f.hoehe%
      POKEL p.io&+52,f.breite%
      POKEL p.io&+56,f.hoehe%
      POKEW p.io&+60,4

      d.name$ = "printer.device"+CHR$(0)
      status% = OpenDevice%(SADD(d.name$),0,p.io&,0)
      IF status%<>0 THEN
        PRINT "Drucker ist nicht frei."
        CALL FreeMem(p.io&,240)
        CALL FreeSignal(p.sigBit%)
        EXIT SUB
      END IF

      CALL SendIO(p.io&)
      IF PEEK(p.io&+31)<>0 THEN: loesch
END SUB

SUB loesch STATIC
      SHARED p.io&,p.sigBit%,f.rastport&
```

```
    SHARED p.port&
    status%    = CheckIO%(p.io&)
    IF status% = 0 THEN
      PRINT "Printer noch in Betrieb!"
      PRINT "Bitte warten!"
    END IF
    fehler%    = WaitIO%(p.io&)
    l.bitmap&  = PEEKL(f.rastport&+4)
    l.x%       = PEEKW(l.bitmap&)*8
    l.y%       = PEEKW(l.bitmap&+2)
    l.tiefe%   = PEEK(l.bitmap&+5)
    FOR loop1%=1 TO l.tiefe%
      l.plane& = PEEKL(l.bitmap&+4+4*loop1%)
      CALL FreeRaster(l.plane&,l.x%,l.y%)
    NEXT loop1%

    CALL CloseDevice(p.io&)
    CALL RemPort(p.port&)
    CALL FreeMem(p.io&,240)
    CALL FreeSignal(p.sigBit%)
    PRINT "Error-Code: ";fehler%
END SUB
```

Eine eingehende Programmbeschreibung wäre hier fehl am Platze; sie gehört eher in ein Buch über das Exec-Betriebssystems. Hier lediglich in Kurzform, was das Programm tut:

Da Ihr BASIC-Programm unmittelbar nach Aufruf der Hardcopy weiterarbeitet, müssen sowohl Rastport wie auch Bitmap inklusive aller Bitplanes in einen Hilfsspeicher kopiert werden, damit sie ungestört vom weiteren Programmablauf ausgedruckt werden. Die Hauptarbeit leisten dabei die Exec-Funktion "CopyMem" (Achtung: Kickstart Version 1.2 benutzen!) und die Grafik-Routine "BltBitMap".

I/O-Block und Replyport werden wie gewohnt eingerichtet. Der Druckvorgang wird jedoch mit "SendIO" aktiviert.

Das SUB "loesch" hat die Aufgabe zu überprüfen, ob der Druckvorgang abgeschlossen ist. Das tut "CheckIO%". Ist das Resultat null, dann wird noch gedruckt. In jedem Fall wird "WaitIO%" aufgerufen. Intern wartet "WaitIO%" auf Vollendung des Drucks

und besorgt dann die ReplyMessage. Außerdem liest diese Routine das Error-Feld des IO-Blocks und liefert den Wert an das Programm.

Nun werden die Bitplanes und die Systemstrukturen an das System zurückgegeben, der Drucker wird geschlossen, das Signalbit freigegeben.

7. Laden von Fremdgrafiken: Der IFF-ILBM-Standard

"IFF" ist eine Abkürzung und steht für "Interchange File Format". Dieses Format erwuchs aus einer Überlegung, die die Firmen "Electronic Arts" und Commodore Amiga in der Anfangszeit des Amiga anstellten: Sowohl für Anwender als auch Programmierer dieses Computers könnte es nur schädlich sein, wenn jedes Grafikprogramm seine Bilder in seiner ganz eigenen Weise abspeichern würde. Als Folge könnte man beispielsweise "Graphicraft"-Bilder nur auf "Graphicraft", "Aegis"-Zeichnungen nur unter "Aegis" und "dPrint"-Grafiken nur mit "dPrint"-Programmen verwenden. Ein Tausch untereinander wäre unmöglich. Deshalb entwickelte man eine Standard-Methode, um Grafiken auf Diskette abzuspeichern. Dieses Standard-Format nennt sich "ILBM": Interleaved Bitmap. Eine "ILBM"-Datei unterteilt sich in mehrere Komponenten. Hier die wichtigsten:

"BMHD" = BitmapHeader

Identifikation: BMHDnnnn

Offset	Typ	Bezeichnung
+ 000	Word	Breite der Grafik
+ 002	Word	Höhe der Grafik
+ 004	Word	X-Position dieser Grafik
+ 006	Word	Y-Position dieser Grafik
+ 008	Byte	Anzahl der Bitplanes (Tiefe)
+ 009	Byte	Masking 0 = kein Masking 1 = Masking 2 = Transparent 3 = Lasso
+ 010	Byte	Datenkompression 0 = keine Kompression 1 = ByteRun1-Algorithmus
+ 011	Byte	unbenutzt

Offset	Typ	Bezeichnung
+ 012	Word	"transparente" Farbe
+ 014	Byte	X-Aspekt
+ 015	Byte	Y-Aspekt
+ 016	Word	Breite der Quellseite
+ 018	Word	Höhe der Quellseite

"CMAP" = ColormapColorMap

Identifikation: CMAPnnnn

Offset	Typ	Bezeichnung	
+ 001	Byte	rot	(0-255)
+ 002	Byte	grün	(0-255)
+ 003	Byte	blau	(0-255)

"BODY" = Bitplanes

Identifikation: BODYnnnn

Bitplane 1
Bitplane 2
(...)
Bitplane n

"CAMG" = Amiga Viewport Modi (Hi-Res, Lo-Res, Lace, etc.)

Identifikation: CAMGnnnn

Offset	Typ	Bezeichnung
+ 000	Long	Viewport-Modi (siehe Kap. 4)

sowie die Colorcycle-Information aus Graphicraft:

"CCRT" - Graphicraft Colorcycle Daten

Identifikation: "CCRTnnnn"

Offset	Typ	Bezeichnung
+ 000	Word	Richtung 0=rückwärts 1=vorwärts
+ 002	Byte	Start (Nr. des Farbregisters: 0-31)
+ 003	Byte	Ende (Nr. des Farbregisters: 0-31)
+ 004	Long	Sekunden
+ 008	Long	Micro-Sekunden

Aus diesen wichtigen Datenblöcken besteht die "ILBM"-Datei einer jeden nach diesem Verfahren gespeicherten Grafik.

Das folgende Programm demonstriert, wie solche Dateien in Ihren Rechner geladen und dargestellt werden können. Dadurch haben Sie die Möglichkeit, beispielsweise ein Anfangsbild für Ihr Grafikprogramm (oder ein Programm ganz anderer Natur) mit einem der bekannten Zeichenprogramme zu erstellen, um es anschließend mit unserer Routine am Programmstart darzustellen.

Hier das Programm:

```
'########################################
'#
'# Programm: Lade ILBM-Bild von Disk
'# Datum: 15.1.87
'# Autor: tob
'# Version: 1.0
'#
'# laedt Bilder aller Modi, incl. Hold-And-
'# Modify, HalfBrite und Graficraft Color
'# Cycle.
'#
'# basiert auf dem "ILBM" IFF Interleaved
'# Bitmap Standard veroeffentlicht am
'# 15. November 1985 von Jerry Morrison
'# (Electronic Arts USA) im "Commodore
'# Amiga ROM Kernel Manual Volume 2",
'# CBM Prod.-Nr. 327271-02 rev 2 vom
'# 12. September 1985, ab Seite H-25
'#
'########################################

PRINT "Suche die .bmap-Dateien..."
```

```
'DOS-Bibliothek
DECLARE FUNCTION xOpen& LIBRARY
DECLARE FUNCTION xRead& LIBRARY
DECLARE FUNCTION Seek& LIBRARY
'xClose()
'Delay()

'EXEC-Bibliothek

DECLARE FUNCTION AllocMem& LIBRARY
DECLARE FUNCTION DoIO% LIBRARY
DECLARE FUNCTION OpenDevice% LIBRARY
DECLARE FUNCTION AllocSignal% LIBRARY
DECLARE FUNCTION FindTask& LIBRARY
'FreeMem()

'GRAPHICS-Bibliothek
DECLARE FUNCTION AllocRaster& LIBRARY
'SetRast()
'LoadRGB4()

LIBRARY "dos.library"
LIBRARY "exec.library"
LIBRARY "graphics.library"

main:       '* Laden
            CLS
            PRINT "ILBM-Lader"
            PRINT
            PRINT "Laedt Standard-IFF-Grafikdateien in den"
            PRINT "Speicher und zeigt das Bild."
            PRINT
            PRINT "Der IFF-Lader unterstuetzt das Graficraft Color-
                   Cycle,"
            PRINT "sowie die Darstellung von Hold-And-Modify (4096
                   Farben)"
            PRINT "und Halfbrite (64 Farben)."
            PRINT
            PRINT "Gemaess den ILBM-Standards werden ggf. 'gepackte'"
            PRINT "Grafikdateien nach dem Byte1Run-Verfahren decodiert
                   und"
            PRINT "dargestellt."
            PRINT
            PRINT "Auf Wunsch kann das Bild auf einem grafikfaehigen
                   Drucker"
            PRINT "ausgedruckt werden."

            LINE INPUT "Name der ILBM-Datei: ";ilbm.file$
            LINE INPUT "Wollen Sie das Bild ausdrucken? (j/n)
            ";jn$
```

```
        LadeILBM ilbm.file$
        ColorCycle -1

        IF jn$="j" THEN
          WINDOW OUTPUT 2
          Hardcopy
        END IF

        WHILE INKEY$="":WEND
        ILBMende

        LIBRARY CLOSE


SUB LadeILBM(ilbm.name$) STATIC
      SHARED disk.handle&,buf.lesen&
      SHARED buf.farbe&,buf.rgb&
      SHARED ilbm.error$,signal%
      SHARED screen.farbe%,amiga.viewport&
      SHARED ilbm.vp.modi&
      SHARED amiga.rastport&

      disk.modeOldFile% = 1005
      disk.name$        = ilbm.name$+CHR$(0)

     '* ILBM-Datei oeffnen
      disk.handle&=xOpen&(SADD(disk.name$),1005)
      IF disk.handle&=0 THEN
        ilbm.error$="ILBM-Datei "+ilbm.name$+" nicht auf Disk/in diesem
                    Directory."
        GOTO ilbm.error.A
      END IF

     '* Disketten Lesebuffer einrichten
      mem.opt&=2^0+2^16
      buf.groesse&=240
      buf.add&=AllocMem&(buf.groesse&,mem.opt&)
      IF buf.add&=0 THEN
        ilbm.error$="Nicht genuegend Zwischenspeicher frei."
        GOTO ilbm.error.A
      END IF

     '* Buffer fuer Chunk-Bereiche unterteilen
      buf.lesen& = buf.add&+0*120
      buf.rgb&   = buf.add&+1*120

     '* Handelt es sich um eine ILBM-Datei?
      disk.gelesen&=xRead&(disk.handle&,buf.lesen&,12)
      ilbm.ID.$   = ""
      FOR loop1%  = 8 TO 11
        ilbm.ID.$ = ilbm.ID.$+CHR$(PEEK(buf.lesen&+loop1%))
      NEXT loop1%
```

```
      IF ilbm.ID.$<>"ILBM" THEN
        ilbm.error$="Datei "+ilbm.name$+" ist keine ILBM-Datei."
        GOTO ilbm.error.A
      END IF

     '* die Daten-Chunks des ILBM lesen
      WHILE (signal%<>1) AND (ilbm.error$="")
        LeseChunk
      WEND

     '* Fehler?
      ilbm.error.A:
      IF ilbm.error$<>"" THEN
        WINDOW CLOSE 2
        SCREEN CLOSE 1
        PRINT ilbm.error$
        EXIT SUB
      END IF

     '* alles ok!
      CALL LoadRGB4(amiga.viewport&,buf.rgb&,screen.farbe%)

     '* ILBM-Datei schliessen?
      IF disk.handle&<>0 THEN
        CALL xClose(disk.handle&)
      END IF
      IF buf.add&<>0 THEN
        CALL FreeMem(buf.add&,buff.groesse&)
        buf.add&=0
      END IF

     '* Viewmodes einstellen
      POKEW amiga.viewport&+32,ilbm.vp.modi&
END SUB

SUB ILBMende STATIC
      WINDOW CLOSE 2
      SCREEN CLOSE 1
END SUB

SUB ColorCycle(modus%) STATIC
      SHARED ccrt.richtung%
      SHARED ccrt.start%,amiga.colortable&
      SHARED ccrt.ende%,screen.farbe%
      SHARED ccrt.secs&,status%
      SHARED ccrt.mics&,amiga.viewport&

     '* vorgesehen?
      IF (status% AND 2^4)<>2^4 THEN
        EXIT SUB
      END IF
```

```
    '* Variablenfelder einrichten
     DIM farbe.original%(screen.farbe%-1)
     DIM farbe.aktuell%(screen.farbe%-1)

    '* alles ok,alte Farben aus Viewport retten
     FOR loop1%=0 TO screen.farbe%-1
       farbe.original%(loop1%)=PEEKW(amiga.colortable&+2*loop1%)
       farbe.aktuell%(loop1%)=farbe.original%(loop1%)
     NEXT loop1%

    '* Color Cycling
     WHILE modus%<>0
      '* Modus?
       IF modus%<0 THEN
         in$=INKEY$
         IF in$<>"" THEN modus%=0
       ELSE
         modus%=modus%-1
       END IF

      '* vorwaerts?
       IF ccrt.richtung%=1 THEN
         ccrt.backup%=farbe.aktuell%(ccrt.start%)
         FOR loop1%=ccrt.start%+1 TO ccrt.ende%
           farbe.aktuell%(loop1%-1)=farbe.aktuell%(loop1%)
         NEXT loop1%
         farbe.aktuell%(ccrt.ende%)=ccrt.backup%

      '* rueckwaerts?
       ELSE
         ccrt.backup%=farbe.aktuell%(ccrt.ende%)
         FOR loop1%=ccrt.start%-1 TO ccrt.ende% STEP -1
           farbe.aktuell%(loop1%+1)=farbe.aktuell%(loop1%)
         NEXT loop1%
         farbe.aktuell%(ccrt.start%)=ccrt.backup%

       END IF
       CALL LoadRGB4(amiga.viewport&,VARPTR(farbe.aktuell%(0)),
       screen.farbe%)
       timeout&=50*(ccrt.secs&+(ccrt.mics&/1000000&))
       CALL Delay(timeout&)
     WEND

    '* Originalfarben wiederherstellen
     CALL LoadRGB4(amiga.viewport&,VARPTR(farbe.original%(0)),
     screen.farbe%)

    '* Felder zurueckgeben
     ERASE farbe.original%
     ERASE farbe.aktuell%
END SUB
```

```
SUB LeseChunk STATIC
      SHARED disk.handle&,buf.lesen&
      SHARED buf.farbe&,buf.rgb&
      SHARED ilbm.error$,signal%
      SHARED screen.farbe%,amiga.viewport&
      SHARED ilbm.vp.modi&,status%
      SHARED amiga.rastport&
      SHARED ccrt.richtung%
      SHARED ccrt.start%,amiga.colortable&
      SHARED ccrt.ende%,screen.farbe%
      SHARED ccrt.secs&
      SHARED ccrt.mics&

     '* Chunk-Kopf lesen
      disk.gelesen& = xRead&(disk.handle&,buf.lesen&,8)
      ilbm.chunk&   = PEEKL(buf.lesen&+4)
      ilbm.ID.$     = ""
      FOR loop1%  = 0 TO 3
        ilbm.ID.$ = ilbm.ID.$+CHR$(PEEK(buf.lesen&+loop1%))
      NEXT loop1%

     '* Der BitMap-Header (BMHD) ?
      IF ilbm.ID.$="BMHD" THEN
       '* Chunk-Inhalt lesen
        disk.gelesen&=xRead&(disk.handle&,buf.lesen&,ilbm.chunk&)

        status%=status% OR 2^0
        ilbm.breite%  = PEEKW(buf.lesen&+0)
        ilbm.hoehe%   = PEEKW(buf.lesen&+2)
        ilbm.tiefe%   = PEEK (buf.lesen&+8)
        ilbm.modus%   = PEEK (buf.lesen&+10)
        screen.breite%   = PEEKW(buf.lesen&+16)
        screen.hoehe%    = PEEKW(buf.lesen&+18)

       '* daraus Darstellungsparameter bilden
        ilbm.bytes%   = ilbm.breite%/8
        screen.bytes% = screen.breite%/8
        screen.farbe% = 2^(ilbm.tiefe%)

       '* alles klarmachen zum Display

       '* HiRes (High Resolution?)
        IF screen.breite%>320 THEN
          screen.modus%=2
        ELSE
          screen.modus%=1
        END IF

       '* Interlace (y=0 - 511) PAL only!!
        IF screen.hoehe%>256 THEN screen.modus%=screen.modus%+2
```

```
  '* ilbm.tiefe%=6 -> HAM/Halfbrite?
   IF ilbm.tiefe%=6 THEN
     ilbm.reg%=-1
   END IF

   tiefe%=ilbm.tiefe%+ilbm.reg%
   SCREEN 1,screen.breite%,screen.hoehe%,tiefe%,screen.modus%
   WINDOW 2,,,0,1

  '* System Parameter
   amiga.fenster&    = WINDOW(7)
   amiga.screen&     = PEEKL(amiga.fenster&+46)
   amiga.viewport&   = amiga.screen&+44
   amiga.rastport&   = amiga.screen&+84
   amiga.colormap&   = PEEKL(amiga.viewport&+4)
   amiga.colortable& = PEEKL(amiga.colormap&+4)
   amiga.bitmap&     = PEEKL(amiga.rastport&+4)
   FOR loop1%=0 TO ilbm.tiefe%-1
     amiga.plane&(loop1%)=PEEKL(amiga.bitmap&+8+loop1%*4)
   NEXT loop1%

  '* fuer HAM/Halfbrite 6. Bitplane einrichten
   IF ilbm.reg%=-1 THEN
     ilbm.reg%=0
     newplane&=AllocRaster&(screen.breite%,screen.hoehe%)
     IF newplane&=0 THEN
       ilbm.error$="Kein Speicher fuer 6. Bitplane frei!"
     ELSE
       POKE  amiga.bitmap&+5,6
       POKEL amiga.bitmap&+28,newplane&
     END IF
   END IF

'* Farb-Tabelle (CMAP) ?
 ELSEIF ilbm.ID.$="CMAP" THEN
  '* Chunk-Inhalt lesen
   disk.gelesen&=xRead&(disk.handle&,buf.lesen&,ilbm.chunk&)

   status%=status% OR 2^1

  '* RGB-Tabelle errechnen
   FOR loop1% = 0 TO screen.farbe% -1
     farbe.rot%   = PEEK(buf.lesen&+loop1%*3+0)
     farbe.gruen% = PEEK(buf.lesen&+loop1%*3+1)
     farbe.blau%  = PEEK(buf.lesen&+loop1%*3+2)
     farbe.rgb%   = farbe.gruen%+16*farbe.rot%+1/16*farbe.
     blau%
     POKEW buf.rgb&+2*loop1%,farbe.rgb%
   NEXT loop1%

  '* Alignment
   IF (ilbm.chunk OR 1)=ilbm.chunk THEN
```

```
    disk.gelesen&=xRead&(disk.handle&,buf.lesen&,1)
  END IF

'* Viewport Modi (CAMG) AMIGA ?
 ELSEIF ilbm.ID.$="CAMG" THEN
  '* Chunk-Inhalt lesen
   disk.gelesen&=xRead&(disk.handle&,buf.lesen&,ilbm.chunk&)

   status%=status% OR 2^3
   ilbm.vp.modi& = PEEKL(buf.lesen&)

'* Color-Cycle-Daten (CCRT) ?
 ELSEIF ilbm.ID.$="CCRT" THEN
  '* Chunk-Inhalt lesen
   disk.gelesen&=xRead&(disk.handle&,buf.lesen&,ilbm.chunk&)

   status%=status% OR 2^4
   ccrt.richtung% = PEEKW(buf.lesen&+0)
   ccrt.start%    = PEEK (buf.lesen&+2)
   ccrt.ende%     = PEEK (buf.lesen&+3)
   ccrt.secs&     = PEEKL(buf.lesen&+4)
   ccrt.mics&     = PEEKL(buf.lesen&+8)

'* Bitplanes (BODY) ?
 ELSEIF ilbm.ID.$="BODY" THEN
   status%=status% OR 2^2

  '* nicht-komprimierte Daten
   IF ilbm.modus%=0 THEN
     FOR loop1%=0 TO ilbm.hoehe%-1
       FOR loop2%=0 TO ilbm.tiefe%-1
         screen.zeile&=amiga.plane&(loop2%)+(loop1%*screen.bytes%)
         disk.gelesen&=xRead&(disk.handle&,screen.zeile&,
                              ilbm.bytes%)
       NEXT loop2%
     NEXT loop1%

  '* komprimierte Daten (ByteRun1-Encoding)
   ELSEIF ilbm.modus%=1 THEN
     FOR loop1%=0 TO ilbm.hoehe%-1
       FOR loop2%=0 TO ilbm.tiefe%-1
         screen.zeile&=amiga.plane&(loop2%)+(loop1%*screen.bytes%)
         zaehler%=0

        '* Decodieren
         WHILE zaehler%<ilbm.bytes%
           disk.gelesen& = xRead&(disk.handle&,buf.lesen&,1)
           code%         = PEEK(buf.lesen&)
          '* Codierung 1: lese n Bytes uncodiert
           IF code%<128 THEN
             disk.gelesen& = xRead&(disk.handle&,screen.
             zeile&+zaehler%,code%+1)
```

```
            zaehler%      = zaehler%+code%+1
          '* Codierung 2: wiederhole naechstes Byte (257-n)-mal
           ELSEIF code%>128 THEN
            disk.gelesen& = xRead&(disk.handle&,buf.lesen&,1)
            disk.byte%    = PEEK(buf.lesen&)
            FOR loop3%=zaehler% TO zaehler%+257-code%
              POKE screen.zeile&+loop3%,disk.byte%
            NEXT loop3%
            zaehler%=zaehler%+257-code%
          '* Codierung 3: no operation
           ELSE
            'nop
           END IF
         WEND
       NEXT loop2%
     NEXT loop1%

    '* andere Decodierungsmethode
     ELSE
      ilbm.error$="Daten-Kompressionsalgorithmus unbekannt."
     END IF

  '* unwichtigen Chunk verarbeiten (GRAB, DEST, SPRT, etc.)
   ELSE
    '* gerade Anzahl Bytes lesen
    IF (ilbm.chunk% OR 1)=ilbm.chunk% THEN
      ilbm.chunk%=ilb.chunk%+1
    END IF

    '* Disk-Cursor verschieben
    mode.current%=0
    stat&=Seek&(disk.handle&,ilbm.chunk%,0)
    IF stat&=-1 THEN
      ilbm.error$="DOS-Fehler. Seek() schlug fehl."
    END IF

   END IF

  '* Fehler-Check
   IF disk.gelesen&<0 THEN
     ilbm.error$="DOS-Fehler. Read() schlug fehl."
  '* EOF (End-Of-File) erreicht?
   ELSEIF disk.gelesen&=0  AND ((status% AND 7)<>7) THEN
     ilbm.error$="ILBM-Datenchunks nicht vorhanden."
     signal%=1
   ELSEIF (status% AND 7)=7 THEN
     signal%=1
   END IF
END SUB

'* Dies ist die Hardcopy-Routine I aus diesem Buch, ins ILBM-Prog
'* integriert:
```

```
SUB Hardcopy STATIC
      mem.opt& = 2^0+2^16
      p.io&    = AllocMem&(100,mem.opt&)
      p.port&  = p.io&+62
      IF p.io& = 0 THEN ERROR 7

      f.fenster&   = WINDOW(7)
      f.rastport&  = PEEKL(f.fenster&+50)
      f.breite%    = PEEKW(f.fenster&+112)
      f.hoehe%     = PEEKW(f.fenster&+114)
      f.screen&    = PEEKL(f.fenster&+46)
      f.viewport&  = f.screen&+44
      f.colormap&  = PEEKL(f.viewport&+4)
      f.vp.modi%   = PEEKW(f.viewport&+32)


      p.sigBit% = AllocSignal%(-1)
      IF p.sigBit% = -1 THEN
        PRINT "Kein Signalbit frei!"
        CALL FreeMem(p.io&,100)
        EXIT SUB
      END IF
      p.sigTask& = FindTask&(0)

      POKE  p.port&+8,4
      POKEL p.port&+10,p.port&+34
      POKE  p.port&+15,p.sigBit%
      POKEL p.port&+16,p.sigTask&
      POKEL p.port&+20,p.port&+24
      POKEL p.port&+28,p.port&+20
      POKE  p.port&+34,ASC("P")
      POKE  p.port&+35,ASC("R")
      POKE  p.port&+36,ASC("T")

      CALL AddPort(p.port&)

      POKE  p.io&+8,5
      POKEL p.io&+14,p.port&
      POKEW p.io&+28,11
      POKEL p.io&+32,f.rastport&
      POKEL p.io&+36,f.colormap&
      POKEL p.io&+40,f.vp.modi%
      POKEW p.io&+48,f.breite%
      POKEW p.io&+50,f.hoehe%
      POKEL p.io&+52,f.breite%
      POKEL p.io&+56,f.hoehe%
      POKEW p.io&+60,4

      d.name$ = "printer.device"+CHR$(0)
      status% = OpenDevice%(SADD(d.name$),0,p.io&,0)
      IF status%<>0 THEN
```

```
        PRINT "Drucker ist nicht frei."
        CALL  FreeMem(p.io&,100)
        CALL  FreeSignal(p.sigBit%)
        EXIT SUB
      END IF

      fehler% = DoIO%(p.io&)

      CALL CloseDevice(p.io&)
      CALL RemPort(p.port&)
      CALL FreeMem(p.io&,100)
      CALL FreeSignal(p.sigBit%)
      PRINT "Fehlercode: ";fehler%
END SUB
```

Für Sie sind die SUBs "LadeILBM", "ColorCycle" und "ILB-Mende" von Interesse. "LadeILBM" verlangt in einem String den Namen des ILBM-Bildes auf Diskette, das geladen werden soll. Natürlich muß sich das Bild im aktiven Diskettenverzeichnis befinden, um auch gefunden zu werden (CHDIR "Verzeichnis"). Dieses SUB lädt das Bild und zeigt es auf dem Bildschirm an. Nun können Sie das SUB "ColorCycle" aufrufen. Sollte ein "CCRT"-Colorcycle-Datenblock gefunden worden sein, übernimmt das Programm das Cycling, was dem Bild einen Effekt der Bewegung verleiht. Das SUB verlangt als Parameter einen Integer-Wert. Ist dieser negativ, dann tauscht der Amiga solange die Farben, bis eine beliebige Taste gedrückt wird. Ist er positiv, dann cyclet der Amiga das Bild Wert-mal. Das SUB "ILBMende" schließlich beendet das Display des Bildes und schließt Screen und Fenster.

Neu in diesem Programm sind die Routinen der DOS-Bibliothek. Leider läßt sich das Laden von ILBM-Dateien nicht über die eingebauten OPEN/INPUT#/CLOSE Befehle des Amiga bewerkstelligen, denn diese unterschlagen Nullen in den Daten. Hier eine kurze Erklärung der verwendeten DOS-Routinen:

```
name$=name$+CHR$(0)
disk.handle&=xOpen&(SADD(name$),1005)
```

name$:	Name der zu öffnenden Datei
1005:	ModeOldFile - Datei existiert bereits

1006:	ModeNewFile - neue Datei dieses Namens wird erzeugt
disk.handle&:	BPTR (Zeiger/4) auf Handler Datenblock wenn 0, dann schlug xOpen fehl.

```
gelesen&=xRead(disk.handle&,buffer&,bytes&)
```

disk.handle&:	Adresse vom xOpen-Aufruf
buffer&:	Adresse eines freien Speicherbereiches
bytes&:	Anzahl der von der aktuellen Disk-Cursor-Position zu lesenden Bytes, die allesamt in den Pufferspeicher passen müssen!
gelesen&:	Anzahl der gelesenen Bytes =0: EOF (End Of File) kleiner als 0: Lesefehler

```
oldpos&=Seek(disk.handle&,offset%,modus%)
```

disk.handle&:	Adresse vom xOpen-Aufruf
offset%:	Anzahl der Bytes, um die der Disk-Cursor verschoben werden soll
modus%:	0 = ab augenblicklicher Position -1 = ab Datei-Anfang 1 = ab Datei-Ende

```
CALL xClose(disk.handle&)
```

disk.handle&:	Handle vom xOpen-Befehl; schließt Datei

```
CALL Delay(ticks).
```

tick	= 1/50 Sekunde
Microsekunde	= 1/1000000 Sekunde

Wartet angegebene Zeit (jedoch nicht busy-waiting; während das Programm wartet, wird zusätzliche Rechenzeit für das System frei.)

Besonderheiten des Programms:

Dieses Programm unterstützt nicht nur die AmigaBASIC Display Modi wie Lo-Res, Hi-Res und Interlace. Zusätzlich können auch ILBM-Grafiken im Halfbrite- (64 Farben) und HAM-Modus (4096 Farben) dargestellt werden. Beide Modi arbeiten mit 6 Bitplanes. Tritt einer der beiden Modi auf, wird eine sechste Bitplane in den Display-Screen eingebaut. Diese Bitplane wird nirgendwo durch "FreeRaster" zurückgegeben, denn sobald der "SCREEN CLOSE"-Befehl den neuen Screen schließt, werden automatisch alle Bitplanes, auch die nachträglich eingebaute sechste, entfernt.

Das Programm ist in der Lage, komprimierte Bitplanes nach dem "ByteRun1"-Verfahren zu dekodieren. Bei diesem Verfahren werden zwei Kontrollcodes verwendet: Ist das gelesene Byte kleiner als 128, dann werden der Byte-Anzahl folgende Bytes direkt übernommen. Ist das Byte größer als 128, dann wird das nächstfolgende Byte (257-Byte)-mal wiederholt (normalerweise wird mit signed bytes von -127 bis +128 gearbeitet, daher die etwas merkwürdige Umrechnung). Ist das Byte =128, passiert nichts (NOP).

8. Die Anwendung: 1024x1024-Punkte-Malprogramm

In den vorangegangenen Kapiteln haben Sie die verschiedenen Grafik-Systemkomponenten des Amiga kennen- und zu programmieren gelernt. Als Abschluß haben wir für Sie ein Programm erstellt, das einmal die Möglichkeiten der Superbitmap-Layer anhand eines kleinen Malprogramms aufzeigt. Mit eingeflossen sind natürlich auch Kenntnisse der Amiga-Zeichensätze, der Zeichenmodi und der verschiedenen Schriftarten. Hier unser Malomat - und was "er" kann:

- voll maus- und menügesteuert
- bis zu 1024x1024 Punkte große Zeichnungen
- Softscrolling über die gesamte Zeichenfläche
- Kreise, Linien, Rechtecke in bewährter Rubberband-Technik
- Freihand-Zeichnen
- Textausgabe in JAM1, JAM2, Complement und Inverse
- bis zu 19 verschiedene Zeichensätze
- Outline-, Kursiv-, Fett-, Underline-Text
- luxuriöse Hardcopy-Funktionen:
    - Ausdruck der gesamten 1024x1024-Punkte-Grafik
    - Ausschnittsvergrößerung/-verkleinerung
    - Verzerrung
- Flächen füllen
- Zeichengrid
- Block löschen
- Kopieren von Bildschirmausschnitten
- selbstdefinierte Pinsel und Pattern

Wegen der enormen Abmessungen der Bitplanes arbeitet dieses Zeichenprogramm mit nur einer einzigen Bitplane, Zeichnungen erscheinen daher in schwarz/weiß. Dieses Programm ist wie geschaffen für Zeichnungen, die anschließend auf den Drucker ausgegeben werden sollen. Wegen der großen Zeichenfläche las-

sen sich auch detaillierte Grafiken erstellen, die dann in Originalgröße oder verkleinert auf einen grafikfähigen Drucker ausgegeben werden können.

```
'#####################################
'#
'# Programm: Superbitmap Zeichenprogramm
'# Datum: 16.4.87
'# Autor: tob
'# Version: 1.0
'#
'#####################################

PRINT "Suche die .bmap-Dateien..."

'GRAPHICS-Bibliothek
DECLARE FUNCTION AskSoftStyle& LIBRARY
DECLARE FUNCTION SetSoftStyle& LIBRARY
DECLARE FUNCTION OpenFont& LIBRARY
DECLARE FUNCTION AllocRaster& LIBRARY

'EXEC-Bibliothek
DECLARE FUNCTION DoIO& LIBRARY
DECLARE FUNCTION OpenDevice& LIBRARY
DECLARE FUNCTION AllocSignal& LIBRARY
DECLARE FUNCTION FindTask& LIBRARY
DECLARE FUNCTION AllocMem& LIBRARY

'DISKFONT-Bibliothek
DECLARE FUNCTION OpenDiskFont& LIBRARY
DECLARE FUNCTION AvailFonts& LIBRARY

'LAYERS-Bibliothek
DECLARE FUNCTION CreateBehindLayer& LIBRARY
DECLARE FUNCTION UpFrontLayer& LIBRARY
DECLARE FUNCTION BehindLayer& LIBRARY

LIBRARY "layers.library"
LIBRARY "graphics.library"
LIBRARY "exec.library"
LIBRARY "intuition.library"
LIBRARY "diskfont.library"

setup:     '* Es geht los:
           PRINT "Mal-O-Mat Zeichenprogramm"
           PRINT "-------------------------"
           PRINT
           PRINT "Wollen Sie mit einem LoRes(1) oder HiRes(2) Schirm
                  arbeiten"
           PRINT "(keinerlei Einfluss auf Groesse der Zeichenflaeche)?"
```

```
            PRINT
            LINE INPUT "Ihre Wahl (1 oder 2) --> ";jn$
            IF jn$="2" THEN
              scrWeite% = 640
              scrMode%  =   2
            ELSE
              scrWeite% = 320
              scrMode%  =   1
            END IF

initPar:    '* Screen Parameter
            scrHoehe%    = 256
            scrTiefe%    =   1
            scrNr%       =   1
            WBenchScrNr% = -1

            '* Fenster Parameter
            windWeite% = scrWeite%-9
            windHoehe% = scrHoehe%-26
            windNr%    = 1
            windTitle$ = "Arbeitsflaeche"
            windMode%  = 16

            '* Fenster Gadgets
            Xoffset%     = 15
            GadX%        = windWeite%-Xoffset%+3
            GadY%        = 11
            GadSX%       = Xoffset%-3
            GadSY%       = GadSX%-1
            GadZahl%     = 5
            GadToleranz% = 1
            Gad$(0)      = "^"
            Gad$(1)      = "v"
            Gad$(2)      = "<"
            Gad$(3)      = ">"
            Gad$(4)      = "H"

            '* CAD Super Bitmap
            superWeite%  = 800
            superHoehe%  = 400
            superFlag%   =   4

            '* Layer Groesse
            layMinX% =  3
            layMinY% = 11
            layMaxX% = windWeite%-8-Xoffset%
            layMaxY% = windHoehe%

            '* Drawing Mode
            draw%    = 4
            modus$   = "FREIHAND"
            drMd%    = 0
```

```
            style%   = 0
            swapper% = 1
            kl%      = 1
            grid1%   = 1
            grid2%   = 1
            fontHoehe% = 8
            DIM get.array%(1)

           '* Printer-Parameter
            printX0%   = 0
            printY0%   = 0
            printX1%   = superWeite%
            printY1%   = superHoehe%
            printSpec% = 4


initDisp:  '* Screen und Fenster oeffnen
            SCREEN scrNr%,scrWeite%,scrHoehe%,scrTiefe%,scrMode%
            WINDOW windNr%,windTitle$,(0,0)-(windWeite%,windHoehe%),
                   windMode%,scrNr%
            WINDOW OUTPUT windNr%
            PALETTE 1,0,0,0
            PALETTE 0,1,1,1

            DIM area.pat%(3):DIM full%(1)
            area.pat%(0) = &H1111
            area.pat%(1) = &H2222
            area.pat%(2) = &H4444
            area.pat%(3) = &H8888
            PATTERN ,area.pat%
            PAINT (100,50),1,1
            full%(0)=&HFFFF
            full%(1)=full%(0)
            PATTERN ,full%
           '* TmpRas einrichten

            buffergroesse& = superWeite%*superHoehe%/8
            buffer&        = PEEKL(WINDOW(8)+12)
            IF buffer&<>0 THEN
              fillflag% = 1
              mem&  = PEEKL(buffer&)
              size& = PEEKL(buffer&+4)
              CALL FreeMem(mem&,size
              opt&  = 2^0+2^1+2^16
              buf&  = AllocMem&(buffergroesse&,opt&)
              IF buf&=0 THEN
                fillflag%=0
                POKEL WINDOW(8)+12,0
              ELSE
                POKEL buffer&,buf&
                POKEL buffer&+4,buffergroesse&
              END IF
```

```
            ELSE
              fillflag%=0
            END IF

initSys:    '* System-Parameter lesen
            windAdd&       = WINDOW(7)
            scrAdd&        = PEEKL(windAdd&+46)
            scrViewPort&   = scrAdd&+44
            scrColMap&     = PEEKL(scrViewPort&+4)
            scrBitMap&     = scrAdd&+184
            scrLayerInfo&  = scrAdd&+224
            scrMode%       = PEEKW(scrViewPort&+32)
            font&          = PEEKL(WINDOW(8)+52)

initSBMap:  '* Superbitmap schaffen
            opt&          = 2^0+2^1+2^16
            superBitmap&  = AllocMem&(40,opt&)
            IF superBitmap&=0 THEN
              PRINT "Hm. Nicht mal 40 Bytes, nein?"
              ERROR 7
            END IF

           '* ...und in Betrieb nehmen
            CALL InitBitMap(superBitmap&,scrTiefe%,superWeite%,super
                 Hoehe%)
            superPlane& = AllocRaster&(superWeite%,superHoehe%)
            IF superPlane& = 0 THEN
              PRINT "Kein Plaaaaatz!"
              CALL FreeMem(superBitmap&,40)
              ERROR 7
            END IF
            POKEL superBitmap&+8,superPlane&

           '* Superbitmap-Layer oeffnen
            SuperLayer& = CreateBehindLayer&(scrLayerInfo&,scrBitMap&,
                          layMinX%,layMinY%, layMaxX%,layMaxY%,
                          superFlag%,superBitmap&)
            IF SuperLayer& = 0 THEN
              PRINT "Heute keine Layer!"
              CALL FreeRaster(superPlane&,superWeite%,superHoehe%)
              CALL FreeMem(superBitmap&,40)
              ERROR 7
            END IF

           '* neuer RastPort
            SuperRast& = PEEKL(SuperLayer&+12)

initPrint:  '* Drucker initialisieren
            opt& = 2^0+2^16
            pio& = AllocMem&(100,opt&)
            IF pio&<>0 THEN
              port&   = pio&+62
```

```
            sigBit% = AllocSignal&(-1)
            IF sigBit%<>-1 THEN
              sigTask& = FindTask&(0)
              POKE  port&+8,4
              POKEL port&+10,port&+34
              POKE  port&+15,sigBit%
              POKEL port&+16,sigTask&
              POKEL port&+20,port&+24
              POKEL port&+28,port&+20
              POKEL port&+34,1347572736&
              CALL AddPort(port&)
              POKE  pio&+8,5
              POKEL pio&+14,port&
              POKEW pio&+18,12
              POKEW pio&+28,11
              POKEL pio&+32,SuperRast&
              POKEL pio&+36,scrColMap&
              POKEL pio&+40,scrMode%
              POKEW pio&+48,superWeite%
              POKEW pio&+50,superHoehe%
              POKEL pio&+52,superWeite%
              POKEL pio&+56,superHoehe%
              POKEW pio&+60,4
            ELSE
              printflag% = 1
              CALL FreeMem(pio&,100)
            END IF
          ELSE
            printflag% = 1
          END IF

prepare:  '* Unsere Move-Gadgets zeichnen
          CALL SetDrMd(WINDOW(8),5)
          FOR loop% = 0 TO GadZahl%-1
            LINE  (GadX%,GadY%+(GadSY%+5)*loop%)-(GadX%+GadSX%,GadY%+
                   GadSY%+(GadSY%+5)*loop%),1,bf
            gadMaus%(loop%) = GadY%+(GadSY%+5)*loop%-4-GadToleranz%
            CALL  Move(WINDOW(8),GadX%+3,GadY%+((GadSY%+5)*loop%)+8)
            PRINT Gad$(loop%)
          NEXT loop%
          CALL SetDrMd(WINDOW(8),1)

          '* Zeichenflaeche vorbereiten
          CALL SetRast(SuperRast&,0)

          '* Kalibrierung zeichnen
          FOR loop% = 0 TO windWeite%-Xoffset% STEP 3
            IF loop%/15 = INT(loop%/15) THEN
              LINE (loop%,windHoehe%)-(loop%,windHoehe%-10)
            ELSE
              LINE (loop%,windHoehe%)-(loop%,windHoehe%-5)
            END IF
```

```
NEXT loop%
FOR loop% = 0 TO windHoehe% STEP 2
  IF loop%/10 = INT(loop%/10) THEN
    LINE (windWeite%-Xoffset%,loop%)-(windWeite%-10X-off
          set%,loop%)
  ELSE
    LINE (windWeite%-Xoffset%,loop%)-(windWeite%-5-Xoff
          set%,loop%)
  END IF
NEXT loop%

'* Layer hervorzaubern
e& = UpFrontLayer&(scrLayerInfo&,SuperLayer&)
GOSUB newpointer

'* Layer unaufdeckbar machen und integrieren
POKEL SuperLayer&+40,windAdd&
backup.rast&  = PEEKL(SuperLayer&+12)
backup.layer& = PEEKL(WINDOW(8))
POKEL SuperLayer&+12,WINDOW(8)
POKEL WINDOW(8),SuperLayer&
SuperRast&=WINDOW(8)

GOSUB koord

'* Menue-Steuerung
MENU 1,0,1,"Service"
  MENU 1,1,1,"Screen loeschen"
  MENU 1,2,1,"Koordinaten Ein"
  MENU 1,3,1,"--------------"
  MENU 1,4,1,"Transparent    "
  MENU 1,5,1,"JAM 2          "
  MENU 1,6,1,"Complement     "
  MENU 1,7,1,"Inverse        "
  MENU 1,8,1,"--------------"
  MENU 1,9,1,"normal/reset   "
  MENU 1,10,1,"kursiv         "
  MENU 1,11,1,"fett           "
  MENU 1,12,1,"unterstrichen  "
  MENU 1,13,1,"outline        "
  MENU 1,14,1,"--------------"
  MENU 1,15,1,"s/w -> w/s     "
  MENU 1,16,1,"Kopfleiste ein/aus"
  MENU 1,17,1,"Q  U  I  T  !  "
MENU 2,0,1,"Zeichnen"
  MENU 2,1,1,"Kreis   "
  MENU 2,2,1,"Rechteck"
  MENU 2,3,1,"Linien  "
  MENU 2,4,1,"freihand"
  MENU 2,5,1,"Text    "
  MENU 2,6,1,"Loeschen"
  MENU 2,7,fillflag%,"Fill    "
```

```
    MENU 2,8,1,"Raster/Grid"
    MENU 2,9,1,"Grid Reset "
    MENU 2,10,1,"Get Area   "
    MENU 2,11,1,"Paint Area "
  MENU 3,0,1,"Font"
    MENU 3,1,1,"Fonts laden"
  MENU 4,0,1,"I/O"
    MENU 4,1,1,"Drucken"
    MENU 4,2,1,"Param. "

  ON MENU GOSUB MenuCtrl
  MENU ON

mcp:  '* Master Control Program (Tron laesst gruessen...)
  WHILE forever=forever
    test%=MOUSE(0)
    mx%=MOUSE(1)
    my%=MOUSE(2)
    GOSUB updateDisp
    CALL SetDrMd(SuperRast&,drMd%)
    enable%=AskSoftStyle&(SuperRast&)
    n&=SetSoftStyle&(SuperRast&,style%,enable%)
   '* zeichnen!
    IF mx%>layMinX% AND mx%<layMaxX% AND test%<0 THEN
      IF draw%=4 THEN
        GOSUB freedraw
      ELSEIF draw%=10 THEN
        GOSUB paintdraw
      ELSEIF draw%=5 THEN
        GOSUB drawtext
      ELSEIF draw%=7 THEN
        GOSUB filler
      ELSE
        GOSUB drawit
        IF draw%=3 AND fetch%=1 THEN
          printX0% = cX%+subox%
          printX1% = 1+cX%+subox%+oldrcX%
          printY0% = cY%+suboy%
          printY1% = 1+cY%+suboy%+oldrcY%
          GOTO continue
        ELSEIF draw%=3 AND grid%=1 THEN
          x1%=cX%+subox%
          y1%=cY%+suboy%
          x2%=cX%+subox%+oldrcX%
          y2%=cY%+suboy%+oldrcY%
          IF x1%>x2% THEN SWAP x1%,x2%
          IF y1%>y2% THEN SWAP y1%,y2%
          breit%= x2%-x1%
          hoch% = y2%-y1%
          IF copy%=0 THEN
            grid1%=breit%
            grid2%=hoch%
```

```
                ELSE
                  g.size&=6+(hoch%+1)*2*INT((breit%+16)/16)
                  IF g.size&>(FRE(0)-1000) THEN
                    BEEP
                  ELSE
                    ERASE get.array%
                    DIM get.array%(g.size&/2)
                    GET (x1%,y1%)-(x2%,y2%),get.array%
                  END IF
                END IF
              END IF
            END IF
          ELSEIF (test%<0 AND mx%>layMaxX%) THEN
          '* Scroll-Gadgets betaetigt?
            IF my%>gadMaus%(4) THEN
              GOSUB ScrollHome
            ELSEIF my%>gadMaus%(3) THEN '<-
              GOSUB ScrollLinks
            ELSEIF my%>gadMaus%(2) THEN '->
              GOSUB ScrollRechts
            ELSEIF my%>gadMaus%(1) THEN 'up
              GOSUB ScrollHoch
            ELSEIF my%>gadMaus%(0) THEN 'down
              GOSUB ScrollRunter
            END IF
          END IF
        WEND

deleteSys: '* System entfernen
        buf&=PEEKL(WINDOW(8)+12)
        IF buf&<>0 THEN
          buffer&=PEEKL(buf&)
          size&=PEEKL(buf&+4)
          CALL FreeMem(buffer&,size&)
          POKEL WINDOW(8)+12,0
        END IF
        IF ptr&<>0 THEN
          CALL ClearPointer(WINDOW(7))
          CALL FreeMem(ptr&,20)
        END IF
        POKEL SuperLayer&+12,backup.rast&
        POKEL WINDOW(8),backup.layer&
        POKEL SuperLayer&+40,0

        CALL DeleteLayer(scrLayerInfo&,SuperLayer&)
        CALL FreeRaster(superPlane&,superWeite%,superHoehe%)
        CALL FreeMem(superBitmap&,40)

        SCREEN CLOSE scrNr%
        WINDOW windNr%,"hi!",,,WBenchScrNr%

        IF printflag%<>1 THEN
```

```
        CALL RemPort(port&)
        CALL FreeSignal(sigBit%)
        CALL FreeMem(pio&,100)
      END IF

      IF oldFont&<>0 THEN CALL CloseFont(oldFont&)
      LIBRARY CLOSE
      END

'*** Das war's. Hier folgen wichtige Unterroutinen! ***

MenuCtrl:  '* Menue wurde benutzt. Was nun?
      menuId   = MENU(0)
      menuItem = MENU(1)

      IF menuId=1 THEN
        IF menuItem     =  1 THEN
          CALL SetRast(SuperRast&,0)
        ELSEIF menuItem =  2 THEN
          GOSUB koord
        ELSEIF menuItem =  4 THEN
          drMd%=0
        ELSEIF menuItem =  5 THEN
          drMd%=drMd% OR 1
        ELSEIF menuItem =  6 THEN
          drMd%=drMd% OR 2
        ELSEIF menuItem =  7 THEN
          drMd%=drMd% OR 4
        ELSEIF menuItem =  9 THEN
          style%=0:drMd%=0:outline%=0
        ELSEIF menuItem = 10 THEN
          style%=style% OR 4
        ELSEIF menuItem = 11 THEN
          style%=style% OR 2
        ELSEIF menuItem = 12 THEN
          style%=style% OR 1
        ELSEIF menuItem = 13 THEN
          outline%=1
        ELSEIF menuItem = 15 THEN
          GOSUB swapcol
        ELSEIF menuItem = 16 THEN
          IF kl%=0 THEN
            kl%=1
          ELSE
            kl%=0
          END IF
        ELSEIF menuItem = 17 THEN
          GOTO deleteSys
        END IF
      ELSEIF menuId = 2 THEN
        grid% = 0
        copy% = 0
```

```
    IF menuItem     = 1 THEN
      modus$ = "CIRCLE"
      draw%  = 1
    ELSEIF menuItem = 2 THEN
      modus$ = "RECHTECK"
      draw%  = 3
    ELSEIF menuItem = 3 THEN
      modus$ = "LINIEN"
      draw%  = 2
    ELSEIF menuItem = 4 THEN
      modus$ = "FREIHAND"
      draw%  = 4
    ELSEIF menuItem = 5 THEN
      modus$ = "TEXT"
      draw%  = 5
    ELSEIF menuItem = 6 THEN
      modus$ = "LOESCHEN"
      draw%  = 6
    ELSEIF menuItem = 7 THEN
      modus$ = "FILL"
      draw%  = 7
    ELSEIF menuItem = 8 THEN
      modus$ = "GRID"
      grid%  = 1
      draw%  = 3
    ELSEIF menuItem = 9 THEN
      grid1% = 1
      grid2% = 1
    ELSEIF menuItem = 10 THEN
      modus$ = "GET AREA"
      draw%  = 3
      grid%  = 1
      copy%  = 1
    ELSEIF menuItem = 11 THEN
      modus$ = "PAINT"
      draw%  = 10
    END IF
  ELSEIF menuId = 3 THEN
    IF fontflag% = 0 THEN
      GOSUB loadFonts
    ELSE
      GOSUB loadFont
    END IF
  ELSEIF menuId=4 THEN
    IF menuItem=1 THEN
      IF printflag%<>1 THEN
        GOSUB hardcopy
      ELSE
        BEEP
      END IF
    ELSEIF menuItem=2 THEN
      GOSUB changePrint
```

```
    END IF
  END IF

  IF kl%=1 THEN
    aus$ = modus$+" / Kopfleiste ausgeschaltet."+CHR$(0)
    CALL WaitTOF
    CALL SetWindowTitles(windAdd&,SADD(aus$),-1)
  END IF

  RETURN

koord:  '* Koordinatenkreuz zeichnen
  CALL SetDrMd(WINDOW(8),2)
  POKEW SuperRast&+34,&HAAAA
  FOR loop%=0 TO superWeite% STEP 50
    LINE (loop%,0)-(loop%,superHoehe%)
  NEXT loop%
  FOR loop%=0 TO superHoehe% STEP 50
    LINE (0,loop%)-(superWeite%,loop%)
  NEXT loop%
  POKEW SuperRast&+34,&HFFFF
  CALL SetDrMd(WINDOW(8),drMd%)
  RETURN

drawit:  '* Multi-Funktions-Zeichner mit Rubberband
  cX%=MOUSE(1)
  cY%=MOUSE(2)
  test%=MOUSE(0)
  mx%=1:my%=1
  ccX%=0:ccY%=0
  oldcX%=0:oldcY%=0
  rcX%=0:rcY%=0
  oldrcX%=0:oldrcY%=0
  CALL SetDrMd(SuperRast&,2)
  subox%=ox%
  suboy%=oy%
  loopflag%=0
  IF (cX% MOD grid1%)>(.5*grid1%) THEN cX%=cX%+grid1%
  IF (cY% MOD grid2%)>(.5*grid2%) THEN cY%=cY%+grid2%
  cX% = cX%-(cX% MOD grid1%)
  cY% = cY%-(cY% MOD grid2%)
  GOSUB oldpos

  WHILE test%<0
    test%=MOUSE(0)
    oldx%=mx%
    oldy%=my%
    oldcX%=ccX%
    oldcY%=ccY%
    oldrcX%=rcX%
    oldrcY%=rcY%
    mx%=MOUSE(1)
```

```
        my%=MOUSE(2)
        IF (mx% MOD grid1%)>(.5*grid1%) THEN mx%=mx%+
        grid1%
        IF (my% MOD grid2%)>(.5*grid2%) THEN my%=my%+
        grid2%
        mx%=mx%-(mx% MOD grid1%)
        my%=my%-(my% MOD grid2%)
        IF mx%=oldx% AND my%=oldy% AND s.fl%=0 THEN
          rep.flag%=1
        ELSE
          rep.flag%=0
          s.fl%=0
        END IF
        IF rep.flag%=0 THEN
          GOSUB oldpos
        END IF
        IF mx%<layMinX%+5 THEN GOSUB ScrollRechts
        IF mx%>layMaxX%-15 THEN GOSUB ScrollLinks
        IF my%<layMinY%+5 THEN GOSUB ScrollRunter
        IF my%>layMaxY%-20 THEN GOSUB ScrollHoch
        GOSUB updateDisp

        ccX%=ABS(mx%-cX%)+ABS(ox%-subox%)
        ccY%=ABS(my%-cY%)+ABS(oy%-suboy%)
        rcY%=my%-cY%+(oy%-suboy%)
        rcX%=mx%-cX%+(ox%-subox%)
        IF rep.flag%=0 THEN
          GOSUB newpos
        END IF

      WEND
      GOSUB newpos
      CALL SetDrMd(SuperRast&,1)
      IF draw%=6 THEN
          x1%=cX%+subox%
          y1%=cY%+suboy%
          x2%=cX%+subox%+oldrcX%
          y2%=cY%+suboy%+oldrcY%
         IIF x2%<x1% THEN SWAP x1%,x2%
          IF y2%<y1% THEN SWAP y1%,y2%
          x1%=x1%+1:y1%=y1%+1
          x2%=x2%-1:y2%=y2%-1
          CALL SetAPen(WINDOW(8),0)
          CALL RectFill(WINDOW(8),x1%,y1%,x2%,y2%)
          CALL SetAPen(WINDOW(8),1)
      ELSEIF (draw%=3 AND (fetch%<>0 OR grid%<>0)) THEN
          REM nichts
      ELSE
          GOSUB newpos
      END IF

RETURN
```

```
newpos:          IF draw%=1 THEN
                     CALL DrawEllipse(SuperRast&,cX%+subox%,cY%+suboy%,
                          ccX%,ccY%)
                 ELSEIF draw%=2 THEN
                     LINE (cX%+subox%,cY%+suboy%)-(cX%+subox%+rcX%,cY%+
                          suboy%+rcY%),swapper%
                 ELSEIF draw%=3 OR draw%=6 THEN
                     LINE (cX%+subox%,cY%+suboy%)-(cX%+subox%+rcX%,cY%+
                          suboy%+rcY%),swapper%,b
                 END IF
                 RETURN

oldpos:          IF draw%=1 THEN
                      CALL DrawEllipse(SuperRast&,cX%+subox%,cY%+suboy%,
                           oldcX%,oldcY%)
                  ELSEIF draw%=2 THEN
                      LINE (cX%+subox%,cY%+suboy%)-(cX%+subox%+oldrcX%,
                           cY%+suboy%+oldrcY%),swapper%
                  ELSEIF draw%=3 OR draw%=6 THEN
                      LINE (cX%+subox%,cY%+suboy%)-(cX%+subox%+oldrcX%,
                           cY%+suboy%+oldrcY%),swapper%,b
                  END IF
                  RETURN
filler:          '* Fuellroutine
                 test%=MOUSE(0)
                 oldx%=MOUSE(1)
                 oldy%=MOUSE(2)
                 PAINT (ox%+oldx%,oy%+oldy%),1,1
                 RETURN

freedraw:        '* Freihand-Zeichner
                 test% = MOUSE(0)
                 oldx%   = MOUSE(1)
                 oldy%   = MOUSE(2)
                 WHILE test%<0
                   oldx% = mx%
                   oldy% = my%
                   mx%   = MOUSE(1)
                   my%   = MOUSE(2)
                   IF mx%<layMinX%+10 THEN GOSUB ScrollRechts
                   IF mx%>layMaxX%-20 THEN GOSUB ScrollLinks
                   IF my%<layMinY%+10 THEN GOSUB ScrollRunter
                   IF my%>layMaxY%-25 THEN GOSUB ScrollHoch
                   LINE (ox%+oldx%,oy%+oldy%)-(ox%+mx%,oy%+my%),swapper%
                   GOSUB updateDisp
                   test% = MOUSE(0)
                 WEND
                 RETURN

paintdraw:       '* Mit Image zeichnen
                 test%=MOUSE(0)
```

```
                WHILE test%<0
                  mx%   = MOUSE(1)
                  my%   = MOUSE(2)
                  IF mx%<layMinX%+10 THEN GOSUB ScrollRechts
                  IF mx%>layMaxX%-20 THEN GOSUB ScrollLinks
                  IF my%<layMinY%+10 THEN GOSUB ScrollRunter
                  IF my%>layMaxY%-25 THEN GOSUB ScrollHoch
                  mx%   = mx%-(mx% MOD grid1%)
                  my%   = my%-(my% MOD grid2%)
                  IF mx%<layMinX%+10 THEN GOSUB ScrollRechts
                  IF mx%>layMaxX%-20 THEN GOSUB ScrollLinks
                  IF my%<layMinY%+10 THEN GOSUB ScrollRunter
                  IF my%>layMaxY%-25 THEN GOSUB ScrollHoch

                  test% = MOUSE(0)
                  PUT (mx%+ox%,my%+oy%),get.array%,OR
                WEND
                RETURN

ScrollHome:     x%=-ox%
                y%=-oy%
                ox%=0
                oy%=0
                GOSUB ScrollDisplay
                RETURN

ScrollRechts:   IF ox%>grid1%-1 THEN
                  x%  = -grid1%
                  ox% = ox%-grid1%
                  GOSUB ScrollDisplay
                END IF
                RETURN

ScrollLinks:    IF ox%<(superWeite%-layMaxX%+layMinX%-grid1%) THEN
                  x% = grid1%
                  IF textWidth%<>0 THEN
                    IF ox%+textWidth%<(superWeite%-layMaxX%+layMin%)
                    THEN
                      x% = textWidth%
                    END IF
                    textWidth% = 0
                  END IF
                  ox% = ox%+x%
                  GOSUB ScrollDisplay
                END IF
                RETURN

ScrollHoch:     IF oy%<(superHoehe%-layMaxY%+layMinY%-grid2%) THEN
                  y%  = grid2%
                  oy% = oy%+grid2%
                  GOSUB ScrollDisplay
                END IF
```

```
              RETURN

ScrollRunter: IF oy%>grid2%-1 THEN
                y%  = -grid2%
                oy% = oy%-grid2%
                GOSUB ScrollDisplay
              END IF
              RETURN

ScrollDisplay: '* scroll it
              CALL ScrollLayer(scrLayerInfo&,SuperLayer&,x%,y%)
              x% = 0
              y% = 0
              s.fl% = 1
              RETURN

updateDisp:   IF kl%=0 THEN
                actu$="> "+modus$+" [F]="+STR$(fontHoehe%)+" [X]="+STR$
                      (ox%+mx%)+"  [Y]="+STR$(oy%+my%)+CHR$(0)
                CALL WaitTOF
                CALL SetWindowTitles(windAdd&,SADD(actu$),-1)
              END IF
              RETURN

loadFonts:    '* Disk-Fonts einladen
              sp$     = modus$
              modus$  = "LADE FONTS."
              GOSUB updateDisp
              opt&    = 2^0+2^16
              bufLen& = 3000
              buffer& = AllocMem&(bufLen&,opt&)
              IF buffer&<>0 THEN
                er& = AvailFonts&(buffer&,bufLen&,3)
                IF er& = 0 THEN
                  eintrag% = PEEKW(buffer&)
                  IF eintrag%>19 THEN eintrag% = 19
                  DIM textAttr&(eintrag%*2)
                  DIM textName$(eintrag%)
                  FOR loop%=0 TO eintrag%-1
                    counter%    = loop%*10
                    fontTitle&  = PEEKL(buffer&+4+counter%)
                    fontH%      = PEEKW(buffer&+counter%+8)
                    textAttr&(loop%*2+1)=PEEKL(buffer&+counter%+8)
                    fontTitle$  = ""
                    check%=PEEK(fontTitle&)
                    WHILE check%<>ASC(".")
                      fontTitle$ = fontTitle$+CHR$(check%)
                      fontTitle& = fontTitle&+1
                      check%     = PEEK(fontTitle&)
                    WEND
                    textName$(loop%) = fontTitle$+".font"+CHR$(0)
                    fontName$        = fontTitle$+STR$(fontH%)
```

```
            fontzaehler      = fontzaehler+1
            MENU 3,fontzaehler,1,fontName$
          NEXT loop%
          CALL FreeMem(buffer&,bufLen&)
          fontflag% = 1
        END IF
      ELSE
        BEEP
      END IF
      modus$ = sp$
      RETURN

loadFont: '* Lade Zeichensatz
      sp$    = modus$
      modus$ = "LADE FONT"
      GOSUB updateDisp
      textBase%            = (menuItem-1)*2
      textAttr&(textBase%) = SADD(textName$(menuItem-1))
      newFont& = OpenDiskFont&(VARPTR(textAttr&(text
      Base%)))
      IF newFont& = 0 THEN
        newFont& = OpenFont&(VARPTR(textAttr&(textBase%)))
      END IF
      IF newFont&<>0 THEN
        IF oldFont&<>0 THEN
          CALL CloseFont(oldFont&)
        END IF
        CALL SetFont(SuperRast&,newFont&)
        oldFont&   = newFont&
        fontHoehe% = INT(textAttr&(textBase%+1)/2^16)
      ELSE
        BEEP
      END IF
      modus$ = sp$
      RETURN

drawtext: '* Text in Grafik-Bitmap schreiben
      IF (mx% MOD grid1%)>(.5*grid1%) THEN mx%=mx%+grid1%
      IF (my% MOD grid2%)>(.5*grid2%) THEN my%=my%+grid2%
      my%=my%-(my% MOD grid2%)
      mx%=mx%-(mx% MOD grid1%)
      CALL Move(SuperRast&,mx%+ox%,my%+oy%+fontHoehe%)
      modus$ = "EINGABE"+CHR$(0)
      CALL WaitTOF
      CALL SetWindowTitles(windAdd&,SADD(modus$),-1)

      modus$ = "TEXT"
      in$    = ""
      WHILE in$<>CHR$(13)
        IF in$<>"" THEN
          CALL SetDrMd(SuperRast&,drMd%)
          enable% = AskSoftStyle&(SuperRast&)
```

```
        n&      = SetSoftStyle&(SuperRast&,style%,
        enable%)
        tempX%  = PEEKW(SuperRast&+36)
        tempY%  = PEEKW(SuperRast&+38)
        rand%   = tempX%-ox%
        IF rand%>layMaxX%-20 THEN
          textWidth% = PEEKW(SuperRast&+60)
          GOSUB ScrollLinks
        END IF
        IF outline% = 0 THEN
          CALL Text(SuperRast&,SADD(in$),1)
        ELSE
          CALL SetDrMd(SuperRast&,0)
          FOR loop1%=-1 TO 1
            FOR loop2%=-1 TO 1
              CALL Move(SuperRast&,tempX%+loop2%,tempY%+
              loop1%)
              CALL Text(SuperRast&,SADD(in$),1)
            NEXT loop2%
          NEXT loop1%
          CALL SetDrMd(SuperRast&,2)
          CALL Move(SuperRast&,tempX%,tempY%)
          CALL Text(SuperRast&,SADD(in$),1)
          tempW% = 0
      END IF
    END IF
    in$ = INKEY$
  '* Funktionstastenbelegung
    IF in$ = CHR$(129) THEN in$ = CHR$(196)
    IF in$ = CHR$(130) THEN in$ = CHR$(228)
    IF in$ = CHR$(131) THEN in$ = CHR$(214)
    IF in$ = CHR$(132) THEN in$ = CHR$(246)
    IF in$ = CHR$(133) THEN in$ = CHR$(220)
    IF in$ = CHR$(134) THEN in$ = CHR$(252)
    IF in$ = CHR$(135) THEN in$ = CHR$(223)
    IF in$ = CHR$(136) THEN in$ = CHR$(167)
    IF in$ = CHR$(137) THEN in$ = CHR$(169)
    IF in$ = CHR$(138) THEN in$ = CHR$(174)
    WEND
    m$ = "TEXT"+CHR$(0)
    CALL WaitTOF
    CALL SetWindowTitles(windAdd&,SADD(m$),-1)

    mx% = 0
    my% = 0
    RETURN

newpointer:  '* Zeichenpointer definieren
    opt&=2^1+2^16
    ptr&=AllocMem&(20,opt&)
    IF ptr&<>0 THEN
      POKEW ptr&+4,256
```

```
                POKEW ptr&+8,640
                POKEW ptr&+12,256
                CALL SetPointer(WINDOW(7),ptr&,3,16,-8,-1)
              END IF
              RETURN

hardcopy:     '* Bitmap ausdrucken
              sp$    = modus$
              modus$ = "HARDCOPY"
              GOSUB updateDisp
              dev$ = "printer.device"+CHR$(0)
              er&  = OpenDevice&(SADD(dev$),0,pio&,0)
              IF er&=0 THEN
                er& = DoIO&(pio&)
                IF er&<>0 THEN BEEP:BEEP
                CALL CloseDevice(pio&)
              ELSE
                BEEP
              END IF
              modus$ = sp$
              RETURN

swapcol:      IF swapper%=0 THEN
                swapper%=1
              ELSE
                swapper%=0
              END IF

              RETURN

changePrint: '* Printer-Parameter aendern
             '* Ausgabe auf das eigene Fenster
              backup.font&=PEEKL(WINDOW(8)+52)
              CALL SetFont(WINDOW(8),font&)
              POKEL SuperLayer&+12,backup.rast&
              POKEL WINDOW(8),backup.layer&
              e& = BehindLayer&(scrLayerInfo&,SuperLayer&)

              CALL SetDrMd(WINDOW(8),1)
              LINE (0,0)-(windWeite%-8-offset%-20,windHoehe%-
              15),1,bf
              LINE (20,10)-(windWeite%-8-offset%-40,windHoehe%-
              25),0,bf
              LOCATE 3,1
              PRINT TAB(4);"DRUCK-PARAMETER/SETTINGS"
              PRINT TAB(4);"-----------------------"
              PRINT
              PRINT TAB(4);"Legen Sie den Druck-Ausschnitt"
              PRINT TAB(4);"mittels des Rechteckes fest!
              PRINT
              FOR t=1 TO 10000:NEXT t
              repeat:
```

```
fetch% = 1
draw%  = 3
modus$ = "FETCH"

'* Ausgabe wieder auf das Layer
e&=UpFrontLayer&(scrLayerInfo&,SuperLayer&)
POKEL SuperLayer&+12,WINDOW(8)
POKEL WINDOW(8),SuperLayer&

GOTO mcp

continue:
fetch%=0
modus$="RECHTECK"
'* Ausgabe auf das eigene Fenster
e& = BehindLayer&(scrLayerInfo&,SuperLayer&)
POKEL SuperLayer&+12,backup.rast&
POKEL WINDOW(8),backup.layer&
  LOCATE 9,1
  PRINT TAB(4);USING "Neuer Start X:####";printX0%
  PRINT TAB(4);USING "Neuer Start Y:####";printY0%
  PRINT TAB(4);USING "Neues Ende  X:####";printX1%
  printX1P%=printX1%-printX0%
  PRINT TAB(4);USING "Neues Ende  Y:####";printY1%
  printY1P%=printY1%-printY0%

  LOCATE 15,1
  PRINT TAB(4) SPACE$(26)
  LOCATE 15,1
  PRINT TAB(4);
  INPUT "Sind die Werte OK (j/n) ";jn$
  IF jn$="n" THEN GOTO repeat

PRINT TAB(4);
INPUT "[1] Normal [2] Verzerrt ";nv%
IF nv%=2 THEN
  PRINT TAB(4);
  INPUT "Absolute X-Ausdehnung";printX%
  PRINT TAB(4);
  INPUT "Absolute Y-Ausdehnung";printY%
  printSpec% = 0
ELSE
  printSpec% = 4
END IF

POKEW pio&+44,printX0%
POKEW pio&+46,printY0%
POKEW pio&+48,printX1P%
POKEW pio&+50,printY1P%
POKEL pio&+52,printX%
POKEL pio&+56,printY%
POKEW pio&+60,printSpec%
```

```
'* Ausgabe wieder auf das Layer
 e&=UpFrontLayer&(scrLayerInfo&,SuperLayer&)
 POKEL SuperLayer&+12,WINDOW(8)
 POKEL WINDOW(8),SuperLayer&
 CALL SetFont(WINDOW(8),backup.font&)
 CALL SetDrMd(WINDOW(8),drMd%)

 RETURN
```

8.1 Bedienungsanleitung

Bevor Sie das Programm starten, sollten Sie sich die Variablendefinition am Anfang des Programms anschauen. Sie können in einem Screen niedriger oder hoher Auflösung arbeiten. Da das aber keinen Einfluß auf die Größe der Zeichnung hat, empfehlen wir der Detailgenauigkeit wegen einen Screen geringer Auflösung.

Auch die Größe der Superbitmap - und damit die Größe der Zeichnung - können Sie selbst festlegen. In der jetzigen Fassung verwendet das Programm eine 400x800 Punkte große Zeichenfläche. Wenn Sie den Speicher dazu besitzen, können Sie die Fläche natürlich auf volle 1024x1024 Punkte, den CAD-Standard, ausdehnen.

Noch eine wichtige Bemerkung: Der Kreis-Befehl funktioniert nur zusammen mit der Kickstart-Disk Version 1.2 oder darüber!

Starten des Programms

Starten Sie den Malomat einfach mit "RUN". Sofern Sie über zwei Disk Drives verfügen, sollte sich in einem Laufwerk Ihre Programmdiskette und im zweiten die Workbench-Disk befinden. Besitzen Sie nur ein Disk Drive, dann sollten Sie nach dem Ladevorgang Ihre Programmdiskette aus dem Laufwerk nehmen und durch die Workbench ersetzen. Sollte es dennoch einmal dazu kommen, daß ein Requester erscheint ("Bitte Disk sowieso einlegen..."), dann wird der Workbench-Screen automatisch aktiviert. Sie gelangen zu unserem Zeichenscreen durch gleichzeiti-

ges Drücken der linken "A"-Amiga-Taste und "M". Sie können natürlich auch den Workbench-Screen nach unten ziehen.

Erste Zeichnungen

Sie befinden sich nun im Zeichenprogramm. Den größten Teil des Bildschirmes beansprucht die Zeichenfläche. Die Fenster-Kopfzeile ist ausgeschaltet. Drücken Sie einmal auf die rechte Maustaste! Es erscheint das Menü:

```
SERVICE     ZEICHNEN    FONT    I/O
```

Unter "SERVICE" finden Sie:

```
Screen löschen
Koordinaten ein
--------------
Transparent
JAM 2
Complement
Inverse
--------------
normal/reset
kursiv
fett
unterstrichen
outline
--------------
s/w    w/s
Kopfleiste ein/aus
QUIT
```

Der erste Menüpunkt löscht die gesamte Zeichnung. Der zweite Punkt schaltet ein Koordinatenraster ein. Wählen Sie diesen Punkt erneut, und das Raster verschwindet wieder.

Die folgenden neun Modi bestimmen die Art der Textausgabe, auf die wir gleich kommen werden. s/w w/s vertauscht Hinter- und Vordergrundfarbe. Das kann recht nützlich sein, wenn man nur Teile der Zeichnung löschen will.

Wählen Sie einmal den Punkt "Kopfleiste ein/aus" an! Sofort wird die aktuelle Position der Maus in der Kopfzeile angezeigt,

zusammen mit dem gerade aktiven Zeichenmodus und der Höhe des augenblicklichen Zeichensatzes. Ist die Kopfzeile eingeschaltet, wird viel Rechenzeit dafür verschwendet: Scrolling und Zeichenfunktionen verlangsamen sich. Wenn Sie also ohne die Kopfzeile auskommen können, lassen Sie sie ausgeschaltet.

Standardmäßig befinden Sie sich zunächst im Zeichenmodus "FREIHAND". Sobald Sie die linke Maustaste drücken, zeichnet die Maus, vergleichbar mit einem Stift. Wenn Sie sich der rechten oder unteren Kante nähern, scrollt das Bild weiter; zusätzliche Teile der Superbitmap werden sichtbar.

Am rechten Bildschirmrand finden Sie fünf kleine Symbolfelder. Fahren Sie mit der Maus einmal auf eines, und drücken Sie die linke Maustaste! Der Bildschirm scrollt in die jeweilige Pfeilrichtung, wenn dort noch Platz ist. Das "H"-Symbol steht für "Home". Es verschiebt die Zeichnung blitzartig wieder in den Ausgangszustand zurück.

Kreise, Rechtecke, Linien

Unter dem Menüpunkt "ZEICHNEN" finden Sie verschiedene Zeichenfunktionen. Solange Sie die linke Maustaste gedrückt halten, können Sie Größe und Richtung der jeweiligen Zeichenoperation frei bestimmen. Lassen Sie die Taste los, wird das Objekt endgültig gezeichnet.

Bildschirmausschnitte löschen

Um nur Teile der Zeichnung zu löschen, wählen Sie den Punkt "Löschen". Sie können nun ein Rechteck beliebiger Größe bestimmen, dessen Inhalt mit der Hintergrundfarbe ausgefüllt wird.

Text ausgeben

Dies ist eine der Zeichenfunktionen. Sie können mit ihr Grafiken beschriften. Nachdem der Text-Modus aktiviert ist, können Sie weiterhin mit der Maus über den Bildschirm fahren. Sobald

Sie die linke Maustaste drücken, erwartet der Amiga eine Texteingabe über die Tastatur. Als Abschluß drücken Sie die RETURN-Taste. Die Funktionstasten sind wie folgt belegt:

F1	F2	F3	F4	F5	F6	F7	F8	F9	F10
Ä	ä	Ö	ö	Ü	ü	ß	#	(C)	(R)

Der Eingabe echter deutscher Texte inklusive Umlaute steht also nichts im Wege.

Die Textausgabe kann variiert werden. Unter SERVICE stehen Ihnen zahlreiche Modi zur Verfügung:

Transparent:	Grafiken werden durch Text nicht gelöscht.
JAM2:	Grafiken werden durch Text überschrieben.
Complement:	Wo schwarz ist, wird es weiß und umgekehrt.
Inverse:	Hinter- und Vordergrundfarbe werden vertauscht (funktioniert nur im Text Normalmodus).
Normal:	alle Schriftstile werden zurückgesetzt.
kursiv:	Schrägschrift
fett:	Fettdruck
unterstrichen:	Text wird unterstrichen.
outline:	Textsilhouette

Diese Modi können nach Belieben miteinander gemischt werden, indem Sie nacheinander die betreffenden Punkte anklicken. Diese Modi lassen sich auch während einer Texteingabe verändern.

Der Druck auf die RETURN-Taste beendet die Eingabe.

Wollen Sie mehrere Zeilen Text ausgeben, die linksbündig und in gleichem Abstand voneinander entfernt sind, dann können Sie das GRID einschalten: Wählen Sie in X-Richtung die Breite eines Buchstabens des Zeichensatzes, in Y-Richtung seine Höhe

(mehr Informationen zum GRID siehe unten!). Jede Textzeile wird wie bisher mit RETURN abgeschlossen; für eine linksbündige Zeile fahren Sie einfach mit der Maus unter das erste Zeichen der darüberliegenden Zeile und drücken den linken Mausknopf.

Benutzung verschiedener Zeichensätze

Der Menüpunkt "FONT" erlaubt Ihnen, Disk-Zeichensätze zur Textausgabe zu verwenden. Beim ersten Anklicken finden Sie dort den Punkt "Fonts laden". Bevor Sie ihn anklicken, sollte sich die Workbench-Diskette in einem der Drives befinden. Jetzt werden alle verfügbaren Zeichensätze gesucht (max. 19, mehr kann MENU nicht verarbeiten). Beim nächsten Anklicken enthält dieses Menü dann eine Liste der zur Verfügung stehenden Zeichensätze. Aus dieser Liste können Sie beliebige Zeichensätze auswählen. "Text" gibt Texte im zuletzt geladenen Zeichensatz aus.

Grafik-Ausdruck

Falls Sie über einen grafikfähigen Drucker verfügen, können Sie Ihre Zeichnungen ausdrucken. Der Menüpunkt "I/O" verfügt über zwei Unterpunkte: "Drucken" und "Param.". Wollen Sie die gesamte Zeichenebene ausdrucken, dann genügt es, den Punkt "Drucken" anzuwählen. Sind Sie hingegen nur an einem Ausschnitt interessiert, dann wählen Sie "Param."! Dort werden Sie aufgefordert, den Druckausschnitt zu markieren. Dazu steht Ihnen ein Rechteck-Rubberband zur Verfügung, mit dem Sie den gewünschten Bereich umrahmen können. Anschließend werden die so ermittelten Daten angezeigt und sicherheitshalber können Korrekturen angebracht werden. Stimmte der Ausschnitt jedoch, können Sie wählen zwischen (1) normalem und (2) verzerrtem Druck. Normaldruck druckt die Grafik in den realen Proportionen aus. Andernfalls können Sie die Anzahl der Punkte in X- und Y-Richtung angeben, die die auszudruckende Grafik auf

dem Drucker einzunehmen hat. Wenn Sie in horizontaler Richtung hier allerdings mehr Punkte angeben, als Ihr Drucker verarbeiten kann, kommt es nicht zum Ausdruck.

Für die Dauer des Druckes sind alle Zeichenfunktionen außer Betrieb.

Grid/Raster

Oft werden in Zeichnungen Diagramme und eine symmetrische Aufteilung erforderlich. Mit der Funktion "Raster/Grid" können Sie ein beliebig großes Zeichengrid einstellen: Wählen Sie ein Rechteck beliebiger Größe! Von nun an werden alle Zeichenoperationen nur noch als Vielfaches dieses Rechteckes ausgeführt, sind also immer symmetrisch zueinander. Sie können die Größe des Grids immer wieder verändern. "Grid Reset" setzt das Grid wieder auf 1x1-Punkt, also den normalen Zeichenmodus.

Flächen füllen

Mit der "Fill"-Funktion lassen sich beliebig große Flächen füllen. Die Flächen müssen von einer lückenlosen schwarzen Linie umrandet sein. Fahren Sie mit der Maus in die Mitte der Fläche, und drücken Sie die linke Maustaste! Bei großflächigen Füllaktionen, insbesondere, wenn sich zahlreiche andere Objekte in der Zeichenfläche befinden, kann die Füll-Operation bei einer 1024x1024-großen Zeichenfläche über eine Minute dauern, in der die zu füllende Fläche durchgerechnet wird.

Eigener Pinsel

Sie können einen beliebigen Teil Ihrer Grafik (die Größe ist allerdings abhängig von Ihrem verbliebenen Speicherplatz) als Pinsel wählen. Dazu selektieren Sie bitte unter "ZEICHNEN" das Feld "Get Area". Nun können Sie einen beliebigen rechteckigen Teil der Grafik "einfangen". Mittels "Paint Area" können Sie nun damit zeichnen.

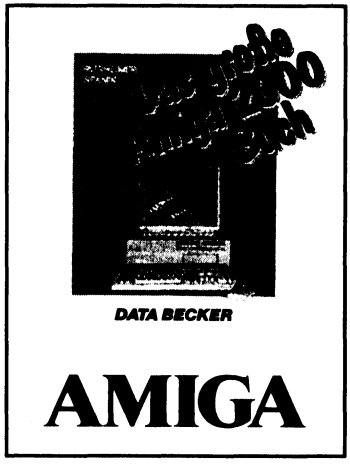
Eigene Muster (Pattern)

Ganz ähnlich funktionieren die eigenen Muster. Jeder Teil Ihrer Grafik kann als Mustervorlage genutzt werden. Gehen Sie so vor: Zeichnen Sie einen kleinen Teil des Musters. Fangen Sie diesen Teil mit "Get Area" ein. Jetzt fangen Sie dieselbe Grafik noch einmal ein, und zwar mit dem "Raster Grid". Gehen Sie nun auf "Paint Area". Ihr Muster kann jetzt als Raster auf den Bildschirm gemalt werden!

Stichwortverzeichnis

Bücher zum Amiga 2000

Der Amiga 2000 wird immer beliebter. Kunststück, hat er doch wirklich das Zeug zum Traumcomputer. Besonders für diejenigen, die einen Amiga wollen, aber auch einen PC brauchen. Das einzige, was den Amiga-2000-Besitzern bisher fehlte, war die passende Literatur. Das hat sich mit diesem Buch geändert.



Aus dem Inhalt:

- Amiga-Grundlagen leichtgemacht
- So stellt man die Akku-Uhr
- Vom richtigen Umgang mit AmigaDOS
- Software und was man damit machen kann
- Software-Installationstips für die Harddisk
- Einbau der PC-Karte
- Speicher erweitern – aber richtig
- Einbau und Einrichten von PC- und Amiga-Harddisk
- Wie man Kickstart wieder ins RAM bringt
- Ton auch für den PC

Rügheimer/Spanik
Das große Buch zum Amiga 2000
Hardcover, 736 Seiten, DM 59,–
ISBN 3-89011-199-8

DAS STEHT DRIN:

Der Amiga ist eine tolle Grafik-Maschine. Bis zu 4096 Farben gleichzeitig, 640 x 512 Bildpunkte Auflösung, Sprites, Bobs und die Geschwindigkeit des Grafikprozessors begeistern einfach jeden Amiga-Anwender. Das neue Supergrafikbuch zum Amiga hilft Ihnen diese Funktionen schnell und sicher in den Griff zu bekommen. Anhand von vielen Beispielprogrammen lernen Sie die Grafikprogrammierung in AmigaBASIC und GFA-BASIC. Denn gerade GFA-BASIC bietet sich durch seine Schnelligkeit besonders für die Intuition-Programmierung an.

Aus dem Inhalt:

- Die Grafik-Befehle des AmigaBASIC (Punkt, Linie, Kreis, Rechteck, Muster, Flächen füllen)
- Laden und Speichern von IFF-Grafiken
- Erstellen von Sprites, Bobs und Animation
- Fenster- und Screen-Programmierung unter Intuition
- Systemprogrammierung mit den Libraries
- Super-Malprogramm durch 1024 x 1024 Bildpunkte
- Die verschiedenen Zeichensätze und Schriftarten von BASIC aus nutzen und anwenden
- Die verschiedenen Grafik-Befehle des GFA-BASIC
- Zugriff auf die ROM-Libraries mit GFA-BASIC
- Multitasking-Hardcopyroutine
- Ausdrucken beliebiger Fenster
- Superschnelles Apfelmännchenprogramm in GFA-BASIC zum Abtippen

UND GESCHRIEBEN HABEN DIESES BUCH:

Jens Trapp und Tobias Weltner sind begeisterte Amiga-Anwender mit langer Programmiererfahrung. Tobias Weltner, der schon im Buch „Amiga Tips & Tricks“ die faszinierenden Möglichkeiten des Rechners beschrieben hat, beweist auch hier, daß schwierige Themen leichtverständlich dargestellt werden können.

ISB N 3-89011-345-1 DM +039.00

DM 39,–
ÖS 304,–
sFr 37,–

DATA BECKER

9 783890 113456 03900