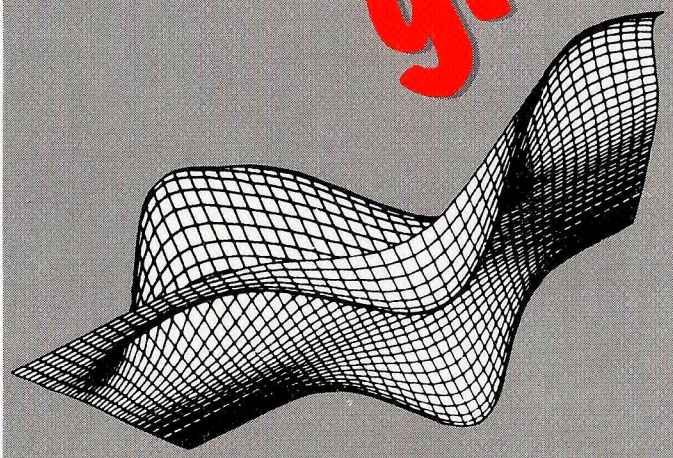
WELTNER
TRAPP
JENNRICH

Super-grafik



Ein **DATA BECKER** Buch

MIT DISKETTE

AMIGA

DAS STEHT DRIN:

Der Amiga ist eine tolle Grafik-Maschine. 4096 Farben gleichzeitig, 640×400 Punkte Auflösung, Sprites, Bobs und die Geschwindigkeit des Blitters begeistern einfach. Das AMIGA SUPERGRAFIK-Buch zeigt Ihnen, wie Sie diese tollen Features in den Griff bekommen. Dabei ist es gleichgültig, ob Sie mit mit BASIC einsteigen, über die Amiga-Libraries die schnellen Routinen von BASIC aus nutzen oder komplette Programme in C schreiben wollen: Dieses Buch zeigt mit vielen Beispielprogrammen, wie man die Grafik des Amiga programmiert.

Aus dem Inhalt:

- Die Grafik-Befehle des AmigaBASIC (Punkt, Linie, Kreis, Rechteck, Muster, Flächen füllen)
- Laden und Speichern von Grafiken (IFF-Format)
- Sprites, Bobs, Animation
- CAD durch 1024×1024-Superbitmap
- Verschiedene Zeichensätze und Schriftarten in BASIC
- Neue Möglichkeiten von BASIC aus durch Zugriff auf die Libraries und Spezial-Chips (4096 Farben gleichzeitig, farbige Muster, Hardcopy von Screens und Windows)
- Grafikprogrammierung von C aus (Punkt, Linie, Rechtecke, Polygone, Farben)
- Die Routinen der Grafik-Bibliothek nutzen
- Das Animationssystem des Amiga (Sprites, Bobs, AnimObs)
- Programmierung von Copper und Blitter (Rasterzeilen-Interrupt, blitzschnelles Kopieren)
- Komplette Beschreibung des Amiga-Grafiksystems (View, Viewport, RastPort, Aufbau der Bitmaps, Screens, Windows)

UND GESCHRIEBEN HABEN DIESES BUCH:

Jens Trapp und Tobias Weltner haben schon mit „64 Tips & Tricks Band II“ ihre Fähigkeiten unter Beweis gestellt. Tobias Weltner hat außerdem in „Amiga Tips & Tricks“ die tollsten Möglichkeiten im Amiga entdeckt. Bruno Jennrich ist begeisterter Grafik-/3-D-Programmierer. Für ihn ist C die richtige Sprache, um auf dem Amiga schnelle Grafik zu erzeugen.

ISB N 3-89011-254-4 DM +059.00

DM 59,–
ÖS 460,–
sFr 57,–

DATA BECKER

9 783890 112541 05900

Weltner
Trapp
Jennrich

Amiga

Supergrafik

DATA BECKER

ISBN 3-89011-254-4
2. überarbeitete und erweiterte Auflage

Wichtiger Hinweis:
Die in diesem Buch wiedergegebenen Schaltungen, 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 Schaltungen, technischen Angaben und Programme in diesem Buch wurden von dem 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 ist der Autor 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 "Supergrafik"-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-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 viele 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!

Abgerundet wird das "Supergrafik"-Buch durch den dritten Teil, in dem Programmierer der Sprache "C" nicht nur mit Fenstern und Screens vertraut gemacht, sondern handfeste Geheimnisse gelüftet werden: Endlich wird ausführlich gezeigt, wie sich das vollautomatische Animationssystem des Amiga benutzen läßt: Bobs, Sprites und die "Virtual Sprite Machine" - Informationen, die sich noch nicht einmal in den ROM-Manuals der Firma Commodore finden!

Ob Sie das "Supergrafik"-Buch zur behutsamen Einführung, als potenten Problemlöser oder als kompetentes Nachschlagewerk nutzen - die vor Ihnen liegenden über 600 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 ja schon viele überzeugt, und das liegt wohl nicht zuletzt an der Grafik.
Beim Amiga wird man - besonders als ehemaliger Home-Computer-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 der 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 Rosetten- und Girlandenmuster

Schon mit dem PSET-Befehl allein lassen sich schöne Grafiken auf den Bildschirm bringen. Diese kann man entweder, wie Sie eben gesehen haben, selber zeichnen, oder man kann den Computer zeichnen lassen. Um letzteres zu verwirklichen, müssen Sie Ihrem Computer nur mitteilen, was er zeichnen soll. Damit es der Computer versteht, müssen wir unsere Vorstellungen in mathematische Formeln verpacken, wie im folgenden Programm. Es erzeugt verschiedene Muster in Rosettenform auf dem Bildschirm.

```
REM Rosetten

pi=3.1415296#
f=.5  'Gibt das Verhaeltnis von Hoehe zu Breite an

INPUT "Anzahl der Schleifen : ",schleifen
CLS

IF schleifen <> -1 THEN schleifen=schleifen+1

REM Voreingestellte Werte
radius=100 'Radius des Ausgangskreises
innen=3    'Anzahl der "inneren" Linien
aussen=3   'Anzahl der "aeusseren" Linien
```

```
FOR t=-innen/10 TO aussen/10 STEP .1
  FOR winkel=0 TO 2*pi STEP .01
    x=radius*COS(winkel)+t*radius*COS(winkel*schleifen)
    y=radius*SIN(winkel)+t*radius*SIN(winkel*schleifen)
    PSET (300+x,y*f+100)
  NEXT winkel
NEXT t
```

Die Form der Rosetten können Sie durch Eingabe verschiedener Werte bestimmen. Die Werte, die Sie als Anzahl der Schleifen eingeben, sollten etwa zwischen 2 und 20 liegen. Aber auch negative Werte erzeugen interessante Figuren.
Die Figuren können Sie außerdem noch variieren, indem Sie mit den voreingestellten Variablenwerten am Programmanfang experimentieren. Das, was Sie am Ende auf dem Monitor sehen, setzt sich aus mehreren sogenannte Epizykloiden zusammen. Das ist keine ansteckende Krankheit, sondern der mathematische Name für Kurven, die durch Rollen eines Kreises um einen zweiten Kreis entstehen. Es werden nicht die Kreise betrachtet, sondern nur Punkte auf oder im äußeren Kreis. Den Weg dieser Punkte können wir auf dem Bildschirm beobachten. Die Formel für Epizykloide haben wir etwas benutzerfreundlich abgeändert. Die allgemeine Formel finden Sie in jeder ausführlicheren mathematischen Formelsammlung wieder.

1.1.3 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 Syntax 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 ein Strich von 40 Pixeln auf den Bildschirm gezeichnet. Es scheint, als würde sich der ganze Strich bewegen. Dabei wird lediglich vorne an den Strich ein neuer Punkt dazu gesetzt und hinten einer gelöscht.

1.1.4 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: Beispielsweise, wenn es um die Benutzung von Farben geht, denn eine Besonderheit dieses Computers sind seine Farben. Damit ist nicht das wunderschöne Grau der Tastatur, sondern die 4096 verschiedenen Farben sind 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 ein neues Screen öffnet, kann man bestimmte Parameter wie Breite, Höhe, Modus und die Tiefe festlegen. Die Tiefe gibt an, wieviel Farben man gebrauchen 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^{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 haben wir den Modus Eins benutzt. Die Breite und Höhe des Screens darf 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 dieser 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
```

In diesen beiden Programmen wird die Farbe des Punktes auf unterschiedliche Weise errechnet. Im ersten Programm werden die X- und Y-Werte durch AND miteinander verknüpft. Sehr schöne Muster entstehen auch, wenn man statt der AND-Verknüpfung die Werte multipliziert oder mit XOR verknüpft.
Im zweiten Programm sieht man ein Quadrat, das außen dunkel ist und zur Mitte hin immer heller wird. Es wirkt fast wie eine Pyramide von oben. Zu diesem Zweck haben wir nur die Farben aus den Registern 21 bis 31 genommen. In diesen Registern sind verschiedene Grauabstufungen, mit denen wir diesen 3-D-Effekt erzielen können.

1.1.5 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 Defaultfarbe (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.

```
REM Demo fuer P(RE)SET

RANDOMIZE TIMER
SCREEN 1,320,200,5,1
WINDOW 2,,(0,0)-(311,185),16,1
l=.8
DIM y(l*100)

WHILE INKEY$=""
COLOR INT(RND*32),INT(RND*32)
CLS

FOR j=1 TO 4
  y(j)=50+RND*80:s(j)=RND*50
NEXT j

FOR x=0 TO 6.2+l STEP .04
  FOR j=1 TO 4 STEP 2
    PRESET ((x-l)*50,y(j)-s(j)*SIN(x-l))
    PRESET ((x-l)*50,y(j+1)-s(j+1)*COS((x-l)*j))
    PSET (x*50,y(j)-s(j)*SIN(x)),INT(RND*32)
    PSET (x*50,y(j+1)-s(j+1)*COS(x*j))
  NEXT j
NEXT x

WEND
```

Bei diesem Programm laufen mehrere geschlängelte Linien über den Bildschirm. Dabei wird, wie beim ersten PRESET-Demoprogramm, vorne ein Punkt an die "Schlange" gesetzt und hinten einer gelöscht. Das Besondere an diesem Programm ist, daß die Vorder- und Hintergrundfarben verändert werden. Wie man sieht, funktioniert der Löschbefehl PRESET ohne Probleme.

1.1.6 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.

1.1.7 Relative Adressierung

Bei PSET und fast allen noch folgenden Grafik-Befehlen des Amiga-BASIC 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 "absolut" genannt.
Die zweite Adressierungsart nennt sich "relativ", 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 anderern 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 unterschiedliche 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, um eine Linie zu bestimmen, immer noch zwei Punkte.

```
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 Auf den Punkt gebracht

Mit PSET kann man fast alles machen, was es in der Welt der Computergrafik gibt. Fast. Denn mit PSET kann man sicher jede beliebige Grafik erstellen oder eine Linie zeichnen lassen. Allerdings wird es dafür häufig sehr lange brauchen, und gerade darauf kommt es häufig an. Der Zeitgewinn mit dem LINE-Befehl ist enorm, wie man sehr einfach feststellen kann:

```
REM Vergleichstest von PSET und LINE

REM Gerade mit PSET
FOR i= 0 TO 180
  PSET(i,i)
NEXT i

REM Gerade mit LINE
LINE (180,0)-(0,180)
```

Der LINE-Befehl eröffnet dem Benutzer sehr viel Möglichkeiten, vielleicht sogar mehr als der PSET-Befehl. Genauso, wie man die Linie als zusammengesetzte Punktreihe betrachten kann, kann man den Punkt zum Sonderfall der Linie degradieren. In diesem Fall wären Anfangs- und Endkoordinaten des LINE-Befehls gleich.

1.2.2 Der Moire-Effekt

Immer, wenn mehrere Linien dicht nebeneinander oder übereinander gezeichnet werden, ist der Moire-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 Moire-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 Moire-Effekt entsteht in der Gegend der Verbindungslinie beider Ecken, denn dort werden viele Linien gezeichnet.
Der Moire-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 Moire-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, die dieses Programm erstellt, anschaut, ist es nicht leicht zu erkennen, daß diese Grafik auf so einfache Weise entstanden ist.

1.2.3 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.

1.2.4 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 herumzuexperimentieren 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 'kooerdinatenkreuz 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.

1.2.4.1 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 einge-

ben 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.

1.2.4.2 Undefinierte Funktionswerte

Dieses Programm gibt Ihnen die Möglichkeit, mathematische Funktionen auf dem Bildschirm zu betrachten. Dabei brauchen Sie sich nicht um die Definitionsmenge einer Menge zu kümmern. Will man zum Beispiel die Kurve 1/x berechnen, tritt an der Stelle x=0 ein Fehler auf, weil 1/0 keinen definierten Wert liefert. Im Programm wird mit der ON-ERROR-GOTO-Anweisung vor der Berechnung des Funktionswerts die Fehlerunterbrechungsreaktionsfähigkeit, wie es das BASIC-Handbuch nennt, aktiviert. Das bedeutet im Klartext, daß der Computer beim Auftreten eines Fehlers das Programm nicht abbricht, sondern zu einer Sprungmarke verzweigt und dort auf den Fehler reagiert.
In diesem Programm erhält der Funktionswert beim Auftreten bestimmter Fehler den Wert 9999. Stößt die Zeichenroutine auf diesen Wert, weiß sie, daß dieser Punkt nicht gezeichnet werden soll.
Neben 'Division by zero' wird auch der Fehler 'Overflow' unterdrückt. Neben diesen provozierten Fehlern besteht auch die Gefahr, daß unvorhergesehene Fehler auftreten. Bei ungewollten Fehlern wird das Programm abgebrochen.
Alle Fehler, die nach der Berechnung der Funktionswerte auftreten, sind unbeabsichtigt. Damit das Programm auf diese Fehler nicht falsch reagiert, muß die Reaktionsfähigkeit auf Fehler wieder ausgeschaltet werden.

1.2.4.3 Scaling

Der Benutzer kann durch Vergrößern und 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 ver-

wirrende Eindrücke über deren Verlauf entstehen. So wirkt z.B. eine fast gerade verlaufende Kurve viel steiler.
Diese Darstellunsweise 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.

1.2.5 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 der beiden 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, wieviel 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.

1.2.6 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 ist 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 vom 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 Amiga-BASIC. 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 Sachen auf:

1. Die Figuren sind 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 sehen Sie 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, um die wir uns erst später kümmern 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; 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 ensteht der Moire-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, 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 wird, 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 auch 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. Beispielsweise wird in Tortengrafiken angegeben, wer nach einer Wahl wieviel Sitze erhält. Die Torte symbolisiert alle möglichen Sitze. Die Sitze einer Partei sind dann "Kuchenstücke" in den Farben der jeweiligen Partei.
Wahlen sind nur eines von vielen Anwendungsgebieten der Tortengrafik.
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 selber 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. Flächen zu färben ist in der Regel recht einfach. Man braucht nur eine begrenzte Fläche und einen Punkt in dieser Fläche, dann können wir 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, ist in bestimmten Fällen recht nützlich, denn es reicht 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 Home-Computern 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 einfach 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. Aber schließ-

lich 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, 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 Programmes 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.

```
REM Drudenfuss

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 zweimal 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, 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, wieviel 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,...). 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 2^n 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 Hexadizimal- 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, was erst nur als Zeichenkette aus Leerstellen und Sternen besteht, wird umgeformt, und dann wird 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 errechneten 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 A's 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 erniedrigt 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.

1.4.3.5 Gesammelte Werke

Mit flächenfüllenden Mustern kann man natürlich Super-Programme schreiben. Hier sind noch einige Beispielprogramme für flächenfüllende Muster.

```
REM Stars and Stripes

DEFINT a-z
OPTION BASE 1
DIM a(16)
SCREEN 1,320,200,5,1
WINDOW 2,,(0,0)-(311,185),16,1

FOR i= 1 TO 16
  READ a(i)
NEXT i
PAINT (0,0),2
LINE (16,16)-(260,146),9,bf
FOR i= 0 TO 5
  LINE(16,26+i*20)-(260,36+i*20),1,bf
NEXT i
PATTERN ,a
LINE (16,16)-(111,80),1,bf

DATA 0,1536,3840,-16,16320,8064
DATA 6528,0,0,96,240,4095,1020
DATA 504,408,0

WHILE INKEY$="": WEND
```

```
WINDOW CLOSE 2
SCREEN CLOSE 1
```

Sicher kennen Sie alle das amerikanische Sternenbanner. Wenn nicht, werden Sie es sofort kennnenlernen, denn es erscheint nach dem Programmstart auf Ihrem Bildschirm. Die berühmten Sterne der Flagge wurden als Muster definiert. Und zwar wurden zwei Sterne untereinander und etwas seitlich verschoben in einem Muster definiert. Auf diese Weise liegen nicht alle Sterne in einer Reihe untereinander.

Eine weitere Möglichkeit, Muster einzusetzen, besteht, wie wir oben schon angedeutet haben, bei 3-D-Effekten. Auch hierzu haben wir ein kleines Demonstrationsprogramm anzubieten:

```
REM 3-D

DEFINT a-z
REM 3-D Wuerfel

OPTION BASE 1
DIM c(4)
SCREEN 1,320,200,3,1
WINDOW 2,,(0,0)-(311,185),16,1
COLOR 0,1
CLS
REM Muster
c(1)=&H1010
c(2)=&H4040
c(3)=&H101
c(4)=&H404
PATTERN ,c
h=94
x=68
y=x/2

REM Grosser Wuerfel
AREA (60,44)
AREA STEP (x,-y)
AREA STEP (x,y)
AREA STEP (0,h)
AREA STEP (-x,y)
AREA STEP (0,-h)
AREAFILL

SWAP c(1),c(4)
SWAP c(2),c(3)
PATTERN ,c
AREA STEP (0,0)
```

```
AREA STEP (-x,-y)
AREA STEP (0,h)
AREA STEP (x,y)
AREAFILL

REM Kleiner Wuerfel
COLOR 4
AREA STEP (0,-h/2)
AREA STEP (x/2,-y/2)
AREA STEP (0,-h/2)
AREA STEP (-x/2,-y/2)
AREA STEP (-x/2,y/2)
AREA STEP (0,h/2)
AREAFILL

SWAP c(1),c(4)
SWAP c(2),c(3)
PATTERN ,c
AREA STEP (0,0)
AREA STEP (x/2,-y/2)
AREA STEP (0,-h/2)
AREA STEP (-x/2,y/2)
AREAFILL

WHILE INKEY$="": WEND

WINDOW CLOSE 2
SCREEN CLOSE 1
```

Man sieht direkt auf die Kante eines 3-D-Würfels. In der vorderen oberen Ecke scheint ein kleiner Würfel zu fehlen. Oder steht da etwa ein zweiter Würfel hervor?
Die Muster, mit denen die verschiedenen Querstriche erzeugt werden, sind fast gleich. Das einzige, was unterschiedlich ist, ist die Reihenfolge der Daten des Musters. Alles, was man machen muß, um von einem Muster ins andere überzugehen, ist die Reihenfolge der Daten umzudrehen. Im Programm erledigen das jeweils zwei SWAP-Anweisungen.

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 einem 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. Dann kann man die Farben, die in den Farbregistern angegeben sind, ändern.
Jede Farbe besteht aus drei Werten, die den Rot-, Grün- und Blauanteil angeben. Jeder Anteil kann einen von 16 Werten annehmen. Das macht dann 4096 Farben.
Der BASIC-Befehl, mit dem man die Farben ändert, heißt PALETTE. Außer den RGB (Rot, Grün und Blau) -Werten 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.

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 gezielter suchen und sicher auch finden. Man kann, während das Programm läuft, alle drei Farbanteile ändern. Auf diese Weise kommt man schnell an alle gewünschten Kombinationen heran. Die Änderungen kann man gleichzeitig auf dem Bildschirm beobachten.

```
REM ColorConstructor

SCREEN 1,320,200,2,1
WINDOW 2,"Farbgestalter",(0,0)-(297,185),31,1
LINE (200,20)-(300,150),3,bf
LOCATE 5
PRINT "> '4'   '5'   '6'"
LOCATE 16
PRINT "< '1'   '2'   '3'"LOCATE 10
PRINT "   R     G     B"
r=1:g=.5:b=0 'orange

WHILE 1
 LOCATE 11
 PRINT USING "  #.##";r;g;b
 PALETTE 3,r,g,b
 WHILE a$=""
  a$=INKEY$
 WEND
  IF a$<>"" THEN
   IF a$="1" AND r>=.0666 THEN r=r-.06666
   IF a$="4" AND r<=.9333 THEN r=r+.06666
   IF a$="2" AND g>=.0666 THEN g=g-.06666
   IF a$="5" AND g<=.9333 THEN g=g+.06666
   IF a$="3" AND b>=.0666 THEN b=b-.06666
   IF a$="6" AND b<=.9333 THEN b=b+.06666
   a$=""
  END IF
WEND
```

Die Farbanteile kann man mit den Tasten "1"-"6" verändern. Diese Tasten sind für unseren Zweck sehr gut auf dem Zehnerblock angeordnet: Die 4 erhöht den Rotanteil, 1 erniedrigt ihn. 5 und 2 sind die Tasten des Grünanteils und 6 und 3 die für Blau.
Die Farbanteile erhöhen und verringern sich immer um 0.0666. Das entspricht genau 1/15. Auf diese Weise erreicht man bei jedem Tastendruck eine Farbänderung.

Übrigens kann man mit PALETTE auch die drei Farben der Maus verändern. Die entsprechenden Register sind 17, 18 und 19. Probieren Sie es aus.

1.5.3 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 sehr wichtig. Deshalb greifen wir an dieser Stelle etwas voraus und greifen direkt in den Speicher, um uns direkt 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. 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.

1.5.4 Animation mit Farbe

Nun wollen wir Ihnen auch gleich einen Nutzen unserer neu-gewonnenen Funktionen vorführen. Man kann sehr einfach bewegte Bilder erzeugen, indem man einfach mehrere Farben ständig vertauscht. Wenn Sie schon einmal mit GraphiCraft oder anderen Malprogrammen gearbeitet haben, wird Ihnen diese Methode sicherlich vertraut sein. Man kann auf diese Weise mit wenigen Handgriffen einen reißenden Bach oder ähnliches auf den Bildschirm zaubern. Mit ein wenig Programmieraufwand ist diese Farbvertauschung auch in BASIC kein Problem. Allerdings sollte man die Anzahl der Farben, die am Tauschprozeß beteiligt sind, klein halten, denn durch das langsame BASIC dauert der Austausch verhältnismäßig lange. Dadurch sehen Bewegungen, die man mit dem Programm simulieren kann, ziemlich abgehackt aus.

```
REM Palette-Umkehrung

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

PALETTE 0,0,.5,0
PALETTE 28,0,.35,.72
PALETTE 29,0,.35,1
PALETTE 30,0,.5,1
PALETTE 31,0,.6 ,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
f1=28:f2=31
FOR j=0 TO 311 STEP 4
FOR i=0 TO 3  STEP .5
 LINE (j+i,120+j/8)-(j+i-4,140+j/8),28+i
 LINE (j+i+1,120+j/8)-(j+i-3,140+j/8),28+i
NEXT i
NEXT j

rotation:
r1=FNrot(f2)-.03
g1=FNgruen(f2)-.03
b1=FNblau(f2)-.03
FOR i=f1 TO f2
  r=FNrot(i)-.03
  g=FNgruen(i)-.03
  b=FNblau(i)-.03
  PALETTE i,r1,g1,b1
```

```
    r1=r
    g1=g
    b1=b
  NEXT i
  GOTO rotation
```

Dieses Programm zeichnet einen abstrakten Fluß. Für den Fluß haben wir die letzten acht Farbregister blau gefärbt.
Wie Sie im Programmteil Rotation sehen können, müssen wir, wenn wir die abgefragten Farbanteile an PALETTE weitergeben, vorher .03 von den Werten abziehen. Damit korrigieren wir einen Fehler, der beim Umrechnen der Gleitkommazahlen in einen Wert zwischen 0 und 15 (denn so werden die Werte im Speicher abgelegt) entsteht.

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. kann man 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 = Hoehe
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 lassen 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, wieviel 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. Dagegen kann man, wenn man genug Speicher zur Verfügung hat, nie zuviel 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 auch ganz gut mit dem Programm malen. Anfangs hat man nur den kleinen Punkt als Pinsel. Dieser Pinsel hat 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";
1     IF MOUSE(0)=0 THEN 1
      ax=MOUSE(1):ay=MOUSE(2)
```

```
2     IF MOUSE(0)<>0 THEN 2
3     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 angegeben 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 übereinanderfallen, wird der Punkt gelöscht. Es wird nur ein Punkt gesetzt, wenn ein ungesetzter und ein gesetzter Punkt übereinanderfallen.
Sicher ist dies nicht für jeden Anwendungszweck das Wahre. 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 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.

```
' 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.
Außer mit Daten im Speicher herumzurangieren, kann er auch noch etwas anderes. Z.B. kann er mit den Daten, die er verschiebt, gleichzeitg 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 mit dem 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, gibt es zu PUT-PSET auch ein Gegenstück mit dem Namen PRESET. Mit dem 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 Spicherplatz 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 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 gespeicherten 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 Verknuepfung

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 alles 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 Home-Computern. 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 beziehungsweise 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 selber 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 zum Erstellen von Objekten, reicht für viele Zwecke aus, ist aber u.a. 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 BASIC strikes back: Eddi II

Als erstes haben wir ein sehr komfortables Programm, 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 II, kann man Objekte mit einer Breite von bis zu 309 und einer Höhe von bis zu 183 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, die man mit dem PUT-Befehl wieder auf den Bildschirm bringen kann, abspeichern.
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 II
REM ****************
REM
REM Jens Trapp  4/87
REM

CLEAR ,42000&    'Arbeitsspeicher vergroessern
OPTION BASE 1
DEFINT a-r,t-z
DEF FNe(b,h,t)=(3+t*h*INT((b+15)/16))

REM Sprite Farben
DIM sr(3),sg(3),sb(3)
sr(1)=1:sg(1)=1  :sb(1)=1   'weiss
sr(2)=0:sg(2)=0  :sb(2)=0   'schwarz
sr(3)=1:sg(3)=.53:sb(3)=0   'orange


REM Menues aendern

MENU 1,0,1,"Windows"
MENU 1,1,1,"Laden    "
MENU 1,2,1,"Speichern"
MENU 1,3,1,"Groesse  "
MENU 1,4,1,"Probe    "
MENU 1,5,1,"Loeschen "
MENU 1,6,1,"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 "
MENU 4,9,0,"Spr. Farbe "
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 standardobjekte
planepick=2^tiefe-1
saveback=1
overlay=1

REM Bildschirm aufbauen
SCREEN 1,320,200,stiefe,1
WINDOW 2,,,16,1
GOSUB schirm


Abfrage:
    a$=INKEY$
    WHILE a$=""
      IF MOUSE(0)<>0 THEN GOSUB zeichnen
      a$=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<184 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
  IF ASC(a$)=139 THEN
    ERASE feld
    DIM feld(FNe(breite,hoehe,stiefe))
    GET (1,1)-(breite,hoehe),feld
    LOCATE 24,1
    PRINT "Bestimmen Sie die Groesse mit der Maus";
    WHILE MOUSE(0)=0 :WEND
    breite=MOUSE(1):IF breite>WINDOW(2) THEN breite=WINDOW(2)
    hoehe=MOUSE(2):IF hoehe>184 THEN hoehe=184
    GOSUB schirm
  END IF
END IF
IF ASC(a$)>47 AND ASC(a$)<58 THEN dicke=ASC(a$)-48: MENU 2,1,1-(tool=1)," Punk
te"+STR$(ASC(a$)-48)
IF a$<>"q" THEN Abfrage


ende:
     WINDOW CLOSE 2
     SCREEN CLOSE 1
     MENU RESET
END


schirm:
     COLOR 1,0:CLS
     feld(3)=tiefe
     PUT (1,1),feld,PSET
sh2: LINE (0,0)-(breite+1,hoehe+1),1,b          'Rahmen der Grafik
     LINE (breite+2,0)-(WINDOW(2),WINDOW(3)),8,bf
     LINE (0,hoehe+2)-(breite+2,WINDOW(3)),8,bf
GOTO tf                                         'Farbpalette


verzweigen:
    titel=MENU(0)
    punkt=MENU(1)
```

```
ON titel GOTO fenster,tools,tiefe,flags
RETURN

tiefe:
    MENU 3,tiefe,1
    planepick=punkt^2
    IF punkt<tiefe THEN
     ERASE feld
     DIM feld(FNe(breite,hoehe,stiefe))
     GET (1,1)-(breite,hoehe),feld
     tiefe=punkt
     GOTO schirm
    END IF
    tiefe=punkt
tf: MENU 3,tiefe,2
    LINE (0,186)-(WINDOW(2),WINDOW(3)),8,bf
    FOR i=0 TO 2^tiefe-1
      LINE(30+i*8,186)-(37+i*8,195),i,bf
    NEXT i
    IF farbe>2^tiefe-1 THEN farbe=1
    IF farbealt>2^tiefe-1 THEN farbealt=1
GOTO farben2

fenster:
ON punkt GOTO laden,speichern,groesse,probe,loeschen,ende
RETURN

tools:
    MENU 2,tool,1
    tool=punkt
    MENU 2,tool,2
    IF tool=6 THEN
      farbealt=farbe
      LINE (0,0)-(breite+1,hoehe+1),farbealt,b
      GOTO farben2
    END IF
RETURN

flags:
ON punkt GOTO fvsprite,collmask,shadmask,saveback,overlay,savebob,planepick,pla
neonoff,sprfarbe
RETURN

fvsprite:
    fvsprite=(fvsprite+1)MOD 2
fvsprite2:
    IF fvsprite=1 THEN
      MENU 4,1,2
      MENU 4,9,1
      FOR i= 1 TO 3
```

```
      PALETTE i,sr(i),sg(i),sb(i)
    NEXT i
    FOR i=2 TO 8
      MENU 4,i,0
    NEXT i
    tiefe=2
    IF breite<16 THEN
      f=0
    ELSE
      f=8
    END IF
    LINE (18,0)-(breite+1,hoehe+1),f,bf
    breite=16
    LINE (0,0)-(breite+1,hoehe+1),farbealt,b
    FOR i=1 TO 5
      MENU 3,i,0
    NEXT i
    MENU 3,2,2
    collmask=0
    shadmask=0
    saveback=0
    overlay=0
    savebob=0
    planeonoff=0
  ELSE
    MENU 4,9,0
    MENU 4,1,1
    PALETTE 1,1,1,1
    PALETTE 2,0,0,0
    PALETTE 3,1,.53,0
    FOR i=2 TO 8
      MENU 4,i,1
    NEXT i
    tiefe=5
    MENU 3,0,1
    FOR i=1 TO 4
      MENU 3,i,1
    NEXT
    MENU 3,5,2
  END IF
  planepick=2^tiefe-1
GOTO tf

collmask:
  collmask=(collmask+1)MOD 2
coll2:
  IF collmask=1 THEN
    b=0
    CALL dateiname("CollisionMask",coll$,b)
    IF coll$="" THEN collmask=0
```

```
    END IF
    MENU 4,2,1+collmask
RETURN

shadmask:
    shadmask=(shadmask +1)MOD 2
shad2:
    IF shadmask=1 THEN
      b=0
      CALL dateiname("ShadowMask",shad$,b)
      IF shad$="" THEN shadmask=0
    END IF
    MENU 4,3,1+shadmask
RETURN

saveback:
    saveback=(saveback+1)MOD 2
    MENU 4,4,1+saveback
RETURN

overlay:
    overlay=(overlay+1)MOD 2
    MENU 4,5,1+overlay
RETURN

savebob:
    savebob=(savebob+1)MOD 2
    MENU 4,6,1+savebob
RETURN

planepick:
    CALL planes("Planepick",planepick)
RETURN

planeonoff:
    CALL planes("PlaneOnOff",planeonoff)
RETURN

SUB planes (b$,p)  STATIC
    WINDOW 3,b$,(0,0)-(150,24),16,1
    PRINT "Beenden mit Return"
pl: FOR i=0 TO 4
      LOCATE 2,2+i*2
      PRINT i;
      LOCATE 3,2+i*2
      PRINT (2^i AND p)/2^i ;
    NEXT i
    a$=""
    WHILE a$=""
      a$=INKEY$
```

```
    WEND
    IF ASC(a$)>=48 AND ASC(a$)<53 THEN p=p XOR 2^(ASC(a$)-48)
    IF ASC(a$)<>13 THEN pl
    WINDOW CLOSE 3
END SUB

SUB dateiname (b$,c$,d) STATIC
    WINDOW 3,b$,(0,0)-(300,40),16,1
    IF c$<>"" THEN
      PRINT "Alter "b$"-Name :"
      PRINT c$
      PRINT "Return fuer gleichen Namen"
    END IF
    INPUT "Neuer Name :",d$
    IF d$<>"" THEN c$=d$
    IF d=1 THEN
      PRINT "Bob- oder Put-Format"
      INPUT "(b/p)";d$
      IF d$="b" THEN
        d=1
      ELSE
        IF d$="p" THEN
          d=0
        ELSE
          d=2
        END IF
      END IF
    END IF
    WINDOW CLOSE 3
END SUB

sprfarbe:
WINDOW 3,"Sprite Farbe",(0,0)-(200,40),16,1
INPUT "Farbe 1,2 oder 3 ";a
IF a>0 AND a<4 THEN
  INPUT "Rotanteil  : ";sr(a)
  INPUT "Gruenanteil: ";sg(a)
  INPUT "Blauanteil : ";sb(a)
  PALETTE a,sr(a),sg(a),sb(a)
END IF
WINDOW CLOSE 3
RETURN

zeichnen:
    x=MOUSE(1)
    y=MOUSE(2)
    xalt=MOUSE(3)
    yalt=MOUSE(4)
    IF y>185 THEN farben
    IF x>breite OR x<1 OR y>hoehe OR y<1 THEN RETURN
```

```
    ON tool GOTO punkte,w,w,w,w,fuellen
w:  IF MOUSE(0)<>0 THEN
      f1=POINT(xalt,yalt)
      PSET (xalt,yalt),-(f1=0)
      f2=POINT(x,y)
      PSET (x,y),-(f2=0)
      PSET (x,y),f2
      PSET (xalt,yalt),f1
    ELSE
      ON tool GOTO ,linien,Rahmen,kaesten,kreise
    END IF
GOTO zeichnen

loeschen:
    LINE (1,1)-(breite,hoehe),0,bf
RETURN

punkte:
    x1=x+dicke:IF x1>breite THEN x1=breite
    y1=y+dicke:IF y1>hoehe THEN y1=hoehe
    LINE (x,y)-(x1,y1),farbe,bf
RETURN

linien:
    LINE(x,y)-(xalt,yalt),farbe
RETURN

Rahmen:
    LINE(x,y)-(xalt,yalt),farbe,b
RETURN

kaesten:
    LINE (x,y)-(xalt,yalt),farbe,bf
RETURN

kreise:
    IF y<>yalt AND x<>xalt THEN
      r=ABS(x-xalt):v=ABS(y-yalt)
      IF v<r THEN
        CIRCLE(xalt,yalt),r,farbe,,,v/r
      ELSE
        CIRCLE(xalt,yalt),v,farbe,,,v/r
      END IF
    END IF
GOTO sh2

fuellen:
    PAINT(x,y),farbe,farbealt
RETURN
```

```
farben:
    IF POINT(x,y)>=0 THEN farbe=POINT(x,y)
farben2:
    LINE (0,186)-(25,195),farbe ,bf
    LINE (0,186)-(25,195),farbealt,b
RETURN

groesse:
    WINDOW 3,"Groesse",(0,0)-(200,30),16,1
    PRINT "Breite = ";breite
    PRINT "Hoehe  = ";hoehe
    PRINT "Taste druecken";
    WHILE INKEY$="":WEND
    WINDOW CLOSE 3
RETURN

probe:
    ERASE feld
    DIM feld(FNe(breite,hoehe,stiefe))
    GET (1,1)-(breite,hoehe),feld
    CLS
    LOCATE 10,5
    PRINT "Ich arbeite"
    GOSUB BFormat
    OBJECT.SHAPE 1,a$
    ON COLLISION GOSUB nochmal
    COLLISION ON
    GOSUB nochmal
    OBJECT.ON
    FOR i=0 TO 100
      COLOR INT(RND*32)
      LOCATE INT(RND*22)+1,RND*50 +1
      PRINT "ralf"
    NEXT i
    WHILE INKEY$="":WEND
    OBJECT.OFF
    OBJECT.STOP
    COLLISION OFF
GOTO schirm

nochmal:
    OBJECT.X 1,10
    OBJECT.Y 1,10
    OBJECT.VX 1,20
    OBJECT.VY 1,15
    OBJECT.START 1
RETURN

laden:  b=1
        MENU 3,tiefe,1
```

```
  CALL dateiname("Load",n$,b)
  IF n$="" OR b=2 THEN RETURN
  OPEN n$ FOR INPUT AS #1
  IF b=0 THEN
    INPUT #1,breite,hoehe,tiefe
    ERASE feld
    DIM feld(FNe(breite,hoehe,stiefe))
    FOR i=4 TO FNe(breite,hoehe,tiefe)
      INPUT #1,feld(i)
    NEXT i
    feld(1)=breite
    feld(2)=hoehe
    feld(3)=tiefe
    planepick=2^tiefe-1
  ELSE
    ColorSet=CVL(INPUT$(4,1)) 'vom Programm
    DataSet=CVL(INPUT$(4,1))  'nicht benutzt
    tiefe=CVL(INPUT$(4,1))
    breite=CVL(INPUT$(4,1))
    hoehe=CVL(INPUT$(4,1))
    flags=CVI(INPUT$(2,1))
    planepick=CVI(INPUT$(2,1))
    planeonoff=CVI(INPUT$(2,1))
    ERASE feld
    DIM feld(FNe(breite,hoehe,tiefe))
    feld(1)=breite
    feld(2)=hoehe
    feld(3)=tiefe
    FOR i=4 TO FNe(breite,hoehe,tiefe)
      feld(i)=CVI(INPUT$(2,1))
    NEXT i
  END IF
IF flags AND 1 THEN
  fvsprite=1
  FOR i=1 TO 3
    a=CVI(INPUT$(2,1))
    sr(i)=(a AND 3840)/3840
    sg(i)=(a AND 240)/240
    sb(i)=(a AND 15)/15
  NEXT i
  GOSUB fvsprite2
ELSE
  collmask=(flags AND 2)/2
  shadmask=(flags AND 4)/4
  saveback=(flags AND 8)/8
  overlay=(flags AND 16)/16
  savebob=(flags AND 32)/32
  MENU 4,1,1
  MENU 4,2,1+collmask
  MENU 4,3,1+shadmask
```

```
        MENU 4,4,1+saveback
        MENU 4,5,1+overlay
        MENU 4,6,1+savebob
        IF shadmask=1 THEN
        b=0:shad$=""
        CALL dateiname("Schadowmask",shad$,b)
        IF shad$<>"" THEN
          OPEN shad$ FOR OUTPUT AS 2
            PRINT #2,breite;hoehe;1;
        END IF
        FOR i=4 TO FNe(breite,hoehe,1)
          a=CVI(INPUT$(2,1))
           IF shad$<>"" THEN PRINT #2,a;
         NEXT i
         IF shad$<>"" THEN CLOSE 2
        END IF
        IF collmask=1 THEN
            b=0:coll$=""
            CALL dateiname("Collisionmask",shad$,b)
            IF coll$<>"" THEN
              OPEN coll$ FOR OUTPUT AS 2
              PRINT #2,breite;hoehe;1;
              FOR i=4 TO FNe(breite,hoehe,1)
               PRINT #2,CVI(INPUT$(2,1));
            NEXT i
            CLOSE 2
          END IF
        END IF
      END IF
      CLOSE 1
GOTO schirm

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 #1,breite,hoehe,tiefe
        FOR i=4 TO FNe(breite,hoehe,tiefe)
          PRINT #1,feld(i)
        NEXT i
      END IF
      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
      a$=a$+MKI$(INT(sr(1)*15)*256+INT(sg(1)*15)*16+sb(1)*15) 'Sprfarbe 1
```

```
    a$=a$+MKI$(INT(sr(2)*15)*256+INT(sg(2)*15)*16+sb(2)*15)  'Sprfarbe 2
    a$=a$+MKI$(INT(sr(3)*15)*256+INT(sg(3)*15)*16+sb(3)*15)  'Sprfarbe 3
  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. feld(1) gibt die Breite, feld(2) die Höhe und feld(3) die Tiefe der Grafik an. Die Größe des Feldes wird jeweils mit einer selbstdefinierten Funktion bestimmt.
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.
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 sehen Sie 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 werden alle Objekte hinein gezeichnet.

1.7.4.2 Ein Programm mit Format

Das erste, was man bei einem Objekt machen sollte, ist seine Größe zu bestimmen. Zwar kann die Größe des Objekts auch später noch geändert werden, doch wird man sich am Anfang doch schon etwas über Form und Ausmaß des Objektes im klaren sein. Deshalb stehen diese Funktionen am Anfang der Erklärung des Programms.
Die Größe des Objekts kann man jederzeit mit dem ersten Menü abfragen. Dort gibt es den Punkt "Größe", der die aktuelle Breite und Höhe angibt, wenn er aktiviert wird.
Diese beiden Werte kann man auf zweierlei Weise beeinflussen. Beide Wege führen über die Tastatur:

1. Die Cursortasten. Die Größe des Objektes wird jeweils um eins vergrößert oder verkleinert.
2. Erst die Help-Taste drücken und dann mit dem Mauszeiger den Punkt angeben, der die neue rechte, untere Ecke sein soll.

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*184 Punkte dürften, wenn man genügend Speicher hat, für die meisten Gelegenheiten ausreichen.

1.7.4.3 Die Tiefe

Die Tiefe gehört sicherlich auch zum Format des Objekts. Sie gibt an, wieviel Farben ein Objekt maximal haben darf. Bei Tiefe 5 sind es 32 (2^5) 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 Länge und 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 entsprechenden 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 führt genau dasselbe wie der BASIC-Befehl LINE aus. 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, blinkt 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 selber 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, in dem 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 Integerwerte 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 Alles hat ein Ende...

Auch für das Verlassen des Programms gibt es zwei Möglichkeiten: Man kann entweder den Menüpunkt "Ende" aktivieren oder einfach die Taste "q" drücken. Wenn Sie die Grafik nicht vorher abgespeichert haben, sind beim Verlassen alle Daten verloren.
Durch das Verlassen des Programms werden das Fenster und der Bildschirm geschlossen. Das normale BASIC-Menü wird wieder hergestellt.

1.7.4.9 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.

1.7.5 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 wird 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 0,0,0,0,0,0,0,8176,0,0,0,0
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 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 Datenliste, 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 SaveBack 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 des Objekts, 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 welche 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 keinster 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 Disk ab.
5. Wir laden die Grafik des zukünftigen Objekts noch einmal herein.
6. Nun setzten 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-Mask. 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 entsprechendem gesetzten 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. Die gleiche Funktion gibt es natürlich auch für die Y-Richtung. Diese Funktionen sind sehr wichtig, zum Beispiel dann, 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 äuße-

ren 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 COLLISOIN-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 Program-

mierer 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- oder Selbstmaske. Die zweite Maske ist die Hit- oder Stoßmaske. Wenn ein Objekt mit einem anderen kollidiert, wird jeweils eine Me- mit der Hitmask 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 Bitebenen

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 als der Bildschirm, in dem sie sich befinden, haben. 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 falsch. 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:

```
'Demonstrationsprogramm zum
'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
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 ebenso gut mit weniger beziehungsweise 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. Oder 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 0,0,0,0,0,0,0,96,0,0
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 verschieden 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, die durch gesetzte Overlay- und SaveBack-Flags bei Bobs zu erreichen sind, geboren.
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. Dafür kann man aber die Farben selber festlegen. Wenn das Spriteflag gesetzt ist, kann man mit dem Menüpunkt "Spr.Farben" die drei Farben für das Sprite festlegen. Die Werte werden genau wie bei der PALETTE-Anweisung eingegeben.

2. Einstieg in das Amiga-Betriebssystem

Bisher kamen alle unsere Programme mit den herkömmlichen Amiga-BASIC-Befehlen aus. An dieser Stelle haben wir aber die Leistungsgrenze von AmigaBASIC erreicht. Wir wollen nun Projekte in Angriff nehmen, die sich mit BASIC-Befehlen allein nicht realisieren lassen: Eine Grafik-Hardcopyroutine, neue Zeichensätze, 1024x1024 Punkte Superbitmap, um nur einiges zu nennen.

Bei jedem anderen Computer wäre nun die Zeit gekommen, umständliche Maschinensprache-Routinen zu schreiben, die dann als Befehlserweiterung die fehlenden Möglichkeiten des AmigaBASIC ausfüllen. Nicht jedoch beim Amiga. Die Antworten zu unseren Projekten gibt es nämlich bereits, und zwar liegen sie im Amiga Betriebssystem. Dieses Betriebssystem besitzt eine Reihe von Bibliotheken (engl. "library"), in denen, sorgsam nach Themenbereich geordnet, hunderte kleiner Maschinensprache-Routinen gespeichert sind. Für fast jedes Programmierproblem kann man hier die Lösung finden. Sie sehen also, es ist gar nicht nötig, umständliche Befehlserweiterungen zu entwickeln. Man braucht lediglich einen Weg zu finden, um an die System-Bibliotheken heranzukommen.

Dieser Weg ist in das AmigaBASIC bereits eingebaut. Es handelt sich um die beiden Befehle "LIBRARY" und "DECLARE FUNCTION (...) LIBRARY". Wir werden gleich genauer auf sie eingehen. Unbedingte Voraussetzung für die Nutzung der System-Bibliotheken ist eine Datei, die mit dem Suffix ".bmap" endet. Für jede der zahlreichen System-Bibliotheken gibt es ein solches ".bmap"-File. Dieses File enthält die Namen der in der jeweiligen Bibliothek gelagerten Maschinenroutinen, neben anderen wichtigen Parametern. Mittels des von Amiga auf der "Extras"-Diskette mitgelieferten "ConvertFd"-Programms können Sie die nötigen ".bmap"-Files schnell erstellen.

Bevor Sie nun weiterlesen, sollten Sie die entsprechenden ".bmap"-Dateien generieren. Im Rahmen dieses Buches werden die Dateien

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

benötigt. Sie sollten diese Dateien nach Möglichkeit in das Unterverzeichnis "LIBS:" auf der Workbench-Diskette kopieren oder dafür sorgen, daß sich die ".bmap"-Dateien immer zusammen mit den jeweiligen sie benötigenden Programmen in einem Directory auf derselben Diskette befinden.

Nachdem diese Voraussetzungen erfüllt sind, kommen wir zur Programmierung. Wie eingangs erwähnt, stehen dazu die beiden Befehle "LIBRARY" und "DECLARE FUNCTION (...) LIBRARY" zur Verfügung. Bevor Sie eine (oder mehrere) der System-Bibliotheken benutzen können, müssen Sie diese öffnen. Das geschieht mit dem "LIBRARY"-Befehl. Für die Grafik-Bibliothek sieht das so aus:

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

Sehen wir uns einmal an, was AmigaBASIC tut, sobald es auf diesen Befehl stößt: Zunächst sucht es nach dem Definitionsfile "graphics.bmap". Ist es nicht im aktuellen Directory zu finden, sucht es AmigaBASIC auf der Workbench-Diskette im Directory "LIBS:". Ist es auch dort nicht vorhanden, kommt es unweigerlich zu einem "File not found"-Error. Sie sehen also, wie wichtig es ist, die entsprechenden ".bmap"-Dateien zu erzeugen und beim Programm zu halten.

Ist die ".bmap"-Datei gefunden, dann öffnet AmigaBASIC die entsprechende System-Bibliothek. Von diesem Zeitpunkt an vergleicht es fortwährend Funktionsaufrufe mit den in "graphics.bmap" gespeicherten Funktionsnamen und ruft gegebenenfalls die entsprechende Routine in der Bibliothek auf.

Wir werden das nun an einem kleinen Beispiel demonstrieren. Die Funktion aus der Grafik-Bibliothek, die wir 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 (darauf kommen wir gleich), die Adresse des auszugebenden Textes sowie die Länge des Textes. Hier das Programm:

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

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

Tippen Sie das Programm sorgfältig ab, wobei die Betonung auf "sorgfältig" liegt (System-Bibliotheken sind sehr sensibel, wenn es zu Fehlbedienungen kommt...). Sobald Sie das Programm starten, wird die Datei "graphics.bmap" gesucht. Anschließend erscheint in der linken oberen Ecke des Bildschirms die Meldung "Hello World!". Der Aufruf "LIBRARY CLOSE" schließt die Bibliothek wieder. Dies ist ein Schönheitsbefehl, denn Sie können ihn auch weglassen: Sobald Sie RUN oder NEW verwenden, schließt AmigaBASIC ohnehin alle bis dahin offenen Bibliotheken.

Kommen wir wieder zurück zu obigem Programm. Zwei Dinge bedürfen der weiteren Erklärung: Das Statement "SADD(a$)" sowie 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 gemeinhin eine Zeichenebene, wobei wir diesen Begriff im späteren Verlauf des Buches noch wesentlich genauer erklären. Im Moment genügt es, wenn Sie sich vorstellen, daß die Adresse des Rastports der Routine angibt, in welches Fenster sie den Text ausgeben soll. Diese Adresse finden Sie für das aktuelle Fenster immer in der Variablen WINDOW(8).

Obiges Programm sollte die Benutzung einer Bibliothek demonstrieren, mehr nicht. Schließlich hätte man dasselbe auch mit einem ganz normalen PRINT-Befehl bewerkstelligen können, nur um vieles einfacher. Darum schauen Sie sich einmal das folgende Programm an. Es ist eine SUB-Routine namens "P", die im wesentlichen nichts weiter ist als ein Ersatz für PRINT, mit der gerade gewonnenen Text-Routine als Herzstück. So rufen Sie sie auf:

```
P "text",mode%

mode%:    0 = PRINT "text";
          1 = PRINT "text"
```

```
'#####################################
'#
'# Programm: Quick-Print
'# Datum: 13.4.87
'# Autor: tob
'# Version: 1.0
'#
'#####################################

PRINT "Suche das .bmap-File..."

'GRAPHICS-Bibliothek
'Text()

LIBRARY "graphics.library"

demo:   demo$=STRING$(80,"*")
        CLS

       '* Schnelles Print
        PRINT "SCHNELLES PRINT:"
        FOR loop1%=0 TO 10
          P demo$,1
        NEXT loop1%

       '* normales PRINT
        PRINT "AmigaBasic's normales PRINT:"
        FOR loop2%=0 TO 10
          PRINT demo$
        NEXT loop2%

        LIBRARY CLOSE

SUB P(was$,mode%) STATIC
      CALL Text(WINDOW(8),SADD(was$),LEN(was$))
      IF mode%=1 THEN PRINT
END SUB
```

Unsere neue Routine hat einen ganz wesentlichen Vorteil gegenüber PRINT: Sie ist mehr als dreimal so schnell! Text erscheint blitzartig, nicht mehr Zeile für Zeile! Mehr noch: Durch ein einfaches "LPRINT" innerhalb des SUBs lassen sich sämtliche Texte auch auf den Drucker ausgeben, ohne daß hinter jedes PRINT ein weiteres LPRINT gehängt zu werden braucht.

Unser erster nutzbringender Zugriff auf eine System-Bibliothek ist damit abgeschlossen und hat Sie hoffentlich neugierig gemacht auf die

vielen weiteren Dinge, die sich mit ihrer Hilfe verwirklichen lassen. Dies hier ist lediglich eine erste Demonstration.

Doch unsere Einführung ist noch nicht ganz abgeschlossen. In unserem Demo-Programm tauchte noch gar nicht der angekündigte Befehl "DECLARE FUNCTION (...) LIBRARY" auf. Scheinbar kann man ohne ihn auskommen. Das liegt jedoch nur an der Beschaffenheit der in der Demo benutzten Text-Routine: Sie führt eine Aufgabe aus, ohne sich wieder bei AmigaBASIC zurückzumelden. Routinen dieser Art lassen sich mittels "CALL" aufrufen, ein "DECLARE"-Befehl ist unnötig. Anders sieht es bei wirklichen Funktionen aus, die einen Wert an das Programm zurückliefern. Eine solche Funktion ist beispielsweise die aus der Grafik-Bibliothek stammende Routine namens "ReadPixel". Sie benötigt als Parameter die Adresse des Rastports (also einen Wegweiser zu dem gewünschten Fenster) sowie eine X- und eine Y-Koordinate. Sie liefert den Farbwert des dort gefundenen Punktes an das Programm zurück. Um sie benutzen zu können, bedarf es einer "DECLARE"-Anweisung:

```
DECLARE FUNCTION ReadPixel% LIBRARY
LIBRARY "graphics.library"
x%=320
y%=125
rastport&=WINDOW(8)
farbe%=ReadPixel%(rastport&,x%,y%)
PRINT "gefundene Farbe = Nr.";farbe%
LIBRARY CLOSE
```

Die erste Programmzeile deklariert die Routine "ReadPixel" als Funktion, denn sie liefert nach dem Aufruf einen Wert zurück. Nachgestellt ist dem Funktionsnamen ein "%"-Zeichen, welches andeuten soll, daß der von der Routine zurückgelieferte Zahlenwert vom Typ "Integer" ist. Fortan wird die Routine unter dem Namen "ReadPixel%" aufgerufen.

Die Grundlagen wären hiermit geschaffen. Sie wissen jetzt, wie Sie eine Bibliothek öffnen können und wie Sie an die in der Bibliothek lagernden Routinen herankommen. Welche Routinen dort lagern und welcher Art die Parameter sind, mit denen sie aufgerufen werden, finden Sie auf den folgenden Seiten. Abschließend zwei Dinge: AmigaBASIC kann mit bis zu fünf Bibliotheken gleichzeitig arbeiten. Sie sollten sich weiterhin immer vor Augen halten, daß die Routinen in den Bibliotheken reine Maschinenroutinen sind. Sie sind besonders schnell, weil sie vergleichsweise unkomfortabel aufgerufen werden müssen. Die Routinen überprüfen die von Ihnen gelieferten Parameter

nicht auf ihre Richtigkeit, sondern führen die Anweisungen blind aus. Ein kleiner Fehler, ein falscher Parameter, ein verkehrter Aufruf, und Sie kommen in den Genuß eines System-Crashs. Das ist nicht weiter schlimm, sieht man von der Tatsache ab, daß der Rechner nach einem solchen Crash von neuem mit der Workbench-Diskette gefüttert werden muß - alle Programme im Speicher sind verloren.

Tippen Sie deshalb Programme sehr sorgfältig ab. Speichern Sie die abgetippten Programme vor dem Gebrauch.

Wie leicht ein Crash passieren kann, wollen wir Ihnen natürlich nicht vorenthalten. Deshalb zur Abschreckung das folgende Programm.

Achtung: Dieses Programm ist fehlerhaft und produziert einen Crash - alle Programme im Speicher werden dadurch gelöscht!

```
      LIBRARY "graphics.library"
      a$="Hello World!"
      laenge%=LEN(a$)
      rasport&=WINDOW(8)
REM Hier ist der Fehler! rasport& statt rastport&
REM rastport& also =0
      CALL Text(rastport&,SADD(a$),laenge%)
      LIBRARY CLOSE
```

Es kommt nach einiger Zeit zu folgendem Display:

```
Software Failure. Press left mouse button to continue.
  Guru Meditation #00000004.00224D6
```

Folgen Sie der Anweisung, oder drücken Sie gleichzeitig die beiden "A"-Amiga-Tasten sowie die "CTRL"-Taste.

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 BASIC-Interpreters: "LIST" und "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 immer in der Variable WINDOW(7) gespeichert. Die dort zu findende Adresse zeigt auf einen Datenblock, der folgendermaßen aufgebaut ist:

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

Datenstruktur "Window"/Intuition/124 Bytes

Offset	Typ	Bezeichnung
+ 000	Long	Zeiger zu nächstem 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
+ 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	Zeiger auf Menü-Header
+ 032	Long	Zeiger auf Titeltext für dieses Fenster
+ 036	Long	Zeiger auf ersten aktiven Requester
+ 040	Long	Zeiger auf Double-Click-Requester
+ 044	Word	Anzahl der das Fenster block. Request
+ 046	Long	Zeiger auf den Screen, in dem Fenster ist
+ 050	Long	Zeiger auf den Rastport des Fensters
+ 054	Byte	Linker Rahmen
+ 055	Byte	Oberer Rahmen
+ 056	Byte	Rechter Rahmen
+ 057	Byte	Unterer Rahmen
+ 058	Long	Zeiger auf Rahmenrastport
+ 062	Long	Zeiger auf das erste Gadget
+ 066	Long	Zeiger auf Eltern-Fenster
+ 070	Long	Zeiger auf Kind-Fenster
+ 074	Long	Zeiger auf 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
+ 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	Zeiger auf Menü-Haken
+ 104	Long	Zeiger auf Screen-Titeltext
+ 108	Word	GZZ-MausX
+ 110	Word	GZZ-MausY
+ 112	Word	GZZ-Breite

+ 114	Word	GZZ-Höhe
+ 116	Long	Zeiger auf externe Daten
+ 120	Long	Zeiger auf User Daten

Jedes Fenster besitzt einen solchen Datenblock, gefüllt mit den entsprechenden Parametern. Um mit diesem Datenblock arbeiten zu können, gehen Sie 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). 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 und durch POKEW manipuliert. Ein Long-Feld schließlich besteht aus vier Bytes und wird analog durch PEEKL ausgelesen und mit Hilfe von POKEL 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 die Datenstruktur Eintrag für Eintrag näher unter die Lupe nehmen, um zu sehen, was man mit ihrer Hilfe bewerkstelligen kann.

Offset 0: Zeiger zu nächstem Fenster

Hier finden Sie die Anfangsadresse eines weiteren Fenster-Datenblocks. Intuition verwaltet alle Fenster in einer Kette. Von jedem Fenster-Datenblock kann man zum nächsten und zum vorangegangenen Block finden. Dieser Eintrag ist jedoch nicht wichtig, denn die eigentlichen Ketten-Zeiger folgen erst später ab Offset 66 und Offset 70.

Offset 4 und 6: Positionen der linken oberen Fensterecke

Diese beiden Word-Felder enthalten die X- und Y-Koordinate der linken oberen Ecke Ihres Fensters relativ zur linken oberen Ecke des Screens, in dem das Fenster liegt. Diese Abfrage liefert Ihnen die Koordinate:

```
fenster&=WINDOW(7)
x%=PEEKW(fenster&+4)
y%=PEEKW(fenster&+6)
PRINT "Obere linke Fensterecke liegt auf"
PRINT "Koordinate (";x%;",";y%;")"
END
```

Für das BASIC-Ausgabefenster ergibt diese Abfrage normalerweise die Koordinate (0,0), denn dieses Fenster liegt in der obersten linken Ecke des Workbench-Screens. Sobald Sie dieses Fenster jedoch verschieben, ändern sich die Werte. Mit diesen Werten werden wir später einige Intuition-Routinen aufrufen können.

Offset 8 und 10: Fenster-Dimensionen

Bekanntlich lassen sich viele Fenster durch das Sizing-Gadget in der rechten unteren Fensterecke vergrößern bzw. verkleinern. Es ist daher für ein Programm nicht immer klar, welche Abmessungen ein Fenster augenblicklich besitzt. Dieses einfache Programm besorgt die aktuellen Fenster-Ausdehnungen:

```
fenster&=WINDOW(7)
breite%=PEEKW(fenster&+8)
hoehe%=PEEKW(fenster&+10)
PRINT "Im Augenblick ist Ihr Fenster ";breite%
PRINT "Pixel breit und ";hoehe%;" Pixel hoch."
END
```

Offset 12 und 14: Maus-Koordinaten

Diese beiden Word-Felder enthalten immer die aktuellen Koordinaten der Maus relativ zur oberen linken Ecke des Fensters. Das erste Feld enthält dabei die Y-, das zweite die X-Koordinate.

Man kann mit Hilfe dieser beiden Einträge ohne geringste Mühe ein kleines Zeichenprogramm schreiben, wie die folgenden Zeilen beweisen:

```
'#####################################
'#
'# Programm: Maus-Zeichner I
'# Datum: 5. April 87
'# Autor: tob
'# Version: 1.0
'#
'#####################################

init:       fenster&=WINDOW(7)
           '* Adresse der Fenster-Datenstruktur

loop:      '* wartet auf Tastendruck
            PRINT "Beliebige Taste fuer Abbruch"
            WHILE INKEY$=""
              maus.y% = PEEKW(fenster&+12)
              maus.x% = PEEKW(fenster&+14)
              PSET (maus.x%,maus.y%)
            WEND
```

Zwei Dinge fallen auf: Erstens beginnt die gezeichnete Linie einige Pixel unterhalb der Maus, zweitens werden Punkte an Stelle einer Linie gezeichnet. Diese beiden Felder enthalten die Maus-Koordinaten relativ zur oberen linken Fensterecke. Die Zeichenebene, in die PSET zeichnet, hat ihren Ursprung jedoch **nicht** in der oberen linken Ecke des Fensters, sondern in der oberen linken Ecke der Zeichenebene (sofern es sich um ein GimmeZeroZero-Fenster handelt; darauf kommen wir später ausführlich zu sprechen). Subtrahieren Sie einfach den Wert 11 vom Y-Wert, den Wert 4 von der X-Koordinate. Das zweite Problem liegt in der relativ langsamen BASIC-Schleife: Die Maus wird schneller bewegt, als BASIC Punkte zeichnen kann. Sie können dies leicht nachprüfen, indem Sie einmal mit der Maus sehr langsam auf dem Bildschirm entlangfahren. Scheinbar entstehen so Linien. Man kann sich jedoch helfen, indem mittels des LINE-Befehls von Maus-Position zu Maus-Position eine Linie gezogen wird. Das folgende Programm beinhaltet beide Änderungen:

```
'#####################################
'#
'# Programm: Maus-Zeichner II
'# Datum: 5. April 87
'# Autor: tob
'# Version: 1.0
'#
'#####################################

init:       fenster&=WINDOW(7)
           '* Adresse der Fenster-Datenstruktur
            maus.y% = PEEKW(fenster&+12)-11
            maus.x% = PEEKW(fenster&+14)-4

loop:      '* wartet auf Tastendruck
            PRINT "Beliebige Taste = Abbruch"
            WHILE INKEY$=""
              altmaus.y% = maus.y%
              altmaus.x% = maus.x%
              maus.y%    = PEEKW(fenster&+12)-11
              maus.x%    = PEEKW(fenster&+14)-4
              LINE (altmaus.x%,altmaus.y%)-(maus.x%,maus.y%)
            WEND
```

Wenn Sie bei diesem Programm über den Bildschirm flitzen, werden Sie eine interessante Entdeckung machen: Es treten vereinzelt "Zacken" auf. Das geschieht immer dann, wenn die Maus-Koordinaten gerade aktualisiert werden, der Y-Wert jedoch noch nicht geändert ist.

Offset 16, 18, 20 und 22: Fenster-Limits

Einige Fenster lassen sich mit Hilfe der Maus vergrößern und verkleinern. Jedes Fenster besitzt aber seine ganz spezielle Mindest- und Maximalgröße. Zwischen diesen beiden Grenzen läßt sich das Fenster stufenlos regulieren. Diese Grenzen liegen hier in der Fenster-Datenstruktur:

```
fenster&=WINDOW(7)
min.x%=PEEKW(fenster&+16)
min.y%=PEEKW(fenster&+18)
max.x%=PEEKW(fenster&+20)
max.y%=PEEKW(fenster&+22)
PRINT "Mindestgroesse: X=";min.x%;
PRINT "Y=";min.y%
PRINT "Maximalgroesse: X=";max.x%;
PRINT "Y=";max.y%
END
```

In diesem Beispiel werden die Grenzen ausgelesen. Genausogut können Sie die Grenzen selbst festlegen, indem Sie eigene Werte mittels POKEW an die entsprechenden Adressen poken:

```
fenster&=WINDOW(7)
min.x%=5
min.y%=7
max.x%=640
max.y%=200
POKEW fenster&+16,min.x%
POKEW fenster&+18,min.y%
POKEW fenster&+20,max.x%
POKEW fenster&+22,max.y%
END
```

Sie müssen allerdings unbedingt darauf achten, daß:

a) die Mindestwerte kleiner sind als die Maximalwerte.
b) die Maximalwerte nicht kleiner sind als das augenblickliche Fenster (Dimensionen lassen sich ab Offset 8 und 10 erfragen!).

Offset 24: Fenster-Modi

Intuition kennt verschiedene Fenster-Typen. Jedes Fenster läßt sich zunächst mit unterschiedlichen Dingen ausrüsten:

a) Vergrößerungs-/Verkleinerungsgadget
b) Verschiebe-Kopfleiste
c) Vordergrund-/Hintergrund-Knöpfe
d) Schließ-Knopf

Außerdem läßt sich die Refresh-Art festlegen. Unter Refresh-Art versteht man die Methode, mit der der Inhalt des Fensters gerettet wird, wenn ein anderes Fenster über dem betreffenden liegt. "Simple Refresh" überläßt diese Aufgabe Ihnen. Im Normalfall bedeutet dies: Wird Ihr Fenster kurzfristig von einem anderen Fenster überschattet, dann ist der überschattete Inhalt Ihres Fensters verloren. "Smart Refresh" beauftragt Intuition, verdeckte Fensterinhalte automatisch im RAM zu speichern, damit diese Teile später wieder zurückkopiert werden können. Diese Methode benötigt bereits mehr Zeit und unter Umständen sehr viel Speicherplatz. Die "Superbitmap"-Methode

schließlich speichert von vornherein den gesamten Fensterinhalt anderswo im RAM. Sie ist sehr speicheraufwendig, hat aber den Vorteil, daß das Fenster einen Ausschnitt aus einer sehr viel größeren Grafik zeigen kann.

Neben diesen Grundattributen eines Fensters gibt es Spezialfenster. Ein "Backdrop"-Fenster liegt immer hinter allen anderen Fenstern und läßt sich nicht nach vorn holen. Es dient beispielsweise als Hintergrund oder Grafikebene. Der sichtbare Workbench-Screen ist beispielsweise nichts anderes als ein Backdrop-Fenster. Es bedeckt den eigentlichen Screen. Ein "GimmeZeroZero"-Fenster ist zweigeteilt: Es besteht aus einem Rahmen und einer Zeichenebene. Diese Methode erlaubt völlig ungezwungenes Zeichnen, denn es ist unmöglich, versehentlich in den Rahmen des Fensters hineinzuzeichnen. Das "BASIC"-Fenster ist beispielsweise ein solches Fenster. Das "Borderless"-Fenster besitzt keinen Rahmen. Das Backdrop-Workbench-Fenster ist beispielsweise zugleich vom Typ "Borderless", denn es hat keinen Rahmen und verschmilzt mit dem Hintergrund.

Die Bit-Belegungen für diese Modi entnehmen Sie bitte dem Diagramm in Kapitel 3.1. Hinweis: Änderungen in diesem Feld machen sich erst bemerkbar, wenn das entsprechende Fenster von Intuition erneut gezeichnet wird, z.B. wenn Sie es verschieben.

Offset 28: Der Menü-Header

Eine besondere Eigenschaft eines jeden Intuition-Fensters ist die Möglichkeit, ein Menü zu erzeugen. Auf Druck der rechten Maustaste erscheinen dann unterschiedliche Menüpunkte auf der obersten Kopfleiste, können angewählt und selektiert werden.

Dieses Feld enthält den Zeiger auf das Intuition-Menüsystem für dieses Fenster.

Offset 32: Der Titeltext dieses Fensters

Jedes Fenster hat seinen Namen. Hier ist die Anfangsadresse auf diesen Titeltext gespeichert. Eine einfache Methode, ihn zu verändern, zeigen die folgenden Zeilen:

```
fenster.name$="Hallo Welt"+CHR$(0)
POKEL WINDOW(7)+32,SADD(fenster.name$)
```

Sobald Sie nun das Fenster etwas verschieben (Intuition also das Fenster neu zeichnet), erscheint der neue Name. Später werden Sie eine Methode kennenlernen, den Fensternamen mit sofortiger Wirkung setzen zu können.

Die Anweisung CHR$(0) ist ein Nullbyte, das am Ende des Textes angehängt wird. Es gibt Intuition zu verstehen, daß der Text beendet ist.

Offset 36, 40 und 44: Requester-Handling

Diese Datenfelder dienen Intuition zur Erinnerung, wieviel Requester (und welche) das Fenster zur Zeit blockieren. BASIC-Programmierer können sie vorerst getrost vergessen.

Offset 46: Kontakt zum Screen

Frei umherschwebende Fenster gibt es nicht. Jedes Fenster erscheint in einem Screen. Zu Anfang gibt es zumindest den "Workbench"-Screen, aber es ist natürlich möglich, weitere mittels des SCREEN-Befehls hinzuzufügen. Hier finden Sie die Adresse der Intuition-Screen-Datenstruktur. Wie jedes Fenster eine eigene Datenstruktur besitzt, hat man auch jedem Screen eine solche mitgegeben. Sie ist natürlich anders aufgebaut als die Fensterstruktur. Wir werden uns später mit ihr genau auseinandersetzen.

Offset 50: Kontakt zum Rastport des Fensters

Hier haben wir endlich eine Spur des in Kapitel 2 andeutungsweise erwähnten Rastports. Der Rastport ist im Grunde nichts anderes als eine weitere Datenstruktur. Man könnte den Rastport als eine Kreuzung definieren. Von dort gehen Wege zum Fenster, zum Screen und sogar bis zu den elementarsten Grafik-Komponenten wie Bitmap und Layers (keine Angst, wir werden das alles später ausführlich angehen!).

Offset 54, 55, 56 und 57: Fensterrahmen

Diese vier Byte-Felder enthalten die Dimensionen der Fensterrahmen in Pixel. Als wir uns mit den Maus-Koordinaten beschäftigten (Offset 12, 14), subtrahierten wir die Werte 11 und 4. Dieselben Werte finden

sich hier wieder; es ist die Höhe des oberen Rahmens sowie die Breite des linken.

Wenn Sie mit einem GimmeZeroZero-Fenster arbeiten, können Sie durch Addieren der Dimensionen des oberen und des linken Fensterrahmens die Zeichenkoordinaten relativ zur oberen linken Ecke des Fensters erfahren.

Wenn Sie mit einem anderen Fenstertyp arbeiten, müssen Sie darauf achten, daß Ihre Grafiken nicht in den Fensterrahmen hineinzeichnen. Das erreichen Sie, indem Sie darauf achten, daß:

a) Ihre X-Koordinaten größer sind als der linke Fensterrahmen, jedoch kleiner als die Fensterbreite (Offset 8) minus die Breite des rechten Fensterrahmens.
b) Für die Y-Koordinaten gilt das gleiche.

Offset 58: Der Rahmen-Rastport

Wie bereits angeschnitten, verfügen alle GimmeZeroZero-Fenster über zwei unabhängige Zeichenebenen: den Fensterrahmen sowie den Fensterinhalt. Die hier gespeicherte Adresse ist der Rastport für den Rahmen eines GimmeZeroZero-Fensters. Das folgende Programm richtet mit Hilfe des Rahmen-Rastports und der Grafik-Funktion "Text" eine Statuszeile im Fensterkopf ein:

```
'########################################
'#
'# Programm: Status-Zeile
'# Datum: 17.12.86
'# Autor: tob
'# Version: 1.0
'#
'########################################

'Dieses Programm errichtet im BorderRastPort eines GemmeZeroZero-Windows
'eine Benutzer-Statuszeile von Laenge x. x ist abhaengig von der aktuel-
'len Fenstergroesse. Durch Error-Check wird verhindert, dass Gadgets zer-
'stoert werden.

PRINT "Suche die .bmap-Dateien..."

'GRAPHICS-Bibliothek
'Text()
'SetAPen()
```

```
'SetDrMd()
'Move()

LIBRARY "graphics.library"

main:       CLS
            Status "STATUS: Demo-Fenster. Bitte Taste druecken!",60
            WHILE INKEY$=""
            WEND
            WHILE jn$<>"j"
              Status "STATUS: Bitte Namen eingeben!",60
              CLS
              LOCATE 1,1
              LINE INPUT "--> ";n$
              Status "Name: "+n$+". Korrekt (j/n) ?",60
              LOCATE 1,1
              PRINT SPACE$(50)
              LOCATE 1,1
              LINE INPUT "--> ";jn$
            WEND
            Status "Versuch wird abgeschlossen. Tschuess! (TASTE!)",0
            CLS
            WHILE INKEY$=""
            WEND

ende:       WINDOW 1,""
            LIBRARY CLOSE
            END


SUB Status(text$,weite%) STATIC
    borderRast&    = PEEKL(WINDOW(7)+58)
    IF borderRast& = 0 THEN
      BEEP
      PRINT "Dies ist kein Fenster vom Typ GimmeZeroZero."
      ERROR 255
    END IF

    fensterWeite% = PEEKW(WINDOW(7)+8)
    maxZeichen%   = INT((fensterWeite%-86)/8)
    TextLaenge%   = LEN(text$)
    IF weite%     = 0 THEN weite%=TextLaenge%
    IF weite%<maxZeichen% THEN maxZeichen%=weite%
    IF TextLaenge%<weite% THEN
      text$ = text$+SPACE$(weite%-TextLaenge%)
    END IF
    CALL SetAPen(borderRast&,1)
    CALL Move(borderRast&,32,7)
    CALL text(borderRast&,SADD(text$),maxZeichen%)
```

```
        CALL SetDrMd(borderRast&,0)
        CALL SetAPen(borderRast&,3)
        CALL Move(borderRast&,31,7)
        CALL text(borderRast&,SADD(text$),maxZeichen%)
        CALL SetDrMd(borderRast&,1)
END SUB
```

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
```

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 an dieses Kapitel, wie es gemacht wird.

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 auszuschalten). 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:

```
screen&=PEEKL(WINDOW(7)+46)
```

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	Zeiger auf nächsten Screen
+ 004	Long	Zeiger auf 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	Zeiger auf Screen-Namenstext
+ 026	Long	Zeiger auf 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	Zeiger auf Standard Font TextAttr
+ 044	----	Viewport des Screens
+ 084	----	Rastport des Screens
+ 184	----	Bitmap des Screens
+ 224	----	Layerinfo des Screens
+ 326	Long	Zeiger auf erstes Screen-Gadget
+ 330	Byte	Detail Pen
+ 331	Byte	Block Pen
+ 332	Word	Backup-Register für Beep(), speichert Col0
+ 334	Long	Zeiger auf externe Daten
+ 338	Long	Zeiger auf User Daten

Sicherlich werden Sie viele Ähnlichkeiten zwischen dieser und der Fenster-Datenstruktur aus Kapitel 2.1 gefunden haben. Einige 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.

3.4.2 Die Intuition-Funktionen für das Screen-Handling

Es folgt hier eine Auswahl der Funktionen aus der Intuition-Bibliothek, die sich mit dem Handling von Screens beschäftigen. Es sind dies die Routinen:

MoveScreen()
ScreenToBack()
ScreenToFront()
WBenchToBack()
WBenchToFront()

Wir haben der Einfacheit halber drei SUBs geschrieben, die all diese Routinen verwenden. Sie nennen sich "ScrollScreen", "ScreenHer" und "ScreenWeg".

ScrollScreen verlangt die Anzahl der Pixel, die der Screen, in dem sich das augenblickliche Ausgabefenster befindet, heruntergeschoben werden soll (negativ = heraufgeschoben). ScreenWeg schickt denselben Screen hinter alle bestehenden Screens, SceenHer holt ihn in den Vordergrund.

Hier sind die SUBs, zusammen mit einer kleinen Demonstration:

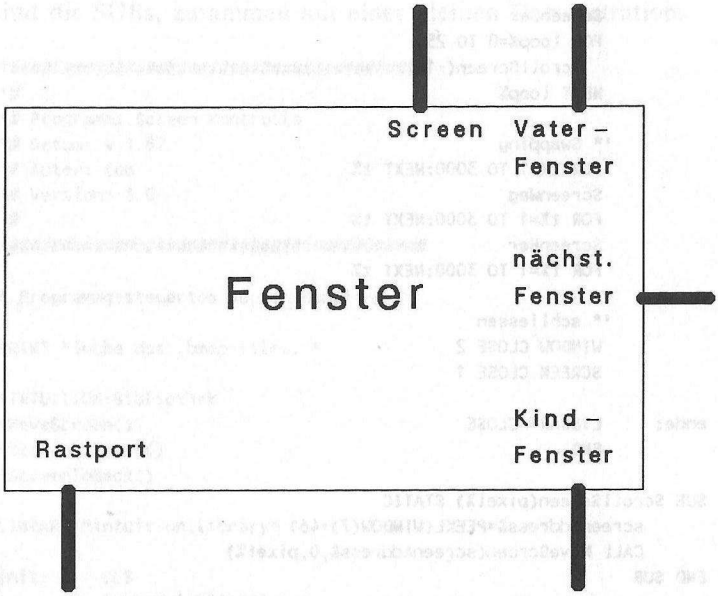
```
'########################################
'#
'# Programm: Screen-Kontrolle
'# Datum: 4.1.87
'# Autor: tob
'# Version: 1.0
'#
'########################################

' Programmgesteuertes Screen-Handling

PRINT "Suche das .bmap-File..."

'INTUITION-Bibliothek
'MoveScreen()
'ScreenToFront()
'ScreenToBack()

LIBRARY "intuition.library"

init:     CLS
          SCREEN 1,320,200,1,1
```

```
        WINDOW 2,"Hallo!",,,1

main:   '* Screens verschieben
        PRINT "Dies ist ein 2. Screen!!!"

        '* runter Screen 1
        WINDOW OUTPUT 2
        FOR loop%=255 TO 0 STEP -1
          ScrollScreen(1)
        NEXT loop%

        '* runter Screen 0
        WINDOW OUTPUT 1
        FOR loop%=255 TO 0 STEP -1
          ScrollScreen(1)
        NEXT loop%

        '* hoch Screen 1
        WINDOW OUTPUT 2
        FOR loop%=0 TO 255
          ScrollScreen(-1)
        NEXT loop%

        '* hoch Screen 0
        WINDOW OUTPUT 1
        ScreenHer
        FOR loop%=0 TO 255
          ScrollScreen(-1)
        NEXT loop%

        '* Swapping
        FOR t%=1 TO 3000:NEXT t%
        ScreenWeg
        FOR t%=1 TO 3000:NEXT t%
        ScreenHer
        FOR t%=1 TO 3000:NEXT t%

        '* schliessen
        WINDOW CLOSE 2
        SCREEN CLOSE 1

ende:   LIBRARY CLOSE
        END

SUB ScrollScreen(pixel%) STATIC
    screenAddress&=PEEKL(WINDOW(7)+46)
    CALL MoveScreen(screenAddress&,0,pixel%)
END SUB
```

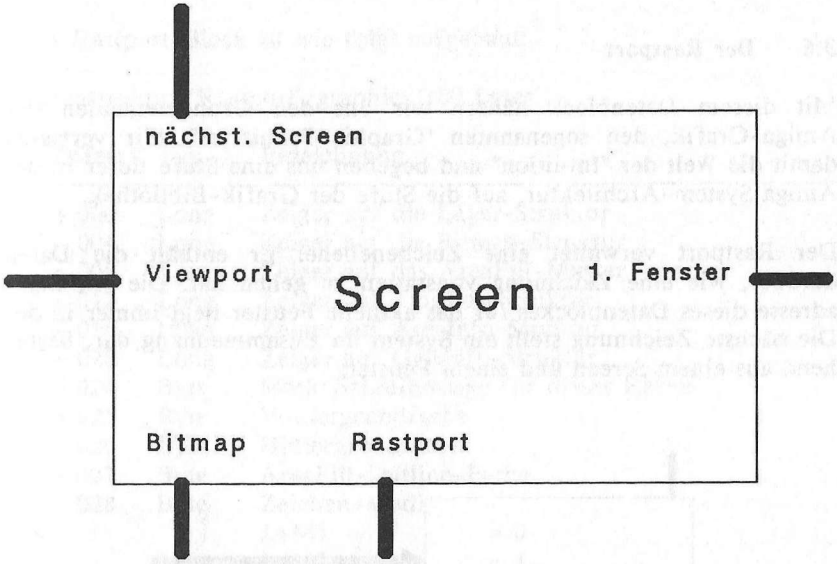
```
SUB ScreenHer STATIC
      screenAddress&=PEEKL(WINDOW(7)+46)
      CALL ScreenToFront(screenAddress&)
END SUB

SUB ScreenWeg STATIC
      screenAddress&=PEEKL(WINDOW(7)+46)
      CALL ScreenToBack(screenAddress&)
END SUB
```

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:



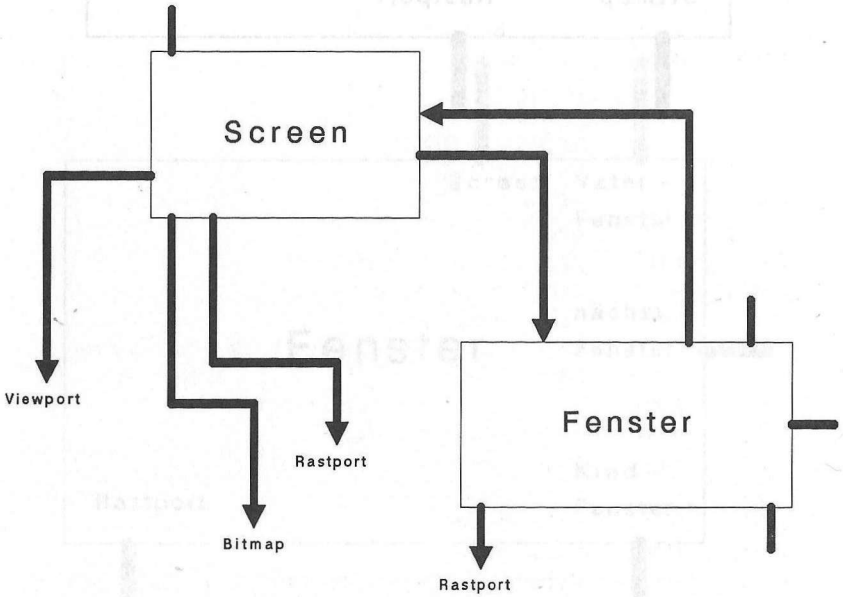
Noch ist das Bild unvollständig. Es fehlen gänzlich Kenntnisse über die Strukturen "Rastport", "Viewport" und "Bitmap".

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 immer in der Die nächste Zeichnung stellt ein System im Zusammenhang dar, bestehend aus einem Screen und einem Fenster:



Variablen WINDOW(8). Ebenso gut läßt sie sich aber auch direkt aus der Fenster-Datenstruktur auslesen:

```
PRINT WINDOW(8)
PRINT PEEKL(WINDOW(7)+50)
```

Der Rastport-Block ist wie folgt aufgebaut:

Datenstruktur "Rastport"/graphics/100 Bytes

Offset	Typ	Bezeichnung
+ 000	Long	Zeiger auf die Layer-Struktur
+ 004	Long	Zeiger auf die Bitmap-Struktur
+ 008	Long	Zeiger auf das AreaFill-Muster
+ 012	Long	Zeiger auf TmpRas-Struktur
+ 016	Long	Zeiger auf AreaInfo-Struktur
+ 020	Long	Zeiger auf 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: 2^n 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	Zeiger auf 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	Zeiger auf 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
```

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: Zeiger auf 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...).

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:

$$bit\&=2^{15}+2^{14}+2^{13}+2^{12}+2^{11}+2^{9}+2^{7}+2^{6}+2^{5}+2^{4}+2^{3}+2^{1}$$

Nun wird dieser Wert in dieses Register geschrieben:

```
POKEW WINDOW(8)+34,bit&
```

Ein Test:

```
LINE (10,10)-(600,10)
```

Sie sehen, es funktioniert.

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
```

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 Druchschnittswert 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 weitere Daten, die sich mit dieser Struktur verbinden lassen.

Offset 70 und folgende: reservierte Datenfelder

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 unser neues 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 Speicherplatz 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
```

Diese sechs Routinen vereinfachen die Arbeit mit Mustern ungemein. Hier die Routinen, zusammen mit einem kleinen Beispielprogramm:

```
'#####################################
'#
'# Programm: Mono-Pattern Module
'# Datum: 27.12.86
'# Autor: tob
'# Version: 1.0
'#
'#####################################

' Einfaches Definieren von Mustern in binae-
' rer Schreibweise

PRINT "Suche die .bmap-Dateien..."

'EXEC-Bibliothek
DECLARE FUNCTION AllocMem& LIBRARY
'FreeMem()

LIBRARY "exec.library"


init:       '* System aktivieren
            CLS
            InitPattern 4
```

```
          '* SetPattern
           SetPat 0,"BBBBBBBBBBBBBBBB"
           SetPat 1,"BAAAAAAAAAAAAAAB"
           SetPat 2,"BAAAABBBBBBAAAAB"
           SetPat 3,"BAAAABBBBBBAAAAB"

          '* Neues Pattern einschalten
           MonoPattern

          '* Pattern auf den Schirm zaubern
           LINE   (10,10)-(100,100),1,bf
           CIRCLE (300,100),100
           PAINT  (300,100),2,1

          '* System wieder ausschalten
           EndPattern

ende:     LIBRARY CLOSE
          END


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
```

Bis jetzt lieferten auch unsere SUBs keine farbigen Muster. Das soll anders werden. Wir fügen die Routine "ColorPattern" hinzu. Sie entspricht "MonoPattern", aktiviert jedoch das Multi-Color-System. Sie können nun, in Abhängigkeit von der Tiefe Ihres Screens, bis zu 32 Farben in den Mustern verwenden. Zur Definition Ihres Musters stehen Ihnen fortan weitere Buchstaben 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)

Das folgende Programm demonstriert MC-Pattern:

```
'########################################
'#
'# 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,"AAAAAAAACCACCAA"
            SetPat 3,"AAAAAAADDADDAAA"
            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
```

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 unsere 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.

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 Aeus
seres!",0
            Outline "  Trotz eines sehr aufwendigen Zeichenprozesses ist die O
utline-Routine",0
            Outline "  aeusserst schnell durch konsequenten Einsatz der Text()
-Funktion.",0
            Outline "  OUTLINE funktioniert natuerlich auch mit anderen Zeiche
nsaetzen...!",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
```

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

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
```

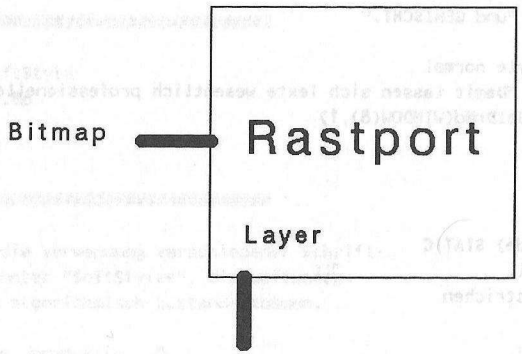
```
'SetDrMd()

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 Zeichensaetze
n."
          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
```

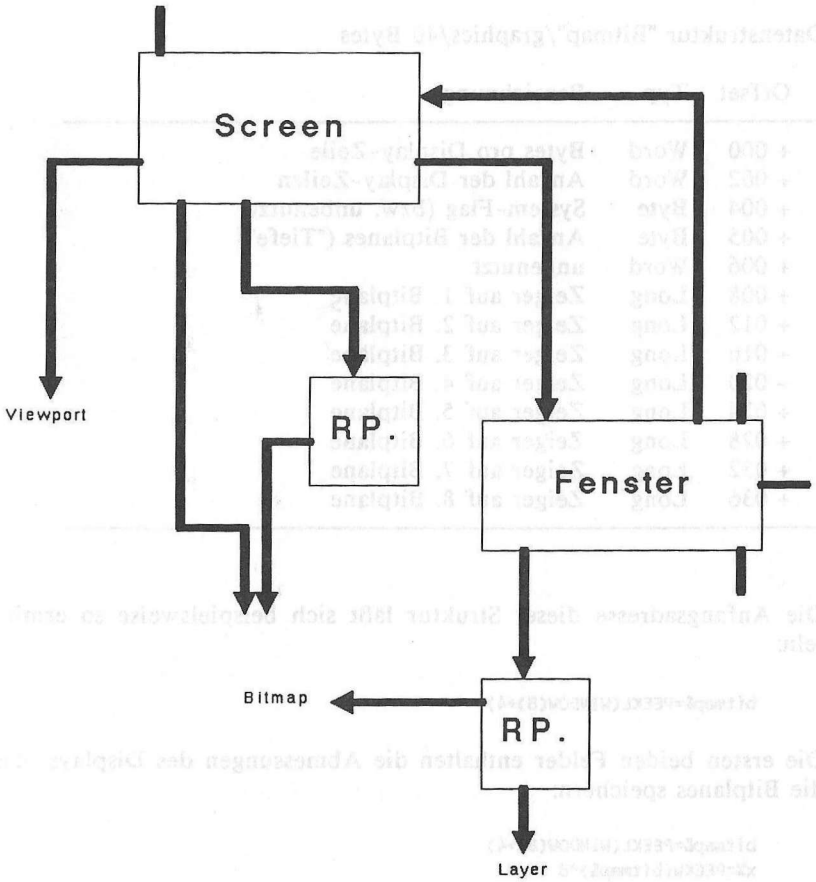
Vielleicht ist Ihnen etwas aufgefallen: Der Kursivdruck des ersten Beispiels sah etwas unförmig aus. Bei diesem Programm erschien er jedoch leserlich. Das Geheimnis ist schnell gelüftet: Der Kursivdruck funktioniert nur im Zeichenmodus JAM1 korrekt, denn durch die Schrägstellung der Zeichen fällt jeweils 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)
```

Die ersten beiden Felder enthalten die Abmessungen des Displays, das die Bitplanes speichern:

```
bitmap&=PEEKL(WINDOW(8)+4)
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

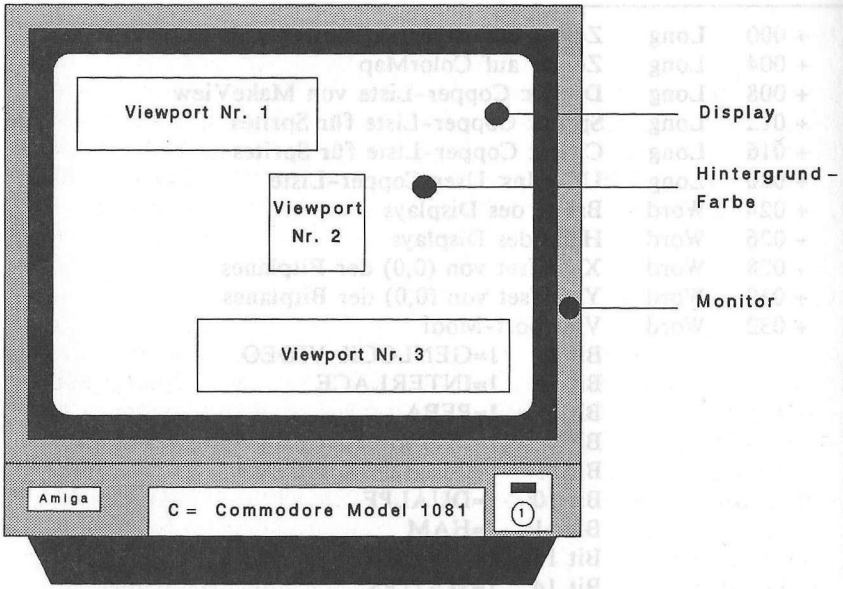
Offset	Type	Bezeichnung
+ 000	Long	Zeiger auf nächsten Viewport
+ 004	Long	Zeiger auf ColorMap
+ 008	Long	DspIns: Copper-Liste von MakeView
+ 012	Long	SprIns: Copper-Liste für Sprites
+ 016	Long	ClrIns: Copper-Liste für Sprites
+ 020	Long	UCopIns: User Copper-Liste
+ 024	Word	Breite des Displays
+ 026	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, 2^5=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 2^5=32 auf 2^6=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 einem 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.

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%+1
6,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 werden 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 übernimmt 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.

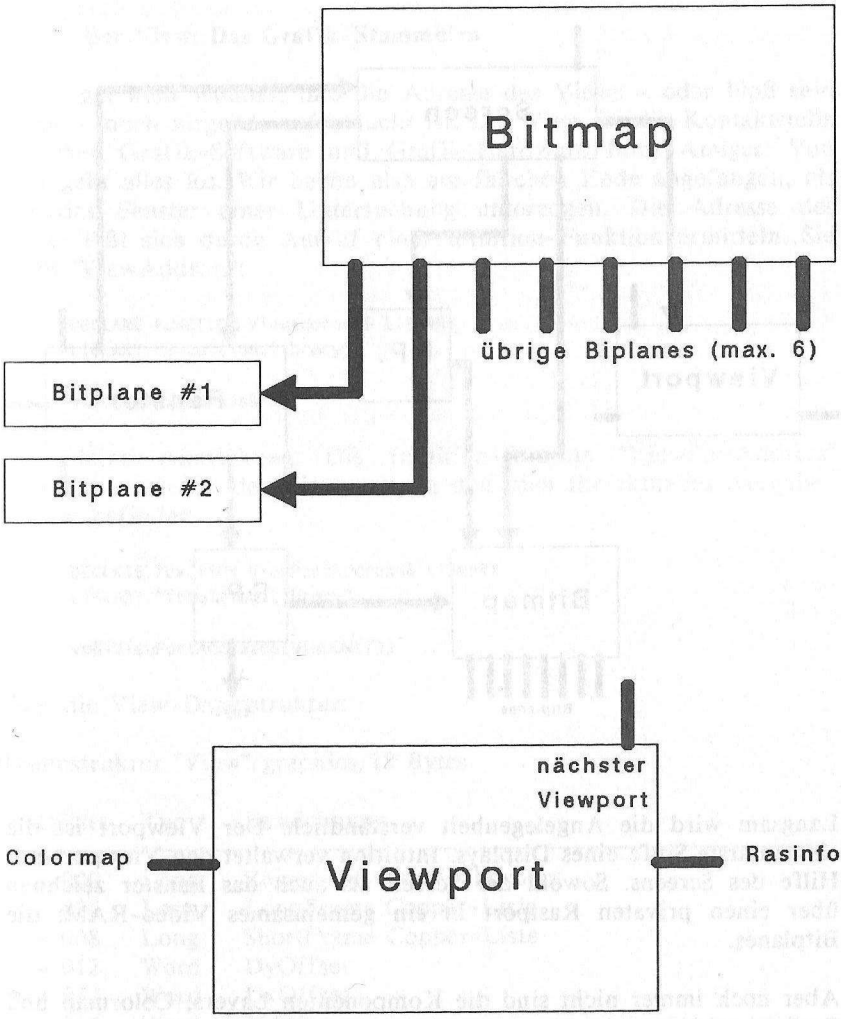
Direkt rechts von dieser HAM-Farbe werden nun sechzehn HAM-Farben des Typs 1 gezeichnet, wobei der Rot-Wert jeweils um eins zunimmt.

Es entsteht so ein Farbmuster, dessen Rot-Intensität nach rechts hin zunimmt, dessen Blau-Intensität nach unten hin größer wird.

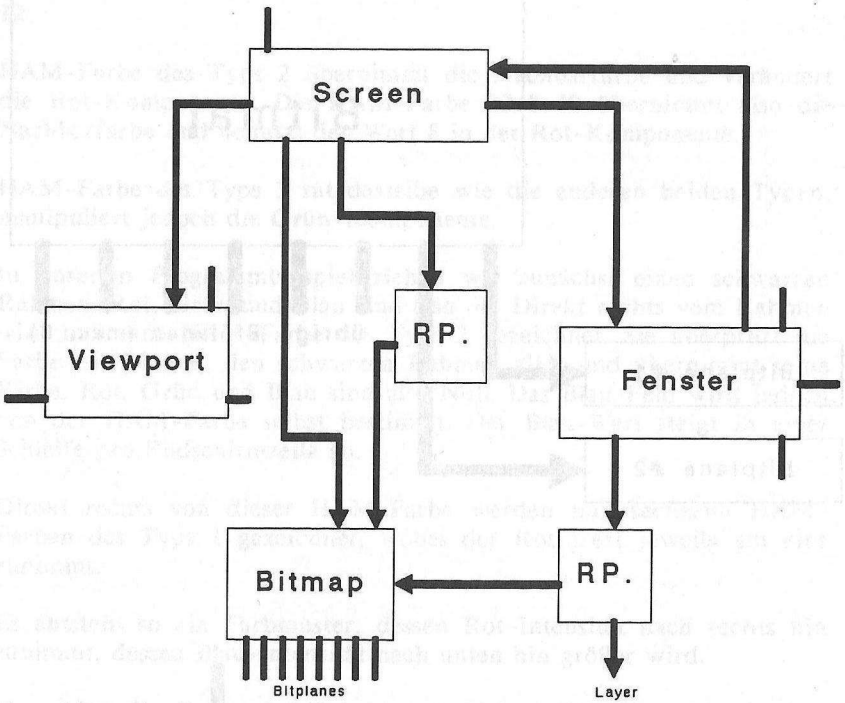
Nun wird die Farbe des Rahmens verändert: Der ehemals schwarze Rahmen wird mittels PALETTE immer grüner. Der Grün-Wert des Rahmens wird dabei sofort von den HAM-Farben unverändert übernommen: Die gesamte Farbgrafik wird also von immer intensiverem Grün durchflutet.

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:


Bitmap
übrige Biplanes (max. 6)
Bitplane #1
Bitplane #2
nächster
Viewport
Colormap
Viewport
Rasinfo

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&
```

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

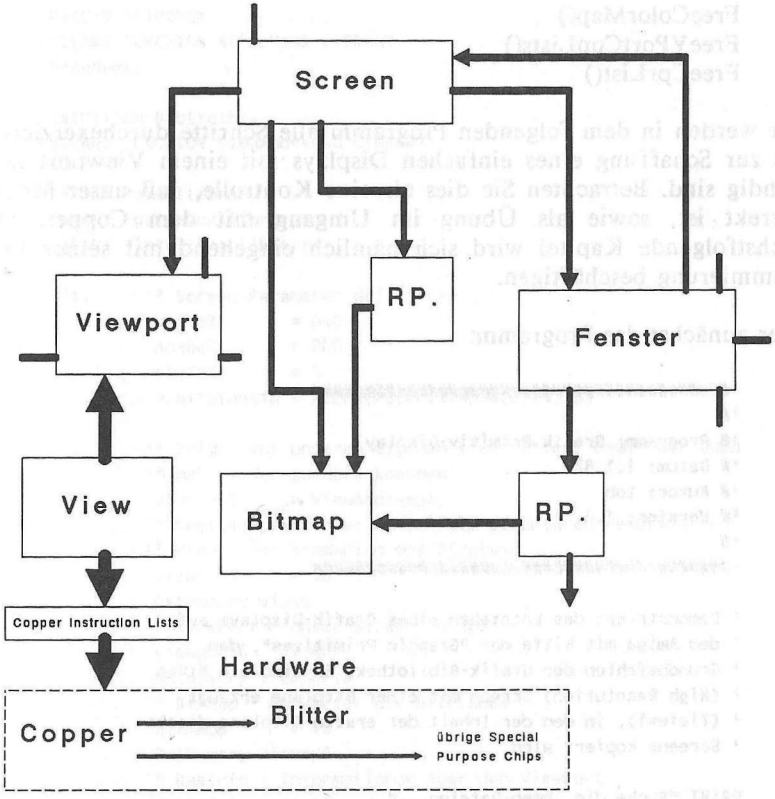
Hier findet sich die Adresse auf die erste Viewport-Struktur des Displays. Von dort findet sich dann die Adresse zu weiteren Viewports, falls es weitere gibt.

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 Instruk-

tionen 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

PZunächst 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 Mög-

lichkeit, 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 B
ildschirm!"

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 UCopList-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&,screenBitMap&,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 fahren 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))
```

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:

```
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. Suchen wir uns zunächst die Grenzen unseres eigenen Layers heraus:

```
fenster&=WINDOW(7)
rastport&=WINDOW(8)
layer&=PEEKL(rastport&)

x0%=PEEKW(layer&+16)
y0%=PEEKW(layer&+18)
x1%=PEEKW(layer&+20)
y1%=PEEKW(layer&+22)

PRINT x0%,y0%
PRINT x1%,y1%

END
```

Das Ergebnis sind die Koordinaten der linken oberen und der rechten unteren Ecke unserer Zeichenebene.

Sie können natürlich auf diese Art eine ganz individuelle Zeichenebene definieren; alles außerhalb dieser Fläche wird "weggeclippt":

```
PRINT "Suche die .bmap-Dateien..."

init:   '* Adressen der Datenstrukturen
        CLS
        fenster&  = WINDOW(7)
        rastport& = WINDOW(8)
        layer&    = PEEKL(rastport&)

        '* Derzeit gueltige Grenzen
        x0% = PEEKW(layer&+16)
        y0% = PEEKW(layer&+18)
        x1% = PEEKW(layer&+20)
        y1% = PEEKW(layer&+22)

        breite% = x1%-x0%
        hoehe%  = y1%-y0%

main:   '* demo
        LINE (x0%,y0%)-(x1%,y1%),2,bf

        '* Neue Grenzen setzen
        nx0% = x0%+.25*breite%
        nx1% = x1%-.25*breite%
        ny0% = y0%+.25*hoehe%
        ny1% = y1%-.25*hoehe%

        POKEW layer&+16,nx0%
        POKEW layer&+18,ny0%
        POKEW layer&+20,nx1%
        POKEW layer&+22,ny1%

        '* So sieht's aus:
        FOR test%=0 TO 40
          PRINT STRING$(50,"*")
        NEXT test%

        CLS
        PRINT "CONT eingeben!"

        STOP

        '* Alte Grenzen wiederherstellen

        POKEW layer&+16,x0%
        POKEW layer&+18,y0%
        POKEW layer&+20,x1%
        POKEW layer&+22,y1%

        END
```

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.

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 "CreateUpfrontLayer"-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%,s
crNr%
          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%,superHoehe%)
          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%,l
ayMinY%,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()
ScrollLayer()

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 Spei-

cherebenen 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%,s
crNr%
            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%,superHoehe%)
            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%,l
ayMinY%,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:

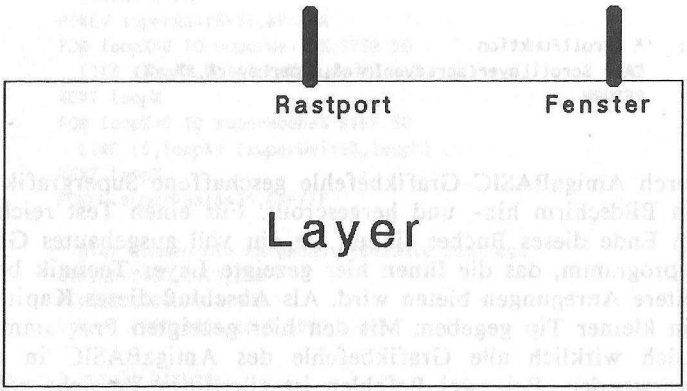
```
CALL SetRast(rastport&,farbe%)
```

rastport&: Adresse des Rastports Ihres Layers/Fensters
farbe%: Die Farbe, mit der Ihr Rastport ausgefüllt werden soll. Zum Löschen normalerweise =0.

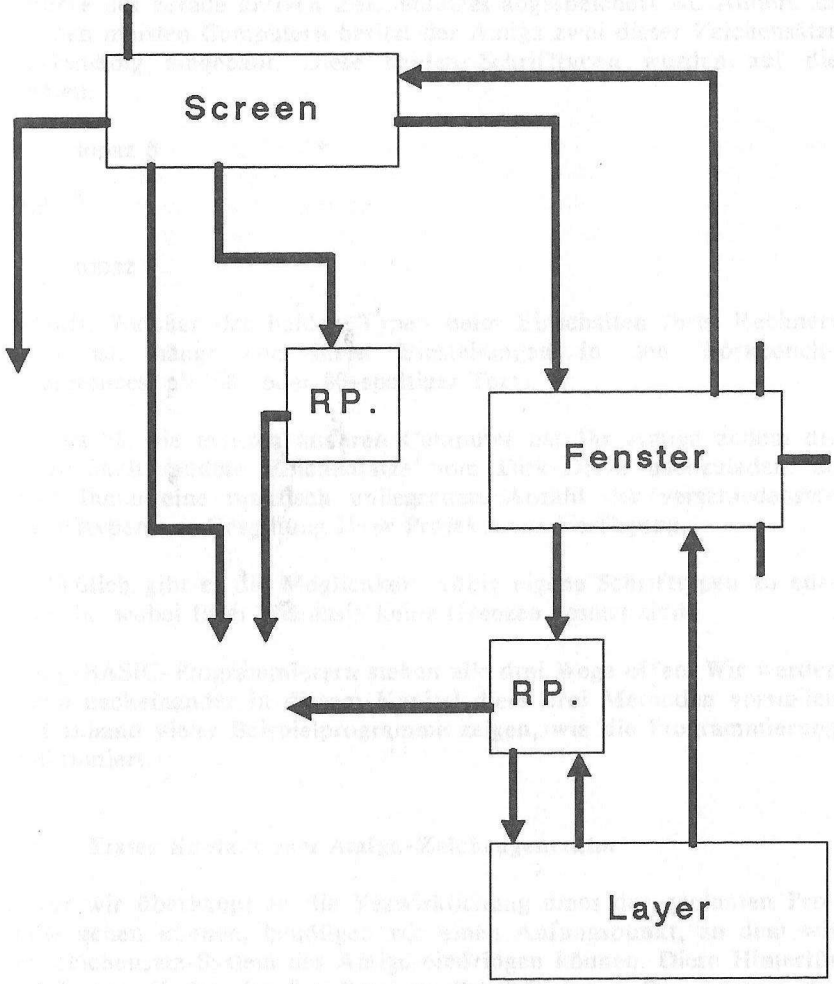
Wenn Sie Text in einen Teil des Layers drucken (LOCATE-Befehl), der unterhalb der unteren Kante Ihres Fensters liegt, dann scrollt AmigaBASIC unglücklicherweise das ehemalige Ausgabefenster, also den linken oberen Teil des Layers, um eine Zeile nach oben. Man umgeht dieses Problem, indem die in Kapitel 2 beschriebene Grafik-Funktion "Text" an Stelle von PRINT benutzt wird.

4.8 Layer im System

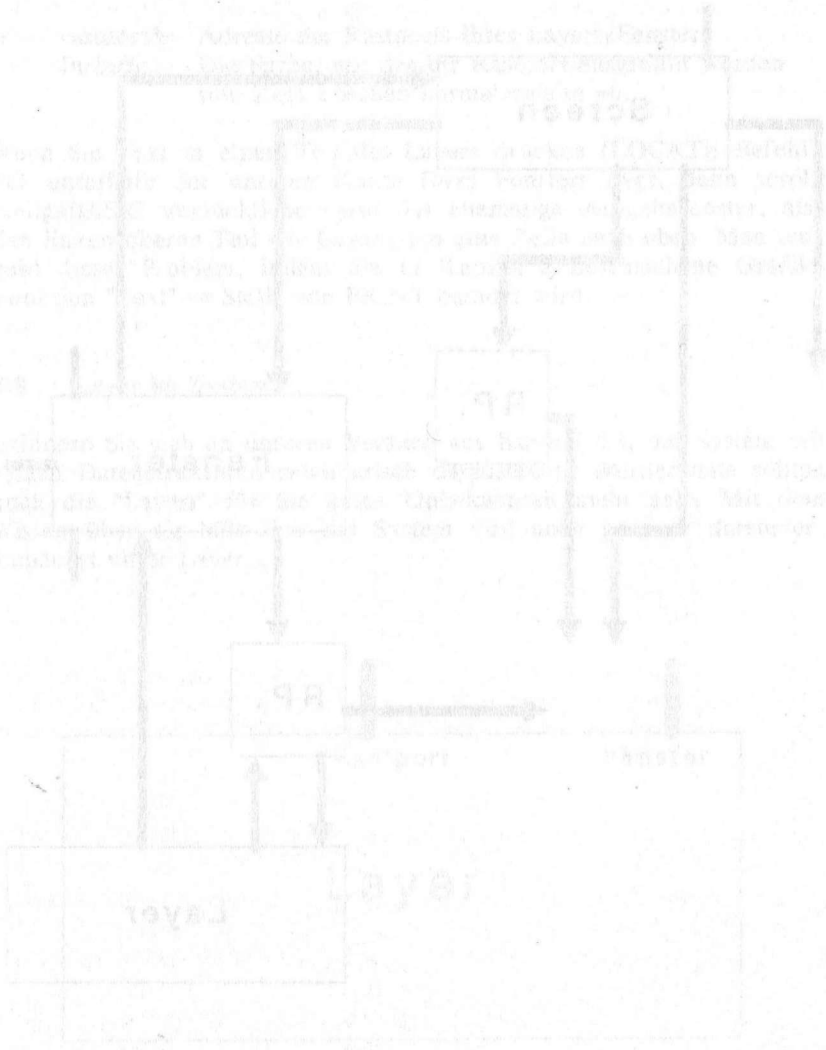
Erinnern Sie sich an unseren Versuch aus Kapitel 4.3, das System mit seinen Datenstrukturen zeichnerisch darzustellen? Mittlerweile sollten auch die "Layers" für Sie keine Unbekannten mehr sein. Mit dem Wissen über sie läßt sich das System nun noch genauer darstellen. Zunächst unser Layer...



...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)
```

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
+ 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 detail-

lierte 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 zeilenweise ab, d.h. jeweils eine Zeile aller Zeichenzeilen. Mit Hilfe des Modulos gelangt man an den Anfang der nächsten Zeile. Dazu später mehr.

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, wesentlich 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
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 wieder 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 ent-

nommen 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.

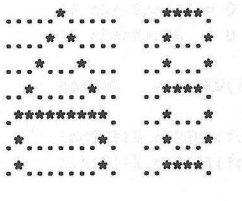
Nun bleibt ein letztes Problem: charSpace setzt die Breite eines Zeichens fest. Für das erste Zeichen sind das in unserem Beispiel 11 Punkte:

```
...........
...........
...........
...........
...........
...........
...........
```

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 Byteoffsets.

Bevor wir ins Detail gehen, zunächst das Programmlisting:

```
'#######################################
'#
'# Programm: Fixed-Width Zeichengenerator
'# 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:

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
+ 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. SchaffeZeichensatz 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%) S
TATIC

      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
```

Mögliche Fehlerquellen:

Es kommt kein Grafikausdruck zustande? Hier eine Checkliste:

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$(f.x%/8)
    pc%    = BltBitMap%(f.bitmap&,0,0,p.bmap&,0,0,f.x%,f.y%,200,255,SADD(temp
A$))
    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 unge-

stört 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
+ 012	Word	"transparente" Farbe
+ 014	Byte	X-Aspekt
+ 015	Byte	Y-Aspekt
+ 016	Word	Breite der Quellseite
+ 018	Word	Höhe der Quellseite

"CMAP" = ColorMap

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 Directo
ry."
      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%,co
de%+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 "ILBMende" 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 lassen 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%,s
crNr%
            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%,superHoehe%)
          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%-10-Xoffset%,loop%)
  ELSE
    LINE (windWeite%-Xoffset%,loop%)-(windWeite%-5-Xoffset%,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%
   IF 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%+r
cY%),swapper%
            ELSEIF draw%=3 OR draw%=6 THEN
                LINE (cX%+subox%,cY%+suboy%)-(cX%+subox%+rcX%,cY%+suboy%+r
cY%),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%+subo
y%+oldrcY%),swapper%
            ELSEIF draw%=3 OR draw%=6 THEN
                LINE (cX%+subox%,cY%+suboy%)-(cX%+subox%+oldrcX%,cY%+subo
y%+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$(menuItem-1))
        newFont& = OpenDiskFont&(VARPTR(textAttr&(textBase%)))
        IF newFont& = 0 THEN
          newFont& = OpenFont&(VARPTR(textAttr&(textBase%)))
        END IF
        IF newFont&<>0 THEN
          IF oldFont&<>0 THEN
            CALL CloseFont(oldFont&)
          END IF
          CALL SetFont(SuperRast&,newFont&)
          oldFont&   = newFont&
          fontHoehe% = INT(textAttr&(textBase%+1)/2^16)
        ELSE
          BEEP
        END IF
        modus$ = 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 gleichzeitiges 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 ein-

fach 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, 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!

9. Grafikprogrammierung in C

Wenn Sie schnelle Grafiken erzeugen wollen, sind Sie hier genau richtig. Sie werden sicher schon festgestellt haben, daß man in BASIC fantastische Grafiken erstellen kann. Aber die Geschwindigkeit, mit der diese erstellt werden, läßt sicher zu wünschen übrig. Deshalb stellt sich über kurz oder lang die Frage, ob es nicht angebrachter wäre, in einer maschinennäheren Sprache mit großen Geschwindigkeitsvorteilen zu arbeiten.

Die Hochsprache C bietet sich hier förmlich an. Geschwindigkeitsmerkmale, wie sie fast nur noch Maschinensprache bietet, und die Übersichtlichkeit der Programme, wie Sie sie von BASIC her kennen, machen C zu *der* Grafiksprache auf dem Amiga.

Um eine Redundanz zum ersten und zweiten Teil dieses Buches zu vermeiden, gehen wir meist nur noch auf die C-spezifischen Merkmale der Grafikprogrammierung ein.

Die hier vorgestellten Programme wurden mit dem Lattice V3.10 Compiler geschrieben. Beachten sollten Sie, daß der Systemstack auf 10240 Bytes erhöht wurde (CLI-Kommando: 'stack 10240'). Doch kommen wir nun zu den Voraussetzungen für die Grafikprogrammierung.

9.1 Die Unterprogramm-Bibliotheken

'Aller Anfang ist schwer' - leider gilt dies auch für den Einstieg in die Grafikprogrammierung in C. Doch wir wollen diese Hürden so diedrig wie möglich halten (Wohlgemerkt: Wir wollen Ihnen hier die Grafikprogrammierung in C, nicht die Programmierung der Sprache C selbst ans Herz legen. Sollten Sie also mit Begriffen wie 'Cast', 'struct' etc. nichts anfangen können, möchten wir Sie bitten, für das weitere Verständnis dieses Buches ein C-Einsteiger-Buch zu Rate zu ziehen Z.B. das Buch 'C für Einsteiger', das auch im Hause DATA BECKER erschienen ist.).

Doch nach diesem allgemeinen Vorgeplänkel wird es jetzt handfest. Bevor Sie die Grafikbefehle, die ja nicht zum Standard Befehlssatz der Sprache C gehören, benutzen können, müssen wir die sogenannten Librariss (zu deutsch: Bibliotheken) benutzen.

In diesen Librarys sind - vereinfacht gesagt - Sprungadressen auf die einzelnen Grafik-ROM-Routinen enthalten. Da, wie Sie vielleicht wissen, schon verschiedene ROM-Versionen (sprich Kickstarts) für den Amiga zu haben sind, hat sich dieses Library-Konzept bewährt. Da nämlich jedes Kickstart seine eigene Speicheraufteilung besitzt, weil die eine Routine hinzu gekommen ist, die andere jedoch fortgenommem wurde, wurde es notwendig, die Adressen der einzelnen Routinen für jede Kickstart-Version parat zu haben. Diese Adressen sind in den Libraries enthalten. So ist es möglich, daß ein Programm sowohl auf Kickstart 1.1 als auch auf Kickstart 1.2 läuft, obwohl alle Routinen nicht mehr an gleicher Stelle im Speicher stehen. Das Funktioniert natürlich nicht, wenn das Programm Routinen benutzt, die nur Kickstart 1.2 enthält.

Für unsere Belange ist in erster Linie die Graphics-Library (sprich: Grafik-Bibliothek) von Bedeutung (Für verschiedene Aufgabenbereiche existieren verschiedene Libraries). Geöffnet wird diese mit:

```
GfxBase = (struct GfxBase*)OpenLibrary("graphics.library",0)
```

GfxBase ist die Struktur, die alle nötigen Informationen der Library enthält. Die Funktion OpenLibrary liefert einen Zeiger auf solch eine vollständig initialisierte GfxBase-Struktur, die den Namen GfxBase haben muß, zurück. Der angegebene String "graphics.library" sorgt dafür, daß auch tatsächlich die Grafik-Bibliothek geöffnet wird (Wie Sie ja oben schon erfahren haben, existieren verschiedene Libraries, die mit OpenLibrary geöffnet werden können). Die Null in der Parameterliste für OpenLibrary gibt an, daß wir die gerade aktuelle Version der Grafik-Bibliothek öfnen wollen.

Doch kommen wir nun zurück zur Grafikprogrammierung. Zuerst müssen wir nämlich einen Bereich schaffen, in dem unsere Grafiken dargestellt und gespeichert werden können.

Dabei kann man eine ähnliche Hierarchie wie in einem größeren Betrieb erkennen:

9.2 Unser Chef: der View

Der View dürfte Ihnen schon aus den ersten beiden Teilen dieses Buches bekannt sein, doch wollen wir aufgrund seiner Wichtigkeit noch einmal auf ihn eingehen:

Er stellt nämlich die Verbindung zwischen unserem 'Kreativitätsbereich' und dem Computer her. Die Struktur, die alle nötigen View-Daten enthält, heißt schlicht und ergreifend 'struct View'. Mit 'struct View View' legen Sie also eine View-Struktur an. Da aber in den einzelnen Variablen dieser Struktur zufällige Werte stehen könnten, wollen wir diese erst einmal initialisieren: 'InitView (&View)' nimmt dies für uns vor.

9.3 Unser Abteilungsleiter: Der Viewport

Nun kommen wir dem Benutzer, also Ihnen, schon einen Schritt näher. Hier bestimmen Sie nämlich die Größe des Kreativitätsbereichs, den Darstellungsmodus und die Anzahl der darstellbaren Farben. Mit all diesen Daten, mit denen Sie den Viewport 'füttern', werden später die sogenannten Copper-Listen berechnet, die vom Spezialprozessor Copper, der für den Bildschirmaufbau zuständig ist, abgearbeitet werden.

Wie aber füttert man den Viewport?

Zuerst einmal wird die Position auf dem Bildschirm festgelegt. Dies geschieht allerdings relativ zum View, der seine Position wiederum aus den Variablen 'View.DxOffset' und 'View.DyOffset' erhält. Diese Variablen werden durch InitView allerdings so initialisiert, daß Ihr View so positioniert wird, wie mit den Preferences (s. das Rechteck in der Mitte des Preferences-Screen) eingestellt.

Geben Sie nun in den Variablen 'ViewPort.DxOffset' und 'ViewPort.DyOffset' jeweils die Werte 0 an, so wird die linke obere Ecke des Viewports exakt mit der des Views zusammenfallen. Geben Sie andere Werte (größer 0) an, so erscheint der Viewport mehr oder weniger in der Mitte des Bildschirms.

Die Größe des Viewports wird in 'ViewPort.DWidth' (Breite) und in 'ViewPort.DHeight' (Höhe) festgelegt. Dabei sollten Sie den Darstellungsmodus beachten. Wollen Sie nämlich 640 Punkte in der Horizontalen (ViewPort.Modes = HIRES) darstellen, müssen Sie in 'ViewPort.DWidth' 640 (anstatt 320 im Normalmodus) angeben. Ebenso wird bei 'ViewPort.Modes = LACE' (Interlaced-Modus) 400 anstatt 200 an 'ViewPort.DHeight' zugewiesen.

Für die Farbdarstellung wird eine Colormap (Farbkarte oder Farbpalette) benötigt. Da für die verschiedenen Farbeinträge (s. Kapitel 14) Speicherplätze benötigt werden, müssen wir uns diese zuweisen lassen.

'ColorMap = (struct ColorMap*) GetColorMap(Anzahl_der_Farben)' legt hier eine ganze Struktur an - die ColorMap-Struktur. Neben Zeigern auf die Farb-Speicherplätze wird hier z.B. auch die Anzahl der darzustellenden Farben des Viewports abgespeichert.

Diese Colormap müssen Sie, nachdem diese initialisiert wurde, dem Viewport zugänglich machen. Dies geschieht entweder, indem Sie den Zeiger auf die vollständig initialisierte Colormap dem Viewport wie folgt zuweisen:

```
ViewPort.ColorMap = ColorMap
```

oder direkt folgenden Aufruf ausführen:

```
ViewPort.ColorMap = (struct ColorMap *) GetColorMap(Anzahl_der_Farben)
```

Bitte beachten Sie, daß der Cast ('...(struct ColorMap*)') unbedingt notwendig ist, um lästige Warnings (Pointers do not point to same Object) zu vermeiden.

Sie können allerdings auch den Rückgabewert der Funktion GetColorMap (sowie den aller übrigen Funktionen) vor dem eigentlichen Programm festlegen:

```
extern struct ColorMap *GetColorMap()
...

main()
...
```

Dann können Sie 'ViewPort.ColorMap = GetColorMap (Anzahl_der_Farben)' aufrufen, ohne daß der Compiler eine Warnung 'ausspuckt'.

Damit in der ViewPort-Struktur keine zufälligen Werte stehen, die unter Umständen das ganze System durcheinanderbringen, gibt es auch für den ViewPort einen Initialisierungsbefehl:

```
InitVPort(&ViewPort)
```

Dieser sollte immer vor der Benutzung und der Zuweisung von Werten an den Viewport ausgeführt werden.

Bevor wir uns dann der Bitmap zuwenden können, müssen wir noch dafür sorgen, daß der View und der Viewport miteinander eine Verbindung eingehen können:

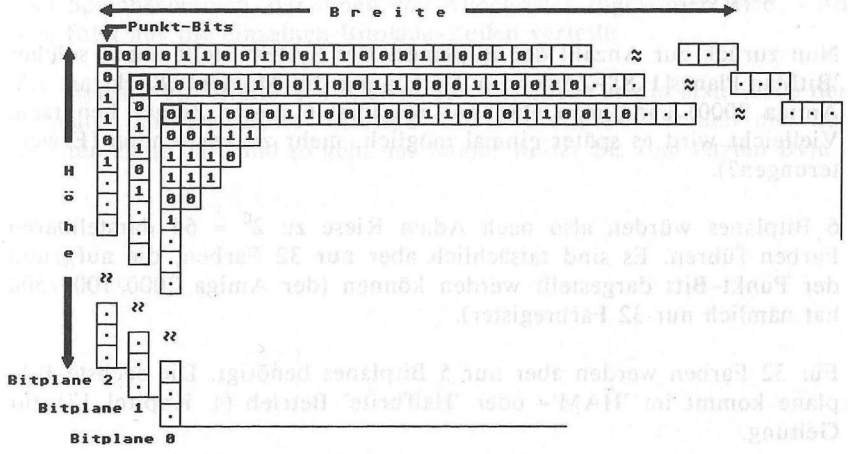
```
View.ViewPort = &ViewPort
```

9.4 Der Arbeiter: die Bitmap

Irgendwo und irgendwie muß die Grafik ja auch im Speicher stehen. Das Wie und Wo regelt die Bitmap. Sie ist in verschiedene BitPlanes (Bit-Lagen) unterteilt. Die Anzahl dieser Bitplanes ist es, die festlegt, wie viele Farben Sie in einem Viewport, der ja auch die Bitmap benutzt, dargestellt werden können. Dabei gilt: je mehr Bitplanes, desto mehr Farben. Die genaue Anzahl der Farben errechnet sich wie folgt:

```
Anzahl_der_Farben = 2^Anzahl_der_BitPlanes
```

Dies hängt damit zusammen, daß diese Bitplanes im grafischen Sinne übereinanderliegen (im Speicher liegen diese Speicherbereiche natürlich hintereinander):



Jeder Punkt wird durch ein Bit in jeder Bitplane repräsentiert. Je nachdem, in welchen Bitplanes das Punkt-Bit gesetzt oder gelöscht ist, kommt die Nummer des Farbregisters (s. Kapitel 14) zustande, mit dessen Farbe der Punkt letztendlich dargestellt wird.

Die Speicherplätze (Bytes) für die einzelnen Bitplanes läßt man sich mit dem AllocRaster-Befehl, einer Unterart des AllocMem-Befehls, zuweisen. Der AllocRaster-Befehl sorgt dafür, daß der zugewiesene Speicherbereich an einer Word-Adresse beginnt. Dies ist deshalb nötig, da das System nur auf gerade, also Word-Adressen zugreifen kann.

Mit 'BitMap.Planes[i] = AllocRaster (Breite,Höhe)' lassen Sie sich den erforderlichen Speicher für eine Bitplane der Bitmap, die 'Höhe' Zeilen hoch und 'Breite' Punkte breit ist, zuweisen. Bitte beachten Sie, daß der Zeiger 'BitMap.Planes[i]' niemals gleich 0 sein darf. Ist dies nämlich der Fall, so konnte nicht mehr genügend Speicher bereitgestellt werden. (Ähnlich ist es auch, wenn die Routinen OpenLibrary oder GetColorMap den Wert 0 liefern. Meist liegt in solchen Fällen eine akute Speicherarmut vor. In solchen Fällen sollten Sie darauf achten, daß das Programm (neben dem CLI oder der Workbench) das einzige Programm ist, das sich in Ihrem Amiga befindet. Testen Sie also stets den Rückgabewert der Funktion, und lassen Sie bei 0 eine passende Meldung ausgeben, damit jeder weiß, welcher Speicher nicht belegt werden konnte. Z.B.:

```
printf (" Kein Speicher mehr für ColorMap vorhanden\n");
exit (0);
```

Nun zurück zur Anzahl der Bitplanes. Das System stellt uns 8 solcher 'BitMap.Planes[1..8]'-Zeiger zur Verfügung. Im Moment (Kickstart 1.2, Amiga 2000) kann man aber nur maximal 6 dieser Zeiger benutzen. Vielleicht wird es später einmal möglich, mehr zu verwenden (Erweiterungen?).

6 Bitplanes würden also nach Adam Riese zu $2^6 = 64$ darstellbaren Farben führen. Es sind tatsächlich aber nur 32 Farben, die aufgrund der Punkt-Bits dargestellt werden können (der Amiga 2000/1000/500 hat nämlich nur 32 Farbregister).

Für 32 Farben werden aber nur 5 Bitplanes benötigt. Die sechste Bitplane kommt im 'HAM'- oder 'Halfbrite'-Betrieb (s. Kapitel 17) zur Geltung.

Leider müssen wir noch auf eine weitere Einschränkung zu sprechen kommen: Im Hi-Res-Betrieb, also bei 640 Punkten Auflösung in X-Richtung, kann man nur mit 4 Bitplanes arbeiten. Dies liegt daran, daß nun ungleich mehr Daten auf den Bildschirm gebracht werden müssen. Dazu steht dem Amiga aber nur ein begrenzter Zeitraum zur Verfügung, der mit 4 Bitplanes voll und ganz ausgeschöpft wird.

Bevor wir nun wieder zu den Bitmaps kommen, wollen wir Ihnen noch zeigen, wie die Bitplanes bzw. der Speicher, den sie belegen, initialisiert wird.

Da nämlich keineswegs garantiert ist, daß alle Bytes des zugewiesenen Speichers gelöscht sind, müssen wir das selbst besorgen (sonst wäre später eine buntes Wirrwarr auf Ihrem Bildschirm die Folge).

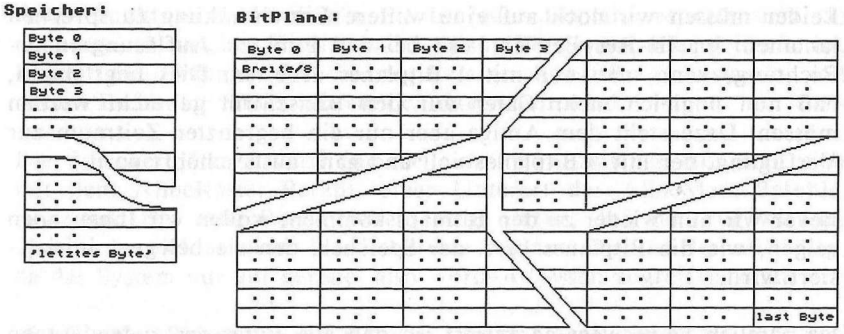
Mit 'BltClear (BitMap.Planes[i], Anzahl_der_Bytes,Flags)' wird der Speicher gelöscht. Die Anzahl der Bytes einer Bitplane, die gelöscht werden sollen, errechnet sich wie folgt:

```
Anzahl_der_Bytes = Breite*Höhe/8
```

Das Macro 'RASSIZE(Breite,Höhe)' errechnet den gleichen Wert (Für den Parameter Flags möchten wir Sie bitten, im Anhang B nachzuschlagen. Normalerweise wird 'Flags' aber gleich 0 gesetzt).

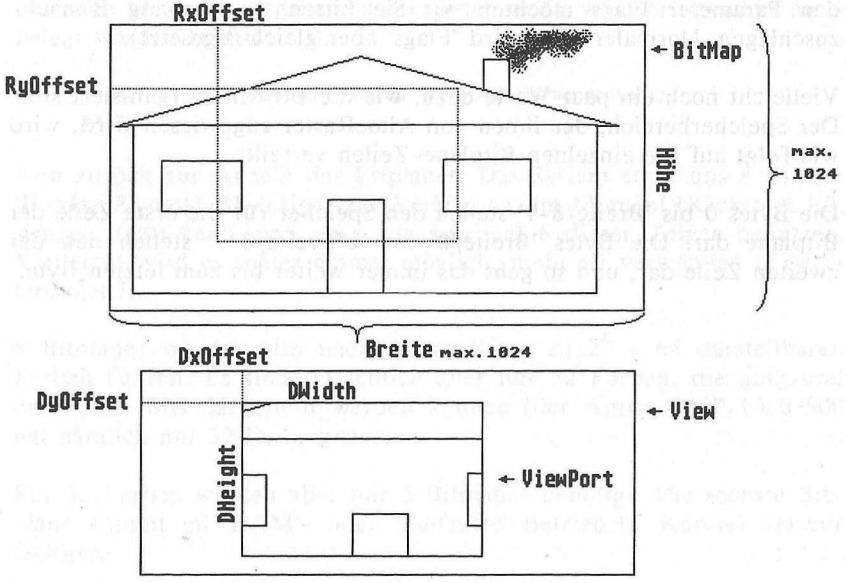
Vielleicht noch ein paar Worte dazu, wie die Bitplanes organisiert sind: Der Speicherbereich, der ihnen von AllocRaster zugewiesen wird, wird wie folgt auf die einzelnen Bitplane-Zeilen verteilt:

Die Bytes 0 bis 'Breite/8-1' stellen den Speicher für die erste Zeile der Bitplane dar. Die Bytes 'Breite/8' bis '2*Breite/8-1' stellen den der zweiten Zeile dar, und so geht das immer weiter bis zum letzten Byte.



Doch nachdem wir uns so intensiv den Bitplanes zugewandt haben, wollen wir uns nun wieder mit der Bitmap (dem 'Betriebsrat' der Bitplanes) beschäftigen:

In der BitMap-Struktur wird nämlich die Größe der Bitmap und die Tiefe (Anzahl der Bitplanes) festgelegt. Die Größe der Bitmap und die des Viewports können sich dabei unterscheiden:



Zu beachten ist nur, daß die Bitmap-Größe auf 1024 * 1024 Zeilen/Punkte beschränkt ist.

Initialisiert wird die Bitmap mit 'InitBitMap (&BitMap, Tiefe ,Breite, Höhe)'.

9.5 Der 'Laufbursche' RasInfo

Nun stehen Bitmap und Viewüort noch ziemlich isoliert nebeneinander. Die Verbindung zwischen beiden wird erst durch die RasInfo-Struktur geschaffen. Diese wird wie die vorigen Strukturen nicht durch eine 'Init...'-Routine für den weiteren Gebrauch präpariert, sondern muß vom Programmierer selbst Element für Element mit sinnvollen Werten belegt werden.

Die Verbindung zwischen Viewport und Bitmap wird mittels der RasInfo-Struktur wie folgt hergestellt:

```
ViewPort.RasInfo = &RasInfo
RasInfo.BitMap   = &BitMap
```

Der Bitmap-Zeiger der RasInfo-Struktur (Raster-Information) ist auch schon fast deren einziges Element. Nur die beiden Variablen RxOffset und RyOffset, die den Punkt angeben, der mit der linken oberen Ecke des Viewports zusammenfallen soll (s. vorige Abbildung), müssen noch initialisiert werden - meistens mit dem Wert 0.

Die letzte RasInfo-Variable - der 'RasInfo.Next'-Zeiger kommt erst mit einem speziellen Darstellungsmodus (s. Kapitel 17) zur Geltung, wird aber meistens gleich 0 gesetzt.

9.6 Unser 'Arbeitstier': der Rastport

Den Rastport bzw die RastPort-Struktur kann man schon fast als Schwerarbeiter bezeichnen. Über ihn laufen (fast) alle Grafik-Ausgaben, denn er enthält die aktuelle Farbe der Punkte, den Zeichenmodus (s. Kapitel 11) und vieles mehr (s. Anhang A).

Er wird - wie üblich - mit 'InitRastPort (&RastPort)' initialisiert. Bei solch einer Initialisierung werden immer gewisse Anfangszustände her-

gestellt, die dann später nach Bedarf und durch geeignete Befehle nach Belieben verändert werden können.

Nach der 'Installation' des Rastports, braucht man ihm nur noch die Adresse der Bitmap, in der ja letztendlich die Grafikbefehle eine sichtbare Wirkung haben, anzugeben:

```
RastPort.BitMap = &BitMap
```

Jetzt sind alle nötigen Strukturen initialisiert und miteinander verkettet. (Der Rastport braucht nicht mit dem Viewport verbunden zu werden, da er nur Informationen für die Grafikbefehle liefert.)

Wir können nun daran gehen, die Darstellung auf dem Bildschirm zu erzeugen - denn bis jetzt sieht man noch nichts.

Dies liegt daran, daß die Copper-Listen, die ja spezielle Programme darstellen, die vom Co-Prozessor Copper abgearbeitet werden, noch nicht erstellt wurden (In diesen Copper-Listen werden besonders die Hardware-Register beeinflußt, die für die Grafikdarstellung benötigt werden (s. Anhang C, Register 'bltcon0/1', 'bplcon0/1/2' etc.).

Diese Copper-Listen werden aufgrund der Informationen, die der initialisierte View, Viewport etc. enthalten mit den Befehlen 'MakeVPort (&View,&ViewPort)' und 'MrgCop (&View)' erstellt.

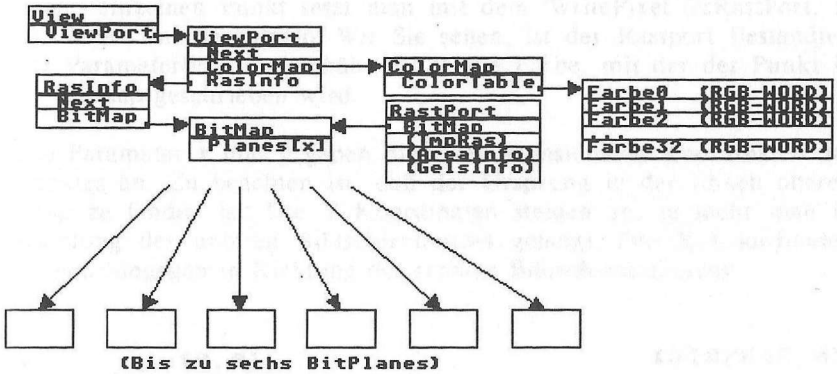
Im ersten Fall (MakeVPort) wird für jeden Viewport (es können durchaus mehrere Viewports in einem View existieren, zu beachten ist nur, daß mindestens ein Unterschied von einer (Raster-)Zeile zwei übereinanderliegenden Viewports bleibt) eine sogenannte Zwischen-Copper-Liste angelegt. Damit alleine kann der Copper aber noch nichts anfangen. Erst wenn alle Viewport-Copper-Listen erstellt und mit MrgCop daraus eine für den Copper verständliche Liste entstanden ist, kann man 'LoadView (&View)'aufrufen.

Nun wird die Copper-Liste vom Copper abgearbeitet, und man kann auf dem Bildschirm der Mühe Lohn erkennen. Vorher sollte man allerdings die aktuelle Copper-Liste bzw. den aktuellen View, der ja die abzuarbeitenden Copper-Listen enthält, in einer Variablen retten (oldview = GfxBase->ActiView). Dabei hilf uns die GfxBase-Struktur, die immer einen Zeiger auf den aktuellen View für uns bereithält.

Vergessen Sie einmal, den alten View zu retten, und vor Ablauf des Programms 'LoadView (oldview)' aufzurufen, sind Sie meistens

rettungslos im Amiga-Dschungel verloren. Da die meisten Programme nämlich vom CLI oder von der Workbench aus gestartet werden, haben Sie keine Möglichkeit mehr, zum Workbench-Screen, in dem auch das CLI-Window enthalten ist, zurückzukehren.

Aus folgender Abbildung können Sie noch einmal entnehmen, wie View, Viewport, Rastport, Bitmap, RasInfo etc. zusammenhängen:



9.7 'Feierabend...

...das läuft wie Honig runter' hieß es in einem bekannten Hit von Peter Alexander. In unseren Programmen artet der Feierabend allerdings noch einmal in Arbeit aus. Erstens muß der belegte Speicher wieder an das System zurückgegeben werden, damit dieses wirtschaftlich damit haushalten kann, und zweitens müssen alle geöffneten Libraries wieder geschlossen werden. Letzteres geschieht mit 'CloseLibrary (Base-Pointer)'. Für die Grafik-Bibliothek muß dieser Base-Pointer 'GfxBase' heißen.

Bei der Rückgabe der Speicherplätze muß man beachten, daß auch die Copper-Listen, die ja mit MakeVPort und MrgCop erzeugt wurden, Speicherplätze belegen.

Die Viewport-Zwischenlisten werden mit 'FreeVPortCopLists (&ViewPort)' freigegeben, wohingegen 'FreeCprList (View.LOFCprList)' und 'FreeCprList (View.SHFCprList)' die für den Copper verständlichen Copper-Listen freigeben ('View.LOF/SHFCprList s. Anhang A).

Den Speicher für jede Bitplane gibt man mit 'FreeRaster (BitMap.Planes[i], Breite, Höhe)' wieder zurück.

Die belegten Speicherplätze für die Colormap werden schließlich mit 'FreeColorMap (&ColorMap)' bzw. 'FreeColorMap (ViewPort.ColorMap)' wieder freigegeben.

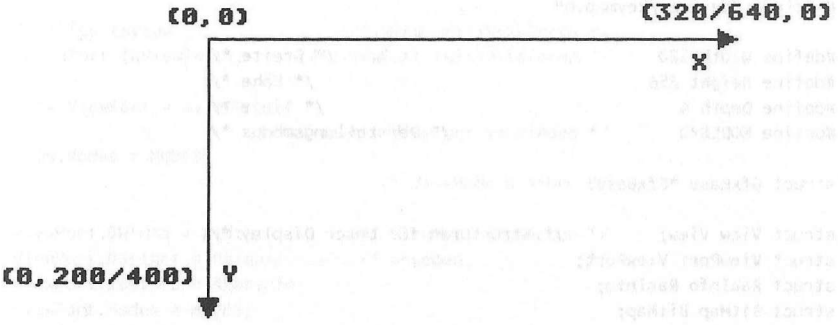
10. Linien und Punkte

Nun kommen wir zu den ersten Grafikbefehlen. Als kleinstes Element einer jeden Computergrafik kann der einzelne Punkt oder das Pixel angesehen werden.

10.1 Gepunktet wird mit WritePixel

Einen einzelnen Punkt setzt man mit dem 'WritePixel (&RastPort, x, y)'-Befehl in die Bitmap. Wie Sie sehen, ist der Rastport Bestandteil der Parameterliste. Er enthält ja z.B. die Farbe, mit der der Punkt in die Bitmap geschrieben wird.

Die Parameter x und y geben die zweidimensionalen Koordinaten des Punktes an. Zu beachten ist, daß der Ursprung in der linken oberen Ecke zu finden ist. Die Y-Koordinaten steigen an, je mehr man in Richtung des unteren Bildschirmrandes gelangt. Die X-Koordinaten steigen hingegen in Richtung des rechten Bildschirmrahmens:



Folgendes Programm öffnet nun einen View, Viewport, etc. und erzeugt mittels WritePixel eine Punkte-Schlange. Jeder Punkt dieser Schlange wird in einer anderen Farbe gezeichnet. Nach dem Zeichnen werden dann die Inhalte der Farbregister mittels der sogenannten ColorMap-Befehle (s. Kapitel 14) rotiert. Dies erzeugt den von Grafikkraft oder Deluxe Paint her bekannten Color-Cycle:

```
/*********************************************************/
/*                    Schlange.c                         */
/*                                                       */
/* Dieses Programm erzeugt mittels WritePixel() eine     */
/* Schlange, die farblich animiert wird.                 */
/*                                                       */
/* Compiled with: Lattice V3.10                          */
/*                                                       */
/* (c) Bruno Jennrich                                    */
/*********************************************************/

#include "exec/types.h"
#include "exec/memory.h"
#include "exec/devices.h"
#include "graphics/gfx.h"
#include "graphics/gfxbase.h"
#include "graphics/gfxmacros.h"
#include "graphics/text.h"
#include "graphics/view.h"
#include "graphics/clip.h"
#include "graphics/copper.h"
#include "graphics/gels.h"
#include "graphics/regions.h"
#include "hardware/blit.h"
#include "devices/keymap.h"

#define Width 320                              /* Breite */
#define Height 256                              /* Höhe */
#define Depth 4                                 /* Tiefe */
#define MODES 0                        /* Darstellungsmodus */

struct GfxBase *GfxBase;

struct View View;          /* Strukturen für unser Display */
struct ViewPort ViewPort;
struct RasInfo RasInfo;
struct BitMap BitMap;
struct RastPort RastPort;
struct View *oldView;

int i,x,y,faktor;

UWORD color = 0;                              /* Farbzähler */

USHORT Colors[16] = {
                    0x000,0x00f,0x0f0,0xf00,
                    0x0ff,0xff0,0xf0f,0xc34,
                    0x646,0x782,0xd23,0x5a9,
```

```
                    0x560,0xacf,0xedf,0xa09
                };
                                /* Initial-Farbtabelle */

char *LeftMouse = (char *)0xbfe001;

VOID Color();                   /* Vorwärtsdeklaration */
VOID Color_Cycle();

/*********************************************************/
/* Es geht los !                                         */
/*********************************************************/

main()
{

   if((GfxBase = (struct GfxBase *)
      OpenLibrary("graphics.library",0)) == NULL)
      {
         printf(" Keine Graphics !!!!!");
         Exit (0);
      }
                                /* Display erstellen */
   oldView = GfxBase->ActiView;    /* rette alten View */

   InitView (&View);            /* View initialisieren */
   InitVPort (&ViewPort);     /* ViewPort initialisieren */

   View.ViewPort = &ViewPort;
                          /* View mit ViewPort verbinden */
   View.Modes = MODES;
                                /* ViewMode angeben */

   ViewPort.DWidth = Width;       /* ViewPort-Parameter */
   ViewPort.DHeight = Height;     /* angeben            */
   ViewPort.RasInfo = &RasInfo;
   ViewPort.Modes = MODES;
   ViewPort.ColorMap = (struct ColorMap *)GetColorMap (16);
   if (ViewPort.ColorMap == 0) goto cleanup1;
                                /* ColorMap anlegen */

   RasInfo.Next = NULL;     /* Verbindung ViewPort-BitMap */
   RasInfo.RxOffset = 0;    /* über RasInfo herstellen    */
   RasInfo.RyOffset = 0;
   RasInfo.BitMap = &BitMap;

   InitBitMap (&BitMap, Depth, Width, Height);
                             /* BitMap initialisieren */
```

```
for (i=0; i<Depth; i++)
   {
      BitMap.Planes[i] = (PLANEPTR)
          AllocRaster (Width,Height);
                                  /* Speicher anfordern */
      if (BitMap.Planes[i] == 0)
         {
            printf ("No Space for BitPlanes\n");
            goto cleanup1;
         }
      else
      BltClear (BitMap.Planes[i],
            RASSIZE (Width,Height),0);
                                  /* BitPlanes löschen */
   }

InitRastPort (&RastPort);   /* RastPort initialisieren */

RastPort.BitMap = &BitMap;
                        /* RastPort-BitMap Verbindung */


LoadRGB4(&ViewPort,&Colors[0],16);
                        /* ColorMap des ViewPorts mit */
                        /* eigenen Farben beschreiben.*/
                        /* Danach MUß MakeVPort(),    */
                        /* MrgCop() und LoadView()    */
                        /* (Intuition:RemakeDisplay())*/
                        /* erfolgen!                  */

MakeVPort (&View, &ViewPort);
MrgCop (&View);
LoadView (&View);

color = 1;                          /* Farbanimation */

x = 0;
y = 0;
faktor = 1;

while (y < Height-1)
   {
      SetAPen (&RastPort,color);
      WritePixel (&RastPort,x,y);

      x += (faktor);
      if ((x >= Width-1) || (x == 0))
         {
            for (i=0; i<7; i++)
               {
```

```
                    Color();
                    SetAPen (&RastPort,color);
                    WritePixel (&RastPort,x,y+i);
                 }
              faktor *= -1;
              y += 6;
           }
        Color();
     }


  while ((*LeftMouse & 0x40) == 0x40)
     {
        Color_Cycle();
     }

           FreeColorMap (ViewPort.ColorMap);    /* Gib   */
           FreeVPortCopLists (&ViewPort);       /* alles */
           FreeCprList (View.LOFCprList);       /* frei  */
           FreeCprList (View.SHFCprList);

  cleanup1: for (i=0; i<Depth; i++)
              {
                 if (BitMap.Planes[i] != NULL)
                 FreeRaster (BitMap.Planes[i],
                             Width,Height);
              }

           LoadView (oldView);
           CloseLibrary(GfxBase);
           return (0);
}

/*********************************************************/
/* Diese Routine sorgt dafür, daß beim Zeichnen der Farb- */
/* zähler hochgezählt wird.                               */
/*-------------------------------------------------------*/
/* Eingabe-Parameter: keine                               */
/*-------------------------------------------------------*/
/* Rückgabe-Werte: keine                                  */
/*********************************************************/

VOID Color()
{
  color++;
  color &= 15;
  if (color == 0) color=1;
}
```

```
/*********************************************************/
/* Diese Routine sorgt dafür, daß die Farben der View-   */
/* Port Colormap rotiert werden                          */
/*-------------------------------------------------------*/
/* Eingabe-Parameter: keine                              */
/*-------------------------------------------------------*/
/* Rückgabe-Werte: keine                                 */
/*********************************************************/

VOID Color_Cycle()
{
   UWORD i,hilf;
   static UWORD Cols[16];

   Cols[0] = Colors[0];       /* Hintergrund-Farbe bleibt */

   hilf = GetRGB4 (ViewPort.ColorMap,15);
   for (i=2; i<16; i++)
      Cols[i] = GetRGB4(ViewPort.ColorMap,i-1);
   Cols[1] = hilf;
                                        /* Umschauffeln */
   LoadRGB4(&ViewPort,&Cols[0],16);
   MakeVPort (&View,&ViewPort);          /* Wichtig !!! */
   MrgCop (&View);
   LoadView (&View);
}
```

10.2 Linien ziehen mit Move und Draw

Natürlich kann der Amiga auch Linien ziehen. Eine Besonderheit dabei ist allerdings, daß dabei weitgehend auf den 68000er verzichtet wird. Fast ausschließlich der Blitter (ein weiterer Spezialprozessor des Amiga) wird damit beauftragt.

Dazu wird zuerst die Anfangskoordinate festgelegt. Das Ziehen von Linien geschieht nämlich nicht, wie von BASIC her gewohnt mit einem Befehl, sondern mit zweien.

Der Move-Befehl setzt die aktuelle Grafik-Position (Grafik-Cursor) auf die angegebene Koordinate:

```
Move (&RastPort,x,y)
```

Diese Koordinate wird dabei auch wieder im angegebenen RastPort abgespeichert (RastPort.cp_x/RastPort.cp_y).

Der nun folgende 'Draw (&RastPort,x,y)'-Befehl zeichnet eine Linie zum hier angegebenen Punkt. Die Koordinate dieses Punktes wird dann automatisch zur neuen Grafik-Cursor-Position auserkoren (Ein weiterer Draw-Befehl - ohne zwischenzeitliches Move - hätte zur Folge, daß eine Linie vom letzten Punkt der ersten Linie zum neu angegebenen Punkt gezogen wird.).

Folgendes Programm macht deutlich, was im Amiga steckt:

```
/*********************************************************/
/*                    Quix.c                             */
/*                                                       */
/* Dieses Programm zeigt, was Geschwindigkeitsmäßig im   */
/* AMIGA, insbesondere beim Linienziehen, steckt.        */
/*                                                       */
/* Compiled with: Lattice V3.10                          */
/*                                                       */
/* (c) Buno Jennrich                                     */
/*********************************************************/

#include "exec/types.h"
#include "graphics/gfx.h"
#include "graphics/rastport.h"
#include "graphics/copper.h"
#include "graphics/view.h"
#include "graphics/gels.h"
#include "graphics/regions.h"
#include "graphics/clip.h"
#include "graphics/text.h"
#include "graphics/gfxbase.h"
#include "hardware/blit.h"

#define DEPTH 2                          /* 2 BitPlanes */
#define WIDTH 320                     /* 320 x 256 Punkte */
#define HEIGHT 256

#define MAX_X WIDTH - 2  /* Rand links/rechts/unten nicht */
#define MAX_Y HEIGHT - 2 /* überschreiben                 */
#define MIN_X 1
#define MIN_Y 10          /* Schrift nicht überschreiben */

#define NOT_ENOUGH_MEMORY -1000
#define MAX_LINES 30                /* Maximal 30 Linien */

struct View View;            /* Unsere eigenen Strukturen */
struct ViewPort ViewPort;
struct RasInfo RasInfo; struct ColorMap *ColorMap;
struct BitMap BitMap;
struct RastPort RastPort;
```

```
SHORT i,j,
      Length;

struct GfxBase *GfxBase;
struct View *oldview; /* Hier wird der alte View gerettet */

USHORT colortable[] = {0x000,0xf00,0x00f,0x0f0};
                          /* Unsrere eigene Farbpalette */

char *QuixString = "Quix - Lines *** (C) BHJ";
char *LeftMouse;                      /* Für CIA - Adresse */

VOID draw();                         /* Vorwärtsdeklaration */
VOID FreeMemory();

/*********************************************************/
/* Es geht los !                                         */
/*********************************************************/

main()
{

   if ((GfxBase = (struct GfxBase *)
      OpenLibrary("graphics.library",0)) == NULL)
      Exit (1);

   oldview = GfxBase->ActiView;       /* Rette alten View */

   InitView(&View);      /* Initialisiere View & ViewPort */
   InitVPort(&ViewPort);
   View.ViewPort=&ViewPort; /* Verbinde beide miteinander */

   InitBitMap(&BitMap,DEPTH,WIDTH,HEIGHT);
                           /* BitMap initialisieren */
   InitRastPort(&RastPort);      /* RastPort     "        */
   RastPort.BitMap = &BitMap;    /* RastPort -> BitMap    */

   RasInfo.BitMap = &BitMap;         /* RasInfo für BitMap */
   RasInfo.RxOffset = 0;
   RasInfo.RyOffset = 0;
   RasInfo.Next = NULL;

   ViewPort.RasInfo = &RasInfo;    /* ViewPort -> RasInfo */

   ViewPort.DWidth = WIDTH;    /* Wie groß ist ViewPort ? */
   ViewPort.DHeight = HEIGHT;
```

```
   ViewPort.ColorMap = (struct ColorMap *)GetColorMap(4);
                                  /* ViewPort's ColorMap */
   LoadRGB4 (&ViewPort,&colortable[0],4);
                        /* Lade ColorMap mit Farbwerten */
   for (i=0; i<DEPTH; i++)
      {
         if ((BitMap.Planes[i] = (PLANEPTR)
            AllocRaster(WIDTH,HEIGHT)) == NULL)
               Exit(NOT_ENOUGH_MEMORY);
                    /* Speicher für BitPlanes reservieren */

         BltClear ((UBYTE *)BitMap.Planes[i],
            RASSIZE(WIDTH,HEIGHT),0);
                          /* BitPlane Speicher löschen */
      }

   MakeVPort(&View,&ViewPort);
                           /* CopperListe für ViewPort */
   MrgCop(&View);
                        /* "Große" CopperListe: für View */

   LoadView(&View);             /* Schalte neues Display an */

   LeftMouse = (char *)0xbfe001;
                           /* CIA Adresse für I/O Ports */
                           /* Wird für linken Mausknopf */
                           /* gebraucht                 */

   draw();                                 /* eigene Routine */

   LoadView(oldview);             /* WorkBench - Display an */
   FreeMemory();                     /* Alles zurückgeben */
   return(0);
}

/*********************************************************/
/* Diese Funktion übernimmt das Zeichnen der Lininen,    */
/* sowie die linke Mausknopfabfrage                      */
/*-------------------------------------------------------*/
/* Eingabe-Parameter: keine                              */
/*-------------------------------------------------------*/
/* Rückgabe-Werte: keine                                 */
/*********************************************************/

VOID draw()
{
   static SHORT i,
          new = 0,
          old = 0,
          full = FALSE,
```

```
        min, temp,
        delay = 1999,
        direction = 1,
        k = 0;

struct {                    /* Koordinaten für eine Linie */
         int x1[MAX_LINES],
             x2[MAX_LINES],
             y1[MAX_LINES],
             y2[MAX_LINES];
       } line;

static int veloc[4] = {-4,5,3,-7};
            /* Was wird zu den Koordinaten addiert ? */

static int start[4] = {110,25,160,74};
             /* Wo wird die erste Linie gezeichnet ? */

static int max[4] = {MAX_X,MAX_Y,MAX_X,MAX_Y};
             /* Erlaubte maximalwerte der Koordinaten */

SetDrMd(&RastPort,JAM1);
SetAPen(&RastPort,3);              /* Zeichenfarbe = Grün */

Length = WIDTH/2-TextLength (&RastPort,
          QuixString,strlen(QuixString))/2;
                        /* Berechne 'Zentralposition' */

Move(&RastPort,Length,RastPort.TxBaseline);
                        /* Grafik-Cursor Positionieren */

Text(&RastPort,QuixString,strlen(QuixString));
                                    /* Text ausgeben */
SetAPen(&RastPort,1);              /* Zeichenfarbe = Rot */

Move(&RastPort,0,9);                /* Rahmen zeichnen */
Draw(&RastPort,WIDTH-1,9);
Draw(&RastPort,WIDTH-1,HEIGHT-1);
Draw(&RastPort,0,HEIGHT-1);
Draw(&RastPort,0,9);


while ((*LeftMouse & 0x40) == 0x40)     /* Maustaste ? */
  {
     for (i=0;i<4;i++)  /* Berechne neue Linienkoords */
        {
           temp = start[i] + veloc[i];
                 /* Ausgangswert + "Geschwindigkeit" */
```

```
    if (temp >= max[i])
                        /* über obere Grenze ? */
       {
          temp = max[i]*2 - start[i] - veloc[i];
          veloc[i] = -veloc[i];
       }

    if (i==1 | i==3) min = MIN_Y;
                    /* Y-Koordinaten betrachten */

    else min = MIN_X;

    if (temp < min)
       {
             /* untere Grenze überschritten ? */
          if (temp < 0) temp = -temp;
          else temp = min;
          veloc[i] = -veloc[i];
       }
    start[i] = temp;
}

if (full == TRUE)
   {
                    /* Letzte Linie löschen */
      SetAPen(&RastPort,0);
      Move(&RastPort,line.x1[old],
                     line.y1[old]);
      Draw(&RastPort,line.x2[old],
      line.y2[old]);

      old++;
      old %= MAX_LINES;
   }

line.x1[new] = start[0]; /* neue Linienkoords */
line.y1[new] = start[1]; /* festlegen         */
line.x2[new] = start[2];
line.y2[new] = start[3];

if (new == MAX_LINES-1) full = TRUE;
            /* MAX_LINES Linien gezeichnet ? */

SetAPen(&RastPort,2); /* Zeichenfarbe =  Blau */

Move(&RastPort,line.x1[new],line.y1[new]);
Draw(&RastPort,line.x2[new],line.y2[new]);
                     /* Zeichne neue Linie */
new++;
new %= MAX_LINES;
```

```
            if (delay != 0)
               {
                  for (i=0; i<delay; i ++);
                                          /* Wart' n' bischen */
                  delay += direction;
                  if (delay == 2000) direction = -1;
               }
            else
               {
                  k++;
                  if (k == 1000) /* 1000 mal 'full power' */
                     {
                        direction = 1;
                        delay += direction;
                        k = 0;
                     }
               }
         }
}

/*********************************************************/
/* Diese Funktion gibt den belegten Speicher für BitMaps  */
/* Copper-Listen etc. zurück.                             */
/*-------------------------------------------------------*/
/* Eingabe-Parameter: keine                               */
/*-------------------------------------------------------*/
/* Rückgabe-Werte: keine                                  */
/*********************************************************/

VOID FreeMemory()
{
   for (i=0;i<DEPTH;i++)
      FreeRaster(BitMap.Planes[i],WIDTH,HEIGHT);
                        /* BitPlane Speicher zurückgeben */
   FreeColorMap(ViewPort.ColorMap);
                        /* ColorMap Speicher, der mit */
                        /* GetColorMap() belegt wurde */
                        /* wieder freigeben           */
   FreeVPortCopLists(&ViewPort);
                              /* Speicher für ViewPort */
                              /* CopperListe           */
                              /* v. MakeVPort() belegt */
   FreeCprList(View.LOFCprList);
   FreeCprList(View.SHFCprList);
                                 /* View CopperListen */
   CloseLibrary(GfxBase);
                                 /* Schliesse Library */
}
```

Beeindruckend, nicht wahr?

Doch noch zwei kleine Tips: Fast alle unsere Programme lassen sich auf Maustastendruck beenden. Dies wurde dadurch erreicht, daß wir die zuständigen Hardware-Register abgefragt haben. Vielleicht können Sie dies gut in Ihren eigenen Programmen verwenden.

Sollte Ihnen das aktuelle Linienmuster (durchgezogene Linie) nicht gefallen, haben Sie die Möglichkeit, mit 'SetDrPt (&RastPort, Muster)' (Set Draw Pattern) ein neues Linienmuster anzugeben. Das 16-Bit-Wort 'Muster' enthält dabei einfach die Punkte der Linie, die gesetzt werden sollen (Voreingestellt war: Muster = 0xffff, was bedeutete, daß die Linie voll durchgezogen wurde).

11. Jetzt kommt Farbe ins Spiel: die Zeichenstifte

Wenn Sie die beiden vorigen Programme ans 'Laufen' gebracht haben, haben Sie sich sicher schon gefragt, wie die Farbe, mit der ein Punkt gesetzt oder eine Linie gezogen werden soll, errechnet wird.

Dazu wollen wir noch einmal in die Tiefen Ihres Amiga hinabsteigen - zu den Bitplanes nämlich. Die einzelnen Bits einer jeden Bitplane, die alle für einen Punkt zuständig sind, werden geORt.

Sind z.B. die Punkt-Bits der BitPlanes 1 und 3 gesetzt, das der Bitplane 2 ist aber gelöscht, so erhält der Punkt die Farbe aus Farbregister $(1*2^0 + 0*2^1 + 1*2^2) = 5$. Man kann hier also die ganz normale binäre Arithmetik anwenden.

Wenn man die Farbe festlegt, mit der ein Punkt gezeichnet werden soll, geht man den umgekehrten Weg: Man legt das Farbregister fest, mit dessen Farbe die Punkte von nun an dargestellt werden sollen, und die Grafikbefehle setzen nun die einelnen Punkt-Bits in die Bitplanes, um dies zu erreichen.

Dabei gilt es zu beachten, daß der Amiga zwei 'Farbstifte', den APen und den BPen kennt. Doch bevor wir deren Funktion erläutern können, müssen wir erst einmal auf die Zeichenmodi eingehen.

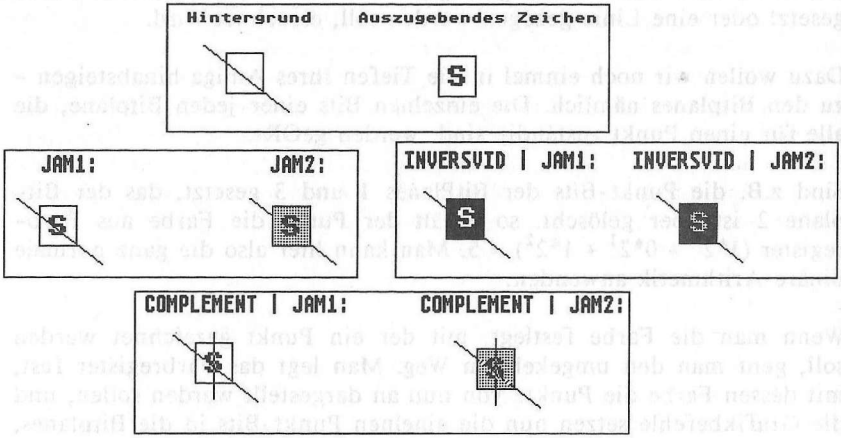
11.2 Die Zeichenmodi werden verändert

Diese Zeichenmodi (Drawmodes) geben an, wie die Punkt-Bits eines Punktes, bevor sie in die zugehörigen Bitplanes geschrieben werden, mit den dort schon vorhandenen Bits verknüpft werden. Dabei stehen die Bitmanipulationen Or, And, Not und Exor zur Verfügung.

Im Normalmodus (SetDrMd (&RastPort, JAM1)) werden die zu setzenden Bits einfach in die BitPlanes hineingeORt.

Im JAM2-Modus (SetDrMd (&RastPort, JAM2)) werden die zu setzenden Punkte geANDet. Das macht beim Setzen einzelner Punkte und dem Ziehen von Linien keinen Unterschied zu JAM1. Wenn man aber Texte ausgeben will, erkennt man den Unterschied sehr schnell. Ein Zeichen wir nämlich als Bitmuster gesetzter und nicht gesetzter Punkte im Speicher abgespeichert. Der Blitter (Block Image Transferrer) der für die Zeichenausgabe zuständig ist, kopiert auch die nicht gesetzten Bits des Zeichens mit in die BitPlane. Wurde nun der JAM2-Modus

eingestellt, wird der Hintergrund mit den nicht gesetzten Bits überschrieben:



Im COMPLEMENT-Modus werden die zu setzenden Bits eines Punktes (auch die einer Linie bzw. Pixels) mit den vorher schon an diesen Stellen in den BitPlanes vorhandenen Bits geEXORt. Die Verknüpfungstabelle sieht also so aus:

Bitplane	Operation: ^	Punkt		Ergebnis
0		0	=	0
0		1	=	1
1		0	=	1
1		1	=	0

Im letzen Modus, dem INVERSVID-Modus, werden die zu setzenden Punkt-Bits umgedreht. Das heißt, daß überall dort, wo ein Punkt-Bit gelöscht werden sollte, nun eines gesetzt wird, und umgekehrt (NOT).

Jedoch sollte man beachten, daß das Verknüpfen mit NOT oder EXOR zuerst nur intern vonstatten geht. Zu sehen bekommt man das Ergebnis nur, wenn man INVERSVID oder COMPLEMENT zusammen mit JAM1 oder JAM2 benutzt (z.B: SetDrMd (&RastPort, INVERSVID | JAM2)). Das Bitmuster wird dann wie bei JAM1 und JAM2 gewohnt in die Bitplanes hineingeORt bzw. geANDet. Das Er-

gebnis ist meist eine kunterbunte Linie oder Text mit farbigen Tupfen.

11.2 Der Vordergrundstift

Nachdem nun die Drawmodes erläutert wurden, kommen wir endlich zu den Zeichenstiften. Mit 'SetAPen (&RastPort, Nummer_des_Farbregisters)' legt man fest, in welcher Farbe ein Punkt im JAM1-Modus gezeichnet werden soll. Bei Punkten und Linien besteht, wie oben schon erwähnt, kein Unterschied zwischen JAM1 und JAM2.

11.3 Der Hintergrundstift

Will man aber Texte ausgeben, kann man mit Hilfe des JAM2-Zeichenmodus und des BPens (Hintergrundstift) farbig unterlegte Texte erzeugen. Dazu legt man einfach mit 'SetBPen (&RastPort, Nummer_des_Farbregisters)' fest, in welcher Farbe die nicht gesetzten Punkte eines Zeichens auf dem Bildschirm erscheinen sollen.

Zur Funktion von APen und BPen sei noch einmal erwähnt, daß diese das Bitmuster bestimmen, das an immer der gleichen Stelle in die Bitmap, Bit für Bit in die einzelnen Bitplanes, hineingeschrieben werden soll.

12. Intuition macht das Leben leicht

Nachdem Sie jetzt schon wissen, wie man Punkte zeichnet, Linien zieht, die Zeichenmodi verändert und neue Zeichenstifte bestimmt, wollen wir Ihnen nun eine wesentlich einfachere Art und Weise vorstellen, View, Viewport etc. anzulegen.

Das Zauberwort dafür heißt Intuition. Diese Benutzeroberfläche ermöglicht es Ihnen, auf einfache Art und Weise Zugriff auf einen Rastport zu erlangen.

Dazu muß allerdings die Intuition-Library geöffnet werden. Dies geschieht ähnlich wie bei der Öffnung der Grafik-Bibliothek. Nur wird hier als Erkennungs-String nicht "graphics.library", sonder logischerweise "intuition.library" angegeben (IntuitionBase = (struct IntuitionBase *) OpenLibrary ("intuition.library",0)).

12.1 Der eigene Screen...

Die eigentliche Darstellungsfläche von Intuition stellen die Screens dar. Screens sind vollkommen initialisierte Viewports mit einer Colormap, einer Bitmap, einem Rastport etc.

Auch ein View wird uns von Intuition zur Verfügung gestellt. Allerdings brauchen wir uns darum gar nicht zu kümmern, da uns Intuition fast alles aus der Hand nimmt.

Im wesentlichen reduziert sich das Öffnen von Screens auf zwei Aktionen:

1. Das Initialisieren einer NewScreen-Struktur und
2. den OpenScreen-Aufruf.

Wie die NewScreen-Struktur im einzelnen initialisiert wird, möchten wir Sie bitten im Anhang nachzuschauen. Zum 'Screen = (struct Screen *) OpenScreen (&NewScreen)'-Befehl wollen wir nur soviel sagen, daß er einen vollkommen funktionstüchtigen, dargestellten Screen erzeugt, auf den wir mittels der Screen-Struktur zugreifen können (Screen und NewScreen sind nicht identisch!). Auf den für (fast) alle Grafikbefehle 'lebenswichtigen' Rastport greift man z.B. mit '&Screen->RastPort' zu.

Das beste Beispiel eines Intuition-Screens finden Sie im Workbench-Screen, auf dem ja alle Aktivitäten der Workbench stattfinden.

Zu erwähnen ist noch, daß OpenScreen sehr viel Speicherplatz belegt. Ist also der Zeiger 'Screen', der auf die initialisierte Screen-Struktur zeigt, gleich 0, so konnte nicht mehr genügend Speicher herangezogen werden. In solch einem Fall gibt das Programm am besten eine kleine Nachricht aus und verabschiedet sich dezent, bzw. kehrt zum CLI oder zur Workbench zurück.

12.2 ...und dann das Window

Für die Ausgabe von Grafiken wird aber auch oft das sogenannte Window (Fenster) verwendet. Solch ein Window hat den Vorteil, daß z.B. Linien, die über den Rand eines Windows hinaus gezeichnet werden, abgeschnitten werden. Beim Screen und bei eigenen Viewports ist dies nicht der Fall. Der 'überstehende' Rest der Linie wird unter Umständen unkontrolliert in den Speicher geschrieben und sorgt so vielleicht für Datenverlust.

Auch die Öffnung eines Windows erfolgt im wesentlichen in zwei Aktionen:

1. Initialisierung einer NewWindow-Struktur und
2. 'Window = (struct Window *) OpenWindow (&NewWindow)'-Aufruf.

Zugriff auf den Rastport eines Windows haben Sie z.B. mit `'&Window->RastPort'`.

12.3 Das Verlassen von Intuition

So, wie Sie auch die Speicherplätze für die eigenen Bitplanes etc. wieder freigeben mußten, müssen Sie auch bei Intuition dafür sorgen, daß die Screens und Windows nach Gebrauch wieder geschlossen werden. Der vorher belegte Speicherplatz steht dann dem System wieder zur Verfügung.

Einen Screen schließt man mit 'CloseScreen (Screen)'. Das Window wird ganz ähnlich geschlossen: 'CloseWindow (Window)'. Allerdings muß das Window vor dem Screen geschlossen werden, sonst kommt es zu "Guru Meditations".

Und natürlich muß man auch dafür sorgen, daß neben der Grafik-Bibliothek auch die Intuition-Bibliothek geschlossen wird:

```
CloseLibrary (IntionBase)
```

13. Das Füllen von Flächen in C

Nachdem wir nun auf die Hilfe von Intuition zugreifen können, wenden wir uns nun den Flächen zu.

13.1 Die farbige Flut: der Flood-Befehl

Mit dem Flood-Befehl kann man zusammenhängende Flächen füllen. Ein - zugegebenermaßen etwas bildhaftes - Beispiel möge das verdeutlichen: Die Wirkung von Flood ist vergleichbar mit einer Wanne, in die Farbe geschüttet wird. Normalerweise (das hängt natürlich von der Menge ab) läuft die Farbe nicht über den Rand hinaus. Die Sache sieht aber ganz anders aus, wenn der Rand schadhaft ist, man das aber zu spät merkt...

Beim Flood-Befehl wird der Rand durch eine geschlossene Umrandungslinie dargestellt, die nicht Rechteckig zu sein braucht. Mit dem Füllen wird an der angegebenen Koordinate, die sich natürlich innerhalb der zu füllenden Fläche befinden muß, begonnen:

```
Flood (&RastPort,Modus,x,y)
```

Zu beachten ist, daß bei dieser Art des Füllens der Modus-Parameter = 0 ist. Das bedeutet: Fülle alle Punkte in der aktuellen Zeichenfarbe im aktuellen Zeichenmodus mit dem aktuellen Füllmuster, und zwar solange, bis du an den Rand stößt.

Dieser Rand wird, wie oben schon erwähnt, durch eine geschlossene Umrandungslinie einer bestimmten Farbe gebildet. Die Farbe dieser Umrandungslinie wird mit 'SetOPen (&RastPort, Farbe_des_Randes)' festgelegt (OPen = AOLPen = Area Out Line Pen = Flächen-Umrandungs-Stift).

Ist der Modus-Parameter aber gleich 1, wird auf etwas andere Art und Weise gefüllt. Die Farbe der Umrandungslinie spielt jetzt keine Rolle mehr. Es wird nämlich nur die zusammenhängende Fläche gefüllt, die aus Punkten der Farbe unter der angegebenen X/Y-Koordinate besteht.

Leider hat der Flood-Befehl eine Eigenart, die dafür sorgt, daß vor dessen Benutzung noch einige Vorkehrungen getroffen werden müssen: er "frißt" Speicher.

Aufgrund eines rekursiven Füllalgorithmus muß man ein sogenanntes temporäres Raster, eine kurzzeitig verwendete (zusätzliche) Bitplane anlegen. Diese Bitplane muß eigentlich nur so groß sein wie das zu füllende Objekt. Auf 'Nummer Sicher' gehen Sie aber, wenn Sie eine komplette Zusatz-Bitplane anlegen (AllocRaster).

Diesen Speicherplatz müssen Sie dann nur noch einer TmpRas-Struktur zugänglich machen und diese wiederum an den Rastport übergeben:

```
InitTmpRas (&TmpRas,&SpeicherAdresse,SpeicherGröße)
RastPort.TmpRas = &TmpRas
```

oder

```
RastPort.TmpRas = InitTmpRas (&TmpRas,&SpeicherAdresse,SpeicherGröße)
```

Vergessen Sie nicht, die Zusatz-Bitplane nach Gebrauch wieder freizugeben!

13.2 Rechtecke gefällig?

Neben dem Füllen zusammenhängender Flächen kann der Amiga natürlich auch einfachere Flächen füllen (Wozu hat er denn den Blitter?).

Mit dem 'RectFill (&RastPort, x1, y1, x2, y2)'-Befehl können Sie Rechtecke füllen. Dabei geben die Koordinaten x1/y1 die linke obere Ecke und die Koordinaten x2/y2 die rechte untere Ecke des zu füllenden Rechtecks an. Bitte achten Sie darauf, daß die linke obere Ecke auch tatsächlicks links und oberhalb der echten unteren Ecke liegt. Wenn nicht, kommt es zu Systemabstürzen, sprich "Gurus".

Eine Besonderheit bei diesem Befehl stellen die Umrandungslinien dar, die um das ausgefüllte Rechteck gezeichnet werden. Die Farbe dieser Linien wird durch den OPen (s. oben) festgelegt.

Wollen Sie allerdings nicht, daß diese Linien gezeichnet werden, brauchen Sie nur das Macro 'BNDRYOFF (&RastPort)', was für 'Boundary Off' steht, zu benutzen. Dieses Macro löscht ganz einfach das AREAOUTLINE-Bit in der Flags-Variablen des Rastports. Wollen zu einem späteren Zeitpunkt aber wieder eine Umrandungslinie um Ihre Rechteck gezeichnet haben, brauchen Sie das AREAOUTLINE-Bit nur wieder zu setzen (RastPort.Flags |= AREAOUTLINE). Ändern

Sie die Farbe des OPens (mit 'SetOPen()'), so wird dieses Bit automatisch gesetzt.

Folgendes Programm macht von dieser Art des Flächenfüllens, die übrigens ohne TmpRas (Zusats-Bitplane) auskommt, Gebrauch:

```
/*********************************************************/
/*                      Hanoi.c                          */
/* Dieses Programm zeigt anhand der "Türme von Hanoi" den */
/* den Einsatz des RectFill() Befehls.                    */
/*                                                        */
/* Compiled with: Lattice V3.10                           */
/*                                                        */
/* (c) Bruno Jennrich                                     */
/*********************************************************/

#include "exec/types.h"
#include "exec/devices.h"
#include "intuition/intuition.h"
#include "intuition/intuitionbase.h"
#include "graphics/gfx.h"
#include "graphics/gfxbase.h"
#include "graphics/gfxmacros.h"
#include "graphics/copper.h"
#include "graphics/gels.h"
#include "graphics/regions.h"
#include "devices/keymap.h"
#include "hardware/blit.h"

#define GFX (struct GfxBase *)
#define INT (struct IntuitionBase *)
#define SCR (struct Screen *)

#define MAXHOEHE 30                /* Maximal 30 Scheiben */

#define WIDTH 640                            /* Breite */
#define HEIGHT 256                           /* Höhe */

#define RPort &Screen->RastPort          /* Unser RastPort */

struct NewScreen NewScreen =
                {
                   0,0,WIDTH,HEIGHT,2,
                   1,0,HIRES,CUSTOMSCREEN,
                   NULL,NULL,NULL
                };

char *String = "Türme von Hanoi mit RectFill()";
```

```
struct Screen *Screen;
struct GfxBase *GfxBase;
struct IntuitionBase *IntuitionBase;

UWORD pattern[4] = {0x9248,0x2492,0x4924,0x9248};
long pos[3];                          /* Für Pin Position */

long stapel[3][MAXHOEHE],             /* Auf welchem Pin, an */
                                      /* welcher Position,   */
                                      /* welche Scheibe ?    */

     hoehe[3] = {0,0,0};              /* Wieviele Scheiben   */
                                      /* auf Pins ?          */
long breite;
long scheiben; /* Wieviele Scheiben hat der Anfangsturm ? */
long high,offset;

char *LeftMouse = (char *)0xbfe001;

VOID turm();                          /* Vorwärtsdeklaration */
VOID init();
VOID build_up();
VOID draw();

/*************************************************************/
/* Es geht los !                                             */
/*************************************************************/

main(argc,argv)                       /* Hole Argument */
int argc;
char *argv[];
{
   long i,j,
        Length;

     if (!(argc==2))
        {
           printf (" Turmhöhe: ");
           scanf ("%ld",&scheiben);
        }
     else
        sscanf(argv[1],"%ld",&scheiben);

     if (scheiben >= MAXHOEHE)
        {
           printf ("Zu viele Scheiben !!!\n");
           exit (1);
        }
```

```
    /* Berechne Pin (X-) Positionen für Darstellung */
pos[0] = WIDTH/6+5;                          /* Links */
pos[1] = WIDTH/2;                            /* Mitte */
pos[2] = (WIDTH/6)*5-5;                     /* Rechts */

breite = (WIDTH/6)/scheiben - 2;
                              /* differenz-breite */

init();

SetRGB4(&Screen->ViewPort,0,0,0,0);   /* Lege Farben */
SetRGB4(&Screen->ViewPort,1,15,15,0); /*       fest */
SetRGB4(&Screen->ViewPort,2,0,15,15);
SetRGB4(&Screen->ViewPort,3,15,0,15);

                            /* Pinne initialisieren */
for (i=0; i<MAXHOEHE; i++)
    for (j=0; j<3; j++)
       stapel[j][i]=0;              /* Pinne löschen */

for (i=0; i<scheiben; i++)
       stapel[0][i]=scheiben-i;
                        /* kleinste Scheibe zu oberst */

hoehe[0] = scheiben;          /* 1 Pin hat N-Scheiben */
SetAfPt(RPort,pattern,2);        /* Setze Füllmuster */
SetOPen(RPort,2);                /* Umrandungsstift */

high = (HEIGHT*3/8)/scheiben;
                      /* Wie hoch ist eine Scheibe ? */
offset = HEIGHT*3/8;

build_up();    /* Das erste mal Darstellung aufbauen */

SetAPen (RPort,3);

                                 /* String ausgeben */
Length=TextLength(RPort,String,strlen(String))/2;
Move (RPort,WIDTH/2-Length,
          Screen->RastPort.TxBaseline);
Text (RPort,String,strlen(String));

                               /* Rekursions Start */
turm (scheiben,1,2,3);

while ((*LeftMouse & 0x40) == 0x40);
                          /* Wart' noch'n bisschen */
CloseScreen(Screen);
CloseLibrary(GfxBase);
CloseLibrary(IntuitionBase);
```

```
    return (0);
}

/*********************************************************/
/* Diese rekursive Prozedur stapelt die Scheiben von     */
/* Pin Eins nach Pin Drei um.                            */
/*-------------------------------------------------------*/
/* Eingabe-Parameter:                                    */
/*    n ::= Anzahl der Scheiben                          */
/*    l ::= Nummer des linken Pins                       */
/*    m ::=   "     "  mittleren Pins                     */
/*    r ::=   "     "  rechten    "                       */
/*-------------------------------------------------------*/
/* Rückgabe-Werte: keine                                 */
/*********************************************************/

VOID turm (n,l,m,r)
long n,l,m,r;
{

   if (n == 1)
      {
            stapel[r-1][hoehe[r-1]] =
                              stapel[l-1][hoehe[l-1]-1];
            hoehe[r-1]++;
            draw(r-1,l-1);
            stapel[l-1][hoehe[l-1]-1] = 0;
            hoehe[l-1]--;
      }
   else
      {
       turm (n-1,l,r,m);
            stapel[r-1][hoehe[r-1]] =
                              stapel[l-1][hoehe[l-1]-1];
            hoehe[r-1]++;
            draw(r-1,l-1);
            stapel[l-1][hoehe[l-1]-1] = 0;
            hoehe[l-1]--;
       turm (n-1,m,l,r);
      }
}

/*********************************************************/
/* Diese Prozedur erröffnet die nöigen Librarys, und     */
/* einen Screen.                                         */
/*-------------------------------------------------------*/
/* Eingabe-Parameter: keine                              */
/*-------------------------------------------------------*/
/* Rückgabe-Werte: keine                                 */
/*********************************************************/
```

```
VOID init ()
{
      GfxBase = GFX OpenLibrary("graphics.library",0);
      IntuitionBase = INT OpenLibrary ("intuition.library",0);
      Screen = SCR OpenScreen (&NewScreen);
}

/*********************************************************/
/* Diese Prozedur baut die drei Türme das erste mal auf. */
/*-------------------------------------------------------*/
/* Eingabe-Parameter: keine                              */
/*-------------------------------------------------------*/
/* Rückgabe-Werte: keine                                 */
/*********************************************************/

VOID build_up()
{
   long i,j;

   WaitTOF();          /* Synchronisiere Ausgabe mit Beam */
   SetRast(RPort,0);
   SetAPen(RPort,1);

   for (j=0; j<3; j++)
      for (i=1; i<=scheiben; i++)
           RectFill (RPort,
                     pos[j]-(stapel[j][scheiben-i]*breite),
                     ((i-1)*high)+offset,
                     pos[j]+(stapel[j][scheiben-i]*breite),
                     (i*high)+offset);
}

/*********************************************************/
/* Diese Prozedur löscht, bzw. setzt die oberste Scheibe */
/* zweier Pinne.                                         */
/*-------------------------------------------------------*/
/* Eingabe-Parameter:                                    */
/*    neu ::= Pin, auf den neue Scheibe gesetzt wird     */
/*    weg ::= Pin, dessen oberste Scheibe weggenommen wird*/
/*-------------------------------------------------------*/
/* Rückgabe-Werte: keine                                 */
/*********************************************************/

VOID draw(neu,weg)
long neu,weg;
{
   SetAPen (RPort,0);                    /* Lösche Scheibe */
   SetOPen (RPort,0);
```

```
    if (hoehe[weg]-1 == 0)                    /*letzter Pin ?*/

    RectFill (RPort,
              pos[weg]-stapel[weg][hoehe[weg]-1]*breite,
              (scheiben-hoehe[weg])*high+offset,
              pos[weg]+stapel[weg][hoehe[weg]-1]*breite,
              (scheiben-hoehe[weg]+1)*high+offset);
    else

    RectFill (RPort,
              pos[weg]-stapel[weg][hoehe[weg]-1]*breite,
              (scheiben-hoehe[weg])*high+offset,
              pos[weg]+stapel[weg][hoehe[weg]-1]*breite,
              (scheiben-hoehe[weg]+1)*high+offset-1);

    SetAPen (RPort,1);              /* Schreibe neue Scheibe */
    SetOPen (RPort,2);

    RectFill (RPort,pos[weg],
              (scheiben-hoehe[weg])*high+offset,
              pos[weg],
              (scheiben-hoehe[weg]+1)*high+offset);

    RectFill (RPort,pos[neu]-stapel[neu][hoehe[neu]-1]*breite,
              (scheiben-hoehe[neu])*high+offset,
              pos[neu]+stapel[neu][hoehe[neu]-1]*breite,
              (scheiben-hoehe[neu]+1)*high+offset);

    Delay(Screen->MouseX/10);
                /* Pause in Abhängigkeit der Mausposition */
}
```

Wenn Sie genau hinsehen, stellen Sie sicher fest, daß die vom Programm gezeichneten Rechtecke nicht vollständig ausgefüllt sind. Sie finden immer wieder kleine Löcher, die den Hintergrund durchscheinen lassen.

Dies haben wir dadurch erreicht, daß wir unser eigenes Füllmuster, wie beim Flood-Befehl schon angesprochen, kreiert haben.

Dabei gilt es, zwei Arten von Füllmustern zu unterscheiden: das einfarbige und das mehrfarbige Muster.

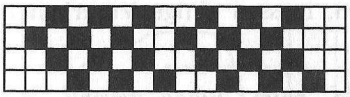
Beiden gemeinsam ist, daß sie nur die Höhe einer Zweierpotenz (also 1, 2, 4, 8... Zeilen) und die Breite von 16 Punkten (Bits) haben können.

Das Füllmuster wird in einem Speicherbereich (z.B. in einem Word-Array) gespeichert.

Für ein einfarbiges Muster muß man nur eine Bitplane festlegen. Die Farbe des APens (BPens) und der Zeichenmodus haben also Auswirkungen auf das Aussehen der Fläche.

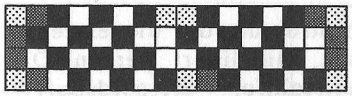
Bei mehrfarbigen Mustern muß man allerdings für jede Bitplane eine eigenes Bitmuster angeben.

einfarbiges Muster:

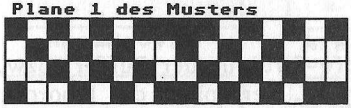


```
UWORD Muster[4] = {0x2A54,
                   0x54A8,
                   0x2A54,
                   0x152A};
```

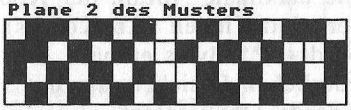
mehrfarbiges Muster:



Plane 1 des Musters



Plane 2 des Musters



```
UWORD Muster[8] = {0xABD5,
                   0x54A8,
                   0x2A54,
                   0x95AB,

                   0x6A56,
                   0xD4A9,
                   0xAA55,
                   0x556A};
```

Diese Bitmuster werden dann unter Verwendung des aktuellen Zeichenmodus direkt in die jeweiligen Bitplanes geschrieben. Hier wirkt nur noch der Zeichenmodus.

Doch wie macht man die Muster für die Grafik-Befehle nutzbar? Dabei hilft uns der 'SetAfPt (&RastPort, &BitMuster, Höhe)'-Befehl (Set Area Fill Pattern).

Die Höhe des Füllmusters wird dabei als Exponent zur Basis 2 angegeben. Ein 8 Zeilen hohes Füllmuster hätte also die 'Höhe' 3 ($2^3 = 8$).

Dies gilt allerdings nur für einfarbige Muster. Bei mehrfarbigen Mustern wird der Exponent mit nNegativem Vorzeichen übergeben, also -3 anstatt 3.

13.3 Polygone füllen: Area... macht's möglich

Eine weitere Art, Flächen zu füllen, stellen die 'Area...'-Befehle dar. Mit diesen Befehlen haben Sie die Möglichkeit, ausgefüllte Polygone zu zeichnen, deren Umriß weiterhin mit Linien in der Farbe des OPens umrandet wird.

Doch um diese Befehle nutzen zu können, müssen einige Vorarbeiten getätigt werden:

Die wichtigste Voraussetzung ist, wie beim Flood-Befehl, die TmpRas-Struktur. Sie muß vollständig initialisiert in dem Rastport, in dem Sie Polygone zeichnen wollen, vorliegen.

Weiterhin muß die sogenannte AreaInfo-Struktur, die die einzelnen 'Stützpunkte' der Polygone aufnehmen soll, initialisiert werden. Dies geschieht mit dem 'InitArea (&AreaInfo, &KoordsBuffer, AnzKoords)'-Befehl.

InitArea sorgt dafür, daß der AreaInfo-Struktur der Puffer (bzw. die Adresse dieses Puffers) zugewiesen wird, der später die Koordinaten enthalten soll. Weiterhin wird auch die maximale Anzahl der Koordinaten festgelegt, die in diesem Puffer platz finden (AnzKoords). Zu beachten ist, daß der Puffer (AnzKoords+1)*5 Bytes enthält. Am einfachsten ist es, wenn man den erforderlichen Speicherplatz mit Hilfe eines char-Arrays (char Buffer[(AnzKoords+1)*5]) anlegt.

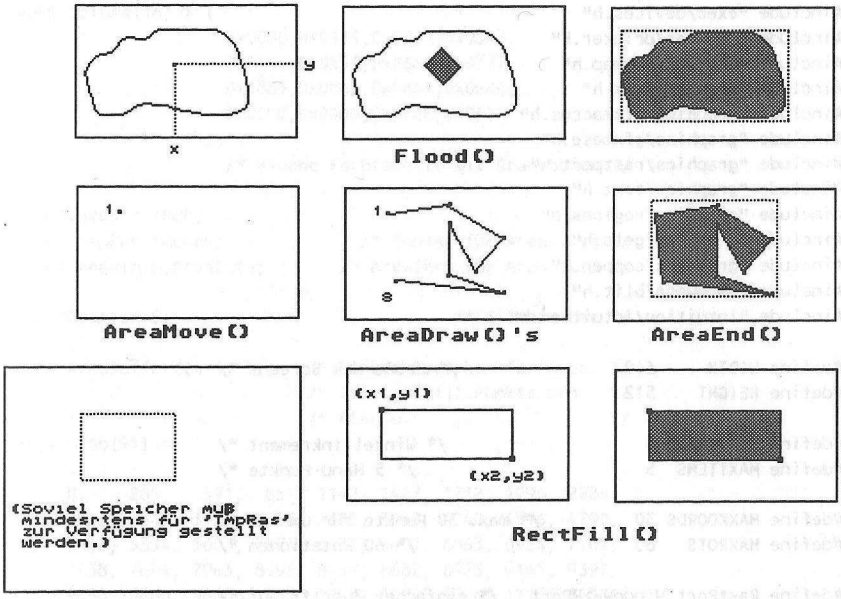
Nachdem die so initialisierte AreaInfo-Struktur an den Rastport übergeben wurde (RastPort.AreaInfo = &AreaInfo) können wir unser Polygon festlegen. Die erste Koordinate muß immer mit 'AreaMove (&RastPort, x, y)' angegeben werden. AreaMove signalisiert, daß ein neues Polygon gezeichnet werden soll.

Nun können mit 'AreaDraw (&RastPort, x, y)' weitere Polygon-Stützpunkte festgelegt werden. x und y geben auch hier die Koordinaten des Punktes innerhalb des Rastports an.

Der letzte Punkt eines Polygons wird mit 'AreaEnd (&RastPort, x, y)' festgelegt. Hier wird auch dafür gesorgt, daß das Polygon gezeichnet und (mit einer Unterart des Flood-Befehls) gefüllt wird.

Wenn Ihnen auch hier die Umrandungslinien nicht zusagen sollten, können Sie diese auch mit dem 'BNDRYOFF (&RastPort)'-Macro unterdrücken.

Folgende Abbildung zeigt Ihnen noch einmal alle Füllmöglichkeiten auf einen Blick:



Zum Abschluß dieses Kapitels haben wir uns noch etwas Besonderes für Sie ausgedacht: Das Programm, das jetzt gleich folgt, erzeugt Rotationskörper (dreidimensional!), aus den von Ihnen mit der Maus eingegebenen Umrissen:

```
/*********************************************************/
/*                    RotateIt.c                         */
/*                                                       */
/* Dieses Programm erzeugt dreidimensionale Rotations-   */
/* körper, die mittels Area..() aufgebaut bzw.           */
/* bzw. dargestellt werden.                              */
/*                                                       */
/* Compiled with: Lattice V3.10                          */
/*                                                       */
/* (c) Bruno Jennrich (The VS.F)                         */
/*********************************************************/

#include "exec/types.h"
#include "exec/memory.h"
#include "exec/devices.h"
#include "devices/printer.h"
#include "devices/keymap.h"
#include "graphics/gfx.h"
#include "graphics/gfxmacros.h"
#include "graphics/gfxbase.h"
#include "graphics/rastport.h"
#include "graphics/text.h"
#include "graphics/regions.h"
#include "graphics/gels.h"
#include "graphics/copper.h"
#include "hardware/blit.h"
#include "intuition/intuition.h"

#define WIDTH     640              /* Größe des Screens */
#define HEIGHT    512

#define INCREMENT 3                /* Winkel inkrement */
#define MAXITEMS  5                /* 5 Menu-Punkte */

#define MAXKOORDS 30          /* max. 30 Punkte für Umriss */
#define MAXROTS   60               /* 60 Rotationen */

#define RastPort Window->RPort   /* einfacher Zugriff auf */
                                 /* Window's RastPort     */

#define MAXAREA (MAXKOORDS-1)*MAXROTS
                                 /* Wie viele Flächen ? */

char ASCString[6];                     /* Für itoa() */

struct GfxBase *GfxBase=0;
struct IntuitionBase *IntuitionBase=0;
struct Screen *Screen=0;
struct Window *Window=0;               /* BasePointers */
```

```
struct MenuItem Items[MAXITEMS];          /* Menu Einträge */
struct IntuiText Texts[MAXITEMS];         /* Menu Texte    */

struct Menu Menus;                                /* Menu */

struct NewWindow NewWindow;
struct NewScreen NewScreen;

char *IString[MAXITEMS] = {
                          "Shading","Hide It",
                          "Koordinaten",
                          "Hardcopy","Quit"
                         };          /* Menu Text-Strings */

UWORD Colors[16] = {
                    0x0000,0x0111,0x0222,0x0333,
                    0x0444,0x0555,0x0666,0x0777,
                    0x0888,0x0bbb,0x0AAA,0x0BBB,
                    0x0CCC,0x0DDD,0x0EEE,0x0FFF
                   };
                    /* eigene Farbtabelle mit Graustufen */

long Status = TRUE;
struct TmpRas TmpRas;                  /* TmpRas für Area...() */
struct AreaInfo AreaInfo;             /* AreaInfo für Area...() */

long *Pointer=0;                                 /* Hilfszeiger */

UBYTE AreaBuffer[(4+1)*5];   /* Punkte Speicher für Area. */
                             /* Immer 4 (+1) Punkte pro   */
                             /* Fläche                    */
int sintab[91] =
  {
     0,   285,   571,   857, 1142, 1427, 1712, 1996, 2280,
     2563, 2845, 3126, 3406, 3685, 3963, 4240, 4516, 4790,
     5062, 5334, 5603, 5871, 6137, 6401, 6663, 6924, 7182,
     7438, 7691, 7943, 8192, 8438, 8682, 8923, 9161, 9397,
     9630, 9860, 10086,10310,10531,10748,10963,11173,11381,
     11585,11785,11982,12175,12365,12550,12732,12910,13084,
     13254,13420,13582,13740,13894,14043,14188,14329,14466,
     14598,14725,14848,14967,15081,15190,15295,15395,15491,
     15582,15662,15749,15825,15897,15964,16025,16082,16135,
     16182,16224,16261,16294,16321,16244,16361,16374,16381,
     16384
  };
                /* Sinustabelle, wird so berechnet:      */
                /* sintab[x]=sin(x) * 16384 (x=1,2..,90) */
                /* Siehe: sincos(), sin(), cos()         */
```

```
int Punkte[MAXROTS][MAXKOORDS][3];
int Mother[MAXROTS][MAXKOORDS][3];
            /* Mother-Array wird aus eingegebenem Umriss */
            /* des Rotationskörpers berechnet, und wird  */
            /* dann als Muster zur Berechnung des Punkte */
            /* Arrays herangezogen. Dies ist nötig, da   */
            /* durch mehrere Transformationen im Raum    */
            /* mit nur einem Array 'irreparable'         */
            /* Schäden auftreten, die das Aussehen des   */
            /* Rotationskörpers negativ beeinflussen.    */

int Area[MAXAREA][4][3];
          /* Hier werden die Flächen, die aus den Punkten */
          /* errechnet werden, abgespeichert.             */

int KoordArray[MAXKOORDS][2];
          /* Hier werden die Koordinaten des eingegebenen */
          /* Umrisses abgespeichert                       */

int x[MAXROTS][MAXKOORDS];
int y[MAXROTS][MAXKOORDS];
          /* Für Transformation aus 3D -> 2D (Bildschirm) */

int Index[MAXAREA+1];                    /* Fürs Sortieren */

long ProzZ = 1500,                    /* Projektions Zentrum */
     ProzY = 0,
     ProzX = 0;

long d = 0;                  /* Entfernung Projektionsebene - */
                             /* Projektionszentrum            */

long bbpx = 0,                        /* Beobachterbezugspunkt */
     bbpy = 0,
     bbpz = 0;

long WinkelX=0,                         /* Darstellungswinkel */
     WinkelY=0,
     WinkelZ=0;

struct IntuiMessage *Message;     /* Intuition's Nachricht */

extern struct MsgPort *CreatePort();
extern struct IOStdReq *CreateStdIO();
                          /* Diese Funktionen werden für */
                          /* Zugriff auf den Printer     */
                          /* (hardcopy()) gebraucht      */

VOID InitScreenWindow();          /* Vorwärtsdeklarationen */
VOID BuildUpMenu();
```

```
VOID CloseIt();
VOID OpenLibs();
VOID IDCMPoff();
VOID IDCMPon();
VOID Rotate();
VOID Show();
long Average();
char *itoa();
VOID Shade();
VOID Init();
VOID Rot();
long sincos();
long sin();
long cos();
long EnterKoords();
VOID make_menu();
struct IORequest *CreateExtIo();
VOID DeleteExtIO();
VOID hardcopy();

/*********************************************************/
/* Diese Funktion initialisiert die angegebenen NewScreen */
/* und NewWindow Strukturen für OpenScreen(), bzw.       */
/* OpenWindow()                                          */
/*-------------------------------------------------------*/
/* Eingabe-Parameter: zu initialisierende NewScreen-     */
/*                              Struktur (NS)            */
/*                    zu initialiserende NewWindow-      */
/*                              Struktur (NW)            */
/*-------------------------------------------------------*/
/* Rückgabe-Werte: keine                                 */
/*********************************************************/

VOID InitScreenWindow (NS,NW)
struct NewScreen *NS;
struct NewWindow *NW;
{
   NS->LeftEdge = 0;        NS->TopEdge = 0;
   NS->Width = WIDTH;       NS->Height = HEIGHT;
   NS->Depth = 4;
   NS->DetailPen = 1;       NS->BlockPen = 0;
   NS->ViewModes = 0;

   if (WIDTH > 320) NS->ViewModes |= HIRES;
   if (HEIGHT > 200) NS->ViewModes |= LACE;

   NS->Type = CUSTOMSCREEN;
   NS->Font = NULL;
   NS->DefaultTitle = "";
   NS->Gadgets = NULL;      NS->CustomBitMap = NULL;
```

```
   NW->LeftEdge = 0;          NW->TopEdge = 0;
   NW->Width = WIDTH;         NW->Height = HEIGHT;
   NW->DetailPen = 6;         NW->BlockPen = 0;
   NW->IDCMPFlags = NULL;
   NW->Flags = BORDERLESS | ACTIVATE | NOCAREREFRESH
                                     | REPORTMOUSE;
   NW->FirstGadget = NULL;
   NW->CheckMark = NULL;      NW->Title = "";
   NW->Screen = NULL;         NW->BitMap = NULL;
   NW->MinWidth = NW->MinHeight =
   NW->MaxWidth = NW->MaxHeight = 0;
   NW->Type = CUSTOMSCREEN;
}

/*********************************************************/
/* Diese Funktion baut das Menu, und dessen Einträge,    */
/* (Items) auf.                                          */
/*-------------------------------------------------------*/
/* Eingabe-Parameter: keine                              */
/*-------------------------------------------------------*/
/* Rückgabe-Werte: keine                                 */
/*********************************************************/

VOID BuildUpMenu()
{
   long i;

   for (i=0; i<MAXITEMS; i++)
      {
         Items[i].LeftEdge = 1;
         Items[i].TopEdge = i*10;
         Items[i].Width = 112;    Items[i].Height = 10;
         Items[i].Flags = ITEMTEXT | ITEMENABLED | HIGHCOMP;
         Items[i].MutualExclude = NULL;
         Items[i].ItemFill = (APTR) &Texts[i];
         Items[i].SelectFill = NULL;
         Items[i].Command = NULL;
         Items[i].SubItem =  NULL;
         Items[i].NextSelect = NULL;

         Texts[i].FrontPen = 6;     Texts[i].BackPen = 0;
         Texts[i].DrawMode = JAM1; Texts[i].LeftEdge = 1;
         Texts[i].TopEdge = 1;
         Texts[i].ITextFont = NULL;
         Texts[i].IText = IString[i];
         Texts[i].NextText = NULL;
      }
```

```
    for (i=0; i< MAXITEMS-1;i++)
        Items[i].NextItem = &Items[i+1];
    Items[MAXITEMS-1].NextItem = NULL;

    Menus.NextMenu = NULL;      Menus.LeftEdge = 10;
    Menus.TopEdge = 0;          Menus.Width = 110;
    Menus.Height = 10;          Menus.Flags = MENUENABLED;
    Menus.MenuName = " Projects";
    Menus.FirstItem = &Items[0];
}

/*********************************************************/
/* Diese Funktion beendet das Programm, nicht ohne vorher */
/* die ganzen geöffneten Librarys, Screens und Windows zu */
/* schliessen, und den extra belegten Speicherplatz wieder*/
/* zurückzugeben.                                         */
/*-------------------------------------------------------*/
/* Eingabe-Parameter: Ausgabe-String (evt. FehlerMeldung) */
/*-------------------------------------------------------*/
/* Rückgabe-Werte: keine                                  */
/*********************************************************/

VOID CloseIt(s)
char *s;
{
    puts(s);
    if (Pointer) FreeMem(Pointer,RASSIZE(WIDTH,HEIGHT/2));
                                    /* TmpRas-Speicher */

    if (Window) {
                ClearMenuStrip(Window);
                CloseWindow(Window);
              }
    if (Screen) CloseScreen(Screen);
    if (GfxBase) CloseLibrary(GfxBase);
    if (IntuitionBase) CloseLibrary(IntuitionBase);

    exit (0);
}

/*********************************************************/
/* Diese Routine erröffnet alles nötige (Librarys, Screen */
/* Window) für das Programm.                              */
/*-------------------------------------------------------*/
/* Eingabe-Parameter: keine                               */
/*-------------------------------------------------------*/
/* Rückgabe-Werte: keine                                  */
/*********************************************************/
```

```
VOID OpenLibs()
{
   InitScreenWindow(&NewScreen,&NewWindow);

   if ((GfxBase = (struct GfxBase *)
            OpenLibrary("graphics.library",0)) == 0)
      CloseIt("No Graphics !!!");

   if ((IntuitionBase = (struct IntuitionBase *)
            OpenLibrary("intuition.library",0)) == 0)
      CloseIt("No Intuition !!!");

   if ((Screen = (struct Screen *)
            OpenScreen(&NewScreen)) == 0)
      CloseIt("No Screen !!!");

   NewWindow.Screen = Screen;

   if ((Window = (struct Window *)
            OpenWindow(&NewWindow)) == 0)
      CloseIt("No Window !!!");

   LoadRGB4 (&Screen->ViewPort, &Colors[0], 16);
                                 /* Neue Farben laden */
   RemakeDisplay();              /* und darstellen    */

   SetRGB4(&Screen->ViewPort,17,4,4,4);     /* Mauszeiger */
   SetRGB4(&Screen->ViewPort,18,8,8,8);     /* Farben     */
   SetRGB4(&Screen->ViewPort,19,13,13,13);

   Init();
   BuildUpMenu();
}

/*********************************************************/
/* Diese Funktion schaltet die Nachrichtenübertragung von */
/* Intuition ab. Das ist deshalb so wichtig, da           */
/* normalerweise alle Intuition-Nachrichten in eine Liste */
/* (Queue) abgespeichert, und dann nach und nach          */
/* abgearbeitet werden. Da aber zeitweise nicht auf       */
/* Intuition reagiert werden kann (z.B. beim Rechnen und  */
/* Zeichnen), sollte man verhindern, daß während dieser   */
/* Zeit gesendete Nachrichten (z.B. Tastendrücke)         */
/* aufgelistet, und danach ziemlich unzusammenhängend     */
/* (für den Benutzer) abgearbeitet werden.                */
/*--------------------------------------------------------*/
/* Eingabe-Parameter: keine                               */
/*--------------------------------------------------------*/
/* Rückgabe-Werte: keine                                  */
/*********************************************************/
```

```
VOID IDCMPoff()
{
   SetMenuStrip(Window,NULL);
   ModifyIDCMP(Window,NULL);
                        /* Menuzeile vom Window entfernen */
}

/*********************************************************/
/* Diese Routine erlaubt Intuition wieder, Nachriten zu   */
/* senden                                                 */
/*-------------------------------------------------------*/
/* Eingabe-Parameter: keine                               */
/*-------------------------------------------------------*/
/* Rückgabe-Werte: keine                                  */
/*********************************************************/

VOID IDCMPon()
{
   ModifyIDCMP(Window, RAWKEY | MOUSEBUTTONS | MENUPICK);
   SetMenuStrip(Window,&Menus);          /* Menu mit Window */
                                         /* verbinden       */
}

/*********************************************************/
/* Diese Routine berechnet aus den Umriss Koordinaten die */
/* dreidimmensionale Rotationsfigur. (Rotation um         */
/* 3d - Y-Achse)                                          */
/*-------------------------------------------------------*/
/* Eingabe-Parameter: rot = Anzahl der Rotationen         */
/*                    ko  = Anzahl der Koordinaten des    */
/*                          Umrisses                      */
/*-------------------------------------------------------*/
/* Rückgabe-Werte: keine                                  */
/*********************************************************/

VOID Rotate(rot,ko)
long rot,ko;
{
   long i,
        j;
   long s,c;

   Move (RastPort,0,RastPort->TxBaseline);
   Text (RastPort, "Berechne Rotations-Matrix ...",29);

   for (i=0; i<rot; i++)
     {
      s=sin((360/rot)*i);      /* Berechne Rotationswinkel */
      c=cos((360/rot)*i);
```

```
      for (j=0; j<ko; j++)
         {
           Punkte[i][j][0] = Mother[i][j][0] =
                (KoordArray[j][0]*c)/16384;          /* x */
           Punkte[i][j][1] = Mother[i][j][1] =
                KoordArray[j][1];                    /* y */
           Punkte[i][j][2] = Mother[i][j][2] =
                -(KoordArray[j][0]*s)/16384;         /* z */

            /* 'Mother' bleibt unverändert, und dient dem */
            /* ganzen Programm als Mutter-Maske          */
         }
      }
   Move (RastPort,0,RastPort->TxBaseline);
   Text (RastPort,"                                    ",29);
}

/*********************************************************/
/* Diese Routine berechnet und zeichnet das zwei-        */
/* dimmensionale Bild aus den dreidimmensionalen         */
/* Koordinaten.                                          */
/*-------------------------------------------------------*/
/* Eingabe-Parameter: rot = Anzahl der Rotationen        */
/*                    ko  = Anzahl der Koordinaten des   */
/*                          Umrisses                     */
/*-------------------------------------------------------*/
/* Rückgabe-Werte: keine                                 */
/*********************************************************/

VOID Show(rot,ko)
long rot,ko;
{
   long i,j,t;

   SetRast(RastPort,0);

   for (i=0; i<rot; i++)
      for (j=0; j<ko; j++)
         {
           t = (d-Punkte[i][j][2])/(ProzZ-Punkte[i][j][2]);
           x[i][j] = WIDTH/2-Punkte[i][j][0]+
                          (ProzX-Punkte[i][j][0])*t;
           y[i][j] = HEIGHT/2-Punkte[i][j][1]+(ProzY-
                     Punkte[i][j][1])*t;
                          /* 3d -> 2d transformation */

           if (j>0)
              {
```

```
                Move(RastPort,x[i][j-1],y[i][j-1]);
                Draw(RastPort,x[i][j],y[i][j]);
                                /* 'Längsverstrebungen' */
            }

        if (i>0)
            {
                Move(RastPort,x[i-1][j],y[i-1][j]);
                Draw(RastPort,x[i][j],y[i][j]);
                                /* 'Querverstrebungen' */
            }
        }

   for (j=0; j<ko; j++)
      {
         Move(RastPort,x[0][j],y[0][j]);
         Draw(RastPort,x[rot-1][j],y[rot-1][j]);
                            /* letzte 'Querverstrebung' */
      }
}

/*********************************************************/
/* Diese Routine berechnet die mittlere Z-Koordinate einer*/
/* angegebenen Fläche                                     */
/*-------------------------------------------------------*/
/* Eingabe-Parameter: i = Welche Fläche                   */
/*-------------------------------------------------------*/
/* Rückgabe-Werte: Mittlere z-Koordinate der Fläche i     */
/*********************************************************/

long Average(i)
long i;
{
   long a,b,c,d;

   a=Area[i][0][2];
   b=Area[i][1][2];
   c=Area[i][2][2];
   d=Area[i][3][2];

   a = a+b+c+d;                     /* Bilde Summe und */
   return (a/4);                    /* Durchschnitt    */
}

/*********************************************************/
/* Diese Routine wandelt die angebene  Integer Zahl nach  */
/* ASCII um.                                              */
/*-------------------------------------------------------*/
```

```
/* Eingabe-Parameter: x = Integer                                    */
/*                    stellen = Wieviele Stellen für                 */
/*                              ASCII String insgesammt              */
/*-------------------------------------------------------------------*/
/* Rückgabe-Werte: Adresse des ASCII-Strings                         */
/*********************************************************************/

char *itoa(x,stellen)
long x,stellen;
{
   stellen--;
   if (x<0)
       {
         ASCString[0] = '-';
         x = -x;
       }
      else ASCString[0] = 32;                    /* ' ' */

   do {
         ASCString[stellen] = (x % 10)+'0';      /* + '0' */
         stellen--;
         x/=10;
      } while (stellen > 0);
   return (ASCString);
}

/*********************************************************************/
/* Diese Routine berechnet und zeichnet die Flächen des              */
/* Rotationskörpers                                                  */
/*-------------------------------------------------------------------*/
/* Eingabe-Parameter: rot - Anzahl der Rotationen                    */
/*                    ko - Anzahl der Punkte pro                     */
/*                         Rotationslinie                            */
/*                    Mode - Farbschatierung ?                       */
/*                    Status - Flächen neuberechnen ?                */
/*-------------------------------------------------------------------*/
/* Rückgabe-Werte: keine                                             */
/*********************************************************************/

VOID Shade(rot,ko,Mode,Status)
long rot,ko,Mode,Status;
{
   long i,j,k;
   long count;          /* Anzahl Flächen des Rot. Körpers */
   long hilf;
   long x,y;
   long t;
   long a,b,c,e;
   long w,col;
```

```
count = 0;

IDCMPoff();          /* Keine Nachrichten von Intuition */

if (Status == FALSE)           /* Flächenneuberechnung */
{                              /* und Neusortierung    */
Move (RastPort,0,RastPort->TxBaseline);
Text (RastPort,"Berechne Flächen... ",20);

for (i=0; i<(rot-1); i++)
   for (j=0; j<(ko-1); j++)
      {
         for (k=0; k<3; k++)
            {
               Area[count][0][k]=Punkte[i][j][k];
               Area[count][1][k]=Punkte[i][j+1][k];
               Area[count][2][k]=Punkte[i+1][j+1][k];
               Area[count][3][k]=Punkte[i+1][j][k];
            }
         count++;
      }


   for (j=0; j<(ko-1); j++)
      {
         for (k=0; k<3; k++)
            {
               Area[count][0][k]=Punkte[rot-1][j][k];
               Area[count][1][k]=Punkte[rot-1][j+1][k];
               Area[count][2][k]=Punkte[0][j+1][k];
               Area[count][3][k]=Punkte[0][j][k];
            }
      count++;
   }

Move (RastPort,0,RastPort->TxBaseline);
Text (RastPort,"Sortiere Flächen... ",20);

x = RastPort->cp_x;
y = RastPort->cp_y;

for (i=0; i<count; i++) Index[i+1] = i;
                        /* Index-Array initialisieren */
for (i=2; i<count+1; i++)
   {
         /* Nullte Area Komponente fürs sortieren !!! */
      hilf = Average(Index[i]);
      Index[0] = Index[i];

      j = i-1;
```

```
      while (hilf < Average(Index[j]))
         {
            Index[j+1] = Index[j];
            j--;
         }
      Index[j+1] = Index[0];

      Move(RastPort,x,y);
      Text(RastPort,itoa((count-i),5),5);
   }

}

count = (ko-1)*rot+1;              /* 'count' muss immer */
SetRast(RastPort,0);               /* bestimmt sein      */

col = 0;
if (Mode == 0)
  {
     SetAPen (RastPort,0);    /* Flächen löschen und    */
     SetOPen (RastPort,9);    /* nur Umrandung zeichnen */
  }
  else
  {
     w = count/9;                  /* Graustufe berechnen */
     SetOPen(RastPort,0);          /* keine Umrandung     */
  }

   for (i=1; i<count; i++)
    {
      e=Index[i];
      for (j=0; j<4; j++)            /* Flächen Zeichnen */
         {
            a=Area[e][j][0];
            b=Area[e][j][1];
            c=Area[e][j][2];

            t = (d-c)/(ProzZ-c);
            x = WIDTH/2-a+(ProzX-a)*t;
            y = HEIGHT/2-b+(ProzY-b)*t;

            if (Mode == 1)              /* in Grau ? */
               {
                 if (i == w)
                   {
                     w += count/9;
                     col ++;
                   }
```

```
                SetAPen (RastPort,4+col);
              }

            if (j==0) AreaMove (RastPort,x,y);
                                    /* Start Polygon */
            else AreaDraw (RastPort,x,y);
                                /* Neuer Polygon Punkt */
          }
        AreaEnd(RastPort);
                            /* Polygonende + zeichnen */
      }
    SetAPen (RastPort,9);
}

/*********************************************************/
/* Diese Routine initialisiert die TmpRas und AreaInfo   */
/* Struktur für den RastPort.                            */
/*-------------------------------------------------------*/
/* Eingabe-Parameter: keine                              */
/*-------------------------------------------------------*/
/* Rückgabe-Werte: keine                                 */
/*********************************************************/

VOID Init()
{
   InitArea (&AreaInfo,AreaBuffer,5);
                           /* immer 4 (+1) Koordinaten */

   if ((Pointer = (long *)AllocRaster(WIDTH,HEIGHT/2)) ==0)
      CloseIt("No free space !!!");
                                /* Speicher für TmpRas */

   RastPort->AreaInfo = &AreaInfo;
   RastPort->TmpRas = (struct TmpRas *)
    InitTmpRas (&TmpRas,Pointer,RASSIZE(WIDTH,HEIGHT/2));
}

/*********************************************************/
/* Diese Routine berechnet die neue Lage des Rotations-  */
/* körpers.                                              */
/*-------------------------------------------------------*/
/* Eingabe-Parameter: WinkelX, WinkelY, WinkelZ bestimmen */
/*                     die neue Lage des Körpers         */
/*                    rot - Anzahl der Rotationen        */
/*                    ko - Anzahl der Linien pro Rotations*/
/*                         Linie                         */
/*-------------------------------------------------------*/
/* Rückgabe-Werte: keine                                 */
/*********************************************************/
```

```
VOID Rot(WinkelX,WinkelY,WinkelZ,rot,ko)
long WinkelX, WinkelY, WinkelZ,rot,ko;
{
   long sx,cx,sy,cy,sz,cz;
   long i,j;
   long hx,hy,hz;

      Move (RastPort,0,RastPort->TxBaseline);
      Text (RastPort,"Berechne neue Position...",25);

      sx=sin(WinkelX);          /* Berechne sin(x,y,z) und */
      cx=cos(WinkelX);          /* cos (x,y,z) nur einmal  */
      sy=sin(WinkelY);
      cy=cos(WinkelY);
      sz=sin(WinkelZ);
      cz=cos(WinkelZ);

      for (i=0; i<rot; i++) /* Berechne Global-Drehungen */
         for (j=0; j<ko; j++)
            {
               hy = Mother[i][j][1]*cx/16384-
                    Mother[i][j][2]*sx/16384;
               hz = Mother[i][j][1]*sx/16384+
                    Mother[i][j][2]*cx/16384;
               Punkte[i][j][0] = Mother[i][j][0];
               Punkte[i][j][1] = hy;
               Punkte[i][j][2] = hz;

               hx = Punkte[i][j][0]*cy/16384+
                        Punkte[i][j][2]*sy/16384;
               hz =-Punkte[i][j][0]*sy/16384+
                        Punkte[i][j][2]*cy/16384;
              Punkte[i][j][0] = hx;
              Punkte[i][j][2] = hz+bbpz;

               hx = Punkte[i][j][0]*cz/16384-
                        Punkte[i][j][1]*sz/16384;
               hy = Punkte[i][j][0]*sz/16384+
                        Punkte[i][j][1]*cz/16384;
               Punkte[i][j][0] = hx+bbpx;
               Punkte[i][j][1] = hy+bbpy;
         }
      Move (RastPort,0,RastPort->TxBaseline);
      Text (RastPort,"                    ",25);
}

/*********************************************************/
/* Diese Funktion holt den Sinuswert aus Tabelle         */
/*-------------------------------------------------------*/
/* Eingabe-Parameter: x für sin(x)                       */
```

```
/*-------------------------------------------------------*/
/* Rückgabe-Werte: sin(x) * 16384                        */
/*******************************************************/

long sincos(x)
long x;
{
   long factor = 1;

   x %= 360;
   if (x>180)                          /* x > 180 Grad */
      {
         x -= 180;
         factor = -1;
      }
   if (x>90) x = 180 - x;
   return (sintab[x]*factor);
}

/*******************************************************/
/* Diese Funktion berechnet den Sinus von x            */
/*-------------------------------------------------------*/
/* Eingabe-Parameter: x                                */
/*-------------------------------------------------------*/
/* Rückgabe-Wert: sin(x) * 16384                       */
/*******************************************************/

long sin(x)
long x;
{
   return (sincos(x));
}

/*******************************************************/
/* Diese Funktion berechnet den Cosinus von x          */
/*-------------------------------------------------------*/
/* Eingabe-Parameter: x                                */
/*-------------------------------------------------------*/
/* Rückgabe-Wert: cos(x) * 16384                       */
/*******************************************************/

long cos(x)
long x;
{
   return (sincos(x+90));          /* cos(x) = sin (90 + x) */
}

/*******************************************************/
/* Diese Funktion erlaubt endlich die Eingabe des Umrisses*/
/*-------------------------------------------------------*/
```

```
/* Eingabe-Parameter: rot - Wieviele Rotationen sind      */
/*                    zugelassen                          */
/*-------------------------------------------------------*/
/* Rückgabe-Werte: Anzahl der einegegeben Koordinaten     */
/*********************************************************/

long EnterKoords(rot)
long rot;
{
   long Class,                  /* zur Spezifizierung der */
        Code;                   /* Intuition Nachricht    */

   long Position;       /* Position an der die Koordinaten */
   long x,y;            /* abgespeichert werden.           */

   long oldkoorx,oldkoory,oldx,oldy;

   long Ende = FALSE;                  /* Eingabe - Ende ? */

   ModifyIDCMP(Window, MOUSEMOVE | MOUSEBUTTONS | RAWKEY);
                          /* Eingabe braucht nur Maus */
                          /* und Tastatur             */

   SetRast(RastPort,0);                    /* BitMap löschen */

   SetAPen (RastPort,9);          /* APen, BPen und DrawMode */
   SetBPen (RastPort,0);          /* festlegen               */

   SetDrMd (RastPort,JAM2);

   Position = 0;   /* Wo werden Koordinaten gespeichert ? */

   Move(RastPort,WIDTH/2,0);                 /* Achsenkreuz */
   Draw(RastPort,WIDTH/2,HEIGHT);

   Move(RastPort,0,HEIGHT/2);
   Draw(RastPort,WIDTH,HEIGHT/2);

   x=Screen->MouseX; /* aktuelle Mausposition für Ausgabe */
   y=Screen->MouseY;

   while (!Ende && (Position < MAXKOORDS))
      {
        if (1 << Window->UserPort->mp_SigBit)
                  /* Nachricht von Intuition gekommen ? */

        while (Message = (struct IntuiMessage *)
                GetMsg(Window->UserPort))          /* Ja */
           {
           Class = Message->Class;  /* Hole die nötigen */
```

```
    Code = Message->Code;     /* Daten          */
    x = Message->MouseX;
    y = Message->MouseY;
    ReplyMsg(Message);          /* Habe Nachricht */
                                /* empfangen !    */

    if (Position == 0)
        {
          oldx = x;
          oldy = y;
        }

    switch (Class)
      {
        case MOUSEBUTTONS:
          if (Code == SELECTDOWN)
            {
                    /* Koordinaten relativ zu */
                    /* Achsenkreuz !!!        */

              if (Position == 0)
                {
                   oldkoorx = x;
                   oldkoory = y;
                }

              KoordArray[Position][0] =
                          x-WIDTH/2;
              KoordArray[Position][1] =
                            HEIGHT/2-y;

              SetDrMd (RastPort,JAM2);
              Move (RastPort,oldkoorx,oldkoory);
              Draw (RastPort,x,y);

                  /* kleines Kreuz zeichnen */
              Move (RastPort,x-3,y-3);
              Draw (RastPort,x+3,y+3);

              Move (RastPort,x-3,y+3);
              Draw (RastPort,x+3,y-3);

              Move (RastPort,x,y);
                    /* alte Cursorposition */

              oldkoorx = x;
              oldkoory = y;

              Position ++;
            }
```

```
                break;

                case MOUSEMOVE:

                 if (Position > 0)
                   {
                    if ((x != oldx) || (y != oldy))
                      {
                         SetDrMd(RastPort,COMPLEMENT|JAM1);

                         Move (RastPort,oldkoorx,oldkoory);
                         Draw (RastPort,oldx,oldy);

                         Move (RastPort,oldkoorx,oldkoory);
                         Draw (RastPort,x,y);
                               /* Linie zu aktueller     */
                               /* Mausposition zeichnen */

                         oldx = x;
                         oldy = y;
                      }
                   }
                break;

                case RAWKEY:
                   if ((Code == 0x50) && (Position > 0))
                        Ende = TRUE;
                                        /* F1 gedrückt */
                break;

            }

        }
        SetDrMd(RastPort,JAM2);
                          /* Mausposition ausgeben */

        Move (RastPort,3*WIDTH/4,RastPort->TxBaseline);
        Text (RastPort,"X: ",3);
        Text (RastPort,itoa((x-WIDTH/2),4),4);
        Text (RastPort," Y: ",4);
        Text (RastPort,itoa((HEIGHT/2-y),4),4);
}

SetDrMd(RastPort,COMPLEMENT|JAM1);

Move (RastPort,oldkoorx,oldkoory);
Draw (RastPort,oldx,oldy);

SetDrMd(RastPort,JAM2);
```

```
   Rotate(rot,Position);     /* Rotationskörper berechnen */
   return (Position);
}

/*********************************************************/
/* Diese Routine sorgt dafür, dass der Benutzer seinen   */
/* Rotationskörper steuern kann                          */
/*-------------------------------------------------------*/
/* Eingabe-Parameter: rot - Anzahl der Rotationslinien   */
/*                    ko - Anzahl der Punkte pro         */
/*                         Rotationslinie                */
/*-------------------------------------------------------*/
/* Rückgabe-Werte: keine                                 */
/*********************************************************/

VOID make_menu(rot,ko)
long rot,ko;
{
   long Ende = FALSE;                  /* Programmende ? */
   long First = TRUE;                /* erstes mal Shading ? */
   long Class,                        /* Intuition's Message */
        Code;
   long x,y;

   IDCMPon();                         /* Nachrichten bitte ! */
   while (!Ende)
      {
        Wait (1 << Window->UserPort->mp_SigBit);
                                        /* Kommt was ? */
         while (Message = (struct IntuiMessage *)
                GetMsg(Window->UserPort))          /* Ja */
            {
            Class = Message->Class;
            Code = Message->Code;
            ReplyMsg(Message);
                        /* Wir haben's gekriegt, danke */

            switch (Class)
               {
                  case RAWKEY:
                     x=y=0;
                     switch (Code)
                        {
                           case 0x4c:      /* Pfeiltasten */
                              y=-1;
                           break;
                           case 0x4d:
                              y=1;
                           break;
```

```
                    case 0x4f:
                       x=-1;
                    break;
                    case 0x4e:
                       x=1;
                    break;
                  }

                 if ((x==0) && (y==0)) Status = TRUE;
                 else
                 {
                   IDCMPoff();   /* kein Intuition */
                   if ((y == -1) && (WinkelX == 0))
                      WinkelX=360;
                   if ((x == -1) && (WinkelZ == 0))
                      WinkelZ=360;
                   WinkelX += y*INCREMENT;
                   WinkelZ += x*INCREMENT;
                   Rot(WinkelX,WinkelY,WinkelZ,rot,ko);
                   Show(rot,ko);
                   IDCMPon();
                   First = TRUE;
                   Status = FALSE;
                 }
               break;

               case MENUPICK:
                  if (Code != MENUNULL)
                    switch(MENUNUM (Code))
                       {
                         case 0:
                           switch (ITEMNUM (Code))
                             {
/* Shading */                    case 0:
                                   IDCMPoff();
                                   if ((First) ||
                                       (!Status))
                                   Shade(rot,ko,1,FALSE);
                                   else
                                   Shade(rot,ko,1,TRUE);
                                   First = FALSE;
                                   Status = TRUE;
                                   IDCMPon();
                                 break;

/* Hide It */                    case 1:
                                   IDCMPoff();
                                   if ((First) ||
                                      (!Status))
                                   Shade(rot,ko,0,FALSE);
```

```
                                    else
                                    Shade(rot,ko,0,TRUE);
                                    First = FALSE;
                                    Status = TRUE;
                                    IDCMPon();
                                 break;

      /* Neue Koordinaten */      case 2:
                                    IDCMPoff();
                                    ko = EnterKoords
                                         (MAXROTS);
                                    WinkelX=0;
                                    WinkelZ=0;
                                        /* WinkelY=0; */
                                    Show(rot,ko);
                                    First = TRUE;
                                    Status = FALSE;
                                    IDCMPon();
                                 break;

        /* Hardcopy */            case 3:
                                    IDCMPoff();
                                    hardcopy();
                                    IDCMPon();
                                 break;

        /* Quit */                case 4:
                                    Ende = TRUE;
                                 break;
                               }
                            }
                         }
                      }

   }
}


/*********************************************************/
/* Ab hier gehts mit der Hardcopy Routine los ! Die beiden*/
/* folgenden Funktionen werden für 'DumpRastPort'        */
/* gebraucht. (Externer Input/OutPut)                    */
/*********************************************************/

struct IORequest *CreateExtIO(ioReplyPort,size)
struct MsgPort *ioReplyPort;
long size;
{
   struct IORequest *ioReq;
```

```
    if (ioReplyPort == NULL) return (NULL);
    ioReq = (struct IORequest *)AllocMem (size,MEMF_CLEAR);
    if (ioReq == NULL) return (NULL);
    ioReq->io_Message.mn_Node.ln_Type = NT_MESSAGE;
    ioReq->io_Message.mn_Length = size;
    ioReq->io_Message.mn_ReplyPort = ioReplyPort;
    return (ioReq);
}

VOID DeleteExtIO (ioExt)
struct IORequest *ioExt;
{
    if (ioExt)
        {
            ioExt->io_Message.mn_Node.ln_Type = 0xff;
            ioExt->io_Device = (struct Device *)-1;
            ioExt->io_Unit = (struct Unit *)-1;
            FreeMem (ioExt,ioExt->io_Message.mn_Length);
        }
}

/*********************************************************/
/* Ab hier steht die eigentlich Hardcopy-Routine         */
/*********************************************************/

VOID hardcopy()
{
    union printerIO
        {
            struct IOStdReq ios;
            struct IODRPReq iodrp;
            struct IOPrtCmdReq iope;
        };

    union printerIO *request;
    struct MsgPort *printerPort;

        printerPort = CreatePort ("printer.port",0);
        request = (union printerIO *)CreateExtIO (printerPort,
                        sizeof(union printerIO));

        if (OpenDevice("printer.device",0,request,0) != 0)
          {
              DeleteExtIO(request);      /* Printer 'öffnen' */
              DeletePort(printerPort);
              CloseIt("Sorry, no printer !!!");
          }

        request->iodrp.io_Command = PRD_DUMPRPORT;
                                  /* DumpRastPort Kommando */
```

```
    request->iodrp.io_RastPort = RastPort;
                              /* Welcher RastPort ? */

    SetRGB4(&Screen->ViewPort,0,15,15,15);
                        /* Hintergrundfarbe = Weiss */

    request->iodrp.io_ColorMap = Screen->ViewPort.ColorMap;
                  /* für Schattierung auf dem Printer */

    request->iodrp.io_Modes = NewScreen.ViewModes;
                              /* Welche ViewModi ? */
    request->iodrp.io_SrcX = 0;    /* linke, obere Ecke */
    request->iodrp.io_SrcY = 0;
    request->iodrp.io_SrcWidth = WIDTH;
    request->iodrp.io_SrcHeight = HEIGHT;
                                   /* rechte, untere */
                                   /* Ecke           */
    request->iodrp.io_DestCols = 0;
    request->iodrp.io_DestRows = 0;
    request->iodrp.io_Special = 0x004;          /* Flag */

    DoIO(request);                            /* Printen */

    CloseDevice(request);
    DeleteExtIO(request);          /* alles schliessen */
    DeletePort(printerPort);
    SetRGB4(&Screen->ViewPort,0,0,0,0);
                        /* Hintergrund wieder normal */
}

/*********************************************************/
/* Jetzt kommt das Hauptprogramm:                        */
/*********************************************************/

main ()
{
   long Koords;

   OpenLibs();                          /* Öffne Libs */

   Koords = EnterKoords(MAXROTS);      /* Hole Umriss */
   Show(MAXROTS,Koords);              /* Zeichne Körper */

   make_menu(MAXROTS,Koords);                /* Menu */

   CloseIt("Bye Bye");                     /* Tschüs */
   return (0);
}
```

Ein paar Hinweise zum Programm:

Nach dem Start können Sie mit der Maus den Umriß eines beliebigen Rotationskörpers eingegeben. Wenn Sie alle Punkte, die den Umriß bestimmen, eingegeben haben, können Sie nach dem Druck von 'F1' den dreidimensionalen Körper auf Ihrem Monitor sehen (Ist die Anzahl der eingegebenen Koordinaten gleich MAXKOORDS, wird dies automatisch, also ohne Tastendruck, vorgenommen).

Mit den Cursor-Tasten können Sie dann diesen Körper im Raum drehen.

Jedoch ist das noch nicht alles, was Sie mit diesem Rotationskörper auf Ihrem Bildschirm anfangen können. Wählen Sie nämlich den Menüpunkt "Shading", wird der Rotationskörper in verschiedenen Graustufen schattiert.

Dies geschieht nach einem recht einfachen Algorithmus: Zuerst wird eine Flächenliste berechnet. Diese Flächenliste enthält die Koordinaten der Eckpunkte aller zu füllenden Flächen. Dann werden diese Flächen nach ihrer mittleren Z-Koordinate sortiert. Danach wird die Fläche, die am weitesten vom Betrachter entfernt ist, zuerst gezeichnet (natürlich mit Area...).

Der Schattierungseffekt wird dadurch erreicht, daß einfach nach einer bestimmten Anzahl von Flächen ein Farbzähler inkrementiert wird, der die Farbe der zu zeichnenden Flächen bestimmt.

Der Menüpunkt "Hide It" funktioniert ähnlich wie "Shading", nur wird hier auf eine Schattierung verzichtet, was bedeutet, daß die Füllfarbe aller Flächen gleich der Hintergrundfarbe ist und daß die Umrandungsfarbe im OPen sich vom Hintergrund unterscheidet. 'Hide It' simuliert einen 'Hidden-Line'-Algorithmus.

Gefällt Ihnen der gerade aktuelle Rotationskörper nicht mehr, können Sie nach Anwahl des Menüpunktes 'Koordinaten' einen neuen Rotationskörper eingeben.

Die Menüpunkte 'Hardcopy' und 'Quit' erklären sich von selbst.

Bitte beachten Sie, daß die Routine zur Berechnung der Lage des Rotationskörpers sehr allgemein gehalten ist. Sie könnten auch eigene Objekte, nicht nur Rotationskörper, in Form von dreidimensionalen Koordinaten angeben, und diese mit Hilfe der Rot- und einer eigenen, etwas modifizierten Show-Routine drehen und darstellen.

14. Die Colormap-Befehle

Nachdem Sie jetzt in der Lage sind, mit den Grafik-"Primitiva", also den einfachen Grafikbefehlen, Grafiken zu erzeugen, wollen wir nun ein wenig tiefer in die Farbdarstellung des Amiga "eintauchen":

Denn Sie können zwar jetzt schon dafür sorgen, daß verschiedenfarbige Punkte auf den Bildschirm gemalt werden (durch Änderung des Vordergrundstifts), aber die Farbe selbst, also die Farbe des Farbregisters, mit der ja ein Pixel dargestellt wird, können Sie noch nicht ändern.

Dafür brauchen wir aber nur den entsprechenden Farbeintrag in der Colormap des Viewports zu verändern und dafür zu sorgen, daß diese Änderung sich auch in den Copper-Listen ausdrückt.

Wie sieht aber solch ein Farbeintrag aus? Grundsätzlich besteht er aus 16 Bits, also einem Word. Jedoch werden beim Amiga zur Zeit nur die untersten 12 Bits genutzt, und zwar so, daß in den Bits 11-8 die Rot-Komponente, in den Bits 7-4 die Grün- und in den Bits 3-0 die Blau-Komponente der Farbe enthalten ist. Aus den Farben Rot, Grün und Blau (daher auch z.B. der Ausdruck RGB-Monitor für Farbmonitor) kann man durch additive Mischung (fast) jeden Farbton herstellen. Beim Amiga kann man $2^{12} = 16^3 = 4096$ verschiedene Farben darstellen.

Sind alle vier Bits der drei Farbkomponenten gesetzt, wird die Farbe Weiß dargestellt (R=15, G=15, B=15). Sind hingegen alle Bits gelöscht, wird die Farbe Schwarz erzeugt (R=0, G=0, B=0).

Farbänderungen nimmt man dann wie folgt vor:

14.1 Eine neue Farbpalette gewünscht?

'LoadRGB4 (&ViewPort, &Palette, Farbeinträge)' sorgt dafür, daß aus dem angegebenen Speicherbereich (Palette) 'Farbeinträge' Words in den dazugehörigen Speicherbereich der Colormap des angegebenen Viewports geladen werden.

Meist ist dieser Speicherbereich ein Word-Array, das von Ihnen initialisiert wurde. Die Werte dieses Arrays werden in die Colormap des angegebenen Viewports geschrieben. Eine Farbänderung ist aber noch nicht zu erkennen. Dies liegt daran, daß der Teil der Copper-Liste,

der die Farbregister 'bearbeitet', noch nicht auf dem neuesten Stand ist.

Das wird aber dadurch erreicht, daß mit MakeVPort, MrgCop und LoadView (RemakeDisplay bei Intuition-Screens) die Copper-Liste komplett neuberechnet wird.

14.2 Oder reicht Ihnen eine einzige Farbänderung?

Bei dem nun folgenden Befehl haben Sie das Problem nicht, die Copper-Liste neu zu berechnen. Denn 'SetRGB4 (&ViewPort, FarbReg, Rot, Grün, Blau)' ändert die Farbe des angegebenen Farbregisters nicht nur in der ColorMap, sondern auch in der Copper-Liste. Die neue Farbe, die ja durch additive Mischung der Farben Rot, Grün und Blau zustande kommt, wird sofort auf dem Bildschirm sichtbar (wenn Punkte in dieser Farbe gezeichnet wurden). Allerdings kann mit SetRGB4 nur die Farbänderung eines einzelnen Farbregisters hervorgerufen werden, die allerdings sofort geschieht.

Natürlich nehmen auch hier die Komponenten Rot, Grün und Blau Werte zwischen 0 und 15 an (%0000 und %1111).

14.3 Welche Farben sind im Angebot?

Das Gegenstück zu SetRGB4 und LoadRGB4 ist 'Farbe = GetRGB4 (ViewPort.ColorMap, Farbreg)'. Dieser Befehl liefert nämlich das Word in 'Farbe', das im angegebenen Farbregister steht. Aus diesem Word können Sie dann mitels der folgenden kleinen Operationen wieder die Rot-, Grün- und Blau-Komponente extrahieren.

```
Rot  = (Farbe>>8) & 15
Grün = (Farbe>>4) & 15
Blau =  Farbe     & 15
```

Folgendes kleine Programm macht sich diese Umrechnung zunutze. Hier wird nämlich der Hintergrund, dessen Farbe ja in Farbregister 0 steht, immer mit der Farbe des Punktes unter dem Mauszeiger eingefärbt:

```
/*********************************************************/
/*                    SimpleColor.c                      */
/*                                                       */
/* Dieses Programm benutzt ReadPixel(), Load-, Get- und  */
/* SetRGB4().                                            */
/*                                                       */
/* Compiled with: Lattice V3.10                          */
/*                                                       */
/* (c) Bruno Jennrich                                    */
/*********************************************************/

#include "exec/types.h"
#include "exec/memory.h"
#include "exec/devices.h"
#include "graphics/gfx.h"
#include "graphics/text.h"
#include "graphics/regions.h"
#include "graphics/gfxbase.h"
#include "graphics/gfxmacros.h"
#include "graphics/copper.h"
#include "graphics/gels.h"
#include "intuition/intuition.h"
#include "devices/keymap.h"
#include "hardware/blit.h"


#define Width 320
#define Height 200
#define Depth 5
#define MODES 0

struct GfxBase *GfxBase;
struct IntuitionBase *IntuitionBase;

struct Screen *Screen;
struct Window *Window;
struct IntuiMessage *Message;
struct RastPort *RPort;

struct NewScreen NewScreen =
           {0,0,
            Width,Height,Depth,
            0,1,
            MODES,
            CUSTOMSCREEN,
            NULL,
            NULL,
            NULL,NULL
           };
```

```
struct NewWindow NewWindow =
          {0,0,
           Width,Height,
           0,1,
           NULL,
           ACTIVATE|BORDERLESS,
           NULL,NULL,
           "Simple-Color-Selection",
           NULL,
           NULL,
           NULL,NULL,NULL,NULL,
           CUSTOMSCREEN
          };

char string[16][4] = {
                      {"0  "},{"1  "},{"2  "},{"3  "},
                      {"4  "},{"5  "},{"6  "},{"7  "},
                      {"8  "},{"9  "},{"A  "},{"B  "},
                      {"C  "},{"D  "},{"E  "},{"F  "}
                     };
                                          /* Hex-Zahlen */
UWORD ROT,
      GRUEN,      /* Rot-, Grün, Blaukomponente der Farbe */
      BLAU;

UWORD dummy;

int Length, x, y, i;

char text[] = "R  G  B";

UWORD Colors[] =
          {
             0x0200,0x0405,0x060A,0x080F,
             0x0214,0x0419,0x061E,0x0823,
             0x0228,0x042d,0x0632,0x0837,
             0x023C,0x0441,0x0646,0x084B,
             0x0250,0x0455,0x065A,0x085F,
             0x0264,0x0469,0x066E,0x0873,
             0x0278,0x047D,0x0682,0x0887,
             0x028C,0x0491,0x0696,0x089B,
             0x02A0,0x04A5,0x06AA,0x08AF
          };                          /* eigene ColorMap */

char *LeftMouse = (char *) 0xbfe001;

extern struct IntuiMessage *GetMsg();
```

```
/*********************************************************/
/* Es geht los !                                         */
/*********************************************************/

main()
{
   if((GfxBase = (struct GfxBase *)
      OpenLibrary("graphics.library",0)) == NULL)
      {
         printf(" Keine Graphics !!!!!\n");
         Exit (0);
      }

   if((IntuitionBase = (struct IntuitionBase *)
      OpenLibrary("intuition.library",0))== NULL)
      {
         printf(" Kein Intuition !!!!!\n");
         goto cleanup1;
      }

   if ((Screen = (struct Screen *)OpenScreen(&NewScreen))
            == NULL)
      {
         printf("Kein Screen !!!!!!\n");
         goto cleanup2;
      }

   NewWindow.Screen = Screen;

   if ((Window = (struct Window *)
       OpenWindow(&NewWindow)) == NULL)
      {
         printf("Kein Window !!!!!!\n");
         goto cleanup3;
      }

   RPort = Window->RPort;

   LoadRGB4(&Screen->ViewPort, &Colors[0], 32);
   RemakeDisplay();              /* Lade eigene ColorMap */
                                 /* und stelle diese dar */
   for (i=0;i<32;i++)
      {
         SetAPen(RPort,i);
         RectFill(RPort, i*(Width/32),(Height/100*90),
                         (i+1)*(Width/32)-1,Height-1);
      }                                  /* Male Rechtecke */

   Length = TextLength(RPort,text,7);
   x = (Width/2)-(Length/2);
```

```
    y = (Height/2)+RPort->TxBaseline;
                                    /* String zentrieren */

    while ((*LeftMouse & 0x40) == 0x40)
       {
            dummy = ReadPixel(RPort,Screen->MouseX,
                                    Screen->MouseY);
               /* Lese Farbe des Punktes unter Mauszeiger */

            SetAPen(RPort,dummy + 15);
            dummy =
                GetRGB4(Screen->ViewPort.ColorMap,dummy);

            ROT = (dummy >> 8) & 15;
            GRUEN = (dummy >> 4) & 15;
            BLAU = dummy & 15;
                              /* Extrahiere Komponenten */

            Move (RPort,x,y);
            Text (RPort,&text[0],7);
            Move (RPort,x,y+20);
                              /* Text Position setzen */

            Text (RPort,&string[ROT][0],3);
            Text (RPort,&string[GRUEN][0],3);
            Text (RPort,&string[BLAU][0],3);
                                    /* Text ausgeben */

            SetRGB4(&Screen->ViewPort,0,ROT,GRUEN,BLAU);
       }

             CloseWindow(Window);
    cleanup3: CloseScreen(Screen);
    cleanup2: CloseLibrary(GfxBase);
    cleanup1: CloseLibrary(IntuitionBase);
    return (0);
    /* Tschüs !!!*/
}
```

Noch ein letztes Wort zu den Namen der Colormap-Befehle:

Die Befehle Load-, Get- und SetRGB4 verdanken die '4' in ihrem Namen nämlich der Tatsache, daß eine Farbkomponente aus 4 Bits besteht.

14.4 Welche Farbe hat denn der Punkt?

Für obiges Programm mußten wir natürlich irgendwie auch an die Farbe des Punktes unterhalb des Mauszeigers herankommen. Dabei half uns der 'FarbReg = ReadPixel (&RastPort, x, y)'-Befehl, der uns die Nummer des Farbregisters liefert, mit dessen Farbe der Punkt an der Koordinate x/y eingefärbt wurde. Dann konnten wir ganz einfach mit GetRGB4 den Eintrag dieses Farbregisters auslesen, die 3 Farbkomponenten extrahieren und mit SetRGB4 in das Hintergrundfarbregister 0 schreiben.

15. Texte ausgeben

Nun reicht es für die verschiedenen Grafik-Anwendungen meistens nicht aus, nur Linien zu ziehen und Punkte zu setzen. Es ist oft der Fall, daß bestimmte Teile einer Grafik beschriftet werden müssen. Eine Möglichkleit dazu wäre, die einzelnen Zeichen mittels Linien nachzubilden. Dies ist aber sehr mühsam.

Deshalb sollte man auf die eingebaute Textausgabe-Routine zurückgreifen:

Mit 'Text (&RastPort, "String", Anzahl_der_Zeichen)' wird der angegebene String an der aktuellen Grafik-Cursor-Position ausgegeben. Die 'Anzahl der Zeichen' bestimmt, wieviele Zeichen des Strings ausgegeben werden sollen.

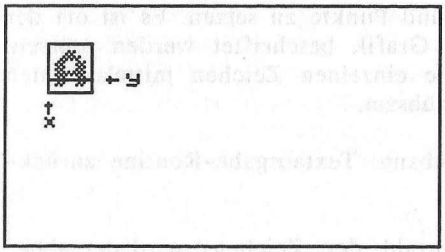
Hierzu vielleicht direkt einen kleinen Tip: Bei der Anzahl der Zeichen hat man sich sehr schnell verzählt. Warum also nicht den Compiler diese Anzahl berechnen lassen? Mit 'Anzahl = strlen ("String")' ist das gar kein Problem. Dieser Befehl ist fast in jeder Standard-Bibliothek eines jeden Compilers zu finden ('c.lib' beim Lattice-Compiler).

So kann man also mit 'Text (&RastPort, "String", strlen("String"))' auf einfache Art und Weise für die Ausgabe eines Textes sorgen.

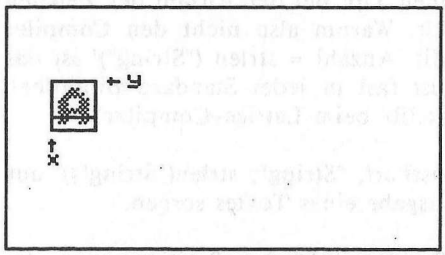
Leider gibt es aber noch ein Problem mit der Positionierung des Strings. Dazu wird zwar die aktuelle Grafik-Cursor-Position herangezogen, jedoch wird die oberste Zeile der Zeichen nicht an die Y-Koordinate der Cursor-Position positioniert. Aber die sogenannte 'Baseline' (Basislinie) wird an die aktuelle Position gesetzt. Dies hat zur Folge, daß der Text ein wenig hoch rutscht.

Addiert man aber zur gewünschten Y-Position des Textes noch die Baseline des aktuellen Zeichensatzes hinzu, wird die oberste Zeile des Strings positioniert:

```
Move (&RastPort,x,y);
```




```
Move (&RastPort,x,y+RastPort.TxBaseline);
```



15.1 Die Textlänge

Da es aber auch notwendig sein kann, die genaue Ausdehnung des Textes im Rastport zu kennen, kann man die Breite des Textes mittels 'Breite = TextLength (&RastPort, "String", strlen("String"))' erfahren. Wie Sie sehen, hat dieser Befehl die gleichen Parameter wie 'Text', nur wird hier kein String ausgegeben, sondern die Breite in Pixeln berechnet.

So können Sie z.B. in CAD-Anwendungen feststellen, ob der auszugebende String genügend Platz hat.

Man kann mit dieser 'Breite' aber auch dafür sorgen, daß der Text zentriert auf dem Bildschirm ausgegeben wird. Dazu berechnet man die X-Koordinate einfach wie folgt:

```
x = Breite_des_Bildschirms/2 - Breite_des_Strings/2
```

Mit 'Move (&RastPort, x, y)' legt man dann die Position fest, an der der Text mitten ausgegeben werden kann (Baseline beachten!).

15.2 Zeichensätze auf dem Amiga

Natürlich hat man sich bei Commodore nicht damit zufrieden gegeben, das Aussehen der Zeichen nur einmal zu definieren. Das würde zur 'Supermaschine Amiga' auch nicht passen.

In den Fonts, das sind 'gepackte', zusammengestauchte Daten-Arrays, sind die verschiedenen Zeichenformen abgespeichert. An diese kommt man über die TextAttr- (Text Attribute) und die TextFont-Strukturen heran:

15.3 Fonts werden geöffnet...

Dazu bestimmen Sie mit Hilfe der TextAttr-Struktur den Namen des zu öffnenden Fonts. Neben dem Namen legen Sie aber auch die gewünschte Größe fest, denn meistens existieren von einem Font mehrere Versionen, die sich alle nur in der Zeichengröße unterscheiden. Das beste Beispiel dafür ist der CLI-Font. In den Preferences haben Sie die Möglichkeit, zwischen 60 und 80 Zeichen pro Zeile zu wählen. Diese Wahlmöglichkeit wurde durch zwei unterschiedlich große Fonts des gleichen Namens ("topaz.font" mit 8 und 9 Zeilen Höhe) geschaffen.

Den Namen des Fonts, den Sie in 'TextAttr.ta_Name = "NAME"' angeben, muß einem File im 'Sys:Fonts'-Unterverzeichnis entsprechen. Dieses File, das das Anhängsel ".font" hat, ist der Header zu allen unterschiedlich großen Fonts dieses Namens.

Aufgrund dieses Headers kann nämlich entschieden werden, ob der angeforderte Font in Größe und Schriftart mit irgendeinem Font dieses Namens übereinstimmt. Sehen wir uns z.B. den 'Ruby.Font' an:

Im 'Sys:Fonts' Unterverzeichnis finden wir unter anderem:

```
"ruby.font" (Header File)
...
ruby (dir)
```

"ruby.font" ist das Header-File, das die Informationen über die Größe der einzelnen Fonts enthält. Im "ruby"-Unterverzeichnis finden wir hingegen diese Files:

```
ruby (dir):

   12   15
   8
```

Sie stehen repräsentativ für drei verschiedene Fonts mit annähernd gleichem Aussehen, die nur jeweils 12, 15 bzw. 8 Zeilen hoch sind.

In 'TextAttr.ta_YSize' gibt man nun einfach die Höhe des gewünschten Fonts an (z.B: 'TextAttr.ta_YSize = 8').

Danach kann man in 'TextAttr.ta_Style' noch festlegen, welche Schriftart (normal, unterstrichen, fett, kursiv) der Font haben soll. Dazu stehen folgende 'Font Style Flags' zur Verfügung:

FS_NORMAL	Der Font wurde ohne spezielle Schriftarten, also normal kreiert.
FSF_UNDERLINED	Die Zeichen des Fonts sind von vorneherein unterstrichen.
FSF_BOLD	Die Zeichen des Fonts sind fett.
FSF_ITALICS	Die Zeichen des Fonts sind kursiv.

In die Variable 'TextAttr.ta_Flags' muß der Benutzer eintragen, ob der Font in der Systemfontliste (FPF_ROMFONT) oder auf der Diskette (FPF_DISKFONT) zu finden ist. Wurde der Font mittels 'RemFont' aus der Systemfontliste gelöscht, ist in 'ta_flags' das 'FPF_REMOVED'-Bit gesetzt.

Im Moment gilt für uns allerdings, daß bis auf den Topaz-Font alle anderen Fonts auf Diskette zu finden sind.

Doch stellt sich jetzt die Frage, wie die Fonts geöffnet werden. Fonts aus der Systemfontliste, also der Topaz-Font, werden mit 'TextFont = OpenFont (&TextAttr)'geöffnet.

Über die TextFont-Struktur hat man dann Zugriff auf den Font. Mit 'SetFont (&RastPort, TextFont)' sorgt man dafür, daß die Zeichen, die mit 'Text' ausgegeben werden, im neuen Gewand erscheinen.

Will man aber einen Font, der nur auf der Diskette zu finden ist, öffnen, muß man zunächst die DiskFont-Library offnen. Dies geschieht mit 'DiskFontBase = (ULONG *) OpenLibrary ("diskfont.library", 0)'. Wie Sie vielleicht schon an dem Cast '(ULONG *)' erkennen konnten, ist DiskFontBase ein Zeiger auf eine ULONG-Variable. Es existiert also keine DiskFontBase-Struktur.

Nun haben Sie aber mit 'TextFont = OpenDiskFont (&TextAttr)' die Möglichkeit, einen diskettengestützen Font zu eröffnen. Diesen können Sie dann auch mit 'SetFont' für den 'Text'-Befehl zugänglich machen. Aber Sie können diesen Font auch mittels'AddFont (s. Anhang B) in die Systemfontliste einreihen, und Sie haben so durch den 'einfacheren' OpenFont-Befehl die Möglichkeit, diesen zu öffnen (wenn kein RemFont erfolgt ist).

15.4 ...und wieder geschlossen

Und natürlich müssen Sie auch die Fonts, wie z.B. auch die Libraries, nach Benutzung wieder schließen. Dazu gibt es den 'CloseFont (TextFont)'-Befehl, der übrigens auch Diskfonts schließt. Wenn Sie Diskfonts benutzen, müssen Sie weiterhin natürlich auch die DiskFont-Library wieder schließen (CloseLibrary (DiskFontBase)).

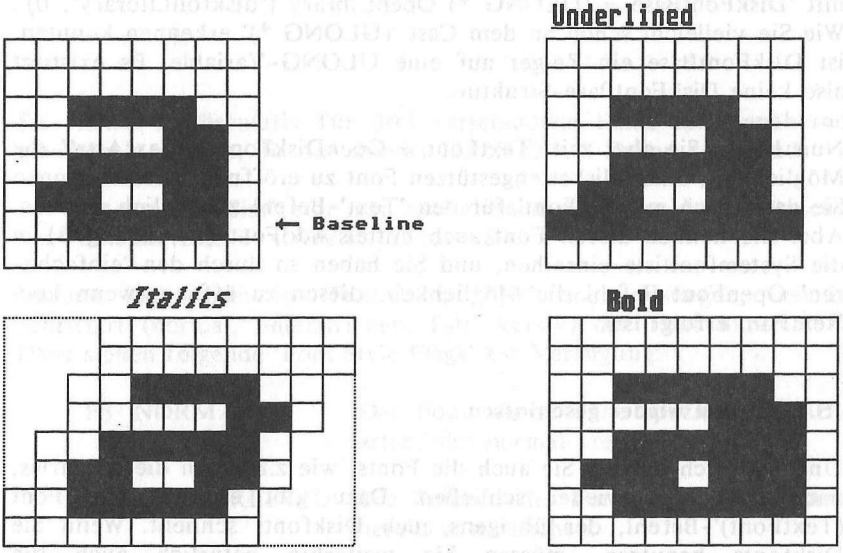
15.5 Softwaremäßig generierte Schriftarten

Oben, bei der Beschreibung der Variable 'TextAttr.ta_Style' haben wir schon von Schriftarten gesprochen. Allerdings waren diese fest im Bitmuster der einzelnen Zeichen verankert.

Hier wollen wir Ihnen aber zeigen, wie Sie nachträglich dafür sorgen können, daß ein Font z.B. kursive Zeichen enthält, obwohl die Bitmuster der Zeichen nichts Kursives an sich haben.

Dazu wird mittels 'SetSoftStyle (&RastPort, StyleBits, StyleEnable)' veranlaßt, daß bei der Ausgabe von Strings die einzelnen Zeichen, bevor sie in die Bitmap des Rastports geschrieben werden, einer Verformung unterzogen werden, z.B. unterstrichen, wenn 'StyleBits = FSF_UNDERLINED' ist.

Bei kursiven Zeichen werden dazu immer zwei Zeilen des Zeichens von unten nach oben um jeweils einen Punkt nach rechts verschoben:



Fette Zeichen werden eigentlich zweimal ausgegeben. Einmal an der angegebenen Position, und ein weiteres Mal an einer um einen Punkt verschobenen horizontalen Position.

Unterstrichene Zeichen werden einfach dadurch erreicht, daß die Baseline des Zeichens voll durchgezogen wird.

Diese softwaremäßige Zeichensatzverformung kommt auch dann zum Zuge, wenn ein Font bis auf die gewünschte Schriftart gefunden werden konnte. Es wird dann dafür gesorgt, daß die gewünschte Schriftart in 'TextAttr.ta_Style' mit 'SetSoftStyle' nachträglich gesetzt wird.

Doch wurde bisher die Bedeutung der Variablen 'StyleEnable' im 'SetSoftStyle'-Aufruf noch nicht erläutert. Diese Variable enthält die Style-Flags, die noch mit 'SetSoftStyle' gesetzt werden dürfen. Hat ein Font nämlich schon von Hause aus die Eigenschaft, kursiv zu sein, macht es keinen Sinn, mittels der Zeichensatzverformung eine doppelte Kursivität hervorrufen zu wollen (die Zeichen wären kaum noch lesbar).

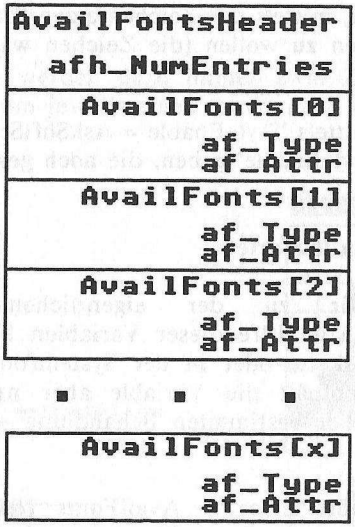
Man läßt sich also mittels 'StyleEnable = AskSoftStyle (&RastPort)' alle die Font-Flags in 'StyleEnable' geben, die noch gesetzt werden dürfen.

15.6 Zeichensätze à la Carte

Nun kommen wir zu der eigentlichen Bedeutung von 'TextAttr.ta_Flags'. Mit Hilfe dieser Variablen können Sie erfahren, ob ein Font auf Diskette oder in der Systemfontliste zu finden ist. Diese Information enthält die Variable aber nur dann, wenn die TextAttr-Struktur einer bestimmten Behandlung - sprich: Funktion - unterzogen wurde.

Diese Funktion heißt 'Error = AvailFonts (&Buffer, Anz_Bytes, Flags)'. Sie füllt den angegebenen Speicherbereich ('Buffer') mit dem sogenannten AvailFontsHeader und darauf folgenden AvailFonts-Strukturen.

'AvailFonts()'



Diese AvailFonts-Strukturen enthalten die TextAttr-Strukturen zu allen erreichbaren Fonts (AvailFonts.af_Attr). Allerdings können Sie mit dem Flags-Parameter noch festlegen, ob Sie nur die TextAttr-Strukturen der SystemFonts (AFF_MEMORY) oder die der diskgestützten Fonts (AFF_DISK) oder aller Fonts (AFF_MEMORY | AFF_DISK) erhalten wollen.

In der Variablen 'AvailFonts[i].af_Type' steht nach 'AvailFonts' dann, ob der Font, dessen TextAttr-Struktur Sie gerade untersuchen auf Diskette (AFF_DISK) oder in der Systemfontliste (AFF_MEMORY) zu finden ist.

Die einzige Variable des 'AvailFontsHeader' enthält nach AvailFonts nur die Anzahl der zugänglichen Fonts (AvailFontsHeader.afh_NumEntries).

Folgendes Programm gibt einen kleinen Text nacheinander in allen möglichen Schriftarten und Fonts aus:

```
/*********************************************************/
/*                    SetFont.c                          */
/*                                                       */
/* Dieses Progrmm demonstriert die Benutzung von         */
/* AvailFonts(), OpenFont(), OpenDiskFont(), SetFont()   */
/* AskSoftStyle() und SetSoftStyle().                    */
/*                                                       */
/* Compiled with: Lattice V3.10                          */
/*                                                       */
/* (c) Bruno Jennrich                                    */
/*********************************************************/

#include "exec/types.h"
#include "exec/memory.h"
#include "exec/devices.h"
#include "devices/keymap.h"
#include "graphics/gfx.h"
#include "graphics/copper.h"
#include "graphics/gels.h"
#include "graphics/gfxbase.h"
#include "graphics/regions.h"
#include "hardware/blit.h"
#include "intuition/intuition.h"
#include "intuition/intuitionbase.h"
#include "libraries/diskfont.h"

#define WIDTH 320
#define GROESSE 1000              /* Wie groß ist Buffer? */
#define RP Screen->RastPort       /* Zugriff auf RastPort */

struct GfxBase *GfxBase;              /* Unsere BasePointer */
struct IntuitionBase *IntuitionBase;
ULONG *DiskfontBase;
                  /* Keine spezielle Base für DiskFonts */

struct NewScreen NewScreen =
                    {
                      0,0,WIDTH,200,2,
                      1,0,
                      0,
                      CUSTOMSCREEN,
                      NULL,
                      "",
                      NULL,NULL
                    };

struct AvailFontsHeader *Buffer;
struct AvailFonts *Availfonts;
                      /* Einmal Header und Zeiger auf */
                      /* AvailFonts-Strukturen        */
```

```
struct Screen *Screen;

int StyleEnable, Style;

/*********************************************************/
/* Es geht los !                                         */
/*********************************************************/

main()
{
   struct TextFont *TextFont;
   BOOL   Error;
   long   i,j,Length;
   char   *LeftMouse = (char *) 0xBFE001;

   if ((GfxBase = (struct GfxBase *)
           OpenLibrary("graphics.library",0)) == NULL)
             {
                 printf ("Sorry, No Graphics !!!\n");
                 goto cleanup3;
             }

   if ((IntuitionBase = (struct IntuitionBase *)
           OpenLibrary("intuition.library",0)) == NULL)
             {
                 printf ("Sorry, No Intuition !!!\n");
                 goto cleanup2;
             }

   if ((DiskfontBase = (ULONG *)
           OpenLibrary("diskfont.library",0)) == NULL)
             {
                 printf ("Sorry, No DiskFonts !!!\n");
                 goto cleanup1;
             }

   Screen = (struct Screen *) OpenScreen (&NewScreen);
   if (Screen == 0)
        {
            printf ("Sorry, No Screen !!!\n");
            goto cleanup0;
        }

   SetRGB4 (&Screen->ViewPort,0,0,0,0);  /* Farben ändern */
   SetRGB4 (&Screen->ViewPort,1,15,0,0);
   SetRGB4 (&Screen->ViewPort,2,0,15,0);
   SetRGB4 (&Screen->ViewPort,3,0,0,15);
```

```
SetRast (&RP,0);
        /* Screen vorher löschen, damit die 'Gadgets' */
        /* verschwinden                                */

Buffer = (struct AvailFontsHeader *)
      AllocMem (GROESSE, MEMF_CLEAR | MEMF_CHIP);

if (Buffer == 0)
         {
            printf ("Sorry, No Buffer !!!\n");
            goto cleanup0;
         }

SetAPen (&RP,3);
Length = TextLength (&RP,"AvailFonts()...",15);
Move (&RP,WIDTH/2-Length/2,30);
Text (&RP,"AvailFonts()...",15);
                              /* Message an Benutzer */

Error = AvailFonts (Buffer, GROESSE, AFF_MEMORY |
                                     AFF_DISK);

Availfonts = (struct AvailFonts *) &Buffer[1];
                 /* AvailFontsHeader überspringen     */
                 /* (Buffer vom Typ AvailFontsHeader) */

for (i=0; i<Buffer[0].afh_NumEntries; i++)
   {
      if (Availfonts[i].af_Type == AFF_DISK)
         TextFont = (struct TextFont *) OpenDiskFont
                    (&Availfonts[i].af_Attr);

     /* Font auf Disk oder in Memory ? */

      else
           TextFont = (struct TextFont *) OpenFont
                      (&Availfonts[i].af_Attr);

      if (TextFont != 0)       /* Font korrekt eröfnet */
      {
         SetFont (&RP, TextFont);
                     /* Font in RastPort einbinden */

         StyleEnable = AskSoftStyle (&RP);
                      /* Welche Styles sind erlaubt ? */

         SetAPen (&RP,0);         /* Bildschirm löschen */

         WaitTOF();          /* Um Blinken zu vermeiden */
```

```
            RectFill (&RP, 0,10,WIDTH,50);
                                /* oberste Zeile löschen */

            SetAPen (&RP,2);
            Length = TextLength (&RP,"Die Fonts des AMIGA:",
                                 20);

            Move (&RP,WIDTH/2-Length/2,30);

            Text (&RP,"Die Fonts des AMIGA:",20);

            for (Style = FSF_ITALIC*2; Style>=0; Style--)
              {
                SetSoftStyle (&RP,Style,StyleEnable);
            /* Alle Styles durch Schleife einmal erzeugen */

                Length = TextLength (&RP,
                  Availfonts[i].af_Attr.ta_Name,
                  strlen (Availfonts[i].af_Attr.ta_Name));

                Move (&RP,WIDTH/2-Length/2,100);

                SetAPen (&RP,0);

                WaitTOF();
                RectFill (&RP, 0,80,WIDTH,110);
                SetAPen (&RP,1);

                Text (&RP,
                  Availfonts[i].af_Attr.ta_Name,
                  strlen (Availfonts[i].af_Attr.ta_Name));
                                /* Fontnamen ausgeben */
                j=0;
                while (((*LeftMouse & 0x40) == 0x40) &&
                          (j<500000)) j++;
                Delay(10);
              }
            CloseFont (TextFont);
          }
        }

    FreeMem (Buffer, GROESSE);
                            /* extra Speicher freigeben */
    cleanup0:   CloseScreen (Screen);
    cleanup1:   CloseLibrary(DiskfontBase);
    cleanup2:   CloseLibrary(IntuitionBase);
    cleanup3:   CloseLibrary(GfxBase);
    return (0);
}
```

16. Die Blitter-Befehle

Nun kommen wir zu Befehlen, die nur dem Amiga bekannt sind. Diese Befehle dienen zur grafischen Ausnutzung des Coprozessors 'Blitter'. Dieser ist nämlich in der Lage, Speicherbereiche von 125 KByte innerhalb einer Sekunde im Speicher zu verschieben und die Kopie dabei noch mit logischen Verknüpfungen zu verändern.

Die Art dieser Verknüpfung wird über soganannte Minterms geregelt. Jedoch können diese Minterms nur bei 2 Befehlen von Ihnen selbst bestimmt werden. Die anderen hier beschriebenen Befehle beschränken sich auf das pure Kopieren bzw. Löschen von Daten.

16.1 Speicherbereiche löschen

Um Speicherbereiche zu löschen, gibt es den BltClear-Befehl. Diesen Befehl haben Sie schon in unserem ersten Programm kennengelernt. Dort haben wir diesen Befehl nämlich zum Löschen unserer selbst 'angelegten' Bitplanes benutzt.

Dazu mußten wir dem Befehl die Adresse des zu löschenden Speichers übergeben. Wieviel Speicher ab dieser Adresse gelöscht wird, haben wir mit den Parametern 'Anz_Bytes' und 'Flags' bestimmt ('BltClear (&Speicher, Anz_Bytes, Flags)').

Ist nämlich Bit 1 des Flags-Parameters gesetzt (Flags = 2), wird das 'Long-Word' (32 Bits) 'Anz_Bytes' so interpretetiert, daß die unteren 16 Bits (0-15) die Anzahl der Bytes (also 8 Punkte) pro zu löschender Linie angeben. Die oberen 16 Bits (16-31) geben dann die Anzahl der zu löschenden Linien an. Da aber diese Routine so ausgelegt ist, daß nur Bitplanes mit der Größe von maximal 1024*1024 Punkten gelöscht werden können, kann der Wert in den unteren 16 Bits höchstens 128 betragen (128*8 = 1024).

Wie Sie sehen, ist diese Art, eine Bitplane zu löschen, etwas umständlich. Deshalb haben wir sie auch nur der Vollständigkeit halber hier erwähnt. Denn einfacher geht es, wenn Sie den Flags-Parameter auf 1 setzen und in 'Anz_Bytes' die tatsächliche Anzahl der zu löschenden Bytes angeben.

Und mit Hilfe des 'RASSIZE (Breite,Höhe)'-Macros, das Ihnen die Anzahl der Bytes berechnet, die für eine Bitplane mit 'Breite' Punkten Breite und 'Höhe' Zeilen benötigt werden, haben Sie ein einfaches Werkzeug zur Berechnung der Anzahl der zu löschenden Bytes an der Hand.

16.2 Der Blitter kopiert Daten...

Natürlich wird diese Kapazität auch zum Kopieren von Daten aus einer Bitmap in die andere benutzt. Dabei hilft uns der 'BltBitMap (&QuellBitMap, X1, Y1, &ZielBitMap, X2, Y2, Breite, Höhe, Minterm, Maske, &TmpA)'-Befehl. Sie können mit ihm nämlich rechteckige Ausschnitte einer (Quell-)Bitmap in eine weitere (Ziel-) Bitmap kopieren.

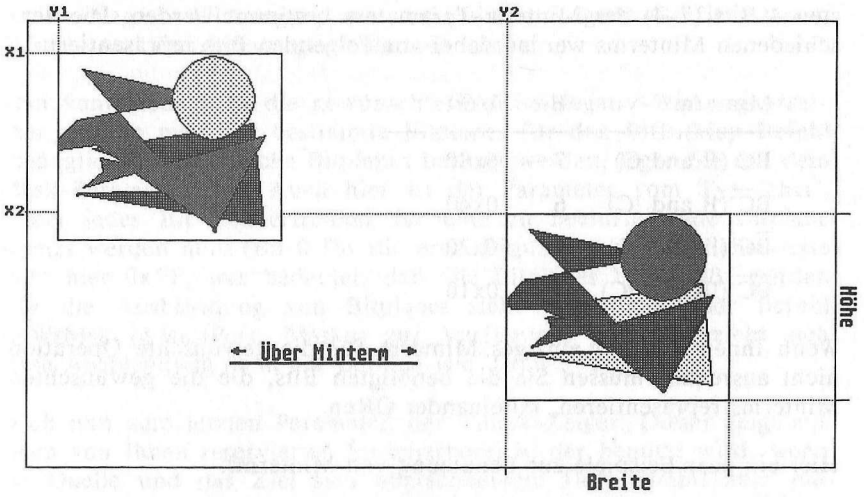
Die Bitmap, aus der Sie die zu kopierenden Daten lesen, und die, in die Sie diese Daten hineinschreiben wollen, können natürlich auch gleich sein.

Doch wie legt man das zu kopierende Rechteck fest? Nun, dazu übergibt man ganz einfach die Koordinaten der linken oberen Ecke des zu kopierenden Rechtecks innerhalb der auszulesenden (Quell-)Bitmap (X1,Y1). Weiterhin bestimmt man die Zielposition des zu kopierenden Rechtecks anhand der linken oberen Ecke (X2,Y2) in der Ziel-Bitmap.

Dann gibt man mit 'Breite' und 'Höhe' die Größe des zu kopierenden Rechtecks in Punkten bzw. Linien an. Dies braucht nur einmal zu geschehen, da die Größe dieses Rechtecks nicht verändert wird. Der Blitter verzerrt ja nicht, sondern kopiert nur.

Leider erlaubt der BltBitMap-Befehl nicht, daß beliebig große Rechtecke im Speicher in eine Bitmap verschoben werden. Die Breite des Rechtecks muß nämlich zwischen 1 und 976 Punkten, die Höhe zwischen 1 und 1023 Linien liegen.

Beachten sollten Sie hierbei, daß das Rechteck vollkommen innerhalb der Ziel-Bitmap liegen muß. Achten Sie auf diese kleine Tatsache nicht, kann es passieren, daß bei einem Überschreiten der Bildschirmränder ein niedlicher kleiner 'Software Failure' bzw. 'Guru Meditation' auftritt.



Doch wenn Sie diese 'Kleinigkeit' beachten, kann gar nichts schiefgehen. Doch nun zu der Art der Verknüpfungen.

Oben haben wir ja schon darauf hingewiesen, daß der Blitter in der Lage ist, während des Kopierens logische Verknüpfungen vorzunehmen. Dies kann z.B. bedeuten, daß die Daten des Quell- und des Zielrechtecks, bevor sie in das Zielrechteck geschrieben werden, miteinander durch 'Und' verknüpft werden.

Wie was geschieht, bestimmen Sie mit den Minterms. Sie bestimmen, wie Quelle und Ziel miteinander verknüpft werden (s. Anhang über die Hardware-Register).

Jedes Bit dieses Parameters (vom Typ 'char') bestimmt, wie Quelle und Ziel 'eingebracht' werden. Jedoch müssen Sie beachten, daß dieser Befehl nur zwei Quellen und ein Ziel zuläßt, wobei Quelle 2 gleich dem Ziel ist. Der Blitter kann aber 3 verschiedene Quellen miteinander in Beziehung setzen und das Ergebnis an einer vierten Stelle abspeichern. Hier werden nur Quelle B und C benutzt, wobei Ziel D und Quelle C gleich sind (s. Hardware-Register (bltcon)).

Aufgrund dieser Tatsache bleiben dem BltBitMap-Befehl von 256 möglichen Minterms nur noch ganze 16 übrig, die mit Hilfe der ober-

sten 4 Bits (7-4) des Minterm-Parameters bestimmt werden. Die verschiedenen Minterms werden dabei von folgenden Bits repräsentiert:

Minterm	Bit	Wert
BC (B and C)	7	0x80
$B\overline{C}$ (B and !C)	6	0x40
$\overline{B}C$ (!B and C)	5	0x20
$\overline{BC}$ (!B and !C)	4	0x10

Wenn Ihnen nun ein einziges Minterm für die gewünschte Operation nicht ausreicht, müssen Sie die benötigten Bits, die die gewünschten Minterms repräsentieren, miteinander ORen.

Hier ein paar Beispiele zur Benutzung von Minterms:

Mit der Kombination (!B and !C) or (!B and C) = !B, was dem Wert 0x30 für 'Minterm' entspricht, überschreiben Sie den Zielbreich mit dem invertiertem Quellbereich.

Mit (!B and !C) or (B and !C) = !C (= 0x50) wird nur das Zielrechteck, natürlich jede Bitplane für sich, invertiert.

'and' kann durch '*' und 'or' durch '+' ersetzt werden. Die Rechengesetze (Kommutativgesetz, Assoziativgesetz etc.) gelten auch hier.

Hier weitere, häufig benutzte Minterms:

0x60: Wo die Punkte von C gelöscht sind, werden die Punkte aus Quelle B gesetzt, aber wo die Punkte der Quelle B gelöscht sind, bleibt C erhalten.

0x80: Hier werden die beiden zu blittenden Rechtecke miteinander geANDet, das heißt, daß Punkte aus B nur dann gesetzt werden, wenn dieselben Punkte in C schon gesetzt waren.

Mit 0xC0 kopiert man Datenbereiche aus Quelle B ins Ziel C, ohne die Daten zu verändern.

Noch ein letztes Wort zu den Verknüpfungen: Die Bitmaps werden Bitplane für Bitplane geblittet. So kann es passieren, daß anstatt

Positiv-Negativ-Wirkungen, wie bei 'einPlane-igen' Grafiksystemen üblich, auch Farbänderungen auftreten können.

Man kann aber auch die gewünschte Positiv-Negativ-Wirkung erreichen, indem man nur bestimmte Bitplanes für den BltBitMap-Befehl zugänglich macht. Welche Bitplanes berührt werden, legen Sie mit dem Mask-Parameter fest. Auch hier ist der Parameter vom Typ 'char', wobei jedes Bit stellvertretend für eine zu beeinflussende Bitplane gesetzt werden muß (Bit 0 für die erste Bitplane usw.). Üblicherweise steht hier 0xFF, was bedeutet, daß alle Bitplanes beeinflußt werden (für die Ausblendung von Bitplanes steht Ihnen auch der Befehl SetWrMsk (&RastPort, Maske) zur Verfügung, jedoch bezieht sich diese Ausblendung dann auf Befehle wie 'Draw', 'Text' usw.).

Doch nun zum letzten Parameter: der TmpA-Zeiger. Dieser zeigt auf einen von Ihnen reservierten Speicherbereich, der benutzt wird, wenn die Quelle und das Ziel sich überschneiden. Dies kommt aber nur dann vor, wenn Sie den BltBitMap-Befehl auf eine einzige Bitmap anwenden. Die Größe dieses Speichers wird durch die Größe des berlappenden Bereiches festgelegt. Sind Sie sich sicher, daß sich Quelle und Ziel nicht überschneiden, brauchen Sie keinen Speicher zu allokieren und übergeben der Funktion für diesen Parameter den Wert Null.

Eine Unterart des BltBitMap-Befehls ist der ClipBlit-Befehl:

16.2.1 Ein verwandter Befehl: ClipBlit

Die Befehle BltBitMap und 'ClipBlit (&QuellRastPort, X1, Y1, &ZielRastPort, X2, Y2, Breite, Höhe, Minterm)' sind fast identisch. Die Unterschiede zwischen beiden bestehen jedoch darin, daß BltBitMap, wie der Name schon sagt, 'niedrigere Bitmaps 'blittet', während ClipBlit Rastport-Strukturen zur Bestimmung der Daten, 'die durch den Blitter gejagt werden sollen', heranzieht. Natürlich können auch hier beide Rastports gleich sein.

Ein weiterer Unterschied besteht darin, daß ClipBlit keinen Mask-Parameter benötigt - er hat ja die Rastports zur Verfügung, in denen die Variable 'Mask', die mit 'SetWrMsk(&RastPort,Maske)' verändert werden kann, enthalten ist.

In bezug auf die anderen Parameter stehen sich BltBitMap und ClipBlit in nichts nach. Jedoch sollte man beachten, daß ClipBlit vor-

nehmlich in einer Intuition-Umgebung seine Anwendung findet, denn ClipBlit ist in der Lage, auf die Besonderheiten von Intuition, wie z.B. die Tatsache, daß eine Grafik-Operation über die Grenzen des Windows hinaus weggeclippt bzw. 'weggeschnitten' wird, einzugehen, weshalb Sie bei der Verwendung dieses Befehls in Intuition-Windows keine Angst vor Systemabstürzen zu haben brauchen.

Hierzu ein kleines 'Blitter'-Programm:

```
/*********************************************************/
/*                    BltBitMap.c                        */
/*                                                       */
/* Dieses Programm zeigt die grundsätzliche Funktion des */
/* BltBitMap(), bzw. ClipBlit() Befehls auf.             */
/*                                                       */
/* Compiled with: Lattice V3.10                          */
/*                                                       */
/* (c) Bruno Jennrich                                    */
/*********************************************************/

#include "exec/types.h"
#include "exec/memory.h"
#include "exec/devices.h"
#include "devices/keymap.h"
#include "graphics/gfx.h"
#include "graphics/text.h"
#include "graphics/regions.h"
#include "graphics/gfxmacros.h"
#include "graphics/gfxbase.h"
#include "graphics/gels.h"
#include "graphics/copper.h"
#include "intuition/intuition.h"
#include "hardware/blit.h"

#define Width 320              /* Bildschirmdefinitionen */
#define Height 200
#define Depth 2
#define MODES 0

char *LeftMouse = (char *) 0xbfe001;

struct GfxBase *GfxBase;
struct IntuitionBase *IntuitionBase;

struct Screen *Screen;
struct Window *Window;
struct IntuiMessage *Message;
struct RastPort *RPort;
```

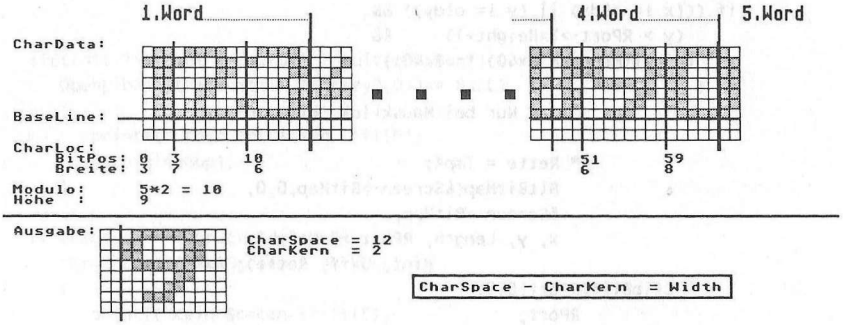
```
struct NewScreen NewScreen =
            {0,0,
             Width,Height,Depth,
             1,0,
             MODES,
             CUSTOMSCREEN,
             NULL,
             NULL,
             NULL,NULL
            };

struct NewWindow NewWindow =
            {0,0,Width,Height,
             1,0,NULL,
             ACTIVATE|BORDERLESS,
             NULL,NULL,
             "Blit-BitMap",
             NULL,
             NULL,
             NULL,NULL,NULL,NULL,
             CUSTOMSCREEN
            };

int Length,
    x,y,
    oldx, oldy,
    i,j,Mint;

char Mins[16][5] = {
                     "$00 ","$10 ","$20 ","$30 ",
                     "$40 ","$50 ","$60 ","$70 ",
                     "$80 ","$90 ","$A0 ","$B0 ",
                     "$C0 ","$D0 ","$E0 ","$F0 "
                   };
APTR TmpA,
     Rette;

BOOL Ende = FALSE;

/*********************************************************/
/* Es geht los !                                         */
/*********************************************************/

main()
{
   if((GfxBase = (struct GfxBase *)
      OpenLibrary("graphics.library",0)) == NULL)
      {
         printf(" Keine Graphics !!!!!");
         Exit (0);
```

```
    }

  if((IntuitionBase = (struct IntuitionBase *)
     OpenLibrary("intuition.library",0))== NULL)
     {
        printf(" Kein Intuition !!!!!");
        goto cleanup1;
     }

  if ((Screen = (struct Screen *)
      OpenScreen(&NewScreen)) == NULL)
     {
        printf("Kein Screen !!!!!!");
        goto cleanup2;
     }

  NewWindow.Screen = Screen;

  if ((Window = (struct Window *)
      OpenWindow(&NewWindow)) == NULL)
     {
        printf("Kein Window !!!!!!");
        goto cleanup3;
     }

  RPort = Window->RPort;
  Length = TextLength (RPort, NewWindow.Title,
                       strlen (NewWindow.Title))+5;

/*
  if ((TmpA = (APTR) AllocMem (RASSIZE(Width,Height),
                     MEMF_CHIP)) == NULL)
     {
        printf("Kein Rasterbuffer  !!!!!!");
        goto cleanup4;
     }                                                  */

  SetDrMd (RPort, JAM2);

  oldx = oldy = -1;

  while (Ende == FALSE)
     {
        x = Screen->MouseX;
        y = Screen->MouseY;

        if (((x == 0) && (y == 0)) &&
            ((*LeftMouse & 0x40) != 0x40)) Ende = TRUE;
                        /* Links, oben klicken == Ende */
```

```
        if (((x != oldx) || (y != oldy)) &&
            (y > RPort->TxHeight+1)      &&
           ((*LeftMouse & 0x40) != 0x40))
           {
                    /* Nur bei Mausklick 'blitten' */

                 /* Rette = TmpA;
                    BltBitMap(&Screen->BitMap,0,0,
                    &Screen->BitMap,
                    x, y, Length, RPort->TxHeight,
                                Mint, 0xff, Rette); */
            ClipBlit(RPort,0,0,
                     RPort,
                     x,y,Length,RPort->TxHeight,
                     Mint);                 /* Blitten */

            SetAPen (RPort,2);
            Move (RPort,Length,RPort->TxBaseline);
            Text (RPort," Minterms: ",12);
            Text (RPort,&Mins[(Mint >> 4)][0],4);

            Mint += 0x10;                   /* Minterm ++ */
            Mint &= 0xF0;

            oldx = x;
            oldy = y;
          }
    }


    /* if (TmpA != 0) FreeMem(TmpA,RASSIZE(Width,Height)); */

    cleanup4: CloseWindow(Window);
    cleanup3: CloseScreen(Screen);
    cleanup2: CloseLibrary(IntuitionBase);
    cleanup1: CloseLibrary(GfxBase);

    return(0);   /* Tschüs */
}
```

16.3 ...und liest Daten

Mit dem 'BltTemplate (&Speicher, BitPosition, Modulo, &Rastport, X,Y, Breite, Höhe)'-Befehl kann man Daten aus einem 'gepackten' Daten-Array auslesen. In solch einem gepackten Daten-Array sind z.B. die Fonts des Amiga organisiert. Bit an Bit verstaut stehen die einzelnen Zeichen im Speicher.



Aber wie kann man aus dieser Organisation die einzelnen Zeichen auslesen?

Nun, da hilft uns der BltTemplate-Befehl. Aber bevor wir näher auf die Parameter dieses Befehls eingehen, möchten wir Ihnen erst einmal ein kleines Programm vorstellen, mit dem wir einen Mini-Zeichensatz simuliert haben:

```
/*********************************************************/
/*                  Figures.c                            */
/*                                                       */
/* Dieses Programm zeigt die Funktionsweise des          */
/* BltTemplate() Befehls am Beispiel eines Pseudo-Fonts  */
/* auf.                                                  */
/*                                                       */
/* Compiled with: Lattice V3.10                          */
/*                                                       */
/* (c): Bruno Jennrich                                   */
/*********************************************************/

#include "exec/types.h"
#include "exec/memory.h"
#include "exec/devices.h"
#include "devices/keymap.h"
#include "graphics/gfx.h"
#include "graphics/text.h"
#include "graphics/regions.h"
#include "graphics/copper.h"
#include "graphics/gels.h"
#include "graphics/clip.h"
#include "graphics/gfxmacros.h"
#include "graphics/view.h"
```

```
#include "graphics/gfxbase.h"
#include "intuition/intuition.h"
#include "hardware/blit.h"

#define Width 640              /* Bildschirmdefinitionen */
#define Height 200
#define Depth 2
#define MODES HIRES

char *LeftMouse = (char *) 0xbfe001;

struct GfxBase *GfxBase;
struct IntuitionBase *IntuitionBase;

struct TextAttr TextAttr =
           {"topaz.font",
            8,
            FS_NORMAL,
            FPF_ROMFONT};

struct Screen *Screen;
struct Window *Window;
struct IntuiMessage *Message;
struct RastPort *RPort;

struct NewScreen NewScreen =      /* Screen-Definitionen */
           {0,0,
            Width,Height,Depth,
            1,0,
            MODES,
            CUSTOMSCREEN,
            &TextAttr,
            NULL,
            NULL,NULL
           };

struct NewWindow NewWindow =            /* Window-Defines */
           {0,0,
            Width,Height,
            1,0,
            NULL,
            ACTIVATE|BORDERLESS,
            NULL,NULL,
            NULL,
            NULL,
            NULL,
            NULL,NULL,NULL,NULL,
            CUSTOMSCREEN
           };
```

```
int i;

BOOL ende = FALSE,
     MOVE;

UWORD Class,
      Code;

UWORD CharData [] = {
   /* 1. Zeile */    0x001E,0x0CFE,0x11F9,0xC3C0,0x0000,
   /* 2. Zeile */    0x6F23,0x1CC0,0x221A,0x3467,0x8000,
   /*     "    */    0xF183,0x2CC0,0x4036,0x3C68,0xC000,
                     0x618E,0x4CF8,0xF863,0xCC78,0xC000,
                     0x6303,0xFF0D,0x8CC1,0xE7D8,0xC000,
                     0x6C43,0x0C07,0x8CC6,0x3198,0x8000,
                     0x7F82,0x0C05,0x8986,0x230f,0x0000,
                     0x003C,0x0CF8,0xF183,0xC600,0x0000,
   /* 9. Zeile */    0x0000,0x0800,0x0000,0x0000,0x0000
                    };
                         /* Zahlen: 1 2 3 4 5 6 7 8 9 0 */

UWORD *ChipCharData,*Help;        /* Blitter kann nur aus */
                                  /* Speicher 'unterhalb' */
                                  /* 512K lesen. Deshalb  */
                                  /* wird CharData[] in   */
                                  /*   'Chip-Accesible'   */
                                  /* Speicher geschrieben */
                                  /* bzw. kopiert !!!     */
UWORD CharHeight = 9;
                       /* Jedes Zeichen ist 9 Zeilen hoch */

UWORD CharLoc[] = {0,3 ,3,7, 10,6, 16,8, 24,7,
                   31,7, 38,7, 45,7, 52,7, 59,7};
                                       /* Bitpos, Breite */

UWORD Modulo = 5*2;
                                 /* 5 Words * 2 = 10 Bytes */

UWORD TxSpacing = 11;
                                    /* größte Breite + 3 */

char *String1 = "Na, sind das nicht tolle Zahlen :";
char *String2 =
"Tja, und ohne Text, sondern mit BltTemplate() ausgegeben.";

VOID Figures();                   /* Vorwärtsdeklaration */
```

```
/*********************************************************/
/* Es geht los !                                         */
/*********************************************************/

main()
{
   if((GfxBase = (struct GfxBase *)
      OpenLibrary("graphics.library",0)) == NULL)
      {
         printf(" Keine Graphics !!!!!");
         Exit (1000);
      }

   if((IntuitionBase = (struct IntuitionBase *)
      OpenLibrary("intuition.library",0))== NULL)
      {
         printf(" Kein Intuition !!!!!");
         goto cleanup1;
      }

   if ((Screen = (struct Screen *)
       OpenScreen(&NewScreen)) == NULL)
      {
         printf("Kein Screen !!!!!!");
         goto cleanup2;
      }

   NewWindow.Screen = Screen;        /* Screen-Struktur für */
                                     /* NewScreen           */

   if ((Window = (struct Window *)
       OpenWindow(&NewWindow)) == NULL)
      {
         printf("Kein Window !!!!!!");
         goto cleanup3;
      }

   ChipCharData = (UWORD*) AllocMem
            (sizeof(CharData),MEMF_CLEAR|MEMF_CHIP);

   if (ChipCharData == 0)
     {
        printf (" Keinen Chip-Memory mehr !!!\n");
        goto cleanup4;
     }

   Help = ChipCharData;
   for (i=0; i<(sizeof(CharData)/sizeof(UWORD));i++)
       {
          *Help = CharData[i];
```

```
        Help++;
    }

RPort = Window->RPort;

SetAPen (RPort,1);
SetDrMd (RPort,JAM1);

Move (RPort,20,20);
Text (RPort,String1,strlen(String1));
                                    /* Text ausgeben */

Move (RPort,20,35);
Figures();                          /* Zahlen ausgeben */

Move(RPort,20,50);

SetDrMd (RPort,JAM2);
Figures();

SetDrMd (RPort,JAM1);
SetAPen (RPort,1);
                                          /* OUTLINE */

Move (RPort,20,76);
Figures();

Move (RPort,20,74);
Figures();

Move (RPort,21,75);
Figures();

Move (RPort,19,75);
Figures();

SetAPen (RPort,2);
Move (RPort,20,75);
Figures();

Move (RPort, 20,100);
Text(RPort,String2,strlen(String2));

while ((*LeftMouse & 0x40) == 0x40);

        FreeMem (ChipCharData,sizeof(CharData));
cleanup4: CloseWindow(Window);
cleanup3: CloseScreen(Screen);
cleanup2: CloseLibrary(IntuitionBase);
cleanup1: CloseLibrary(GfxBase);
```

```
    return (0); /* Tschüs */
}

/*********************************************************/
/* Diese Funktion ist für das Lesen der Daten aus dem    */
/* Array zuständig.                                      */
/*-------------------------------------------------------*/
/* Eingabe-Parameter: keine                              */
/*-------------------------------------------------------*/
/* Rückgabe-Werte: keine                                 */
/*********************************************************/

VOID Figures()
{
   int i;

   for (i=0; i < 10; i++)
      {
         BltTemplate(ChipCharData,CharLoc[i*2],Modulo,
                     RPort,RPort->cp_x,RPort->cp_y,
                     CharLoc[i*2+1], CharHeight);
                    /* Hol' Daten aus CharData Array */

         Move (RPort,RPort->cp_x+TxSpacing,RPort->cp_y);
                     /* Neue Position für Ausgabe */
      }
}
```

In unsererem Programm haben wir den Mini-Zeichensatz, der die Ziffern von 9-0 beeinhaltet, in einem Array (CharData) von 'UWORDs' abgespeichert. Die Anfangsadresse dieses Arrays müssen wir dem BltTemplate-Befehl übergeben.

Damit der BltTemplate-Befehl auch weiß, ab welcher Bit-Position ein auszugebendes Zeichen beginnt, haben wir ein weiteres Array angelegt, das sowohl den Beginn eines Zeichens (Charloc, jeweils Element 0, 2, 4...) als auch dessen Breite in Pixel (CharLoc, Element 1, 3, 5...) angibt. Die Höhe unserer Zeichen haben wir konstant gehalten (die Fonts des Amiga erlauben auch keine variable Höhengestaltung der einzelnen Zeichen).

Doch nun zur Verwendung der Arrays in bezug auf BltTemplate:

'CharData' ist unser Speicher, aus dem wir die Daten lesen wollen, also müssen wir als ersten Parameter diese Adresse übergeben. Die Bit-

Position unseres Zeichens wird aus dem Array 'CharLoc' gelesen, deren geradzahligen Elemente (0, 2, 4...) diese Position angeben. Die Angabe der Bit-Position ermöglicht, daß ein Zeichen nicht mit einem neuen Byte beginnen muß, sondern an einer beliebigen Stelle innerhalb eines Bytes beginnen kann. Dies spart viel Speicherplatz.

Als nächstes übergibt man dem BltTemplate-Befehl den 'Modulo'. Dieser Wert bestimmt, wie viele Bytes und somit Punkte (1 Byte = 8 Pixel) man zur aktuellen Bit-Position addieren muß, um die Daten für die nächste Zeile lesen zu können, oder anders ausgedrückt: Der Modulo gibt die Breite des Arrays in Bytes an.

Hat man dann den Rastport, in den die gelesenen Daten geschrieben werden sollen, angegeben, bestimmt man die Position innerhalb der Bitmap mit 'X,Y', in der die Daten ausgegeben werden sollen, wobei hier die linke obere Ecke des zu kopierenden Rechtecks positioniert wird.

'Breite, Höhe' geben die Breite des zu kopierenden Rechtecks in Punkten bzw. die Höhe dieses Rechtecks in Zeilen an. Der Wert für 'Breite' kann aus den ungeraden Charloc-Elementen gelesen werden.

Alle diese Parameter bestimmen also die Größe des auszulesenden Rechtecks sowie die Position dieses Rechtecks auf dem Bildschirm. Jedoch gibt es bei diesem Befehl nicht die Möglichkeit, die Daten, bevor sie in die Bitmap geschrieben werden, einer logischen Verknüpfung zu unterziehen.

17. Die Darstellungsmodi des Amiga

Nun kommen wir zu den in vorigen Kapiteln schon oft zitierten Darstellungsmodi. Dort haben wir ausführlich und mit Hilfe von Programmen beschrieben, wie Sie einen Bildschirm öffnen und in diesem Ihre eigenen Grafiken mittels der Grafikbefehle programmieren können.

Wir haben in diesen Kapiteln auch mehrfach auf die Darstellungsmodi hingewiesen. Aber welche überhaupt existieren, und wie man sie alle in einem Viewport festlegt - bei manchen Modi genügt es nämlich nicht, nur die Modes-Variable des Views bzw. Viewports zu verändern - haben wir noch nicht ausdrücklich gezeigt.

Doch zuerst Grundsätzliches über die Darstellungsmodi: Man kann sie nämlich in vier Rubriken unterteilen.

Unter die erste Rubrik fallen die Modi, die die Auflösung, also die Anzahl der darstellbaren Punkte sowohl in vertikaler als auch in horizontaler Richtung, bestimmen.

Zweitens gibt es da die Farb-Modi, die entscheidenden Einfluß auf die Anzahl der darstellbaren Farben nehmen.

Die Spezial-Modi, die eine weitere Rubrik darstellen, befassen sich weniger mit den von der Hardware-Seite her zu ermöglichenden Darstellungsmodi, sondern vielmehr mit der softwaremäßigen Ausnutzung der schon vorhandenen.

Die letzte Rubrik - wie könnte es anders sein? - beinhaltet die Modi 'Sonstiges', worunter auch die Darstellung von Sprites fällt.

Doch zunächst:

17.1 Die 'Auflösungsmodi'

Die Qualität eines Grafik-Computers, was der Amiga ja nun mal ist, wird in besonderem Maße durch dessen Auflösungsvermögen bestimmt. Denn je mehr Punkte auf dem Monitor erscheinen können, desto besser ist der Computer (natürlich spielt die Geschwindigkeit eines Computers auch eine große Rolle, für uns aber im Moment weniger!).

Mit maximal 640x400 Punkten im 'Interlaced'-Modus liegt der Amiga zwar nicht in der Spitzenklasse, aber er liegt doch immerhin im guten 'Mittelklasse'-Bereich. Die Stärke des Amiga liegt ja schließlich nicht allein in seinem Auflösungsvermögen, sondern auch in der Darstellung von bis zu 4096 Farben (auf einmal!).

Doch welche verschiedenen Auflösungsmodi sind möglich? Folgende Tabelle gibt Ihnen darüber einen kleinen Überblick:

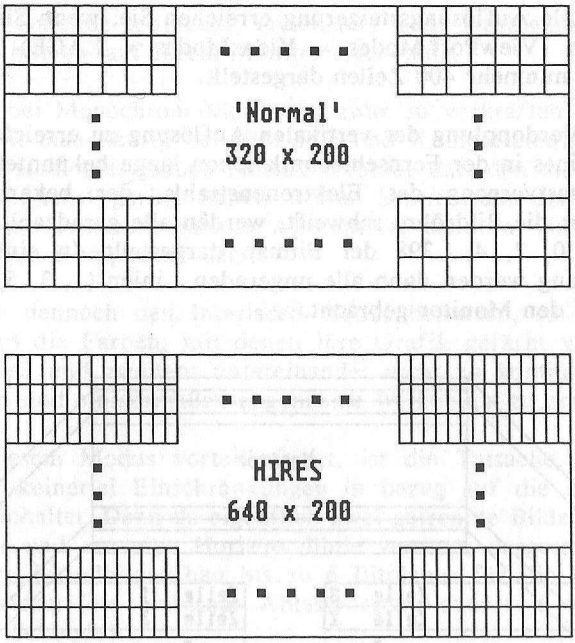
1. 320 * 200 Punkte mit 32/64 Farben (Lo-Res)
2. 320 * 400 Punkte mit 32/64 Farben
3. 640 * 200 Punkte mit 16 Farben (Hi-Res)
4. 640 * 400 Punkte mit 16 Farben

Bitte beachten Sie, daß sich die hier angegebenen Werte auf ein 'Monitor-Stück' der Größe des Workbench-Screens beziehen. Wenn Sie sich der Technik des 'Overscans' bedienen, wobei Sie den abtastenden Elektronenstrahl über die 100%-Schärfe-Zone hinaus für Ihre Grafiken benutzen, können Sie bis zu 720x512 Punkte darstellen, wobei an den Rändern des Monitors dann Unschärfen auftreten.

Wie Sie erkennen können, unterscheiden sich die einzelnen Auflösungen in X- und Y-Richtung, wenn überhaupt, nur um den Faktor zwei. Doch wie kann man die verschiedenen Auflösungen einstellen?

Dazu gibt es zwei Flags, die in der Modes-Variablen des Views bzw. Viewports gesetzt werden können: Hi-Res und Lace.

Das Hi-Res-Flag wird gesetzt, wenn Sie die horizontale, also die X-Auflösung, von 320 auf 640 Punkte erhöhen wollen (ViewPort.Modes = View.Mode = HIRES). Natürlich müssen Sie beachten, daß Sie bei einer Verdopplung der Auflösung auch den Speicherplatz der Bitmap, die Sie in diesem Modus darstellen wollen, verdoppeln müssen.



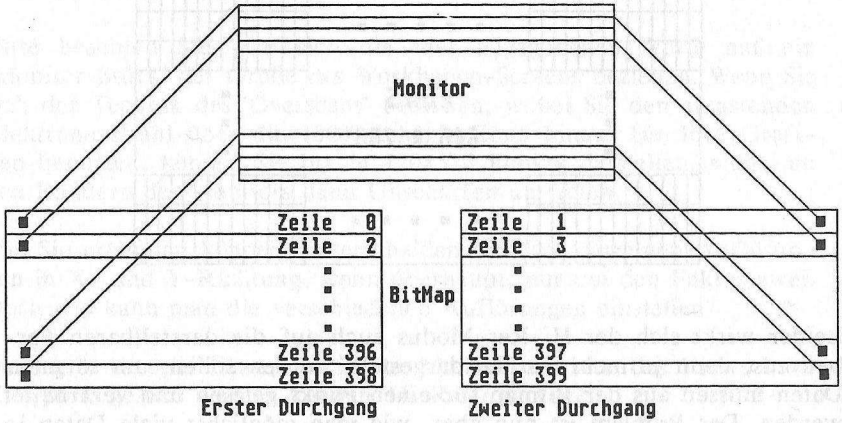
Leider wirkt sich der Hi-Res-Modus auch auf die darstellbaren Farben aus, denn je mehr Farben dargestellt werden sollen, um so mehr Daten müssen aus der Bitmap für einen Punkt gelesen und verarbeitet werden. Das Problem ist nun aber, wie man möglichst viele Daten in möglichst wenig Zeit verarbeiten kann.

Wenn man nämlich im Normal-Modus arbeitet, hat der Elektronenstrahl nur ein ca. 1 mm^2 großes Quadrat für einen Punkt abzutasten. Da der Elektronenstrahl immer mit der gleichen Geschwindigkeit über die Bildröhre 'fegt', hat die Grafik-Hardware des Amiga im Normal-Modus naturgemäß mehr Zeit, die benötigten Daten aus dem Speicher zu lesen und darzustellen als im Hi-Res-Modus, in dem zwei Punkte auf ca. 1 mm^2 nebeneinander Platz finden müssen.

Somit wird auch die Anzahl der darstellbaren Farben stark eingeschränkt. Während Sie nämlich im Normal-Modus bis zu 6 Bitplanes auf den Monitor 'bringen' können, können Sie im Hi-Res-Modus nur 4 Bitplanes darstellen, was bedeutet, daß Ihnen nur noch maximal 16 Farben zur Verfügung stehen.

Eine vertikale Auflösungssteigerung erreichen Sie, wenn Sie das Lace-Flag setzen (ViewPort.Mpdes = View.Modes = LACE). So werden anstatt 200 nunmehr 400 Zeilen dargestellt.

Um diese Verdopplung der vertikalen Auflösung zu erreichen, bedient man sich eines in der Fernsehtechnik schon lange bekannten Tricks: In einem Abtastvorgang des Elektronenstrahls, der bekanntlich 'von hinten' über die Bildröhre schweift, werden alle geradzahligen Linien, also Linie 0, 2, 4,.. 398 der Bitmap dargestellt. In einem zweiten Abtastvorgang werden dann alle ungeraden Linien (1, 3, 5,... 399) der Bitmap auf den Monitor gebracht.



Aufgrund dieser Zweiteilung des Abtastvorgangs wird die Bildwechselfrequenz, also die Anzahl der kompletten Bildschirmaufbauten pro Sekunde, halbiert. Dies hat aber zur Folge, daß das Bild, das nun in einer 30stel anstatt in einer 60stel Sekunde aufgebaut wird, ein wenig flackert.

Leider kann es passieren, daß das wenige Flackern zu einem ausgewachsenen Flimmern avanciert. Denn bei Viewports, die helle Farben verwenden, merkt man sehr, daß die Farben schnell an Intensität verlieren. Dieser Intensitätsverlust läßt sich auf die geringe Nachleuchtzeit des Monitors zurückführen, aber die Phosphorpartikel, die der Elektronenstrahl bei seinem Abtastvorgang zum Leuchten bringt, dürfen ja auch nicht zu lange nachleuchten. Denn wenn Sie nicht den Interlaced-Modus benutzen, wird für den Aufbau eines Bildes ja bloß

eine 60stel Sekunde gebraucht. Treten hier lange Nachleuchtzeiten auf, können 'Schlieren' auf Ihrem Monitor erscheinen.

Dies mag bei Monochrom-Monitoren zwar zu verkraften sein (diesen Effekt sieht man häufig bei IBM PCs und Kompatiblen), bei Color-Monitoren stellt dies jedoch einen Störfaktor dar, den man bei Commodore möglichst gering halten wollte. Somit stellt der Interlaced-Modus keine ernstzunehmende Alternative, sondern vielmehr einen schwammigen Kompromiß dar.

Wollen Sie dennoch den Interlaced-Modus benutzen, sollten Sie beachten, daß die Farben, mit denen Ihre Grafik gefärbt wird, erstens nicht zu hell und zweitens untereinander nicht zu kontrastreich sind. Denn dann sind einigermaßen angenehme Wirkungen zu erzielen.

Was an diesem Modus vorteilhaft ist, ist die Tatsache, daß dessen Benutzung keinerlei Einschränkungen in bezug auf die darstellbaren Farben beinhaltet. Denn da eigentlich zwei getrennte Bilder, ein wenig zeitversetzt und um eine Horizontallinie versetzt, dargestellt werden, können pro Einzelbildaufbau bis zu 6 Bitplanes für die Grafik verwendet werden (ein einzelner Abtastvorgang dauert ja immer noch eine 60stel Sekunde).

Sollten Sie noch Bedenken hinsichtlich der Organisation der Interlaced-Bitmaps haben, können wir Sie beruhigen: Sie müssen nur genug Speicher reservieren (doppelt soviel wie normal, im Laced- und Hi-Res-Modus dann natürlich 4mal soviel wie sonst) den Rest, sprich die Aufteilung des 'geraden' und 'ungeraden' Bildes, übernimmt der Amiga.

Bitte beachten Sie, daß der Auflösungsmodus des Views bestimmt, ob in den Viewports ein bestimmter Auflösungsmodus gesetzt werden kann. Erlaubt Ihr View z.B. nicht den Hi-Res-Modus, können Sie diesen auch nicht in einem Ihrer Viewports benutzen. Andersherum geht dies aber. Wenn Sie nämlich im View das Hi-Res-Flag gesetzt haben, können Sie trotzdem den Viewport in 'Lo-Res' darstellen.

17.2 Die Farbmodi

Doch nun zu den Farb- bzw. Colormodi des Amiga, denn farblich gesehen leistet der Amiga Erstaunliches:

Denn er ist erstens in der Lage, 64 Farben auf einmal und ohne große Probleme darzustellen, und zweitens kann er alle 4096 Farben, die er erzeugen kann, auch auf einmal darstellen.

17.2.1 Der EXTRA_HALFBRITE-Modus

In diesem Modus können Sie, wie oben schon erwähnt, bis zu 64 Farben auf einmal in einem Low-Resolution-Viewport darstellen.

Denn hier werden 6 Bitplanes benötigt, was im Lo-Res-Modus ja möglich ist. Außer diesen 6 Bitplanes müssen Sie weiterhin das EXTRA_HALFBRITE-Flag in der Modes-Variablen des Viewports setzen.

Doch was bewirkt Halfbrite? Nun, um dies zu erläutern, müssen wir ein wenig in die Tiefen des Amiga hinabsteigen.

Dem Anhang über die Hardware-Register können Sie entnehmen, daß der Amiga nur 32 Color-Register enthält, um diese alle zu nutzen, würden also schon 5 Bitplanes reichen. Wozu also die sechste Bitplane?

Hier kommt der Halfbrite-Modus ins Spiel: Denn wenn Sie außer den Bits in den 'untersten' 5 Bitplanes auch noch das Bit eines Punktes in der sechsten Bitplane gesetzt haben, dann wird dieser Punkt zwar noch in der Farbe des Farbregisters, das in den untersten 5 Bitplanes bestimmt wurde, dargestellt, aber nur mit halber Intensität. Das ist also das Geheimnis des Halfbrite(Halbhell)-Modus. (Diese Halbierung der Intensität wird dadurch erreicht, daß der Inhalt der Farbregisters intern halbiert bzw. nach rechts 'geshiftet' wird.)

Wenn Sie also bei der Farbwahl mit SetAPen zum eigentlichen Wert des Farbregisters noch 32 addieren, wird jeder neu gesetzte Punkt mit halber Farbintensität als normal dargestellt.

17.2.2 'Hold and Modify' für 4096 Farben

Jetzt kommen wird zum 'Überbären' der Farbgrafik: dem Hold and Modify Modus.

Alle 4096 Farben kann man in ihm auf den Bildschirm 'zaubern'. Aber leider sind hier vor den Erfolg ein paar 'Steine in den Weg gelegt' worden, aber wir helfen Ihnen, diese aus dem Weg zu räumen.

Beschäftigen wir uns zuerst einmal mit den Voraussetzungen, die für diesen Modus geschaffen werden müssen: Zuerst müssen auch hier 5, besser aber 6 Bitplanes für die HAM-Bitmap zur Verfügung gestellt werden (Dies schließt natürlich aus, daß HAM in Verbindung mit Hi-Res benutzt werden kann!).

Weiterhin müssen Sie natürlich ein Flag in der Modes-Variablen des Viewports setzen: das HAM-Flag. (ViewPort.Modes = HAM, hier spielt der View auch keine Rolle).

Um die Funktionsweise dieses Modus zu erklären, wollen wir Sie noch einmal an den Aufbau eines Farbregisters in der Colormap eines Viewports erinnern: Die 16 Bits, die für ein Farbregister herangezogen

werden, enthalten in den Bits 11-8 die Rot-, in den Bits 7-4 die Grün- und in den Bits 3-0 die Blaukomponente der Farbe.

Im 'Hold and Modify'-Modus werden nun für die Farbbestimmung eines Punktes zwei Farbkomponenten des linken 'Vorgängerpunktes' übernommen (Hold) und eine dritte Farbkomponente modifiziert (Modify) bzw. von Ihnen mit einem neuen Wert belegt.

Welche Farbkomponente verändert wird und welche beiden Farbkomponenten unverändert übernommen werden, bestimmen Sie mit dem Bitmuster in den Bitplanes 5 und 6 eines Punktes.

Der Wert %01.... (=0x10) veranlaßt, daß die Rot- und Grünkomponente des Vorgängerpunktes übernommen und die neue Blaukomponente eines Punktes den Wert der Bitplanes 4-1 bekommt.

Schreiben Sie in die Bitplanes 5 und 6 den Wert %10....(0x20), werden die Grün- und die Blaukomponente übernommen und die Rotkomponente wird verändert.

Setzen Sie beide Bits eines Punktes in den Planes 5 und 6, also den Wert %11.... (0x30), dann wird die Grünkomponente verändert.

Was aber passiert, wenn die Bitplanes 5 und 6 an einer Stelle gleich 0 sind? Nun, dann werden die Bitplanes 4-1 dazu benutzt, die Farbe eines Punktes mittels der normalen Colormap darzustellen. Diese 16 normal darstellbaren Farben können z.B. als Ausgangspunkte für die Modifikation benutzt werden, von denen aus Sie dann einfach und schnell Farbänderungen vornehmen können.

Wollen Sie z.B. die Rotkomponente eines Punktes mittels Modifikation ändern, setzen Sie mit SetAPen den Vordergrundstift, der festlegt, welche Bits der Bitplanes für einen Punkt auf 1 und auf 0 gesetzt werden, auf 0x20 und addieren zu diesen 0x20 noch die neue Rot-Intensität (0-15) hinzu. Der dann gesetzte Punkt (z.B. mit WritePixel) übernimmt dann die Grün- und Blaukomponente des Vorgängerpunktes und setzt die neue Rotkomponente auf den angegebenen Wert.

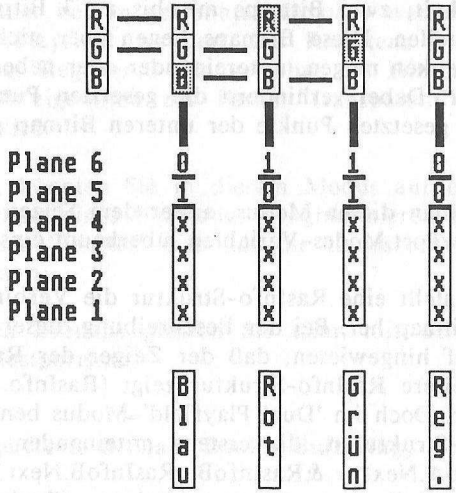
Wollen Sie die Farben aus der Colormap benutzen, brauchen Sie mit SetAPen nur das gewünschte Farbregister, ohne Addition eines weiteren Wertes, anzugeben.

Zu beachten ist, daß der 'Hold and Mofiy'-Modus nicht zeilenübergreifend funktioniert. Der letzte Punkt einer Zeile hat keinen Einfluß

auf den ersten Punkt der nächsten Zeile. Wollen Sie also einen Punkt mit der X-Koordinate 0 setzen, müssen Sie beachten, daß bei seinem 'Vorgängerpunkt', der tatsächlich gar nicht existiert, alle Farbkomponenten gelöscht sind.

Noch ein letztes Wort zu den Bitplanes. Oben haben wir davon gesprochen, daß 5 oder 6 Bitplanes für den HAM-Modus zu reservieren sind. Sollten Sie sich entscheiden, nur 5 Bitplanes zu benutzen (vielleicht, weil Sie nicht mehr genügend freien Speicher haben), dann sollten Sie bedenken, daß Bitplane 6 als gelöscht angenommen wird. Das heißt, daß Sie nur noch Punkte mit den Farben der Colormap setzen oder die Blaukomponente eines Punktes verändern können.

In folgender Abbildung sehen Sie noch einmal die Funktionsweise des HAM-Modus:



Hold and Modify

17.3 Die 'Special Modes'

Die bisher erläuterten Modi befaßten sich alle mehr oder weniger mit der Darstellung der Punkte einer Bitmap. Die hier beschriebenen Modi gehen jedoch näher auf die Organisation der Bitmaps ein. Wie Sie sich überlagernde und gepufferte Bitmaps erzeugen können, erläutern wir im folgenden.

17.3.1 Dual Playfield

Haben Sie schon einmal versucht, mehrere Bitmaps in einem Viewport darzustellen? Die Antwort auf diese Frage dürfte wohl von den meisten von Ihnen mit 'Nein' beantwortet worden sein, denn wer kommt schon auf solch eine Idee?

Doch der Amiga macht's möglich. Im 'Dual Playfield'-Modus haben Sie die Möglichkeit, zwei Bitmaps mit bis zu 3 Bitplanes in einem Viewport darzustellen. Diese Bitmaps liegen aber nicht, wie manche jetzt vielleicht denken mögen, untereinander oder nebeneinander, sondern aufeinander. Dabei verhindern die gesetzten Punkte der oberen Bitmap, daß die gesetzten Punkte der unteren Bitmap gesehen werden können.

Doch wie kann man diesen Modus, außer dem Setzen des DUALPF-Flags in der ViewPort.Modes-Variablen, überhaupt einstellen?

Wie Sie wissen, stellt eine RasInfo-Struktur die Verbindung zwischen Viewport und Bitmap her. Bei der Beschreibung dieser Struktur haben wir schon darauf hingewiesen, daß der Zeiger der RasInfo-Struktur, der auf eine weitere RasInfo-Struktur zeigt (RasInfo.Next), zunächst ohne Belang war. Doch im 'Dual Playfield'-Modus benötigen wir zwei solcher RasInfo-Strukturen, die erstens miteinander verbunden sein müssen, (RasInfoA.Next = &RasInfoB; RasInfoB.Next = NULL;) und zweitens auf jeweils eine Bitmap zeigen (RasInfoA.BitMap = &BitMapA; RasInfoB.BitMap = BitMapB).

Playfield A ist dabei durch die Bitmap repräsentiert, auf die die erste RasInfo-Struktur zeigt, die außerdem noch mit dem Viewport verbunden ist (ViewPort.RasInfo = &RasInfoA;). Playfield B wird analog mit Bitmap Nummer Zwei assoziiert und normalerweise von Playfield A verdeckt, das heißt, daß da, wo in Playfield A Punkte gesetzt wurden (irgendein Bit in den Bitplanes der Bitmap A ist gesetzt), die Punkte aus Playfield B nicht gesehen werden können. Sind jedoch in Playfield

A Punkte gelöscht, kann man durch diese hindurch das Playfield B sehen. (Der Begriff Playfield (dt.: etwa Spielfeld), der eigentlich mit einer Bitmap gleichzusetzen ist, kommt wohl daher, daß hier zwei Bereiche geschaffen werden, in denen man sich nach Lust und Laune 'austoben' kann.)

Wenn Sie dann dann alle Vorbereitungen getroffen haben - die ViewPort.Modes-Variable auf DUALPF gesetzt, die beiden Bitmaps, RasInfos und die Rastports initialisiert - können Sie den Bildschirm normal, das heißt mittels MakeVPort, MrgCop und LoadView weiter öffnen.

Ein kleiner Hinweis sei noch gestattet: Wenn Sie die Videoprioritäten zwischen Playfield A und Playfield B ändern wollen, so daß Playfield B über Playfield A liegt, brauchen Sie nur das Flag PFBA zusätzlich zu DUALPF setzen (ViewPort.Modes = DUALPF | PFBA).

Die Farben dieser beiden getrennten Bitmaps werden von den Farbregistern 0-7 (Playfield A) und den Registern 8-15 (Playfield B) bestimmt, wobei Register 0 wie gewohnt die Hintergrundfarbe stellt und Register 8 eigentlich zur Farbe 'Transparent' aus Playfield B herangezogen wird.

Wie Sie sehen, könnten Sie in diesem Modus auf einfache Art und Weise das Cockpit eines Flugzeugs programmieren (z.B. in Playfield A), durch dessen 'Fenster' man auf die Landschaft (in Playfield B) herabblicken kann.

Auf die beiden Bitmaps greifen Sie dann mit den zwei zusätzlich initialisierten Rastports zu.

17.3.2 Gepufferte Bitmap (Double Buffering)

Hat Sie nicht schon immer interessiert, wie man Grafiken im Hintergrund aufbaut, während eine andere gerade dargestellt wird? Nun, wenn dem so ist, dann lesen Sie ruhig weiter, denn hier erfahren Sie, wie eben das zu machen ist.

Um diese sogenannten 'zweifach gepufferten' Displays zu erstellen, brauchen wir zuerst einmal zwei Bitmaps, denn irgendwo müssen die Grafiken ja abgespeichert werden. Diese beiden Bitmaps sollten allerdings gleich groß sein!

Der Rest, der dann noch zu erledigen ist, ist eigentlich sehr einfach: Sie öffnen einen Bildschirm wie gewohnt, indem Sie View, Viewport usw. initialisieren.

Dann lassen Sie die Zeiger der RasInfo-Struktur auf die erste Bitmap zeigen und machen wie gewohnt weiter (MakeVPort, MrgCop). LoadView wird aber noch nicht ausgeführt, denn Sie sichern sich jetzt erst einmal die Adressen der beiden von MrgCop generierten Hardware-Copper-Listen z.B. in einem Array:

```
struct CprList *Copper[2][2];
...

Copper[0][0] = View.LOFCprList;
Copper[0][1] = View.SHFCprList;
```

Dann lassen Sie den RasInfo-Zeiger für die Bitmaps auf die zweite Bitmap zeigen und generieren mit MakeVPort und MrgCop die zweite Hardware-Copper-Liste für die zweite Bitmap. Allerdings müssen Sie vorher die beiden Zeiger auf die Hardware-Copper-Listen des Views wieder auf Null gesetzt haben, da MrgCop sonst annimmt, daß schon eine Copper-Liste existiert und keine zweite generiert zu werden braucht.

Die so zum zweitenmal generierten Copper-Listen retten Sie sich auch (Copper[1][0] = View.LOFCprList; Copper[1][1] = View.SHFCprList;). Dann können Sie getrost LoadView ausführen, wodurch dann die zweite Bitmap auf dem Bildschirm dargestellt wird.

So stellen Sie dann eine Bitmap dar, während Sie die andere mittels der RastPort-Strukturen, die Sie natürlich noch zusätzlich für beide Bitmaps initialisiert haben, kräftig bearbeiten können. Ist die zweite Bitmap fertiggestellt, laden Sie in eine View-Struktur einfach die vorher geretteten Adressen der Copper-Listen für die Bitmap (View-LOFCprList = Copper[x][0]; und View.SHFCprList = Copper[x][1];), und führen wieder LoadView aus. Und so geht das immer weiter. Natürlich müssen Sie auch dafür sorgen, daß richtig auf die Bitmaps zugegriffen wird. Dazu sollten Sie die Rastports der beiden Bitmaps auch als Array organisieren, in dem Sie ähnlich wie beim 'Copper'-Array hin- und herschalten können.

In dem nun folgenden Programm finden Sie alle die oben beschriebenen Modi, auf mehrere Viewports verteilt, in einem View dargestellt.

```
/*********************************************************/
/*                    The_Modes.c                        */
/*                                                       */
/* Dieses Programm zeigt alle möglchen Darstellungsmodi  */
/* auf einem Bildschirm in 4 ViewPorts.                  */
/*                                                       */
/* Compiled with: Lattice V3.10                          */
/*                                                       */
/* (c) Bruno Jennrich                                    */
/*********************************************************/

#include "exec/types.h"
#include "graphics/gfx.h"
#include "graphics/rastport.h"
#include "graphics/copper.h"
#include "graphics/view.h"
#include "graphics/gels.h"
#include "graphics/regions.h"
#include "graphics/clip.h"
#include "exec/exec.h"
#include "graphics/text.h"
#include "graphics/gfxbase.h"
#include "hardware/dmabits.h"
#include "hardware/custom.h"
#include "hardware/blit.h"
#include "devices/keymap.h"

#define DUAL1 3  /* Indizes für DualPlayfield (Zugriff auf Rast- */
#define DUAL2 4  /* und ViewPort)                                */

#define DBUFF1 DUAL1                /* Double-Buffer      */
#define DBUFF2 DUAL2+1              /* Indizes            */

UWORD BUFF[2] = {DBUFF1,DBUFF2};    /* Für Double-Buffer  */
                                    /* (werden getriggert) */

int Trigger;                           /* Triggervariable */

struct View View;
struct ViewPort ViewPorts[4];           /* Vier ViewPorts */


struct RasInfo RasInfos[6];         /* Sechs BitMaps. Für  */
struct BitMap BitMaps[6];           /* die 'oberen' 3 Modi */
struct RastPort RastPorts[6];       /* jeweils eine, dann  */
                                    /* für DualPF 2 und    */
                                    /* für Double Buffer   */
                                    /* eine weitere ( = 6) */

SHORT i,j,k,l,n;                       /* Hilfsvariablen */
```

```
struct GfxBase *GfxBase;

struct View *oldview;          /* Um alten View zu retten */

UWORD ColorTable[4][16] = {{
                            0x000,0x00F,0x0F0,0xF00,
                            0xFF0,0x0FF,0xF0F,0xABC,
                            0xFF2,0xD30,0x7CF,0x4C3,
                            0x7A8,0x9C6,0x1D6,0xFD9
                           },
                           {
                            0x000,0x00F,0x0F0,0xF00,
                            0xFF0,0x0FF,0xF0F,0xABC,
                            0xFF2,0xD30,0x7CF,0x4C3,
                            0x7A8,0x9C6,0x1D6,0xFD9
                           },
                           {
                            0x000,0x100,0x200,0x300,
                            0x400,0x500,0x600,0x700,
                            0x800,0x900,0xA00,0xB00,
                            0xC00,0xD00,0xE00,0xF00,
                           },
                           {
                            0x000,0x00F,0x0F0,0x00f,
                            0xFF0,0x0FF,0xF0F,0x000,
                            0xFF2,0xD30,0x7CF,0x4C3,
                            0x7A8,0x9C6,0x1D6,0xFD9
                           }};
                     /* Farbpalette für eigene ViewPorts */
int x,y;
int rot,gruen,blau;                    /* Für HAM-Aufbau */
int Length;

char *LeftMouse = (char *) 0xBFE001;       /* HARDWARE !!! */

char *LaceString = "Uff !! Ganz schön eng !";
char *HAMString = "Der HAMmer";
char *HighString = "Das ist der HIRES Power !!!";
char *Pf2String = "Nix' los im Vordergrund !!!";


struct cprlist *LOF[2];       /* Double Buffer CopperLists */
struct cprlist *SHF[2];

int Ecken[4][2] = {{50,23},            /* Für Bewegung des */
                   {270,23},           /* Rechtecks        */
                   {50,43},
                   {270,43}};
```

```
int Veloc[4][2] = {{2,-3},          /* 'Geschwindigkeiten' */
                   {3,2},
                   {-3,-2},
                   {-2,3}};


VOID Make();
VOID FreeMemory();

/*********************************************************/
/* Hier gehts ab !!!                                     */
/*********************************************************/

main()
{
   if ((GfxBase = (struct GfxBase *)
      OpenLibrary("graphics.library",0)) == NULL)
      Exit (1000);

   oldview = GfxBase->ActiView;
                                   /* Rette alten View */

   InitView(&View);              /* Initialisiere neuen   */
   for (i=0; i<4; i++)           /* View, sowie ViewPorts */
      InitVPort(&ViewPorts[i]);

   View.ViewPort = &ViewPorts[0];    /* Verbinde View mit */
                                     /* erstem ViewPort   */

   View.Modes = HIRES | LACE;    /* Höchster Auflösungs-   */
                                 /* modus für View, damit  */
                                 /* LACE und HIRES in View-*/
                                 /* Ports möglich werden   */

   InitBitMap(&BitMaps[0],6,320,65);
                        /* 1.  LACE | HALFBRITE BitMap */

   InitBitMap(&BitMaps[1],4,640,33);
                                   /* 2.  HIRES BitMap */

   InitBitMap(&BitMaps[2],6,320,67);
                                    /* 3.  HAM BitMap */

   InitBitMap(&BitMaps[DUAL1],3,320,67);
   InitBitMap(&BitMaps[DUAL2],3,320,67);
                          /* 4.  DualPlayfield BitMaps */
```

```
InitBitMap(&BitMaps[DBUFF2],3,320,67);
                          /* Double Buffering BitMap */
                          /* DUALPF1 BitMap wird     */
                          /* gebuffert               */

for (i=0; i<6; i++)
   {
      RasInfos[i].BitMap = &BitMaps[i];
      RasInfos[i].RxOffset = 0;
      RasInfos[i].RyOffset = 0;
      RasInfos[i].Next = NULL;
                    /* Alle RasInfos initialisieren */
   }


ViewPorts[0].DxOffset = 0;    ViewPorts[0].DyOffset = 0;
ViewPorts[0].DWidth = 320;    ViewPorts[0].DHeight = 64;
ViewPorts[0].RasInfo = &RasInfos[0];
ViewPorts[0].Modes = LACE | EXTRA_HALFBRITE;
ViewPorts[0].Next = &ViewPorts[1];
         /* LACE | EXTRA_HALFBRITE ViewPort (erster) */


ViewPorts[1].DxOffset = 0;    ViewPorts[1].DyOffset = 33;
ViewPorts[1].DWidth = 640;    ViewPorts[1].DHeight = 32;
ViewPorts[1].RasInfo = &RasInfos[1];
ViewPorts[1].Modes = HIRES;
ViewPorts[1].Next = &ViewPorts[2];
                        /* HIRES ViewPort (zweiter) */

ViewPorts[2].DxOffset = 0;    ViewPorts[2].DyOffset = 66;
ViewPorts[2].DWidth = 320;    ViewPorts[2].DHeight = 66;
ViewPorts[2].RasInfo = &RasInfos[2];
ViewPorts[2].Modes = HAM;
ViewPorts[2].Next = &ViewPorts[3];
                          /* HAM ViewPort (dritter) */

ViewPorts[3].DxOffset = 0;  ViewPorts[3].DyOffset = 133;
ViewPorts[3].DWidth = 320;  ViewPorts[3].DHeight = 66;
ViewPorts[3].RasInfo = &RasInfos[DUAL1];
ViewPorts[3].Modes = DUALPF | PFBA;
                 /* Playfield 2 liegt vor Playfield 1 */

ViewPorts[3].Next = NULL;
/* Dual-Playield und Double Buffer ViewPort (vierter) */

RasInfos[DUAL1].Next = &RasInfos[DUAL2];
                          /* DUALPF RasInfos verbinden */
```

```
ViewPorts[0].ColorMap=(struct ColorMap*)GetColorMap(16);
ViewPorts[1].ColorMap=(struct ColorMap*)GetColorMap(16);
ViewPorts[2].ColorMap=(struct ColorMap*)GetColorMap(16);
ViewPorts[3].ColorMap=(struct ColorMap*)GetColorMap(16);
       /* Für jeden ViewPort ColorMap-Speicher holen */

for (i=0;i<4;i++)
         LoadRGB4(&ViewPorts[i],&ColorTable[i][0],16);
            /* Jedem ViewPort seine eigenen Farben */

          /* Hole jetzt den Speicher für die BitMaps */
for (i=0; i<6; i++)
   {
      if ((BitMaps[0].Planes[i] =
         (PLANEPTR)AllocRaster(320,65)) == NULL)
         Exit (1000);              /* LACE Requirements */
      BltClear ((UBYTE *)BitMaps[0].Planes[i],
                        RASSIZE(320,65),0);
   }

for (i=0; i<4; i++)
   {
     if ((BitMaps[1].Planes[i] =
        (PLANEPTR)AllocRaster(640,33)) == NULL)
        Exit (1000);              /* HIRES Requirements */
     BltClear ((UBYTE *)BitMaps[1].Planes[i],
                       RASSIZE(640,33),0);
   }

for (i=0; i<6; i++)
   {
      if ((BitMaps[2].Planes[i] =
         (PLANEPTR)AllocRaster(320,67)) == NULL)
         Exit (1000);              /* HAM Requirements */
      BltClear ((UBYTE *)BitMaps[2].Planes[i],
                  RASSIZE(320,67),0);
   }

for (i=0; i<3; i++)
   {
      if ((BitMaps[DUAL1].Planes[i] =
         (PLANEPTR)AllocRaster(320,67)) == NULL)
         Exit (1000);              /* DUALPF1 Requirements */
      BltClear ((UBYTE *)BitMaps[DUAL1].Planes[i],
                  RASSIZE(320,67),0);
   }

for (i=0; i<3; i++)
   {
```

```
      if ((BitMaps[DUAL2].Planes[i] =
         (PLANEPTR)AllocRaster(320,67)) == NULL)
         Exit (1000);          /* DUALPF2 Requirements */
      BltClear ((UBYTE *)BitMaps[DUAL2].Planes[i],
                RASSIZE(320,67),0);
   }

for (i=0; i<3; i++)
   {
      if ((BitMaps[DBUFF2].Planes[i] =
         (PLANEPTR)AllocRaster(320,67)) == NULL)
         Exit (1000);    /* Double Buffer Requirements */
      BltClear ((UBYTE *)BitMaps[DBUFF2].Planes[i],
                RASSIZE(320,67),0);
   }

for (i=0; i<6; i++)
   {
      InitRastPort (&RastPorts[i]);
      RastPorts[i].BitMap = &BitMaps[i];
                          /* RastPorts vorbereiten */
   }

MakeVPort(&View,&ViewPorts[0]);    /* Für jeden View-  */
MakeVPort(&View,&ViewPorts[1]);    /* Port die Copper- */
MakeVPort(&View,&ViewPorts[2]);    /* Liste berechnen  */
MakeVPort(&View,&ViewPorts[3]);

MrgCop (&View);              /* Alle ViewPort Listen zu */
                             /* einer zusammen'mergen'  */

LOF[0] = View.LOFCprList;       /* Listen für Double-  */
SHF[0] = View.SHFCprList;       /* Buffer sichern      */

ViewPorts[3].RasInfo = &RasInfos[DBUFF2];
RasInfos[DBUFF2].Next = &RasInfos[DUAL2];
    /* RasInfo für DUALPF & Double Buffer präparieren */

View.LOFCprList = NULL;
View.SHFCprList = NULL;                /* Wichtig !!!!!! */
                  /* Sonst keine zweite CopperListe */

MakeVPort(&View,&ViewPorts[0]);          /* Dasselbe wie */
MakeVPort(&View,&ViewPorts[1]);          /* oben nochmal */
MakeVPort(&View,&ViewPorts[2]);          /* für Double-  */
MakeVPort(&View,&ViewPorts[3]);          /* Buffering    */

MrgCop (&View);

LOF[1] = View.LOFCprList; /* Zweite CopperListe retten */
```

```
SHF[1] = View.SHFCprList;

          /* Ab jetzt werden die BitMaps beschrieben, */
          /* und die Bilder aufgebaut                 */

          /* Das erste ist der HALFBRITE | LACE Modus */
for (x=0; x<16 ; x++)
     {
        SetAPen(&RastPorts[0],x);
        RectFill(&RastPorts[0],x*(320/16)+1,1,
                               (x+1)*(320/16)-1,31);
        SetAPen(&RastPorts[0],x+32);
                                   /* Für Halfbrite */
        RectFill(&RastPorts[0],x*(320/16)+1,32,
                               (x+1)*(320/16)-1,62);
     }

SetAPen (&RastPorts[0],2);

Move (&RastPorts[0],0,0);          /* Rahmen zeichnen */
Draw (&RastPorts[0],319,0);
Draw (&RastPorts[0],319,63);
Draw (&RastPorts[0],0,63);
Draw (&RastPorts[0],0,0);

SetAPen(&RastPorts[0],5);            /* Text ausgeben */
SetDrMd (&RastPorts[0],JAM1);
Length=TextLength(&RastPorts[0],LaceString,
                  strlen(LaceString));
Move (&RastPorts[0],320/2-Length/2,
                  64/2+RastPorts[0].TxBaseline);
Text (&RastPorts[0],LaceString,strlen(LaceString));

                                   /* HIRES Viewport */
for (i=1;i<31;i++)
   {                               /* Ein paar Linien */
     SetAPen(&RastPorts[1],i%16);  /* zeichnen        */
     Move(&RastPorts[1],1,i);
     Draw(&RastPorts[1],638,30-i);
   }

for (i=1;i<638;i++)
   {
     SetAPen(&RastPorts[1],i%16);
     Move(&RastPorts[1],i,1);
     Draw(&RastPorts[1],638-i,31);
   }

SetAPen(&RastPorts[1],2);
```

```
Move (&RastPorts[1],0,0);                    /* N'en Rahmen */
Draw (&RastPorts[1],639,0);
Draw (&RastPorts[1],639,31);
Draw (&RastPorts[1],0,31);
Draw (&RastPorts[1],0,0);

SetDrMd(&RastPorts[1],JAM1);                      /* Text */
SetAPen(&RastPorts[1],0);
Length = TextLength(&RastPorts[1],HighString,
                    strlen(HighString));
Move (&RastPorts[1],640/2-Length/2,
                    32/2+RastPorts[1].TxBaseline);
Text (&RastPorts[1],HighString,strlen(HighString));

                     /* Nun der Hold and Modify Modus */
SetAPen(&RastPorts[2],15+0x20);

Move (&RastPorts[2],0,0);                        /* Rahmen */
Draw (&RastPorts[2],319,0);
Draw (&RastPorts[2],319,65);
Draw (&RastPorts[2],0,65);
Draw (&RastPorts[2],0,0);

SetDrMd(&RastPorts[2],JAM1);
SetAPen(&RastPorts[2],14+0x30);

x = 0;
y = 1;
rot = 0;
gruen = 0;

for (i=0; i<64; i++)                          /* 64 Zeilen */
   {
   x = 320/2-(64+8)/2;
   for (l=0; l<4; l++)
      {
         SetAPen(&RastPorts[2],(rot + 0x20));   /* Rot */
         WritePixel(&RastPorts[2],x,y+i);
         x++;
         SetAPen(&RastPorts[2],(gruen + 0x30));
         WritePixel(&RastPorts[2],x,y+i);       /* Grün */
         x++;
         for (blau=0; blau<16; blau++)
            {
               SetAPen(&RastPorts[2],(blau + 0x10));
               WritePixel(&RastPorts[2],x,y+i);/* Blau */
               x++;
            }
         gruen++;
      }
```

```
    if (gruen == 16){gruen = 0; rot ++;}
    }

l = 2;
Length = TextLength(&RastPorts[2],HAMString,
                    strlen(HAMString));
for (i=1; i<(66-RastPorts[2].TxHeight);
          i += RastPorts[2].TxHeight)
{
   SetAPen(&RastPorts[2],l);
   l++;
   Move (&RastPorts[2],1,
                    i+RastPorts[2].TxBaseline);
   Text (&RastPorts[2],HAMString,strlen(HAMString));
   Move (&RastPorts[2],320-Length-1,          /* Texte */
                    i+RastPorts[2].TxBaseline);
   Text (&RastPorts[2],HAMString,strlen(HAMString));
}

                                  /* DUALPF2 'bemalen' */
SetAPen(&RastPorts[DUAL2],1);

RectFill (&RastPorts[DUAL2],0,0,319,6);
RectFill (&RastPorts[DUAL2],0,59,319,65);

RectFill (&RastPorts[DUAL2],0,6,16,59);
RectFill (&RastPorts[DUAL2],303,6,319,59);

SetAPen(&RastPorts[DUAL2],2);

Length = TextLength(&RastPorts[DUAL2],Pf2String,
                    strlen(Pf2String));
Move (&RastPorts[DUAL2],320/2-Length/2,
                  66/2+RastPorts[DUAL2].TxBaseline);
Text (&RastPorts[DUAL2],Pf2String,strlen(Pf2String));

Trigger = 0;

SetAPen(&RastPorts[BUFF[0]],10);  /* Farben für beide  */
SetAPen(&RastPorts[BUFF[1]],10);  /* BitMaps setzen    */

while ((*LeftMouse & 0x40) == 0x40)
   {                    /* Bei Mausclick Programmende */

     SetRast (&RastPorts[BUFF[Trigger]],0);
     Move(&RastPorts[BUFF[Trigger]],Ecken[0][0],
                                    Ecken[0][1]);
                        /* Zeichne ein paar Linien */
```

```
        for (i=1; i<4; i++)
           {
              Draw(&RastPorts[BUFF[Trigger]],Ecken[i][0],
                                              Ecken[i][1]);
           }

        Draw(&RastPorts[BUFF[Trigger]],Ecken[0][0],
                                       Ecken[0][1]);
        for (i=0; i<4; i++)
           {
              Ecken[i][0] += Veloc[i][0];
              Ecken[i][1] += Veloc[i][1];

              if ((Ecken[i][0] <= 0) || Ecken[i][0] >= 319)
                 {
                    Ecken[i][0] -= Veloc[i][0];
                    Veloc[i][0] = -Veloc[i][0];
                 }

              if ((Ecken[i][1] < 0) || (Ecken[i][1] >= 65))
                 {
                    Ecken[i][1] -= Veloc[i][1];
                    Veloc[i][1] = -Veloc[i][1];
                 }
              }

        Make(&View,Trigger);          /* Schalte View um */
        Trigger ^= 1;       /* andere BitMap 'anschalten' */
     }


   LoadView(oldview);                 /* Alten View Laden */
   FreeMemory();
   return(0);
}

/*********************************************************/
/* Diese Funktion sorgt für das Umschalten der beiden    */
/* View CopperListen.                                    */
/*-------------------------------------------------------*/
/* Eingabe-Parameter: View: der die neuen Listen erhält, */
/*                    Trigger: Welche CopperListe ?      */
/*-------------------------------------------------------*/
/* Rückgabe-Werte: keine                                 */
/*********************************************************/

VOID Make(View,Trigger)
struct View *View;
int Trigger;
   {
```

```
      View->LOFCprList = LOF[Trigger];
      View->SHFCprList = SHF[Trigger];
      LoadView(View);
      WaitTOF();
   }

/*********************************************************/
/* Diese Funktion gibt den durch die BitMaps belegten    */
/* Speicher, sowie den restlichen reservierten Seicher an */
/* das System zurück                                     */
/*-------------------------------------------------------*/
/* Eingabe-Parameter: keine                              */
/*-------------------------------------------------------*/
/* Rückgabe-Werte: keine                                 */
/*********************************************************/

VOID FreeMemory()
{
   for (i=0; i<6; i++)
         FreeRaster(BitMaps[0].Planes[i],320,64);
                                    /* LACE | HALFBRITE */

   for (i=0; i<4; i++)
         FreeRaster(BitMaps[1].Planes[i],640,32);
                                              /* HIRES */

   for (i=0; i<6; i++)
         FreeRaster(BitMaps[2].Planes[i],320,66);
                                                /* HAM */

   for (i=0; i<3; i++)
         FreeRaster(BitMaps[DUAL1].Planes[i],320,66);
                                    /* DualPlayfield 1 */

   for (i=0; i<3; i++)
         FreeRaster(BitMaps[DUAL2].Planes[i],320,66);
                                    /* DualPlayfield 2 */

   for (i=0; i<3; i++)
         FreeRaster(BitMaps[DBUFF2].Planes[i],320,66);
                                      /* Double Buffer */

   for (i=0;i<4;i++)
      FreeColorMap(ViewPorts[i].ColorMap);
                               /* ColorMaps zurückgeben */

   for (i=0; i<4; i++)
      FreeVPortCopLists(&ViewPorts[i]);
                        /* ViewPort CopperListen zurückgeben */
```

```
    FreeCprList(LOF[0]);
    FreeCprList(SHF[0]);
    FreeCprList(LOF[1]);
    FreeCprList(SHF[1]);
                                    /* Free the Copper ! */

    CloseLibrary(GfxBase);
}
```

17.4 Andere Modi

Nun kommen wir zu der Rubrik 'Sonstiges'. Aber 'Sonstiges' hat's in sich: Hier erfahren Sie nämlich alles über die Sprites.

Doch vorher, nur der Vollständigkeit halber, noch ein Modus, der eigentlich gar kein Modus ist:

17.4.1 VP_HIDE

Setzen Sie dieses Modes-Flag, dann wird der Viewport, in dem es gesetzt wurde, bei der Generierung der Copper-Listen 'übergangen'. Dies hat zur Folge, daß er später auch nicht auf dem Monitor erscheint. (Dieses Flag wird zum Beispiel dann gesetzt, wenn Sie einen Screen in den Hintergrund verbannen.)

17.4.2 Sprites

Nun kommen wir zu dem Thema Sprites. Sicherlich haben Sie von diesen Errungenschaften der Amiga-Technik schon in Computerzeitschriften usw. gelesen. Aber Sie werden sicher auch gelesen haben, daß z.B. die VSprites nicht funktionierten. Aber dennoch zeigen wir Ihnen (im nächsten Kapitel), wie es geht.

Doch bevor wir uns näher mit den VSprites beschäftigen, wollen wir Ihnen erst einmal die Hardware-Sprites ans Herz legen.

17.4.2.1 Die Hardware-Sprites

Wie der Name schon sagt, sind Hardware-Sprites 'Errungenschaften' der Hardware des Amiga (bei den VSprites 'mischt' noch eine geschickte Copper-Programmierung mit). Von diesen Hardware-Sprites

gibt es insgesamt 8 Stück, deren Aussehen Sie selbst und unabhängig voneinander bestimmen und die Sie frei und unabhängig voneinander über den Bildschirm bewegen können.

Allerdings sind diesen Kleingrafiken gewisse Beschränkungen auferlegt: Erstens sind sie immer nur 16 Punkte breit, dafür aber beliebig hoch, und zweitens können sie nur in 3, mit einem kleinen Trick aber in 15 Farben dargestellt werden.

Doch wie initialisiert man diese Sprites? Nun, zuerst müssen Sie eine 'SimpleSprite'-Struktur anlegen (z.B. mit 'struct SimpleSprite SimpleSprite;'). Diese dient allen Sprite-Befehlen - GetSprite, ChangeSprite und MoveSprite - als 'Schlüssel' zu ihrem Hardware-Sprite. Natürlich sollten Sie beachten, daß Sie, bevor Sie Sprites, egal ob Hardware- oder VSprites, überhaupt benutzen können, das SPRITES-Flag im Viewport (in der Modes-Variablen) setzen müssen.

Haben Sie diese beiden Voraussetzungen erfüllt - die 'Simple-Sprite'-Struktur angelegt und das SPRITES-Flag gesetzt - kann es mit der Programmierung eigentlich schon losgehen.

Als ersten Schritt auf dem Weg zu den eigenen Sprites müssen Sie die Funktion GetSprite aufrufen. Diese überprüft, ob Sie überhaupt das von Ihnen gewünschte Sprite benutzen können.

Mit 'status = GetSprite (&SimpleSprite, SpriteNummer)' können Sie beim System anfragen, ob das Sprite mit der Nummer 'SpriteNummer' noch zu haben ist (Die Sprites sind von 0 bis 7 durchnumeriert). Denn es ist ja möglich, daß erstens andere, gleichzeitig im Speicher ablaufende Programme schon das Sprite benutzen, oder daß das Sprite für die Darstellung der VSprites benutzt wird. Wenn nämlich zwei Benutzer auf ein und dasselbe Sprite zugreifen wollen, gibt's Streit, der sich darin äußert, daß die Sprites nicht mehr vernünftig dargestellt werden.

Wenn Sie also in 'SpriteNummer' bestimmt haben, welches Sprite (0-7) Sie 'haben' wollen, wird Ihnen zur Bestätigung dies auch in der Variablen 'Status' mitgeteilt, die dann die Nummer des Ihnen zugewiesenen Sprites enthält. Ist es Ihnen aber egal, welches Hardware-Sprite Sie benutzen wollen, dann geben Sie in 'SpriteNummer' einfach -1 an, und in 'Status' steht dann, welches Sprite Sie bekommen haben. Steht in Status aber '-1', bedeutet das in beiden Fällen, daß das von Ihnen gewünschte bzw. alle Sprites schon 'anderweitig beschäftigt' ist bzw. sind.

Damit die beiden Sprite-Befehle aber wissen, auf welches Sprite sich die 'SimpleSprite'-Struktur bezieht, wird auch dort, in der Variablen 'SimpleSprite.num', die Nummer des Hardware-Sprites, das nach GetSprite mit dieser Struktur assoziiert wird, abgespeichert.

Somit reicht es also, wenn Sie für die anderen Sprite-Befehle nur die 'SimpleSprite'-Struktur des zu beeinflussenden Sprites angeben.

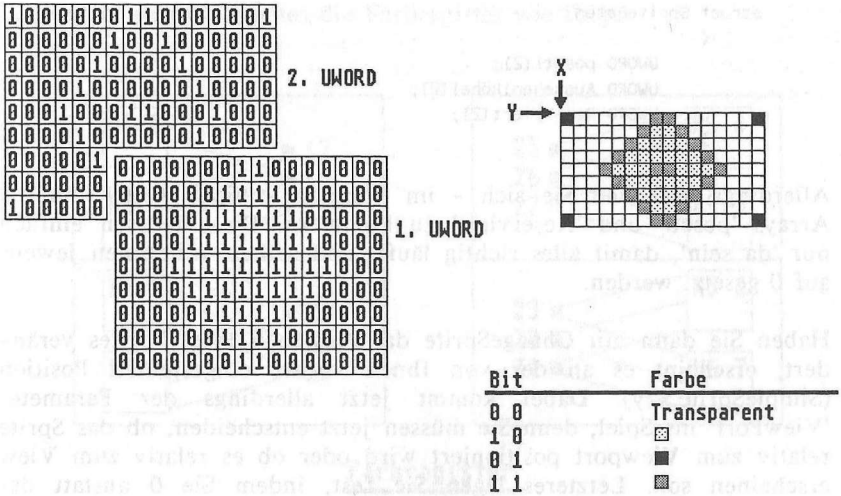
Beim Programmende sollten Sie aber vorher mit 'FreeSprite(Status)' das von Ihnen 'in Beschlag' genommene Sprite wieder zur erneuten Benutzung an das System zurückgeben - also die 'Status'-Variable bitte 'im Hinterkopf' behalten und nicht für andere Zwecke mißbrauchen.

Doch zurück zur Initialisierung der Sprites. Haben Sie nämlich ein Sprite bekommen, müssen Sie jetzt die Höhe, das heißt die Anzahl der Linien des Sprites bestimmen. Dazu weisen Sie der Variablen 'SimpleSprite.height' einfach den gewünschten Wert zu.

Danach bestimmen Sie in den Variablen 'SimpleSprite.x' und 'SimpleSprite.y' die Ausgangsposition des Sprites, also die Initial-Position, an der das Sprite zuerst auf dem Bildschirm zu sehen ist, nachdem Sie den nächten Sprite-Befehl - ChangeSprite - ausgeführt haben.

Dieser verändert nämlich das Aussehen Ihres Sprites, doch wie wird dieses bestimmt?

Wie oben schon erwähnt, sind Hardware-Sprites - ohne Tricks - immer dreifarbig. Um drei Farben darzustellen, werden aber mindestens zwei Farbinformationen (= Bits) pro Punkt benötigt. Dazu wird jede Sprite-Zeile in zwei aufeinanderfolgenden UWORDs (#define UWORD unsigned int (s. 'exec/types.h')), die ja 16 Bit breit sind, definiert.



Somit könnten Sie theoretisch vier anstatt drei Farben darstellen. Doch die Farbe 'Transparent', die dargestellt wird, wenn beide Bits an einer Stelle in den UWORDs einer Sprite-Zeile gleich 0 sind, ist doch keine Farbe, oder?

Um das Aussehen der Sprites festzulegen, erweist es sich also als angenehm, ein zweidimensionales Array zu definieren, in dem alle Sprite-Zeilen in jeweils zwei UWORDs definiert werden:

```
UWORD Aussehen [Höhe][2] =
                {
                    0x....,  0x....,  /* erste Zeile */

                    ...

                    0x....,  0x....   /* letzte Zeile */
                }
```

Leider müssen Sie dem 'ChangeSprite(&ViewPort, &SimpleSprite, &SpriteDaten)'-Befehl nicht die Anfangsadresse eines solchen Arrays übergeben. Für die korrekte Darstellung der Sprites werden nämlich noch 4 extra UWORDs benötigt, die als Speicher für das Sprite-System dienen. Zwei UWORDs werden vor das UWORD-Array gesetzt, das das Aussehen bestimmt, und zwei weitere 'hintendran' gehängt, so daß sich für 'SpriteDaten', dessen Adresse 'ChangeSprite' übergeben werden muß, folgende Struktur ergibt:

```
struct SpriteDaten
    {
        UWORD posctl[2];
        UWORD Aussehen[Höhe][2];
        UWORD Reserviert[2];
    }
```

Allerdings brauchen Sie sich - im Moment - nicht weiter um die Arrays 'posctl' und 'Reserviert' zu kümmern. Diese müssen einfach nur 'da sein', damit alles richtig läuft, und sollten von Ihnen jeweils auf 0 gesetzt werden.

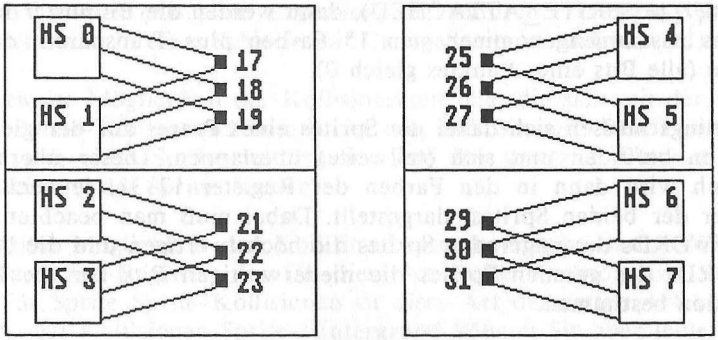
Haben Sie dann mit ChangeSprite das Aussehen Ihres Sprites verändert, erscheint es an der von Ihnen vorher eingestellten Position (SimpleSprite.x/y). Dabei kommt jetzt allerdings der Parameter 'ViewPort' ins Spiel, denn Sie müssen jetzt entscheiden, ob das Sprite relativ zum Viewport positioniert wird oder ob es relativ zum View erscheinen soll. Letzteres legen Sie fest, indem Sie 0 anstatt der Adresse des Viewports übergeben.

Somit haben wir dann schon alles Nötige getan, um ein Hardware-Sprite auf den Bildschirm zu bringen. Um es dann noch zu bewegen, können Sie mit 'MoveSprite(&ViewPort, &SimpleSprite, X,Y);' das Sprite an eine neue Koordinate setzen, wobei auch hier 'ViewPort' bestimmt, ob das Sprite relativ zum View oder zum Viewport positioniert wird.

Allerdings ist es so, daß Sprites nur Lo-Res-Positionen einnehmen können. Wenn Sie also ein Sprite in einem Hi-Res- und Lace-Viewport bewegen wollen, müssen Sie beachten, daß dabei z.B. die Position 320/200 bedeutet, daß sich das Sprite erst in der Mitte des Bildschirms befindet. Um ein Sprite also in diesen Modi um einen Punkt zu bewegen, müssen Sie tatsächlich in diesen die Position jeweils um 2 ändern.

Leider müssen wir Sie aber noch auf ein Manko hinweisen: Die Hardware-Sprites können nicht unabhängig voneinander eingefärbt werden, das heißt, daß jeweils zwei Sprites vier Farbregister teilen. (Diese Farbregister sind dieselben, die auch in Ihrem Rastport die Farben der gesetzten Punkte bestimmen).

So 'teilen' sich die Sprites die Farbregister wie folgt:



Farbregister und Hardware-Sprites

Dabei bestimmen folgende Bitkombinationen für einen Sprite-Punkt, welches Farbregister für einen Punkt zur Darstellung verwendet wird:

1. UWORD	2. UWORD	Farbregister für Sprite 0/1	2/3	4/5	6/7
1	0	17	21	25	29
0	1	18	22	26	30
1	1	19	23	27	31

Das Bitmuster 00 läßt den Hintergrund durchscheinen, solch ein Punkt ist also transparent.

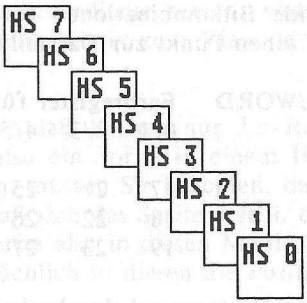
17.4.2.2 Hardware-Sprites in 15 Farben

Aber wie lassen sich denn nun Sprites in 15 Farben darstellen? Tja, wie Sie sehen, kann man sagen, daß die Sprites in 4 Paare unterteilt sind - je ein 'ungerades' und ein 'gerades' Sprite bilden ein Paar (Sprite 0/1, 2/3, 4/5, 6/7).

Wenn Sie dann, nachdem Sie die 'SpriteData'-Struktur für beide Sprites eines Paares initialisiert haben, in beiden Sprites eines Paares das 'SPRITE_ATTACHED'-Bit (attached = verbunden) setzen (posctl[1] |= SPRITE_ATTACHED), dann werden die Bitmuster beider Sprites zusammengenommen, um 15 Farben plus Transparent darzustellen (alle Bits eines Punktes gleich 0).

Allerdings müssen sich dabei die Sprites eines Paares auf der gleichen Position befinden und sich (teilweise) überlappen. Dieser überlappte Bereich wird dann in den Farben der Register 17-31, je nach Bitmuster der beiden Sprites, dargestellt. Dabei muß man beachten, daß die UWORDs des ungeraden Sprites die höchstwertigen und die beiden UWORDs des geraden Sprites die niederwertigen Bits für die Farbselektion bestimmen.

Damit kommen wir zu einer weiteren Eigenschaft der Sprites. Diese haben nämlich eine festgefügte Videopriorität. Wenn sich alle Sprites überlappen, dann ist also Sprite 0 'das oberste Sprite', 'darunter' ist dann Sprite 1, und das Sprite mit der geringsten Videopriorität ist Sprite 7.



17.4.2.3 Wann sind Sprites zusammengestoßen?

Nachdem Sie oben also gesehen haben, wie man Hardware-Sprites initialisiert und steuert, haben Sie sich sicher schon gedacht, daß solche Kleingrafiken gut in 'Action-Spielen' zu verwenden sind.

Doch wie können Sie feststellen, ob z.B. ein Geschoß, das durch ein Sprite dargestellt wird, sein Ziel getroffen oder verfehlt hat? Dazu gibt es drei Möglichkeiten.

Die erste und einfachste ist die, daß Sie die Position des Sprites (Ihres Geschosses o.ä.) laufend kontrollieren, und wenn die Position dieses Sprites und die Position eines anderen Objektes gleich sind, evtl. eine Aktion auslösen (z.B. eine Explosion, die dadurch erreicht wird, daß ein Sprite z.B. in einer Schleife mit ChangeSprite verändert wird).

Eine zweite Möglichkeit der Kollisionskontrolle, die sich mit der Kollision Sprite-Hintergrund befaßt, ist die, daß Sie mit 'ReadPixel' die Umgebung eines Sprites untersuchen und bei gesetzten Punkten z.B. die Richtung des Sprites ändern.

Die dritte Möglichkeit der Kollisionskontrolle besteht darin, daß Sie das Hardware-Register, das für diese Kontrolle zuständig ist, abfragen. Für Sprite-Sprite-Kollisionen ist diese Art der ersten Art vorzuziehen, bei Kollisionen Sprite-Hintergrund können Sie aber leider nur testen, 'ob' oder 'ob nicht'. Sie können zwar bestimmte Bitplanes aus der Kontrolle 'herausnehmen' (s. Hardware-Register 'clxdat', und 'clxcon'), aber auch dies bietet meist nicht solche Differenzierungsmöglichkeiten wie die 'ReadPixel'-Methode.

Wie Sie das Hardware-Register 'clxdat', das für die Kollisionskontrolle zuständig ist, abfragen können, und welche Bits was bedeuten, können Sie dem Anhang über die Hardware-Register entnehmen.

In folgendem Programmbeispiel findet diese Art der Kontrolle sowie die 'ReadPixel'-Methode ihre Anwendung:

```
/*************************************************************/
/*                    Breakout.c                             */
/*                                                           */
/* Dies ist das allseits bekannte Spiel Brakout. Hier        */
/* wird besonders auf die Verschiedenen Arten der            */
/* 'Collision Detection' eingegangen.                        */
/*                                                           */
/* Compiled with: Lattice V3.10                              */
/*                                                           */
/* (c) Bruno Jennrich                                        */
/*************************************************************/

#include "exec/types.h"
#include "exec/memory.h"
#include "exec/tasks.h"
#include "exec/devices.h"
#include "graphics/gfx.h"
#include "graphics/gfxbase.h"
#include "graphics/gfxmacros.h"
```

```
#include "graphics/regions.h"
#include "graphics/copper.h"
#include "graphics/gels.h"
#include "graphics/sprite.h"
#include "intuition/intuition.h"
#include "hardware/blit.h"
#include "hardware/custom.h"
#include "devices/printer.h"
#include "devices/keymap.h"

#define WIDTH 320
#define HEIGHT 200

#define RP Window->RPort

struct GfxBase *GfxBase=0;
struct IntuitionBase *IntuitionBase=0;

struct Screen *Screen=0;
struct Window *Window=0;

struct NewWindow NewWindow;
struct NewScreen NewScreen;

struct SimpleSprite BumperL, BumperR, Ball;

UWORD Ball_Data[] = {              /* Sprite Definitionen */
                     0,0,
                     0x0ff0,0x0000,
                     0x0ff0,0x0000,
                     0x0ff0,0x0000,
                     0x0ff0,0x0000,
                     0x0ff0,0x0000,
                     0x0ff0,0x0000,
                     0,0
                   /* 1.Word,2.Word */
                   };

UWORD BumperR_Data[] = {
                     0,0,
                     0xffff,0x0000,
                     0xffff,0x0000,
                     0xffff,0x0000,
                     0xffff,0x0000,
                     0xffff,0x0000,
                     0xffff,0x0000,
                     0xffff,0x0000,
                     0xffff,0x0000,
                     0,0
                   };
```

```
UWORD BumperL_Data[] = {
                       0,0,
                       0xffff,0x0000,
                       0xffff,0x0000,
                       0xffff,0x0000,
                       0xffff,0x0000,
                       0xffff,0x0000,
                       0xffff,0x0000,
                       0xffff,0x0000,
                       0xffff,0x0000,
                       0,0
                     };
                        /* SetPointer braucht eine Adresse */
                        /* ungleich 0x00000000             */
UWORD *NewPointer;

UWORD *ChipBall,*ChipBumperR,*ChipBumperL,*Help;
                       /* Sprite-Daten müßen im Chip_Mem */
                       /* (untersten 512K) stehen !!     */

UWORD Colors[32] = {
                       0x000,0x00f,0x0f0,0xf00,
                       0x0ff,0xff0,0xf0f,0xaaa,
                       0x789,0xa2e,0x5ad,0x8ed,
                       0xdb9,0xa56,0x789,0xabc,
                       0xfd0,0xccc,0x8e0,0xfb0,
                       0xfac,0x93f,0x06d,0x6fe,
                       0xfac,0x0db,0x4fb,0xf80,
                       0xf90,0xc5f,0xc80,0x999
                     };
                          /* Farbdefinitionen des Screens */
char ASCString [11];                  /* Für 'itoa()' */

struct TextAttr TextAttr = { "topaz.font",
                             8,0,FPF_DISKFONT|FPF_ROMFONT
                           };
                            /* Immer 80 Zeichen pro Zeile */

WORD Spr1, Spr2, Spr3;                  /* Sprite Identifier */

long score = 0,                                 /* Punktestand */
     balls = 3;                          /* Wieviele Bälle ? */
long New,                                    /* Neuer Level ? */
     count;                  /* Wieviele Bricks abgeschossen ? */

extern struct Custom custom;
                /* Direkt Zugriff auf die Hardware Register */
```

```
VOID InitScreenWindow();          /* Vorwärtsdeklarationen */
VOID CloseIt();
VOID OpenLibs();
VOID DelKasten();
VOID Score();
VOID itoa();
VOID AwaitClick();

/*********************************************************/
/* Hier spielt sich alles ab !!!                         */
/*********************************************************/

main ()
{
   long x,y,
        vx=-1,vy=-1,                  /* Richtung des Balls */
        px,py,                        /* Position des Balls */
        MouseX,                       /* Position des Bumpers */
        Color;                        /* Farbe für Bricks */

   BOOL Ende = FALSE;                 /* Programmende ? */
   long i=0;
   long level=0;                      /* Spielschwierigkeit */
   char *LeftMouse = (char *)0xBFE001;
                                      /* Zugriff aus CIA */
   UWORD Collision;                   /* Welche Kollision */


   OpenLibs();

   LoadRGB4 (&Screen->ViewPort,Colors,32);
                                      /* Lade eigene Farben */
   RemakeDisplay();      /* Farben in Screen einbinden !! */

   NewPointer = (UWORD *)AllocMem(sizeof(UWORD)*2,
       MEMF_CLEAR|MEMF_CHIP);

   if (NewPointer == 0) CloseIt("No NewPointer !!!\n");


   SetPointer(Window,NewPointer,0,0,0,0);
                                      /* Mauspointer löschen */
                     /* NewPointer-Zeiger muss != 0 sein */

   MoveSprite (&Screen->ViewPort,&BumperL,0,200);
   MoveSprite (&Screen->ViewPort,&BumperR,0,200);
                                /* Bumper weg aus dem ViewPort */
   custom.clxcon = 0x0;
                    /* Collision Control Register setzen */
                    /* Gib uns jede Kollision !          */
```

```
                /* Wir müßen sowieso das                */
                /* Gewünschte selbst 'rausfiltern !  */

while (!Ende)                          /* Programmende ? */
{
   balls = 3;
   level = 0;
   score = 0;

   while (balls > 0)                        /* Noch Bälle ? */
      {
         New = FALSE;
         count = 0;                /* Alle Bricks vorhanden */

         MoveSprite (&Screen->ViewPort,&Ball,0,200);
                                   /* Ball weg aus ViewPort */
         SetRast (RP,0);                /* BitMap löschen */
         SetAPen (RP,1);              /* Vordergrundfarbe */
         SetDrMd (RP,JAM1);              /* Zeichenmodus */

         Move (RP,0,HEIGHT-1);          /* Frame around */
         Draw (RP,0,10);
         Draw (RP,WIDTH-1,10);
         Draw (RP,WIDTH-1,HEIGHT-1);

         for (y=4; y<16; y++)          /* Bricks aufbauen */
            for (x=2; x<15; x++)
               {
                  Color = 2 + level%64 - (x+y)%2;
                  if ((Color == 0) ||
                     (Color == 32)) Color ++;
                  SetAPen (RP,Color);
                  RectFill (RP,x*19+1,y*8+1,
                     (x+1)*19-1,(y+1)*8-1);
               }

                     /* Balken je nach Level malen */
         if (level%4 == 1)
            {
               SetAPen (RP,1);
               RectFill (RP,30,18*8,100,19*8-1);
               RectFill (RP,WIDTH-100,18*8,
                           WIDTH-30,19*8-1);
            }

         if (level%4 == 2)
               RectFill (RP,WIDTH/2-75,17*8,
                     WIDTH/2+75,18*8-1);
```

```
if (level%4 == 3)
   {
      SetAPen (RP,1);
      RectFill (RP,30,18*8,100,19*8-1);
      RectFill (RP,WIDTH-100,18*8,
                     WIDTH-30,19*8-1);
      RectFill (RP,WIDTH/2-75,17*8,
                    WIDTH/2+75,18*8-1);
   }

level++;
Score (score,level,balls);
AwaitClick();              /* Warte auf Mausclick */

px = Screen->MouseX;           /* Berechne     */
if (px >= WIDTH-1) px -= 14;  /* Ballposition */
if (px > WIDTH/2) vx *= -1;
py = 192-7;


while ((balls > 0) && (New == FALSE))
                       /* Ein Ball im Spiel ? */
   {
      MouseX = Screen->MouseX;

      MoveSprite (&Screen->ViewPort,
                  &BumperL,MouseX-16,192);
      MoveSprite (&Screen->ViewPort,
                  &BumperR,MouseX,192);
            /* Bumper wird mit Mausgesteuert */

      i++;        /* Immer ein bischen Warten */
      if (i > 3)
         {
            px += vx;     /* neue Ballposition */
            py += vy;

            Collision = custom.clxdat;
                  /* Kollisionsregister lesen */
            i=0;
            if (py > 200)
                     /* Ball durchgelassen ? */
               {
                  balls--;
                  Score(score,level,balls);
                  vy = vx = -1;
                  while (((*LeftMouse & 0x40)
                                    == 0x40) &&
                     (balls != 0))
                     {
```

```
                        MouseX = Screen->MouseX;
                        MoveSprite
                        (&Screen->ViewPort,
                         &BumperL, MouseX-16,192);
                        MoveSprite
                        (&Screen->ViewPort,
                         &BumperR, MouseX,192);
                     }

                  px = MouseX;
                  if (px >= WIDTH-2) px -= 14;
                  if (px > WIDTH/2) vx *= -1;
                  py = 192-7;
               }

            if ((Collision & 0x3000) > 0)
               {          /* Ball crashed to */
                  vy = -1; /* Bumper          */
                           /* S2 mit S4 od S6 */
                  if (px <= 0) px = 1;
                  if (px >= WIDTH-14)
                              px=WIDTH-15;
               }

/* Alternative Collision-Control: Aktuelle Position des   */
/* Sprites wird mit Position eines weiteren Sprites ver-  */
/* glichen. Bei Übereinstimmung in bestimmten Grenzen     */
/* wird dann eine Kollision registriert:                  */

/*             if (py >= 192-6)
               {
                  if ((px >= (MouseX-20)) &&
                  (px <= (MouseX+16)))
                     {
                        vy = -1;
                        py = 192-7;
                     }
               }                                        */

            else
               {
                  /* Mit ReadPixel() die Kanten */
                  /* des Balls abchecken, und   */
                  /* gegebenenfalls reagieren   */

                  if (ReadPixel(RP,px+4,py+3)>0)
                              /* rechte Kante */
                     {
                        vx *= -1;
                        DelKasten(px+4,py+3,level);
```

```
                    }

                if (ReadPixel(RP,px+11,py+3)>0)
                                    /* linke Kante */
                    {
                    vx *= -1;
                    DelKasten(px+11,py+3,level);
                    }

                if (ReadPixel(RP,px+8,py)>0)
                                    /* obere Kante */
                    {
                    vy *= -1;
                    DelKasten(px+8,py,level);
                    }

                if (ReadPixel(RP,px+8,py+6)>0)
                                    /* untere Kante */
                    {
                    vy *= -1;
                    DelKasten(px+8,py+6,level);
                    }
                }
            MoveSprite (&Screen->ViewPort,
                        &Ball,px,py);
                       /* Ball auf neue Position */
            }
        }
    }

    MoveSprite (&Screen->ViewPort,&Ball,0,200);
    SetDrMd (RP,JAM2);
    SetAPen (RP,2);
    Move (RP,WIDTH/2-TextLength (RP,"Game Over",9)/2,100);
    Text (RP,"Game Over",9);
                                    /* Spiel zu Ende */
    AwaitClick();

    while ((*LeftMouse & 0x40) != 0x40)  i++;
                        /* Wie lange wird gedrückt ? */

    if (i > 10000) Ende = TRUE;     /* Programmende ? */
  }
  ClearPointer(Window);                     /* Ja */
  CloseIt("Bye Bye");
  return(0);
}
```

```
/*********************************************************/
/* Diese Funktion Initialisiert die nötigen NewScreen    */
/* und NewWindow Strukturen, und öffnet den Screen und   */
/* das Window. Weiterhin werden Hardware-Sprites         */
/* allociert und  verändert                              */
/*-------------------------------------------------------*/
/* Eingabe-Parameter: Zeiger auf die zu initialisierenden */
/*                    NewScreen und NewWindow Strukturen */
/*-------------------------------------------------------*/
/* Rückgabe-Werte: keine                                 */
/*********************************************************/

VOID InitScreenWindow (NS,NW)
struct NewScreen *NS;
struct NewWindow *NW;
{
   int i;

   NS->LeftEdge = 0;        NS->TopEdge = 0;
   NS->Width = WIDTH;       NS->Height = HEIGHT;
   NS->Depth = 6;
   NS->DetailPen = 1;       NS->BlockPen = 0;
   NS->ViewModes = 0;

   if (WIDTH > 320) NS->ViewModes |= HIRES;
   if (HEIGHT > 200) NS->ViewModes |= LACE;
   NS->ViewModes |= SPRITES | EXTRA_HALFBRITE;
          /* Modus SPRITES für die Benutzung der Sprites */

   NS->Type = CUSTOMSCREEN;
   NS->Font = &TextAttr;
   NS->DefaultTitle = "";
   NS->Gadgets = NULL;      NS->CustomBitMap = NULL;

   NW->LeftEdge = 0;        NW->TopEdge = 0;
   NW->Width = WIDTH;       NW->Height = HEIGHT;
   NW->DetailPen = 6;       NW->BlockPen = 0;
   NW->IDCMPFlags = NULL;
   NW->Flags = BORDERLESS | ACTIVATE | RMBTRAP |
           NOCAREREFRESH | REPORTMOUSE;
   NW->FirstGadget = NULL;
   NW->CheckMark = NULL;    NW->Title = "";
   NW->Screen = NULL;       NW->BitMap = NULL;
   NW->MinWidth = NW->MinHeight =
   NW->MaxWidth = NW->MaxHeight = 0;
   NW->Type = CUSTOMSCREEN;

   Spr1 = GetSprite (&Ball,2);        /* Allociere Sprites */
   Spr2 = GetSprite (&BumperR,4);     /* für eigen         */
   Spr3 = GetSprite (&BumperL,6);     /* Gebrauch          */
```

```
if ((Spr1 == -1) | (Spr2 == -1) | (Spr3 == -1))
    CloseIt("No Sprites !!!\n");

Ball.x = BumperL.x = BumperR.x = 0;  /* Position und   */
Ball.y = BumperL.y = BumperR.y = 0;  /* Höhe festlegen */

Ball.height = 6;
BumperL.height = 8;
BumperR.height = 8;

if ((Screen = (struct Screen *)
          OpenScreen(&NewScreen)) == 0)
   CloseIt("No Screen !!!");

NewWindow.Screen = Screen;

if ((Window = (struct Window *)
          OpenWindow(&NewWindow)) == 0)
   CloseIt("No Window !!!");

ChipBall = (UWORD*)
    AllocMem(sizeof(Ball_Data),MEMF_CLEAR|MEMF_CHIP);
ChipBumperR = (UWORD*)
    AllocMem(sizeof(BumperR_Data),MEMF_CLEAR|MEMF_CHIP);
ChipBumperL = (UWORD*)
    AllocMem(sizeof(BumperL_Data),MEMF_CLEAR|MEMF_CHIP);

if ((ChipBall == 0) |
   (ChipBumperR == 0) |
   (ChipBumperL == 0))  CloseIt("No ChipMem !!!\n");

Help = ChipBall;
for (i=0; i<(sizeof(Ball_Data)/sizeof(UWORD));i++)
   {
      *Help = Ball_Data[i];
      Help++;
   }

Help = ChipBumperR;
for (i=0; i<(sizeof(BumperR_Data)/sizeof(UWORD));i++)
   {
      *Help = BumperR_Data[i];
      Help++;
   }

Help = ChipBumperL;
for (i=0; i<(sizeof(BumperL_Data)/sizeof(UWORD));i++)
   {
      *Help = BumperL_Data[i];
```

```
        Help++;
      }

   ChangeSprite (&Screen->ViewPort,&Ball,ChipBall);
   ChangeSprite (&Screen->ViewPort,&BumperL,ChipBumperL);
   ChangeSprite (&Screen->ViewPort,&BumperR,ChipBumperR);
                /* Dem jeweiligen Sprite die jeweiligen */
                /* Daten-Blöcke zuweisen                */
}

/*******************************************************/
/* Diese Routine gibt den gesammten vom Programm belegten */
/* Speicher (Window,Screen, Sprites...) frei.          */
/*-----------------------------------------------------*/
/* Eingabe-Parameter: Fehlermessage (String)           */
/*-----------------------------------------------------*/
/* Rückgabe-Werte: keine                               */
/*******************************************************/

VOID CloseIt(s)
char *s;
{
   puts(s);
   if (Spr1 != -1) FreeSprite(Spr1);   /* Sprites zurück  */
   if (Spr2 != -1) FreeSprite(Spr2);   /* an System       */
   if (Spr3 != -1) FreeSprite(Spr3);

   if (NewPointer != 0) FreeMem(NewPointer,
                        sizeof(UWORD)*2);

   if (ChipBall != 0) FreeMem(ChipBall,sizeof(Ball_Data));
   if (ChipBumperR != 0)
       FreeMem(ChipBumperR,sizeof(BumperR_Data));
   if (ChipBumperL != 0)
       FreeMem(ChipBumperL,sizeof(BumperL_Data));

   if (Window) CloseWindow(Window);  /* Window schliessen */

   if (Screen) CloseScreen(Screen);  /* Screen schliessen */
   if (GfxBase) CloseLibrary(GfxBase);   /* Librarys  "   */
   if (IntuitionBase) CloseLibrary(IntuitionBase);
   exit (0);                             /* Tschüs !!! */
}

/*******************************************************/
/* Diese Funktion erröffnet die nötigen Librarys, und  */
/* ruft InitScreen() (s. o.) auf.                      */
/*-----------------------------------------------------*/
/* Eingabe-Parameter: keine                            */
/*-----------------------------------------------------*/
```

```
/* Rückgabe-Werte: keine                                          */
/*****************************************************************/

VOID OpenLibs()
{
   if ((GfxBase = (struct GfxBase *)
            OpenLibrary("graphics.library",0)) == 0)
      CloseIt("No Graphics !!!");

   if ((IntuitionBase = (struct IntuitionBase *)
            OpenLibrary("intuition.library",0)) == 0)
      CloseIt("No Intuition !!!");

   InitScreenWindow(&NewScreen,&NewWindow);
}

/*****************************************************************/
/* Diese Funktion löscht den abgeschossenen Brick vom            */
/* Bildschirm, und erhöht Punktestand.                           */
/*---------------------------------------------------------------*/
/* Eingabe-Parameter: x,y Position des Balles,                   */
/*                    Spielstärke                                */
/*---------------------------------------------------------------*/
/* Rückgabe-Werte: keine                                         */
/*****************************************************************/

VOID DelKasten(x,y,level)
long x,y,level;
{
   if ((x != 0) && (y < 17*8) && (x <= WIDTH -16)
                  && (y > 16))
                     /* Ball im gültigen Bereich ? */
      {
         x /= 19;              /* Berechne Brick-Position */
         y /= 8;

         SetAPen (RP,0);                    /* Löschen ? */
         RectFill (RP,x*19+1,y*8+1,
               (x+1)*19-1,(y+1)*8-1);

         score += (16-y)*10;           /* Score erhöhen ? */
         count ++;          /* Ein Brick mehr abgeschossen */
         Score (score,level,balls);
                        /* Neuen Score ausgeben (dazu  */
                        /*       wird Level übergeben) */
         if (count == 13*12) New = TRUE;
                                    /* Neuer Level ? */
      }
}
```

```
/*********************************************************/
/* Diese Funktion gibt den Punktestand, die Spielstärke  */
/* und die Anzahl der verbliebenen Bälle aus             */
/*-------------------------------------------------------*/
/* Eingabe-Parameter: score (Punktestand),               */
/*                    level (Spielstärke),               */
/*                    balls (verbleibende Bälle )        */
/*-------------------------------------------------------*/
/* Rückgabe-Werte: keine                                 */
/*********************************************************/

VOID Score(score,level,balls)
long score,level,balls;
{
   SetDrMd(RP,JAM2);
   SetAPen (RP,2);
   Move (RP,0,RP->TxBaseline);
   Text (RP,"Level:  ",8);
   itoa (level,2);
   Text (RP,"  Score: ",9);
   itoa (score,10);
   Text (RP,"  Balls: ",9);
   itoa (balls,2);
   SetDrMd(RP,JAM1);
}


/*********************************************************/
/* Diese Funktion wandelt eine Integer-Zahl nach ASCII,  */
/* und gibt diesen String aus                            */
/*-------------------------------------------------------*/
/* Eingabe-Parameter: Zu wandelnde Zahl,                 */
/*                    wieviele Stellen für diese Zahl ?  */
/*-------------------------------------------------------*/
/* Rückgabe-Werte: keine                                 */
/*********************************************************/

VOID itoa(num,stellen)
long num,stellen;
{
   long rette;

   rette = stellen;
   stellen --;        /* Stellenanzahl um eins vermindern */
                      /* wegen +- 1 Fehler                */
   ASCString[0] = 32;       /* Nulltes Zeichen gleich ' ' */

                                  /* Wandle Integer */
   do {
        ASCString[stellen] = (num % 10) + '0';
```

```
        stellen --;
        num /= 10;
      } while (stellen > 0);
   Text (RP,ASCString,rette);              /* Gib Text aus */
}

/*********************************************************/
/* Diese Funktion wartet auf einen Mausclick             */
/*-------------------------------------------------------*/
/* Eingabe-Parameter: keine                              */
/*-------------------------------------------------------*/
/* Rückgabe-Werte: keine                                 */
/*********************************************************/

VOID AwaitClick()
{
   long MouseX;
   char *LeftMouse = (char *)0xBFE001;

   while ((*LeftMouse & 0x40) == 0x40)
      {
        MouseX = Screen->MouseX;

        MoveSprite (&Screen->ViewPort,&BumperL,
                          MouseX-16,192);
        MoveSprite (&Screen->ViewPort,&BumperR,
                          MouseX,192);
      }
}
```

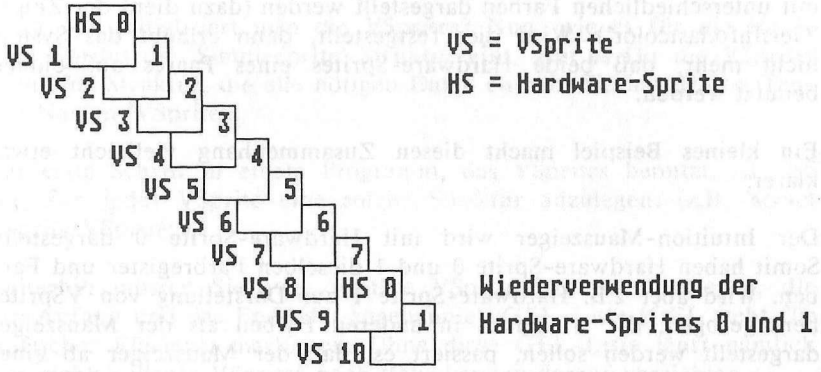
18. Das Animationssystem des Amiga

Nachdem Sie den Umgang mit den Hardware-Sprites kennengelernt haben, wollen wir Sie nun eine Stufe weiter führen: zu den VSprites und Bobs. Diese sogenannten grafischen Elemente (GELs) sind, ähnlich wie die Hardware-Sprites, Kleingrafiken. Doch werden diese unterschiedlich gesteuert.

18.1 Die VSprites

Die VSprites (das 'V' steht für 'virtuell' = scheinbar) werden mit Hilfe der Hardware-Sprites und einer geschickten Copper-Programmierung dargestellt.

So wird zur Darstellung eines VSprites ein Hardware-Sprite einmal verwendet und dann an einer 'tieferen' Position nochmals dargestellt:



Wenn also ein VSprite unter Zuhilfenahme eines Harware-Sprites auf den Bildschirm gebracht wurde, wird, bevor der Elektronenstrahl das untere Ende des Bildschirms erreicht hat, dasselbe Hardware-Sprite noch einmal (oder mehrmals) zur Darstellung eines weiteren VSprites benutzt.

Aber leider sind Ihnen bei der Benutzung der VSprites Beschränkungen auferlegt. Bevor ein Hardware-Sprite zur Darstellung eines weiteren VSprites benutzt werden kann, muß der Elektronenstrahl nämlich eine 'Ruhepause' von mindestens einer Rasterzeile eingelegt haben,

denn die DMA, die für die Darstellung der Hardware-Sprites zuständig ist, ist zwar sehr schnell, aber dem Tempo des Elektronenstrahls nicht gewachsen, um mehrere VSprites mit einem Hardware-Sprite in ein und derselben Rasterzeile mehrmals darstellen zu können.

Somit beschränkt sich die maximale Anzahl von VSprites auf einer Rasterzeile auf 8, denn es gibt ja acht Hardware-Sprites. Steuern Sie allerdings eines dieser Hardware-Sprites selber, und haben Sie dieses mittels GetSprite vom VSprite-System 'abgezogen', sollten Sie beachten, daß Sie eigentlich zwei Sprites 'wegnehmen'.

Da nämlich alle VSprites unterschiedliche Farben haben können, die kurz vor ihrer Darstellung mit dem Copper in die zuständigen Hardware-Register geladen werden, ändert sich ja nicht nur die Farbe des Hardware-Sprites, durch das ein VSprite dargestellt wird, sondern auch die Farbe des 'anderen' Hardware-Sprites, denn wie Sie wissen, bilden Harware-Sprites ja Paare, die sich die Farbregister 'teilen'.

Um so einem 'Farbspiel' vorzubeugen, wird getestet, ob die VSprites mit unterschiedlichen Farben dargestellt werden (dazu dient der Zeiger 'GelsInfo.lastcolor'). Wird dies festgestellt, dann erlaubt das System nicht mehr, daß beide Hardware-Sprites eines Paares auf einmal benutzt werden.

Ein kleines Beispiel macht diesen Zusammenhang vielleicht etwas klarer:

Der Intuition-Mauszeiger wird mit Hardware-Sprite 0 dargestellt. Somit haben Hardware-Sprite 0 und 1 dieselben Farbregister und Farben. Wird aber z.B. Hardware-Sprite 1 zur Darstellung von VSprites herangezogen, die wiederum in anderen Farben als der Mauszeiger dargestellt werden sollen, passiert es, daß der Mauszeiger ab einer bestimmten Position, nämlich ab der, an der das nämliche VSprite dargestellt wird, schlagartig die Farbe wechselt, weil mittels des Coppers die neuen Farben in die zuständigen Register geschrieben wurden. Um dies zu vermeiden, müßten Sie also auch Sprite 1 aus den 'Klauen' der VSprites befreien.

Welche Sprites für die VSprite-Software zur Verfügung stehen, bestimmen Sie in der Variablen 'GelsInfo.sprRsvrd'. Jedes gesetzte Bit dieser Variablen kennzeichnet das zugehörige Hardware-Sprite als benutzbar für die VSprite-Software. Somit ist diese Variable von anderer Bedeutung als 'GfxBase->SpriteReserved'. Hier steht nämlich drin, welche Sprites zur eigenen Benutzung 'abgezogen' wurden. Im

Normalfall, also im 'Intuition'-Betrieb, ist dort Bit 0 gesetzt (= 0x01), was bedeutet, daß Sprite 0 im 'Eigenbetrieb' läuft und so nicht von den VSprites benutzt werden darf. Um dies in der GelsInfo-Struktur widerzuspiegeln, müßte dann vom Benutzer in 'GelsInfo.sprRsrvd' der Wert 0xfe geschrieben werden. So wird festgelegt, daß alle Hardware-Sprites außer Sprite 0 zur Darstellung der VSprites benutzt werden dürfen.

Beachten sollten Sie aber hierbei, daß der Benutzer dafür sorgen sollte, daß bei verschiedenfarbigen VSprites Hardware-Sprite-Paare aus dem Verkehr gezogen werden.

Denn wenn ein Hardware-Sprite-Paar zur Darstellung zweier verschiedenfarbiger VSprites auf ein und derselben Linie herangezogen würde, müßten ja zwischen beiden Hardware-Sprites immer die Farbregister geändert werden. Das 'schafft' der Copper aber nicht mehr.

Somit beschränkt sich die Anzahl darstellbarer VSprites auf einer Rasterzeile auf vier, wenn verschiedene Farben verwendet werden.

Doch wie initialisiert man die VSprites? Nun, wie es für die Hardware-Sprites die 'SimpleSprite'-Struktur gibt, gibt es für die VSprites auch eine Struktur, die alle nötigen Daten enthält. Sie hat den treffenden Namen 'VSprite'.

Der erste Schritt in einem Programm, das VSprites benutzt, ist also der, für jedes VSprite eine solche Struktur anzulegen. (z.B. 'struct VSprite VSprite;')

Weiterhin müssen Sie zwei weitere VSprite-Strukturen anlegen, die den Anfang und das Ende der sogenannten GEL-Liste (GEL steht für grafisches Element) markieren. Ohne diese GEL-Liste läuft nämlich 'gar nichts'. Weder VSprites noch Bobs können darauf verzichten.

Um diese GEL-Liste aufzubauen, benötigen wir eine weitere Struktur: die 'GelsInfo'-Struktur. Bitte weisen Sie den Zeigern 'nextLine', 'lastcolor' und der Variablen 'sprRsvd' die erforderlichen Werte bzw. Speicherplätze zu, da Sie sonst damit rechnen müssen, daß sich nichts auf dem Bildschirm tut (Im Anhang finden Sie eine komplette Beschreibung der 'GelsInfo'-Struktur).

Haben Sie alle Variablen und Zeiger initialisiert, müssen Sie mit 'InitGels (&AnfangsVSprite, &EndVSprite, &GelsInfo)' die GelsInfo-Struktur initialisieren (hier werden an 'GelsInfo.gelHead' und

'GelsInfo.gelTail' die Adressen der beiden angegebenen VSprite-Strukturen übergeben).

Diese beiden VSprites markieren, wie schon gesagt, den Anfang und das Ende der GEL-Liste. Alle weiteren GELs (VSprites und Bobs) werden mit AddVSprite bzw. AddBob in diese Liste eingetragen. Die GEL-Liste wird dann von den Befehlen SortGList und DrawGList zur ordnungsgemäßen Darstellung benötigt.

Jetzt wenden wir uns wieder unseren VSprites zu. Natürlich müssen Sie auch das Aussehen des VSprites bestimmen. Dazu gibt es den Zeiger auf die Daten, die dieses bestimmen (VSprite.ImageData). Diese Daten sind genauso wie 'Aussehen[Höhe][2]' bei den Hardware-Sprites organisiert. Sie geben also nur die Zeilen des VSprites in je zwei UWORDs an. Extra Speicherplätze wie bei den Hardware-Sprites brauchen Sie hier nicht (dazu ist ja die VSprite-Struktur und die GEL-Liste da).

Die Höhe des VSprites gibt man ähnlich wie bei den Hardware-Sprites in der Variablen 'VSprite.Height' an. Positioniert werden sie mit Hilfe der Variablen 'VSprite.X' und 'VSprite.Y'. Aber auch hier sollten Sie darauf achten, daß diese Variablen für VSprites in Hi-Res- oder Lace-Viewports immer um den Wert 2 in der jeweiligen Richtung verändert werden müssen, damit sie sich tatsächlich 'von der Stelle' bewegen.

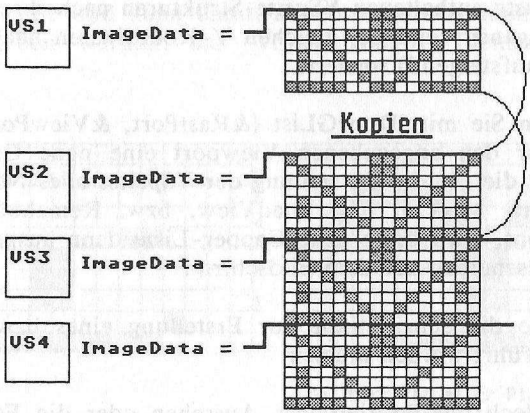
Wie schon oben erwähnt, können VSprites auch verschiedene Farben haben. Diese Farben werden mit Hilfe des Zeigers 'VSprite.SprColors' festgelegt, der auf ein dreidimensionales UWORD-Array zeigt, in dem die Farben enthalten sind. Diese Farben sind so kodiert, wie Sie es von den Farbregistern her gewöhnt sind: (Bit 11-8 enthalten die Rot-, Bit 7-4 die Grün- und Bit 3-0 die Blaukomponente der Farbe). Bitte beachten Sie die Einschränkungen, die entstehen können, wenn Sie Ihre VSprites untereinander zu farbenfroh gestalten.

Aufgrund der 'Paarung' der Hardware-Sprites, mit deren Hilfe die VSprites ja dargestellt werden, kann es passieren, daß bei mehreren VSprites auf einer Rasterzeile (Y-Position) bestimmte VSprites nicht mehr dargestellt werden, also 'ausfallen', weil der Amiga nicht in der Lage ist, so schnell zwischen verschiedenen Farbdefinitionen umzuschalten!

Am besten wäre es deshalb natürlich, wenn Sie alle VSprites mit den gleichen Farben ausstatten, was der Amiga daran merkt, daß alle

'SprColors'-Zeiger der VSprites auf ein und dieselbe Speicherstelle zeigen. Um dies zu testen, muß der Variablen 'GelsInfo.lastcolor' genügend Speicherplatz zugewiesen worden sein. Hier werden nämlich die Zeiger auf die letzten Farbdefinitionen der einzelnen VSprites abgespeichert, um dann mit der Farbdefinition eines neu zu zeichnenden VSprites verglichen zu werden.

Sollten Ihre VSprites aber alle dasselbe Aussehen erhalten, machen Sie bitte nicht den Fehler und lassen alle 'ImageData' Zeiger auf dieselbe Adresse zeigen. Die Speicherzugriffszeit des Amiga erlaubt nicht, mehrere VSprites mit dem gleichen 'ImageData'-Zeiger auszustatten. In diesem Fall ist es das beste, die Daten in einen Speicherbereich zu kopieren und die 'ImageData'-Zeiger dann auf die verschiedenen Anfangsadressen der Daten innerhalb des von Ihnen angelegten Bereiches zeigen zu lassen.



Mehrere VSprites mit gleichem Aussehen

Jetzt müssen Sie noch die Flags-Variable des VSprites auf VSPRITE setzen (VSprite.Flags = VSPRITE), da die VSprite-Struktur, wie Sie später noch erfahren werden, auch für Bobs benutzt werden, und das System somit weiß, daß es sich tatsächlich um ein VSprite handelt.

Allerdings war dies noch nicht alles: Obwohl es gar nicht möglich ist, VSprites mit einer Breite von mehr als 16 Punkten bzw. einem UWORD zu generieren, müssen Sie in 'VSprite.Width' den Wert 1 ein-

tragen. Wenn Sie das versäumen, kann es passieren, daß Ihr VSprite später nicht auf dem Bildschirm erscheint.

Außerdem sollten Sie die 'Tiefe' des VSprites (VSprite.Depth), obwohl auch hier andere Werte unmöglich sind, mit 2 angeben, um so zu veranlassen, daß das System auch 'wirklich Bescheid weiß' (Im Gegensatz zu den Hardware-Sprites können Sie auch keine 'attached' VSprites erzeugen).

Haben Sie das getan, fügen Sie das VSprite mit 'AddVSprite (&VSprite, &RastPort)' in die vorher dem Rastport mit 'RastPort.GelsInfo = &GelsInfo' zugewiesene GelsInfo-Struktur ein. Jetzt kann es mit dem Darstellen losgehen!

Dazu müssen Sie zuerst einmal die GEL-Liste mit 'SortGList (&RastPort)' sortieren. Damit der Copper nämlich entlastet wird und die größtmögliche Anzahl von VSprites dargestellen kann, werden alle in der GEL-Liste enthaltenen VSprite-Strukturen nach ihren Y-Koordinaten (absteigend) und bei gleichen Y-Koordinaten nach ihren X-Koordinaten (aufsteigend) sortiert.

Danach können Sie mit 'DrawGList (&RastPort, &ViewPort);' veranlassen, daß für den angegebenen Viewport eine neue Copper-Liste generiert wird, die für die Darstellung der VSprites alles Nötige regelt. Mit MakeVPort, MrgCop und LoadView, bzw. RemakeDisplay für Intuition-Screens, wird diese neue Copper-Liste dann ausgeführt, und die VSprites erscheinen auf dem Bildschirm.

Das waren also die Schritte, die zur Erstellung eines bzw. mehrerer VSprites ausgeführt werden müssen.

Wollen Sie jedoch die Position, das Aussehen oder die Farben eines VSprites verändern, können Sie das durch Verändern der dafür bestimmten Zeiger und Variablen tun, müssen aber hinterher dafür sorgen, daß die GEL-Liste neu sortiert (SortGList) und auch neu gezeichnet (DrawGList, MakeVPort, MrgCop, LoadView) wird.

18.1.1 VSprites in Kollisionen

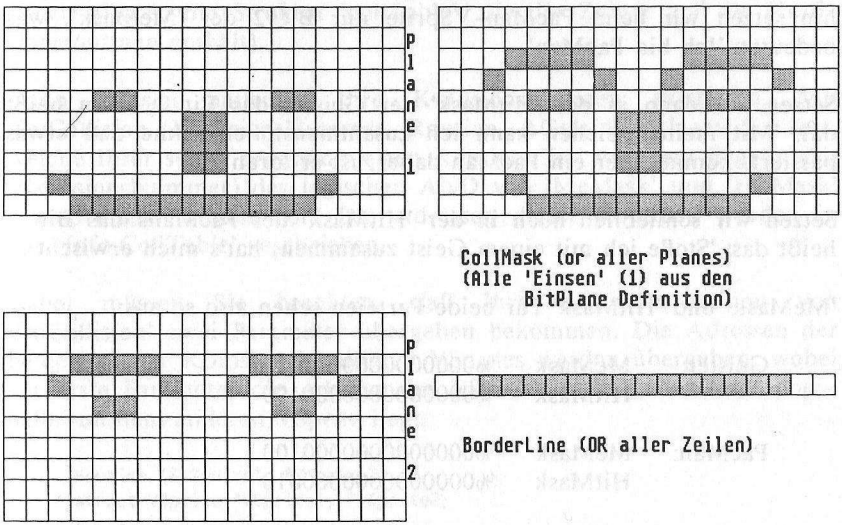
Die oben erläuterten Schritte reichen zwar, um ein VSprite auf dem Bildschirm darzustellen und dieses zu bewegen, aber Kollisionen kann man so noch nicht abfragen, was z.B. für Spiele erforderlich ist.

Doch dabei helfen uns zwei weitere Zeiger in der VSprite-Struktur weiter: 'BorderLine' und 'CollMask'. Außerdem werden die Variablen 'VSprite.MeMask' und 'VSprite.HitMask' sowie der Zeiger 'collTable' (Kollisionstabelle) der 'GelsInfo'-Struktur benötigt (Dieser muß auch bei Nichtbenutzung der Kollisionsmöglichkeiten initialisiert werden.).

Doch zuerst zu 'BorderLine' und 'CollMask'. Diese beiden Zeiger zeigen auf zwei von Ihnen angelegte Speicherplätze.

'BorderLine' enthält nach dem Aufruf von 'InitMasks' das logische OR aller Zeilen des VSprites und braucht somit nur ein UWORD zu umfassen. Diese 'BorderLine' wird zur schnellen Kollisionskontrolle benötigt. Erst wenn mittels der BorderLine eine Kollision 'in Aussicht gestellt wurde', wird mit der 'CollMask' untersucht, ob wirklich eine Kollision stattgefunden hat.

Die 'CollMask' enthält nämlich das logische OR der beiden UWORDs pro VSprite-Zeile und kann damit als ein eindimensionales UWORD-Array angegeben werden, das 'VSprite.Height' Elemente enthalten muß.



Haben Sie an diese beiden Zeiger den benötigten Speicherplatz zugewiesen, können Sie mittels 'InitMasks(&VSprite)' veranlassen, daß die

beiden Kollisionsmasken 'BorderLine' und 'CollMask' für das angegebene VSprite (Bob) initialisiert werden. Natürlich muß dazu auch schon der Zeiger 'ImageData' auf den richtigen Speicherbereich zeigen, denn aus diesem werden 'CollMask' und 'BorderLine' ja errechnet.

Haben Sie diese beiden Masken ordnungsgemäß initialisiert, können Sie mit den Variablen 'MeMask' und 'HitMask' für jedes VSprite bestimmen, welche Kollisionen gemeldet werden sollen. Denn die Routine 'DoCollision(&RastPort)', die alle VSprites in der GEL-Liste auf Kollisionen testet, testet nicht nur, ob eine Kollision stattgefunden hat, sondern selektiert auch, ob diese Kollision überhaupt gemeldet bzw. die zugehörige Routine ausgeführt werden soll.

Nehmen wir einmal an, daß Sie das Spiel 'PacMan' mit Hilfe der VSprites programmieren wollen. Wenn die Geister miteinander zusammenstoßen, soll dabei nichts passieren, aber wenn ein Geist Ihren PacMan berührt, verlieren Sie diesen.

Mit den Variablen 'MeMask' und 'HitMask' können Sie dies nun der 'DoCollision'-Routine mitteilen. Dafür setzen wir bei allen Geistern z.B. das Bit 1 der 'MeMask'. Das bedeutet: 'Ich bin ein Geist'. Weiterhin setzen wir beim PacMan-VSprite nur Bit 2 der 'MeMask', was bedeutet: 'Ich bin PacMan'.

Setzen wir dann in der 'HitMask' der Geister das Bit 2, dann heißt das: 'Mit meinesgleichen kann ich zusammenstoßen, ohne daß etwas passiert, kommt aber ein PacMan daher, ist er 'dran'.

Setzen wir schließlich noch in der 'HitMask' des PacMans das Bit 1, heißt das: 'Stoße ich mit einem Geist zusammen, hat's mich erwischt'.

'MeMask' und 'HitMask' für beide Parteien sehen also so aus:

Geister:	MeMask	%0000000000000010
	HitMask	%0000000000000100
PacMan:	MeMask	%0000000000000100
	HitMask	%0000000000000010

Wenn 'DoCollision' dann eine Kollision feststellt, wird die 'MeMask' des VSprites, das links oberhalb des anderen VSprites liegt, mit dessen 'HitMask' geANDet. Das Ergebnis-Bit dieses logischen ANDs gibt

dann die Kollisionsroutine an, die von 'DoCollision' für diese Kollision angesprungen wird.

Stoßen also z.B. zwei Geister zusammen, wird folgende Routine von DoCollision angesprungen:

> Geist1.MeMask & Geist2.HitMask = %10 & %100 = %000 (also keine Kollision).

Stoßen ein Geist und ein PacMan zusammen:

> Geist.MeMask & PacMan.HitMask = %10 & %10 = %10 (hier wird Routine 1 aufgerufen).

Stoßen ein PacMan und ein Geist zusammen, das heißt, daß hier der PacMan links oberhalb vom Geist liegt (oben war es umgekehrt):

> PacMan.MeMask & Geist.HitMask = %100 & %100 = %100 (hier wird Routine 2 angesprungen).

Doch wie legt man die Kollisionsroutinen fest? Nun, zuerst einmal muß der Zeiger 'collTable' der GelsInfo-Struktur initialisiert werden. (Dazu gibt es die Struktur 'collTable', die die Zeiger auf die Kollisionsroutinen enthält).

Dann müssen natürlich Ihre Kollisionsroutinen existieren. Mit 'SetCollision (KollisionsNummer, Routine, &GelsInfo)' legen Sie fest, welche Ihrer selbst programmierten Routinen bei welchem Ergebnisbit (KollisionsNummer) des logischen AND von 'MeMask' und 'HitMask' angesprungen werden soll. Die Adressen dieser Routinen werden in 'GelsInfo.CollTable' geschrieben.

Dabei müssen Sie beachten, daß Ihre Kollisionsroutinen von 'DoCollision' zwei Parameter übergeben bekommen. Die Adressen der beiden an der Kollision beteiligten VSprites werden übergeben, wobei der erste Parameter die Adresse des VSprites enthält, das links oberhalb von dem anderen VSprite liegt:

```
Routine (&VSprite1, &VSprite2)
struct VSprite *VSprite1, *VSprite2;
{...}
```

So können Sie also Kollisionen zwischen VSprites (und Bobs) abfragen. Aber wie testet man Kollisionen mit dem Rand? Nun, dazu legt man

zuerst in den Variablen 'topmost' (oben), 'bottommost' (unten), 'leftmost' (links) und 'rightmost' (rechts) der GelsInfo-Struktur fest, in welchem Rechteck der Bitmap sich die VSprites bewegen dürfen, ohne daß eine Rand-Kollision gemeldet wird.

Wird von einem VSprite, dessen Bit 0 in der 'HitMask' gesetzt wurde (es ist für Rand-Kollisionen reserviert), festgestellt, daß es die Grenzen berührt, wird die Kollisionsroutine Nummer 0 aufgerufen (Sie wird mit SetCollision (0, Routine, &GelsInfo); festgelegt). Diese hat dabei folgende Parameter:

```
RandKontrolle (VSprite, Flags)
struct VSprite *VSprite;
BYTE Flags;
{...}
```

wobei 'VSprite' das VSprite ist, das mit dem Rand zusammengestoßen ist, und Flags angibt, mit welchem Rand es zusammengestoßen ist. (TOPHIT = oberer Rand, BOTTOMHIT = unterer Rand, LEFTHIT = linker Rand, RIGHTHIT = rechter Rand).

Leider unterstützt die VSprite-Software nicht die Kollision von VSprites mit dem Hintergrund, so daß Sie hier z.B. auf die 'ReadPixel'-Methode zurückgreifen müssen.

Folgendes Programm zeigt Ihnen, wie Sie VSprites erstellen und diese auf Kollisionen testen können:

```
/*********************************************************/
/*                    Tribars.c                          */
/*                                                       */
/* VSprites pur.                                         */
/*                                                       */
/* Compiled with: Lattice V3.10                          */
/*                                                       */
/* (c) Bruno Jennrich                                    */
/*********************************************************/

struct VSpriteExt {              /* Unsere eigenen Daten */
                    int vx,vy;   /* die an die VSprite   */
                  };             /* Struktur angehängt   */
                                 /* werden               */

#define VUserStuff struct VSpriteExt
                   /* Muß vor den #INCLUDES geschehen! */
```

```
#include "exec/types.h"
#include "exec/memory.h"
#include "exec/devices.h"
#include "devices/keymap.h"
#include "graphics/gfx.h"
#include "graphics/gfxmacros.h"
#include "graphics/copper.h"
#include "graphics/gels.h"
#include "graphics/collide.h"
#include "graphics/gfxbase.h"
#include "graphics/regions.h"
#include "hardware/blit.h"
#include "hardware/custom.h"
#include "intuition/intuition.h"
#include "intuition/intuitionbase.h"
#include "libraries/diskfont.h"
#include "hardware/dmabits.h"

#define RP Screen->RastPort       /* Zugriff auf RastPort */

#define MAXVSPRITES 16              /* Wieviele VSprites ? */
struct GfxBase *GfxBase;                  /* BasePointer */
struct IntuitionBase *IntuitionBase;

struct NewScreen NewScreen =              /* Unser Screen */
                    {
                     0,0,320,199,1,
                     1,0,
                     0,
                     CUSTOMSCREEN,
                     NULL,
                     "",
                     NULL,NULL
                    };
struct Screen *Screen;

struct VSprite Anfang, Ende,          /* VSprites für GEL */
               VSprite[MAXVSPRITES]; /* Liste            */

UWORD VSBorderLine[MAXVSPRITES];   /* VSprites Borderline */

UWORD VSCols[3] =
        {0xfff,0x057,0x889};          /* Farben für alle */

struct GelsInfo GelsInfo;           /* GelsInfo muß vor       */
                                    /* Benutzung der VSprites */
                                    /* ordnungsgemäß          */
                                    /* initialisiert werden!  */
```

```
WORD Nextlines[8] = {0,0,0,0,0,0,0,0};
                          /* Nextline-Array für GInfo */
WORD *lastColors[8] = {0,0,0,0,0,0,0,0};
                         /* lastColor-Array für GInfo */

struct collTable collTable;
                        /* Kollisions-Einsprung Tabelle */
UWORD Tribar[18][2] =
          {
               {0xe000,0xc000},      /* Wie sehen die  */
               {0xf800,0xc000},      /* VSprites aus ? */
               {0xfe00,0xc000},
               {0xff80,0xc000},
               {0xf3e0,0xcc00},
               {0xf0f8,0xcf00},
               {0xf03e,0xc7c0},
               {0xf00f,0xc1f0},
               {0xf043,0xc07c},
               {0xf0f0,0xc0ff},
               {0xf3e0,0xc3ff},
               {0xff80,0xcffc},
               {0xfe00,0xfff0},
               {0xf800,0xffc0},
               {0xe000,0xff00},
               {0x8000,0xfc00},
               {0x0000,0xf000},
               {0x0000,0xc000}
          };

VOID RandKontrolle();          /* Unsere Routine, die bei */
                               /* Kollisionen mit dem     */
                               /* Rand aufgerufen wird    */

/*********************************************************/
/* Es geht los !                                         */
/*********************************************************/

main()
{
   long i,j,k;
   int Length;
   UWORD *Tribs, *HilfTrib,*VSCollMask,*HilfColl;
   char   *LeftMouse = (char *) 0xBFE001;
                                  /* Linke Maustaste */

   if ((GfxBase = (struct GfxBase *)
          OpenLibrary("graphics.library",0)) == NULL)
             {
                printf (" No Graphics!\n");
```

```
            exit(0);
          }

if ((IntuitionBase = (struct IntuitionBase *)
        OpenLibrary("intuition.library",0)) == NULL)
          {
            printf ("No Intuition!\n");
            goto cleanup1;
          }

if ((Screen = (struct Screen *)
       OpenScreen (&NewScreen)) == NULL)
         {
           printf ("No Screen!\n");
           goto cleanup2;
         }

Tribs = (UWORD *)AllocMem
 (18*2*sizeof(UWORD)*MAXVSPRITES,MEMF_CLEAR|MEMF_CHIP);
                      /* Obwohl alle VSprites gleich */
                      /* aussehen, sollte für jedes  */
                      /* ein extra 'Chip' Speicher-  */
                      /* bereich belegt werden, weil */
                      /* sonst die DMA 'ausrastet'   */
if (Tribs == 0)
  {
    printf (" No Memory for Tribs !!!\n");
    goto cleanup3;
  }

HilfTrib = Tribs;

for (i=0; i<MAXVSPRITES; i++)
  for (j=0; j<18; j++)
    for (k=0; k<2; k++)
        {
          *HilfTrib = Tribar[j][k];
          HilfTrib++;
        }

VSCollMask = (UWORD *)
  AllocMem(18*sizeof(UWORD)*MAXVSPRITES,MEMF_CHIP);

if (VSCollMask == 0)
  {
    printf (" No Memory for CollMask !!!\n");
    goto cleanup4;
  }
```

```
SetRGB4 (&Screen->ViewPort,0,0,0,0);
SetRGB4 (&Screen->ViewPort,1,13,7,1);
                                    /* N' bischen Farbe */
SetRast (&RP,0);

Length = TextLength (&RP,"Tribars in Action !!!",21);
Move (&RP,320/2-Length/2,100);
Text (&RP,"Tribars in Action !!!",21);

BltClear (&Anfang, sizeof(struct VSprite),0);
BltClear (&Ende, sizeof(struct VSprite),0);
BltClear (&GelsInfo, sizeof(struct GelsInfo),0);
BltClear (VSprite, sizeof(struct VSprite)*MAXVSPRITES,0);
BltClear (&collTable, sizeof (collTable),0);
                /* zur Sicherheit mal alles löschen */

GelsInfo.sprRsrvd = 0xfc;
                        /* Alle Sprites außer 0 u.1 */
                        /* für VSprites             */

GelsInfo.nextLine = Nextlines;
GelsInfo.lastColor = lastColors;
GelsInfo.collHandler = &collTable;

GelsInfo.leftmost = 23;             /* Grenzen für      */
GelsInfo.rightmost = 300;           /* Rand-Kollisionen */
GelsInfo.topmost = 9;
GelsInfo.bottommost = 191;

InitGels (&Anfang, &Ende, &GelsInfo);
                       /* GelsInfo initialisieren   */
RP.GelsInfo = &GelsInfo;  /* und in RastPort einbinden */

SetCollision(0,RandKontrolle,&GelsInfo);

HilfTrib = Tribs;
HilfColl = VSCollMask;
for (i=0; i<MAXVSPRITES; i++)
   {
      VSprite[i].Width = 1;              /* 1 WORD breit */
      VSprite[i].Height = 18;          /* 18 Zeilen hoch */
      VSprite[i].Flags = VSPRITE; /*VSprite ist VSprite*/
      VSprite[i].Depth = 2;                /* 2 'Planes' */
      VSprite[i].ImageData = HilfTrib;/*eigenes Tribar */
      VSprite[i].MeMask = 0;     /* keine GEL Kollision */
      VSprite[i].HitMask = 1;    /* aber mit Rand       */

      VSprite[i].CollMask= HilfColl;
      VSprite[i].BorderLine = &VSBorderLine[i];
```

```
        VSprite[i].SprColors = VSCols;          /* Farben */
        VSprite[i].X = 23+i*(320/MAXVSPRITES-4);
        VSprite[i].Y = 9+i*(200/MAXVSPRITES-2);
                                                /* Position */

        VSprite[i].VUserExt.vx = 5;   /* eigene          */
        VSprite[i].VUserExt.vy = 5;   /* Geschwindigkeit */

        HilfTrib += 2*18;
        HilfColl += 18;
                                    /* nächstes Tribar */

        InitMasks (&VSprite[i]); /* CollMask und          */
                                 /* Borderline berechnen */

        AddVSprite (&VSprite[i], &RP);
                          /* VSprite In Liste einreihen */
    }

  while ((*LeftMouse & 0x40) == 0x40)
    {
        for (i=0; i<MAXVSPRITES; i++)
          {
            VSprite[i].Y +=
                  VSprite[i].VUserExt.vy;
            VSprite[i].X +=
                  VSprite[i].VUserExt.vx;
                                  /* VSprites bewegen */
          }
        SortGList(&RP);                 /* neu Sortieren */
        DoCollision(&RP);          /* Kollisionen testen */
        DrawGList(&RP, &Screen->ViewPort);
                           /* Copper-Liste generieren */
        WaitTOF();
        RemakeDisplay();                /* und ausführen */
    }


        FreeMem(VSCollMask,
                  18*sizeof(UWORD)*MAXVSPRITES);
  cleanup4:  FreeMem(Tribs,
                 18*2*sizeof(UWORD)*MAXVSPRITES);
                                 /* Speicher freigeben */
  cleanup3:  CloseScreen(Screen);
  cleanup2:  CloseLibrary(IntuitionBase);
  cleanup1:  CloseLibrary(GfxBase);
  return(0);
}
```

```
/*********************************************************/
/* Diese Routine wird von DoCollision() aus angesprungen */
/* wenn ein VSprite den Rand (border) berührt            */
/*-------------------------------------------------------*/
/* Eingabe-Parameter: VSprite, das den Rand berührt hat  */
/*                    Border - an welchen Rand           */
/*-------------------------------------------------------*/
/* Rückgabe-Werte: keine                                 */
/*********************************************************/


VOID RandKontrolle (VSprite, Border)
struct VSprite *VSprite;
BYTE Border;
{
   if ((Border & TOPHIT)  == TOPHIT)          /* Oben */
               VSprite->VUserExt.vy *=-1;
   if ((Border & BOTTOMHIT)  == BOTTOMHIT)    /* Unten */
               VSprite->VUserExt.vy *=-1;
   if ((Border & LEFTHIT)  == LEFTHIT)        /* Links */
               VSprite->VUserExt.vx *=-1;
   if ((Border & RIGHTHIT)  == RIGHTHIT)      /* Rechts */
               VSprite->VUserExt.vx *=-1;
}
```

Leider müssen wir Sie hier noch auf ein Problem bei der Kollisionskontrolle hinweisen. Die Routine 'DoCollision', die alle GELs der GEL-Liste auf Kollisionen testet, funktioniert nicht immer einwandfrei. Es kann also sein, daß Kollisionen unentdeckt bleiben.

Weiterhin möchten wir Sie hier noch auf ein Flag der VSprites hinweisen: GELGONE. Es bezieht sich allerdings nicht nur auf VSprites, sondern, wie die gesamte Kollisionskontrolle auch auf Bobs. Wurde dieses Flag (in VSprite.Flags) nämlich vom System gesetzt, so heißt das, daß das GEL (VSprite oder Bob) vollkommen aus den von Ihnen in der GelsInfo-Struktur angegebenen Grenzen verschwunden ist.

Wenn Sie also entdecken, daß dieses Flag gesetzt ist, könnten Sie z.B. mit RemVSprite veranlassen, daß das VSprite aus der GEL-Liste genommen wird (beim nächsten DrawGList wird es also nicht mehr 'mitberechnet'). Für Bobs wird dazu RemIBob oder das Macro RemBob verwendet, doch dazu kommen wir erst im nächsten Kapitel.

18.2 Ein weiteres GEL: Das Bob

Wie VSprites, so sind auch die Bobs (Blitter Objekts) grafische Elemente (GELs). Somit benötigen auch diese zur Darstellung eine GelsInfo-Struktur, die initialisiert und in einen Rastport eingebunden werden muß. Doch was sind Bobs überhaupt? Nun, auf anderen Computern wurden sie 'Shapes' genannt. Da der Amiga diese aber mit Hilfe des Blitters steuert, heißen sie hier Bobs (Blitter Objects).

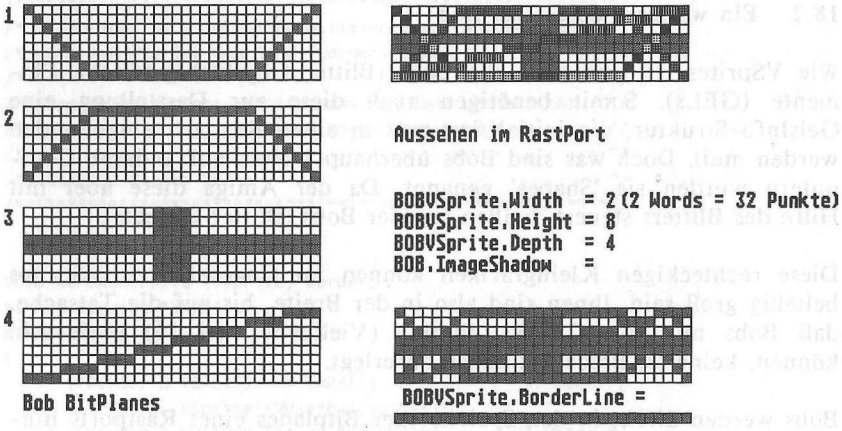
Diese rechteckigen Kleingrafiken können im Gegensatz zu VSprites beliebig groß sein. Ihnen sind also in der Breite, bis auf die Tatsache, daß Bobs nur 16, 32, 48... Punkte (Vielfache von 16) breit sein können, keinerlei Beschränkungen auferlegt.

Bobs werden direkt in den Speicher der Bitplanes eines Rastports hineingeschrieben. Somit ergibt sich automatisch, daß Bobs in der Auflösung des Rastports dargestellt werden und genauso vielfarbig sein können wie der Rastport.

Dafür definieren Sie das Aussehen der Bobs Bitplane-weise. Sie schreiben dem Blitter vor, welche Bitkombinationen (abgespeichert in UWORDs) er in den Speicher der Bitplanes hineinschreiben soll.

Doch es reicht natürlich nicht, nur das Aussehen eines Bobs in einem UWORD-Array zu definieren. Der Blitter muß schließlich wissen, wie groß das Bob sein soll, an welcher Stelle es positioniert werden und wie viele Bitplanes es enthalten soll.

Dazu wird für jedes Bob eine eigene VSprite-Struktur initialisiert. Ja, Sie haben richtig gelesen: Jedes Bob hat seine eigene VSprite-Struktur. Das heißt nicht, daß Bobs irgendwie als VSprites dargestellt werden. Nein, wie schon erwähnt, werden sie direkt in die Bitmap hineingeschrieben.



In der VSprite-Struktur legen Sie die Größe des Bobs in den Variablen 'BOBVSprite.Height' und 'BOBVSprite.Width' fest, wobei die in 'BOBVSprite.Width' angegebene Breite die Anzahl der Words (jeweils 16 Bits) angibt. Weiterhin müssen Sie die Adresse der ersten Bob-Bitplane angeben (BOBVSprite.ImageData).

Die Position wird wie bei den VSprites festgelegt. Mit 'BOBVSprite.X' wird die X- und in 'BOBVSprite.Y' die Y-Koordinate des Bobs festgelegt.

Für die Kollisionskontrolle werden auch hier die Variablen 'MeMask' und 'HitMask' in den VSprite-Strukturen der Bobs wie gewohnt initialisiert. Die Kollisionskontrolle unterscheidet sich nämlich nicht von der für die VSprites. Das heißt aber wiederum, daß Sie Speicher für die 'Borderline', der natürlich so 'breit' wie eine Zeile des Bobs sein muß, sowie den Speicher für die 'CollMask', der eine Bob-Bitplane groß sein muß, bereitstellen müssen. Mit 'InitMasks (&BOBVSprite)' werden diese dann auch für Bobs initialisiert.

Doch kommen wir nun zu den Unterschieden zwischen VSprites und Bobs. Zuerst fällt auf, daß das VSprite-Flag VSPRITE bei der Benutzung von Bobs natürlich nicht gesetzt wird (Wir wollen ja schließlich ein Bob und kein VSprite darstellen).

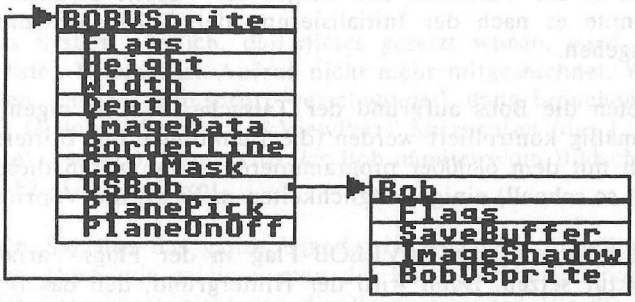
Weiterhin brauchen Sie für das Bob keine eigenen Farbdefinitionen wie für die VSprites (VSprite.SprColors), da Bobs in den Rastport geschrieben werden und somit auch dessen Farben benutzen. Aber hier

kommen wir gleich auf eine Einschränkung zu sprechen. Wollen Sie nämlich VSprites und Bobs in einem Rastport benutzen, sollten Sie darauf achten, daß Ihr Bob keine Farben aus den Farbregistern 17 bis 31 benutzt. In anderen Worten: Erzeugen Sie keine Bobs, die mehr als 4 Bitplanes 'tief' sind. Da nämlich unter Umständen bei der Darstellung von VSprites die Farbregister zwischenzeitlich mit anderen Werten belegt werden können, kann es passieren, daß Ihr Bob flackert bzw. Teile des Bobs kurzzeitig in anderen Farben dargestellt werden. Sollten Sie aber mit 16 Farben nicht auskommen, könnten Sie immerhin noch die Farbregister 16, 20, 24 und 28 benutzen, die ja für die Farbe 'transparent' von den Sprites und somit auch von den VSprites benutzt werden.

Die Tiefe der Bobs bestimmen Sie wie bei den VSprites mit der Variablen 'BOBVSprite.Depth'. Allerdings ist diese bei Bobs nicht konstant, sondern richtet sich danach, wie Sie das Bob definiert haben. Allerdings gilt hier: Je mehr Bitplanes für das Bob definiert wurden, um so mehr Farben enthält es.

Doch ist natürlich die VSprite-Struktur nicht die einzige zur Beschreibung eines Bobs. Denn die 'Bob'-Struktur enthält weitere Zeiger und Variablen, um Bobs näher zu bestimmen.

Dazu müssen Sie die 'Bob'- und die zugehörige 'VSprite'-Struktur miteinander verbinden. Zuerst zeigt ein Zeiger der Bob-Struktur auf die zugehörige VSprite-Struktur (Bob.BobVSprite = &VSprite), so wie von der VSprite.Struktur ein Zeiger auf die Bob-Struktur zeigt (VSprite.VSBob = Bob). Es versteht sich von selbst, daß Bob und zugehörige VSprite-Struktur so miteinander verbunden werden müssen.



Die Relevanten Variablen für die Darstellung von Bobs

In der Bob-Struktur ist aber ein weiterer Zeiger für die farbliche Darstellung von Bobs besonders wichtig: Bob.ImageShadow. Dieser 'ImageShadow' (Bildschatten) ist dasselbe wie die 'CollMask' (Kollisionsmaske), also das logische OR aller Bitplane-Zeilen des Bobs.

Aber warum zwei Zeiger auf ein und dasselbe? Nun, so gleich sind 'ImageShadow' und 'CollMask' nicht. Sie können nämlich die 'CollMask' nachträglich (nach InitMasks) verändern, so daß z.B. nicht alle gesetzten Bits der Bitplanes des Bobs für eine Kollision verantwortlich sind, sondern z.B. nur noch eine Bitplane zur Kollisionskontrolle herangezogen wird.

In Verbindung mit den 'VSprite'-Variablen 'PlanePick' und 'PlaneOnOff' bekommt der 'ImageShadow' aber eine entscheidende Bedeutung: Die Bits in 'PlanePick' bestimmen nämlich, in welche Bitplane des Rastports die Bitplanes des Bobs hineingeschrieben werden sollen. Normalerweise schreiben Sie dort 0xff hinein, was bedeutet, daß alle aufeinanderfolgenden Bitplanes des Bobs in alle aufeinanderfolgenden Bitplanes des Rastports kopiert werden. Steht dort aber z.B. der Wert %00000101 = 0x05, dann bedeutet das, daß die erste Bitplane des Bobs in die erste Bitplane des Rastports und die zweite Bitplane des Bobs in die dritte Bitplane des Rastports kopiert werden soll.

Die Variable 'PlaneOnOff' bestimmt dann, was mit den nicht aktivierten Bitplanes des Rastports geschehen soll. Normalerweise steht hier 0x00, was bedeutet, daß in alle Bitplanes des Rastports, die nicht mit 'PlanePick' selektiert wurden, nichts hineingeschrieben wird. Steht dort aber z.B. 0x2, dann heißt das, daß in Bitplane 2 des Rastports der 'ImageShadow' geschrieben wird.

Haben Sie so die Variablen und Zeiger der VSprite-Struktur initialisiert, könnte es nach der Initialisierung der GelsInfo-Struktur, fast schon losgehen.

Doch bieten die Bobs aufgrund der Tatsache, daß sie eigentlich nur softwaremäßig kontrolliert werden (die Funktionen des Blitters könnte man auch mit dem 68000er programmieren - nur wären diese Routinen nicht so schnell) einige Möglichkeiten mehr als die VSprites.

So können Sie z.B. das SAVEBOB-Flag in der Flags-Variablen der Bob-Struktur setzen. Dann wird der Hintergrund, den das Bob über-

schreibt, nicht gerettet, und das Bob agiert wie ein Pinsel: Sie überstreichen den ganzen Hintergrund 'erbarmungslos'.

Wenn Sie aber wollen, daß der Hintergrund, den das Bob überschreibt, gerettet und, nachdem das Bob bewegt wurde, wieder dargestellt wird, müssen Sie das SAVEBACK-Flag setzen. Diesmal allerdings in der 'VSprite.Flags'-Variablen. Dafür müssen Sie aber auch genügend Speicher für so viele Bob-Bitplanes reservieren, wie in den Rastport geschrieben werden (Bitte beachten Sie dabei 'PlanePick' und 'PlaneOnOff'). Die Restaurierung des Hintergrundes übernimmt die Bob-Software. Die Adresse dieses 'Hintergrund'-Speichers übergeben Sie an 'Bob.SaveBuffer'.

Mit dem OVERLAY-Flag, das auch in der VSprite-Struktur gesetzt wird, können Sie verhindern, daß die gelöschten Bits des Bobs, die man ja aus dem 'ImageShadow' erkennen kann, auch in den Rastport geschrieben werden. Solche Bits eines Bobs, die nicht gesetzt sind, lassen dann den Originalhintergrund durchscheinen. Das Bob wird sozusagen in die Bitplanes des Rastports geORt. Allerdings müssen Sie den 'ImageShadow' bei der Benutzung des 'OVERLAYs' initialisiert haben (am besten mit 'Bob.ImageShadow = BOBVSprite.CollMask', nachdem InitMasks ausgeführt wurde).

Doch waren dies noch nicht alle Flags:

Die obigen Flags befaßten sich alle mit der Darstellung von Bobs. Es gibt aber auch Flags, die sich mit der Nicht-Darstellung von Bobs befassen. Wenn Sie nämlich feststellen, daß ein Bob 'gegangen' ist, sprich, daß das GELGONE-Flag in 'VSprite.Flags' gesetzt ist, dann haben Sie mit dem Macro 'RemBob (&Bob)' die Möglichkeit, das BOBSAWAY-Flag zu setzen.

Merkt das System nämlich, daß dieses gesetzt wurde, wird das Bob beim nächsten DrawGList-Aufruf nicht mehr mitgezeichnet. Wenn Sie aber wollen, daß das Bob sofort 'verschwindet', dann brauchen Sie nur 'RemIBob (&Bob, &RastPort, &ViewPort)' aufzurufen (das I steht für 'immediate' = sofort). Dann wird das Bob nämlich vom Bildschirm und aus der GEL-Liste entfernt.

Nun haben Sie also das ganze 'Handwerkszeug' zur Erstellung eines Bobs parat. Doch wie zeichnet man die Bobs in den Rastport? Nun, DrawGList übernimmt nicht nur die Vorbereitungen für die Darstellung der VSprites, sondern veranlaßt auch, daß die Bobs in den Rast-

port gezeichnet werden. Natürlich muß auch vor DrawGList die GEL-Liste sortiert worden sein (SortGList).

Folgendes Programm zeigt Ihnen, wie Sie Bobs erstellen, diese bewegen und auf Kollisionen testen können:

```
/*********************************************************/
/*                      Bobs.c                           */
/*                                                       */
/* Bobs pur                                              */
/*                                                       */
/* Compiled with: Lattice V3.10                          */
/*                                                       */
/* (c) Bruno Jennrich                                    */
/*********************************************************/

struct VSpriteExt {              /* Unsere eigenen Daten */
                  int vx,vy;     /* die an die VSprite   */
                };               /* Struktur angehängt   */
                                 /* werden               */

#define VUserStuff struct VSpriteExt
                   /* Muß vor den #INCLUDES geschehen! */

#include "exec/types.h"
#include "exec/memory.h"
#include "exec/devices.h"
#include "devices/keymap.h"
#include "graphics/gfx.h"
#include "graphics/gfxmacros.h"
#include "graphics/copper.h"
#include "graphics/gels.h"
#include "graphics/collide.h"
#include "graphics/gfxbase.h"
#include "graphics/regions.h"
#include "hardware/blit.h"
#include "hardware/custom.h"
#include "intuition/intuition.h"
#include "intuition/intuitionbase.h"
#include "libraries/diskfont.h"
#include "hardware/dmabits.h"

#define RP Screen->RastPort       /* Zugriff auf RastPort */

#define MAXBOBS 6                      /* Wieviele Bobs ? */
struct GfxBase *GfxBase;                   /* BasePointer */
struct IntuitionBase *IntuitionBase;
```

```
UWORD *SaveBuffer;
UWORD *DBufBuffer;

struct NewScreen NewScreen =                /* Unser Screen */
                    {
                      0,0,320,199,3,
                      1,0,
                      0,
                      CUSTOMSCREEN | CUSTOMBITMAP,
                      NULL,
                      "",
                      NULL,NULL
                    };
struct Screen *Screen;

struct BitMap BitMap [2];

struct VSprite Anfang, Ende,          /* VSprites für GEL */
              BobVSprite[MAXBOBS];    /* Liste            */

struct Bob Bob[MAXBOBS];
UWORD *CollMask;                /* für Chip-Mem allocation */

UWORD BorderLine[MAXBOBS][2];          /*  Bobs Borderline */

struct GelsInfo GelsInfo;       /* GelsInfo muß vor        */
                                /* Benutzung der VSprites  */
                                /* ordnungsgemäß           */
                                /* initialisiert werden!   */

WORD Nextlines[8] = {0,0,0,0,0,0,0,0};
                                /* Nextline-Array für GInfo */

WORD *lastColors[8] = {0,0,0,0,0,0,0,0};
                               /* lastColor-Array für GInfo */

struct collTable collTable;
                          /* Kollisions-Einsprung Tabelle */

struct DBufPacket *DBufPackets,*HelpPack; /* für Chip-Mem */

UWORD *Image,*Hilf;
UWORD Baloon[46*2*2] =          /* 46 Zeilen a 2 WORDs */
            {                   /* und das ganze 2 mal */

                0x0007,0xf000,       /* Bob-Plane 1 */
                0x0019,0x6400,
                0x0073,0x3300,
                0x00e3,0x3880,
                0x01c7,0x1c40,
```

```
0x03c7,0x1c20,
0x0387,0x1e20,
0x0787,0x1e10,
0x078f,0x0e10,
0x0f8f,0x0e08,
0x0f0f,0x0f08,
0x0f0f,0x0f08,
0x0f0f,0x0f08,
0x0f0f,0x0f08,
0x0f0f,0x0f08,
0x0f0f,0x0f08,
0x0f0f,0x0f08,
0x0f0f,0x0f08,
0x070f,0x0f10,
0x070f,0x0f10,
0x0307,0x1f20,
0x0307,0x1f20,
0x0187,0x1e40,
0x0187,0x1e40,
0x0087,0x1e80,
0x0087,0x1e80,
0x0043,0x1d00,
0x0043,0x3d00,
0x0023,0x3e00,
0x0023,0x3a00,
0x0003,0x3800,
0x001f,0xfc00,
0x000f,0xf800,
0x000f,0xf800,
0x0007,0xf000,
0x0004,0x1000,
0x0004,0x1000,
0x0006,0x3000,
0x0002,0xa000,
0x0002,0xa000,
0x0003,0xe000,
0x0003,0xe000,
0x0007,0xf000,
0x0007,0xf000,
0x0007,0xf000,
0x0007,0xf000,

0x0000,0x0000,      /* Bob-Plane 2 */
0x0006,0x9800,
0x000c,0xcc00,
0x001c,0xc700,
0x0038,0xe380,
0x0038,0xe3c0,
0x0078,0xe1c0,
```

```
                0x0078,0xe1e0,
                0x0070,0xf1e0,
                0x0070,0xf1e0,
                0x00f0,0xf0f0,
                0x00f0,0xf0f0,
                0x00f0,0xf0f0,
                0x00f0,0xf0f0,
                0x00f0,0xf0f0,
                0x00f0,0xf0f0,
                0x00f0,0xf0f0,
                0x00f0,0xf0f0,
                0x00f0,0xf0e0,
                0x00f0,0xf0e0,
                0x00f8,0xe0c0,
                0x00f8,0xe0c0,
                0x00f8,0xe180,
                0x0078,0xe180,
                0x0078,0xe100,
                0x0078,0xe100,
                0x003c,0xe200,
                0x003c,0xe200,
                0x001c,0xc000,
                0x001c,0xc400,
                0x001c,0xc400,
                0x0000,0x0000,
                0x0000,0x0000,
                0x0000,0x0000,
                0x0000,0x0000,
                0x0004,0x1000,
                0x0004,0x1000,
                0x0006,0x3000,
                0x0002,0xa000,
                0x0002,0xa000,
                0x0003,0xe000,
                0x0003,0xe000,
                0x0007,0xf000,
                0x0007,0xf000,
                0x0007,0xf000,
                0x0007,0xf000,
        };

VOID RandKontrolle();           /* Unsere Routine, die bei */
                                /* Kollisionen mit dem     */
                                /* Rand aufgerufen wird    */

VOID Titsch();                  /* Routine, die bei Gel-Gel */
                                /* Kollision aufgerufen     */
                                /* wird.                    */
```

```
/*********************************************************/
/* Es geht los !                                         */
/*********************************************************/

main()
{
   long i,j;
   int Length;
   long toggle;
   char   *LeftMouse = (char *) 0xBFE001;
                                        /* Linke Maustaste */

   if ((GfxBase = (struct GfxBase *)
           OpenLibrary("graphics.library",0)) == NULL)
              {
                 printf (" No Graphics!\n");
                 exit(0);
              }

   if ((IntuitionBase = (struct IntuitionBase *)
           OpenLibrary("intuition.library",0)) == NULL)
              {
                 printf ("No Intuition!\n");
                 goto cleanup1;
              }

   InitBitMap (&BitMap[0], NewScreen.Depth,
                NewScreen.Width, NewScreen.Height);

   InitBitMap (&BitMap[1], NewScreen.Depth,
                NewScreen.Width, NewScreen.Height);

   for (i=0; i<2; i++)
         for (j=0; j<NewScreen.Depth; j++)
            {
               BitMap[i].Planes[j] = (PLANEPTR) AllocRaster
                   (NewScreen.Width, NewScreen.Height);
               if ((BitMap[i].Planes[j]) == NULL)
                  {
                     printf (" No Space for BitMaps\n");
                     goto cleanup2;
                  }
               else BltClear (BitMap[i].Planes[j],
               RASSIZE(NewScreen.Width, NewScreen.Height),0);
            }


   NewScreen.CustomBitMap = &BitMap[0];

   Screen = (struct Screen *) OpenScreen (&NewScreen);
```

```
if (Screen == 0)  {
                    printf ("No Screen!\n");
                    goto cleanup3;
                  }

Image = (UWORD *)
  AllocMem (MAXBOBS*2*2*46*sizeof(UWORD),MEMF_CHIP);
                              /* Bobdefinition:        */
                              /* 2 WORD breit, 46 hoch */
                              /* 2 Planes tief         */
SaveBuffer = (UWORD *)
  AllocMem (MAXBOBS*3*2*46*sizeof(UWORD),MEMF_CHIP);
                              /* 2 WORD breit, 46 Hoch */
                              /* 3 Planes (PlaneOnOff)!*/
DBufBuffer = (UWORD *)
  AllocMem (MAXBOBS*3*2*46*sizeof(UWORD),MEMF_CHIP);

CollMask = (UWORD *)
  AllocMem (MAXBOBS*2*46*sizeof(UWORD),MEMF_CHIP);
                              /* Bobs    Collision     */
                              /* Maske                 */
                              /* (46 Zeilen a 2 Words) */


DBufPackets = (struct DBufPacket *)
  AllocMem(sizeof(struct DBufPacket) * MAXBOBS,
           MEMF_CLEAR|MEMF_CHIP);

if ((SaveBuffer == 0) | (DBufBuffer == 0) |
    (CollMask == 0) | (DBufPackets == 0) |
    (Image == 0))
      {
        printf (" No Chip Memory for Bobs !!!\n");
        goto cleanup4;
      }

Hilf = Image;

for (i=0; i<2*2*46; i++)
   {
      *Hilf = Baloon[i];
      Hilf++;
   }                          /* BOB in Chip-Mem kopieren */

SetRGB4 (&Screen->ViewPort,0,0,5,15);
                                 /* N' bischen Farbe */
SetRast (&RP,0);
SetAPen (&RP,2);

RP.BitMap = &BitMap [1];
```

```
Length = TextLength (&RP,"Baloons in Action !!!",21);
Move (&RP,320/2-Length/2,100);
Text (&RP,"Baloons in Action !!!",21);

RP.BitMap = &BitMap [0];
Move (&RP,320/2-Length/2,100);
Text (&RP,"Baloons in Action !!!",21);


BltClear (&Anfang, sizeof(struct VSprite),0);
BltClear (&Ende, sizeof(struct VSprite),0);
BltClear (&GelsInfo, sizeof(struct GelsInfo),0);
BltClear (BobVSprite, sizeof(struct VSprite)*MAXBOBS,0);
BltClear (Bob, sizeof(struct Bob)*MAXBOBS,0);
BltClear (&collTable, sizeof (collTable),0);
                /* zur Sicherheit mal alles löschen */

GelsInfo.sprRsrvd = 0xfc;
                        /* Alle Sprites außer 0 u.1 */
                        /* für VSprites !!!         */

GelsInfo.nextLine = Nextlines;
GelsInfo.lastColor = lastColors;
GelsInfo.collHandler = &collTable;

GelsInfo.leftmost = 1;              /* Grenzen für      */
GelsInfo.rightmost = 318;           /* Rand-Kollisionen */
GelsInfo.topmost = 13;
GelsInfo.bottommost = 198;

InitGels (&Anfang, &Ende, &GelsInfo);
                        /* GelsInfo initialisieren   */
RP.GelsInfo = &GelsInfo;  /* und in RastPort einbinden */

SetCollision(0,RandKontrolle,&GelsInfo);
SetCollision(1,Titsch,&GelsInfo);

HelpPack = DBufPackets;
for (i=0; i<MAXBOBS; i++)
   {
      BobVSprite[i].Width = 2;       /* 2 WORDs breit  */
      BobVSprite[i].Height = 46;     /* 46 Zeilen hoch */
      BobVSprite[i].Flags = OVERLAY|SAVEBACK;
                                     /* VSprite ist Bob */
      BobVSprite[i].Depth = 2;       /* 2 'Planes'      */
      BobVSprite[i].ImageData = Image;
      BobVSprite[i].MeMask = 0x2;    /* GEL Kollision */
      BobVSprite[i].HitMask = 0x3;   /* aber mit Rand */
      BobVSprite[i].PlanePick = 0x05;/* Plane 1 und 3 */
      BobVSprite[i].PlaneOnOff = 0x02; /* Plane 2     */
```

```
        BobVSprite[i].CollMask = CollMask+i*2*46;
        BobVSprite[i].BorderLine = &BorderLine[i][0];

        BobVSprite[i].X = 11+i*(320/MAXBOBS-10);
        BobVSprite[i].Y = 15+i*(200/MAXBOBS-10);
                                          /* Position */

        BobVSprite[i].VUserExt.vx = 1;/* eigene          */
        BobVSprite[i].VUserExt.vy = 1;/* Geschwindigkeit */

        Bob[i].Flags = 0;
        Bob[i].BobVSprite = &BobVSprite[i];
        BobVSprite[i].VSBob = &Bob[i];
        Bob[i].ImageShadow = CollMask+i*2*46;
                        /* Image Shadow muß im ChipMem */
                        /* angesiedelt sein.           */

        Bob[i].SaveBuffer = SaveBuffer+i*3*2*46;
        Bob[i].DBuffer = HelpPack;

        HelpPack->BufBuffer = DBufBuffer+i*3*2*46;
        HelpPack++;

        InitMasks (&BobVSprite[i]);
                              /* CollMask und         */
                              /* Borderline berechnen */

        AddBob (&Bob[i], &RP);
                               /* In Liste einreihen */
    }
SetRGB4 (&Screen->ViewPort, 0x5+0x2,0,0,0);
SetRGB4 (&Screen->ViewPort, 0x1+0x2,15,0,0);
        /* Plane 2  wird immer mit Shadow beschrieben */

SetRGB4 (&Screen->ViewPort, 0x4+0x2,15,15,15);

toggle = 1;

while ((*LeftMouse & 0x40) == 0x40)
    {
        for (i=0; i<MAXBOBS; i++)
            {
                BobVSprite[i].Y +=
                      BobVSprite[i].VUserExt.vy;
                BobVSprite[i].X +=
                      BobVSprite[i].VUserExt.vx;
                                  /* VSprites bewegen */
```

```
        }
        SortGList(&RP);                    /* neu Sortieren */
        DoCollision (&RP);          /* Kollisionen testen */
        DrawGList (&RP, &Screen->ViewPort);
                                   /* Copper-Liste generieren */
        WaitTOF();
        RemakeDisplay();
        Screen->ViewPort.RasInfo->BitMap = &BitMap[toggle];
        RP.BitMap = &BitMap[toggle];
        toggle ^= 1;
      }

cleanup4:
    if (Image != 0)
        FreeMem(Image,MAXBOBS*2*2*46*sizeof(UWORD));
    if (SaveBuffer != 0)
        FreeMem(SaveBuffer,MAXBOBS*3*2*46*sizeof(UWORD));
    if (DBufBuffer != 0)
        FreeMem(DBufBuffer,MAXBOBS*3*2*46*sizeof(UWORD));
    if (CollMask != 0)
        FreeMem(CollMask,MAXBOBS*2*46*sizeof(UWORD));
    if (DBufPackets != 0)
        FreeMem(DBufPackets,sizeof(struct DBufPacket) *
                                              MAXBOBS);

              CloseScreen (Screen);
cleanup3:     for (i=0; i<2; i++)
                 for (j=0; j<NewScreen.Depth; j++)
                    if ((BitMap[i].Planes[j]) != NULL)
                       {
                         FreeRaster (BitMap[i].Planes[j],
                         NewScreen.Width, NewScreen.Height);
                       }
cleanup2:     CloseLibrary(IntuitionBase);
cleanup1:     CloseLibrary(GfxBase);
    return(0);
}

/*********************************************************/
/* Diese Routine wird von DoCollision() aus angesprungen */
/* wenn ein VSprite den Rand (border) berührt            */
/*-------------------------------------------------------*/
/* Eingabe-Parameter: VSprite, das den Rand berührt hat  */
/*                    Border - an welchen Rand           */
/*-------------------------------------------------------*/
/* Rückgabe-Werte: keine                                 */
/*********************************************************/
```

```
VOID RandKontrolle (VSprite, Border)
struct VSprite *VSprite;
BYTE Border;
{
   if ((Border & TOPHIT)  == TOPHIT)            /* Oben */
               VSprite->VUserExt.vy *=-1;
   if ((Border & BOTTOMHIT)  == BOTTOMHIT)      /* Unten */
               VSprite->VUserExt.vy *=-1;
   if ((Border & LEFTHIT)  == LEFTHIT)          /* Links */
               VSprite->VUserExt.vx *=-1;
   if ((Border & RIGHTHIT)  == RIGHTHIT)        /* Rechts */
               VSprite->VUserExt.vx *=-1;
}


VOID Titsch(VSpritelinksoben, VSpriterechtsunten)
struct VSprite *VSpritelinksoben,
               *VSpriterechtsunten;
{
    VSpritelinksoben->VUserExt.vx *= -1;
    VSpritelinksoben->VUserExt.vy *= -1;

    VSpriterechtsunten->VUserExt.vx *= -1;
    VSpriterechtsunten->VUserExt.vy *= -1;
}
```

18.2.2 Bobs in gepufferten Bitmaps

In obigem Beispiel haben wir die Technik des 'Double Buffering' verwendet, denn wenn Sie viele, große, oder viele und große Bobs bewegen wollen, kann es passieren, daß Sie sehen, wie das Bob gezeichnet, kurz darauf bei einer Bewegung des Bobs der Hintergrund wieder restauriert und das Bob neu gezeichnet wird. Dies kann sich aber als unangenehmes Geflackere bemerkbar machen. Deshalb schreibt man die Bobs in eine Bitmap, während eine zweite dargestellt wird, und schaltet die Bitmaps um, wenn man mit dem Zeichnen fertig ist.

In obigem Programm haben wir die Bobs in einen Intuition-Screen gezeichnet, was uns zu der Frage bringt, wie sich dieser Darstellungsmodus mit Intuition verträgt.

Die Modi Hi-Res, Lace, HAM und Halfbrite werden von Intuition automatisch erkannt und eingestellt. Aber die Modi Dualplayfield und 'Double Buffering' muß der Benutzer selbst programmieren. Wie das Double Buffering in Intuition funktioniert, haben Sie ja in obigem Programmbeispiel gesehen: Man deklariert den Screen als Screen mit

Custombitmap, initialisiert zwei eigene, gleichgroße Bitmaps und weist diese dann immer abwechselnd der RasInfo-Struktur des Screens zu. Danach berechnet man dann mit RemakeDisplay die Copper-Listen für die jeweils neue Bitmap. So werden sogar eventuelle VSprites nebenbei mit dargestellt (die Aktualisierung der Copper-Listen ist für die Bobs nicht unbedingt notwendig).

Wenn man den DUALPF-Modus in Intuition-Screens installieren will, geht man ähnlich vor. Man deklariert den Screen als Custombitmap-Screen und legt auch hier seine eigenen Bitmaps, die nicht mehr als drei Bitplanes enthalten dürfen, an. Eine weitere RasInfo-Struktur, die auf die eine Bitmap zeigt (RasInfo2.BitMap = EigeneBitMap2), wird an die schon vorhandene 'angehangen' (Screen->ViewPort.RasInfo.Next = &RasInfo2;). Die erste Bitmap wird an die schon vorhandene RasInfo-Struktur gegeben (Screen->ViewPort.RasInfo.BitMap = &EigeneBitMap1;). Nach RemakeDisplay werden dann zwei Bitplanes dargestellt (Natürlich muß vorher der DUALPF-Modus in der NewScreen-Struktur angegeben worden sein).

Allerdings sollten Sie beachten, daß der Screen in diesen beiden Modi nicht bewegt werden darf, da das dazu führen würde, daß gleichzeitig zwei neue Copper-Listen für den Screen berechnet werden (Multitasking). Das kann aber zu Konflikten bzw. Systemabstürzen führen.

Deshalb sollten Sie über Screens, die diese beiden Modi unterstützen, immer ein Window legen, das nicht bewegt werden kann (Wir haben darauf verzichtet, da unsere Programme bei einem Mausklick sowieso beendet werden und ein Neuberechnen der Copper-Listen vom Programm so gar nicht in Frage kommt.)

Doch wieder zurück zu den Bobs. Im 'Double Buffer'-Betrieb müssen natürlich auch die Bobs so geartet sein, daß sie diesen unterstützen können. Wenn nämlich der Hintergrund des Bobs gerettet werden soll (VSprite.Flags = SAVEBACK), dann muß der Hintergrund aus beiden Bitmaps gerettet werden.

Dazu müssen aber weitere Speicherplätze reserviert werden. Außerdem muß die Position, an der der Hintergrund wieder restauriert werden soll, für beide Bitmaps bekannt sein. Für die erste Bitmap wird diese Position in den Variablen 'VSprite.OldX' und 'VSprite.OldY' gespeichert. Der Hintergrund selbst steht dabei in 'Bob.SaveBuffer'. Doch wo steht der Hintergrund der zweiten Bitmap? Wo dessen Position? Nun, hierfür gibt es das 'DBufPacket'. Diese 'DBufPacket'-Struktur wird dem Bob 'mitgeteilt' (Bob.DBuffer = &DBufPacket). Das System weiß

dann, daß die Bobs im 'Double Buffer'-Betrieb sind, wenn dieser Zeiger ungleich 0 ist. Allerdings müssen entweder alle oder kein Bob den 'Double-Buffer'-Betrieb unterstützen.

Damit das 'Double Buffering' dann auch richtig funktioniert, müssen Sie der Variablen 'DBufPacket.BufBuffer' nur noch die Adresse eines Speicherbereiches zu übergeben, der genauso groß ist wie der 'SaveBuffer', um den Hintergrund der zweiten Bitmap enthalten zu können. Den Rest erledigt das System. Toll, nicht?

Aber außer dem 'Double Buffering' gibt es noch eine weitere Methode, 'nervöses' Flackern bei der Darstellung von Bobs, deren Hintergrund gerettet wird, zu verhindern: Sie warten einfach auf den Elektronenstrahl bzw. darauf, daß er sich auf der obersten Zeile des Monitors befindet (WaitTOF hilft uns hierbei). Bevor der Elektronenstrahl die erste Rasterzeile, also die erste Zeile, auf der schon Bitmap-Daten dargestellt werden, erreicht hat, haben Sie schon den alten Hintergrund gerettet und das neue Bob gezeichnet bzw. DrawGList ausgeführt. Danach baut der Elektronenstrahl dann das Bild weiter auf. Was zwischendurch passiert ist, kann aber nicht dargestellt werden, da der Elektronenstrahl sich ja zur Zeit 'wo anders' befand.

Leider funktioniert diese Methode aber nur bei einer kleinen Anzahl nicht allzu großer Bobs, die sich außerdem nicht zu nah am oberen Bildschirmrand bewegen, denn der Elektronenstrahl ist 'verteufelt' schnell, und die Darstellung der Bobs braucht Zeit.

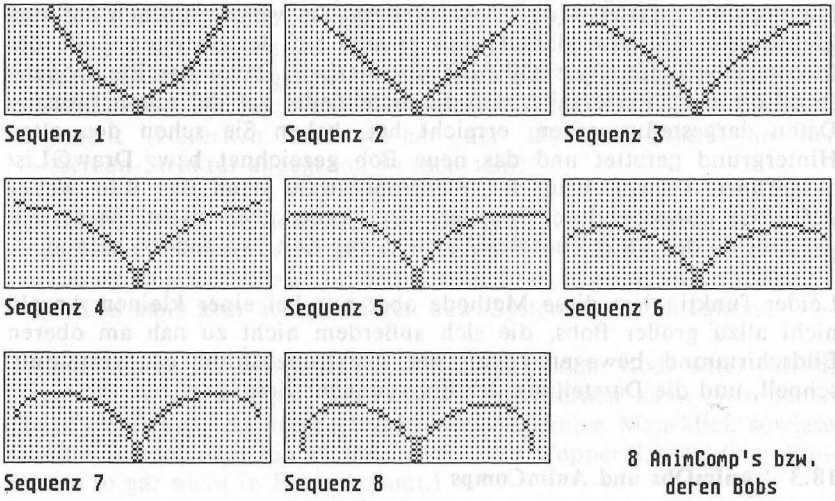
18.3 AnimObs und AnimComps

Doch ist die alleinige Darstellung von 'statischen' Bobs ein wenig fad. Deshalb wurde speziell für die Bobs das Animationssystem entwickelt. Schon zu Beginn des C-Teils haben Sie eine Art der Animation kennengelernt: das 'Color-Cycling' oder die Farbanimation. Dort wurden die Inhalte der Farbregister zyklisch verändert bzw. 'gescrollt', so daß ein Farbregister den Inhalt des vorigen und dieses wieder den Inhalt seines Vorgängers erhielt usw.

Eine zweite Art der Animation haben Sie auch schon kennengelernt: Als Sie nämlich Bobs und VSprites bewegt haben, haben Sie diese animiert. Sie werden jetzt natürlich sagen: 'Das konnte mein 64er auch'. Das stimmt, aber konnte er auch von der Betriebssystem-Software her Bobs animieren? Konnte er so ohne weiteres ein Bob in

verschiedenen 'Sequenzen' zeichnen? Tja, meine Damen und Herren, der Amiga macht's möglich.

Er kennt nämlich die sogenannten 'Animationsobjekte' (AnimObs). Diese AnimObs werden durch sogenannte 'Animationskomponenten' (AnimComps) beschrieben. Diese AnimComps enthalten dann wiederum Ihre vorher definierten Bobs. So können Sie z.B. den Flug einer Möwe in verschiedene Bob-Sequenzen aufteilen, diese Bobs dann mit den AnimComps verbinden und diese wiederum zu einwm AnimOb zusammenzufassen.



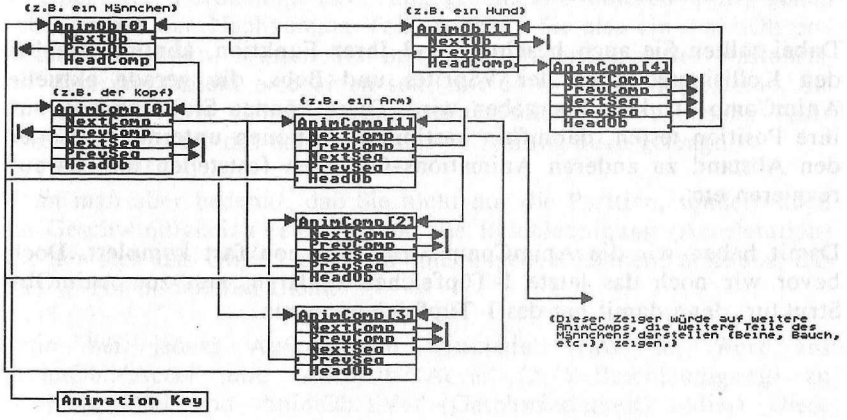
Doch wie geschieht das im einzelnen? Nun, die Bobs werden zuerst wie gewohnt erstellt. Nur müssen Sie außer den Flags, die bestimmen, wie das Bob in die Bitmap geschrieben wird (OVERLAY, SAVEBACK, etc.), auch das BOBISCOMP-Flag der Bobs setzen. Damit 'weiß' die Animations-Software, daß Ihr Bob zu einer Animationskomponente gehört.

Damit das System aber auch weiß, zu welcher Animationskomponente das Bob gehören soll, müssen Sie den Zeiger 'Bob.BobComp' auf die zugehörige 'AnimComp'-Struktur lenken. Wie Sie daraus erkennen können, braucht jedes 'animierte' Bob seine AnimComp-Struktur.

Doch wie verbindet man die AnimComps untereinander? Nun, zuerst gibt es dafür die Zeiger 'AnimComp.NextComp' und 'AnimComp.PrevComp'. Wie Sie sich sicher schon denken können, zeigen diese beiden Zeiger auf den Vorgänger und Nachfolger eines AnimComps. Doch Vorsicht. Bei einer Ring-Sequenz dürfen Sie diese Zeiger nur bei einem einzigen AnimComp dieses Ringes setzen, diese beiden Zeiger der anderen AnimComps müssen dabei auf 0 gesetzt werden. Sonst wird angenommen, daß ein AnimComp der Sequenz z.B. zu einem zweiten Vogel und nicht zu dem gerade animierten Vogel gehört.

Damit komen wir auch schon zu den Zeigern 'NextSeq' und 'PrevSeq'. Wie schon gesagt, können Sie von einem Gegenstand mehrere Positionen definieren (als Bobs) und diese Positionen dann vom Animationssystem nach und nach 'durchblättern' lassen. Das ist fast genauso wie beim Daumenkino. Jede Seite des Daumenkinos, die ja eine neue Position eines Objektes zeigt, wird hier durch verschiedene Bobs dargestellt. Daß die verschiedenen Seiten an Ihnen 'vorbeischnarren', erledigt das Animationssystem.

Dazu müssen Sie allerdings festlegen, wie die 'Seiten' geblättert werden sollen. Die Zeiger 'NextSeq' und PrevSeq' zeigen dabei auf die Nachfolger- und Vorgänger-'Seite', sprich 'AnimComp'. Diese Zeiger bilden dabei einen gechlossenen 'Ring', das heißt, daß die AnimComps erstens untereinander mit den Zeigern 'NextSeq' und 'PrevSeq' verbunden sind, daß weiterhin der Vorgänger des ersten AnimComps das letzte AnimComp ist und der Nachfolger des letzten AnimComps das erste AnimComp ist.



Die AnimOb-Struktur kann nämlich mehrere Sequenz-Ringe enthalten. So können in einem AnimOb z.B. Arme, Beine, Bauch und Kopf eines Männchens zusammengefaßt sein, wobei sich Arme, Beine und Kopf bewegen. Diese 3 Komponenten würden dann als Ring-Sequenzen definiert und von Animate automatisch 'abgespult'. Der Bauch würde dabei dann z.B. nur durch eine einzige AnimComp-Struktur repräsentiert - ein Bauch bewegt sich beim Gehen meistens nicht.

Dabei zeigen dann also die Zeiger 'PrevComp' und 'NextComp' der AnimComps, die das Männchen darstellen, 'hin und her'. Doch wie werden diese AnimComps mit dem AnimOb verbunden? Nun, auch hier gibt es wieder einen Zeiger, der von der AnimOb-Struktur auf die AnimComps zeigt und umgekehrt. Die Hin-Richtung (keine Angst, hier kommt keiner auf die Guillotine) wird durch den Zeiger 'AnimOb.HeadComp' übernommen, der auf das erste AnimComp aller 'Unter'-Objekte zeigt. (Die Komponenten sind ja untereinander durch die 'PrevComp' und 'NextComp'-Zeiger verbunden). Die Rückrichtung wird aber in jedem AnimComp festgelegt. Der Zeiger 'AnimComp.HeadOb' zeigt immer auf das AnimOb, von dem die AnimComps ein Teil sind. So haben Sie in Ihren Routinen, denen ja die gerade aktuelle AnimComp-Struktur übergeben wird, Zugriff auf das zugehörige AnimOb.

Aber das reicht noch nicht, um eine AnimOb-Struktur vollständig zu initialisieren. Denn wo soll das 'Ding' überhaupt auf dem Monitor erscheinen? Diese entscheidende Frage wird mit Hilfe der Variablen 'AnimOb.AnX' und 'AnimOb.AnY' gelöst. Leider können Sie hier nicht einfach sagen, daß Ihr AnimOb z.B. an der Koordinate 160,100 positioniert werden soll. Denn die Koordinaten des AnimObs werden in einer Art 'Festkomma-Zahl' angegeben. Die unteren 6 Bit geben dabei immer den Nachkomma-Teil an. Wenn Sie also ein AnimOb positionieren wollen, müssen Sie immer die Koordinate des Rastports, auf dem das Objekt erscheinen soll, mit 64 (= 2^6) multiplizieren. Sie werden sich jetzt sicher fragen, warum solch eine exotische Positionsangabe? Nun, das haben wir uns zu Anfang auch gefragt.

Wenn man aber bedenkt, daß Sie nicht nur die Position, sondern auch die Geschwindigkeit (Velocity) und die Beschleunigung (Acceleration) mit der sich das Objekt bewegen soll, angeben können, erscheint das alles in einem anderen Licht.

Denn bei jedem Aufruf von 'Animate' wird der Wert aus 'AnimOb.XAccel' und 'AnimOb.YAccel' (X/Y-Beschleunigung) zu 'AnimOb.XVel' und 'AnimOb.YVel' (Geschwindigkeit) addiert. Diese

Wenn Sie dann das RINGTRIGGER-Flag in den AnimComps (AnimComp.Flags = RINGTRIGGER), die diesen Ring bilden, setzen, weiß das System, daß die verschiedenen AnimComps 'geblättert' werden sollen. Dabei können Sie bestimmen, wie lange ein bestimmtes AnimComp zu sehen sein soll.

Die Variable 'AnimComp.TimeSet' gibt nämlich an, wie viele 'Animate'-Aufrufe (Animate ist die Systemroutine, die alle AnimObs bearbeitet) das AnimComp sichtbar bleibt. Die Variable 'TimeSet' wird nämlich zu Anfang einer Sequenz in die Variable 'Timer' kopiert, die dann kontinuierlich, bei jedem 'Animate', dekrementiert wird. Ist 'Timer' dann irgendwann gleich 0, wird das nächste 'AnimComp' des Sequenz-Rings dargestellt.

Doch wieder zurück zur Definition der AnimComps und AnimObs. Denn, wie das Bob 'weiß', zu welchem AnimComp es gehört, so muß auch das AnimComp 'wissen', durch welches Bob es repräsentiert wird. Dazu wird einfach der Zeiger 'AnimComp.AnimBob' auf das spezielle Bob gelenkt.

Als kleinen Leckerbissen können Sie noch eine von Ihnen geschriebene Routine angeben, die von Animate jedesmal bei der Darstellung dieses AnimComps aufgerufen wird. Der Zeiger 'AnimComp.AnimCRoutine' zeigt dabei auf Ihre Funktion. Wenn Sie natürlich keine Funktion definieren wollen, setzen Sie diesen Zeiger einfach auf 0, bei einer eigenen Funktion sollten Sie aber darauf achten, daß diese einen Rückgabewert vom Typ 'Word' hat. Es werden zwar keine Werte an das System zurückgegeben, aber aufgrund der Deklaration in der AnimComp-Struktur sollten Sie dies beachten, um so 'Warnings' des Compilers auszuschalten.

Dabei sollten Sie auch beachten, daß Ihrer Funktion, ähnlich wie bei den Kollisionsroutinen der VSprites und Bobs, die gerade aktuelle AnimComp-Struktur übergeben wird. Diese können Sie dann z.B. auf ihre Position testen, daraufhin bestimmte Aktionen unternehmen oder den Abstand zu anderen Animations-Objekten feststellen und darauf reagieren etc.

Damit haben wir die AnimComp-Struktur schon fast komplett. Doch bevor wir noch das letzte I-Tüpfelchen erklären, erst zur AnimOb-Struktur, denn damit hat das I-Tüpfelchen zu tun.

Drehbewegung eines Rades) und die Positionsänderung asynchron verlaufen. Das heißt, daß z.B. die Drehbewegung eines Rades im krassen Widerspruch zum Weg liegt, den 'es zurücklegt', weil z.B. die Geschwindigkeit laufend erhöht wurde, das Rad sich aber mit der gleichen Geschwindigkeit weiterdreht).

Natürlich können Sie für Ihr Animationsobjekt auch eine Routine festlegen, die von Animate aufgerufen wird. Der Zeiger auf diese Routine heißt 'AnimORoutine'. Sie ist auch vom Typ Word, nur wird ihr natürlich das AnimOb und nicht die Komponente übergeben.

Somit hätten wir alle Animationsstrukturen - das Bob, die Komponente und das Objekt - initialisiert und müssen jetzt nur noch dafür sorgen, daß es auch auf den Bildschirm gebracht wird.

Zuerst brauchen wird dafür natürlich wieder eine für den Bob-Gebrauch vollständig initialisierte GelsInfo-Struktur, die an einen Rastport übergeben wurde.

In diese werden dann mit 'AddAnimOb (&AnimOb, &Schlüssel, &RastPort)' alle Bobs eines AnimObs bzw. dessen Komponenten eingetragen. Der 'Schlüssel', den Sie dabei auch übergeben müssen, ist einfach ein Zeiger auf ein AnimOb (struct AnimOb *Schlüssel = 0), der beim ersten AddAnimOb-Aufruf gleich 0 gesetzt worden sein muß.

Da nämlich für die AnimObs selber keine Liste existiert, muß man ja irgendwie wissen, welches AnimOb zuletzt in die GEL-Liste eingetragen wurde, um sicherzustellen, daß die Objekte ordnungsgemäß miteinander verbunden werden. Die AnimObs sind nämlich auch untereinander verkettet (über die Zeiger AnimOb.PrevOb und AnimOb.NextOb). Dazu zeigt der Schlüssel dann immer auf das zuletzt eingefügte AnimOb.

AddAnimOb sorgt aber nicht nur dafür, daß die Objekte miteinander verbunden werden und die Bobs in die GEL-Liste eingetragen werden, sondern setzt auch die 'Timer'-Variablen der AnimComps auf den Wert, den Sie in 'TimeSet' festgelegt haben, damit der 'Timer' dekrementiert werden kann.

Sind alle AnimObs mit AddAnimOb 'bearbeitet', kann es eigentlch schon losgehen. Rufen Sie nämlich Animate auf, dann wird die 'Timer'-Variable aller gerade aktuellen AnimComps dekrementiert und gegebenenfalls, wenn 'Timer == 0' ist, die nächste Sequenz aktiviert.

Geschwindigkeit wird dann zur tatsächlichen Position in 'AnimOb.AnX' bzw. 'AnimOb.AnY' addiert.

Wenn die Position, Geschwindigkeit und Beschleunigung aber in normalen Punkten angegeben würden, wäre Ihr AnimOb evtl. 'tierisch' schnell, und Sie hätten nur kurze Freude daran, weil es vielleicht nach dem fünften Animate-Aufruf schon aus dem Rastport 'entflohen' ist. Um eben dies zu vermeiden, wird ein AnimOb nur alle 64 'Punkte' tatsächlich einen Punkt weiterbewegt.

Jetzt tritt aber ein auf den ersten Blick unlösbares Problem auf: Mit dieser 64er-Multiplikation könnte man ohne weiteres leben, aber daß man auf normalem Wege nicht alle möglichen Positionen eines Rastports mit dem AnimOb 'anfahren' kann, gibt schon zu denken. Denn für die Position (AnX und AnY) stehen jeweils nur 16 Bit bereit. Somit ergibt sich ein Wertebereich von ±32768. Wenn man diese Werte aber durch 64 teilt, bekommt man nur Werte von -512 bis 512. Wie soll man also ein Objekt in einem Hi-Res-Rastport an Position 639 (in X-Richtung) kriegen? Da dies auf einfachem Wege nicht zu erreichen ist, müssen wir hier einen - uneleganten - Trick anwenden.

Das I-Tüpfelchen der AnimComps kommt jetzt nämlich zum Tragen. Denn Sie können die Position der einzelnen Animationskomponenten relativ zum Animationsobjekt, zu dem diese gehören, angeben (auch in 64er-Schritten!).

Wenn Sie also für Ihre Animationskomponenten einen Offset von 128*64 in 'AnimComp.XTrans' angeben, können Sie alle Positionen des Bildschirms anfahren, wobei bei Positionen, die kleiner als 128 sind, Werte zwischen -128 und 0 in 'AnimOb.AnX' angegeben werden. (Für die Y-Position reicht der Wertebereich von ±512 für alle Auflösungen. Um aber Ihre Komponenten in Y-Richtung relativ zum AnimOb zu positionieren, müssen Sie die Variable 'AnimComp.YTrans' benutzen).

Zwei weitere Variablen können auch zur Positionsänderung eines Objektes herangezogen werden: RingXTrans und RingYTrans. Die Werte dieser Variablen werden immer unverändert zur augenblicklichen Position addiert. Wenn Sie also auf Geschwindigkeiten und Beschleunigungen verzichten wollen, setzen Sie die jeweiligen AnimOb-Variablen (XVel, YVel, XAccel und YAccel) gleich 0, und initialisieren Sie 'RingXTrans' und 'RingYTrans' mit den gewünschten Werten. (Achten Sie darauf, daß die Geschwindigkeit 'angemessen' ist, sonst könnte es in Sequenz-Animationen passieren, daß die interne Bewegung des Objektes (z.B. der Flügelschlag eines Vogels oder die

```
#include "intuition/intuition.h"
#include "intuition/intuitionbase.h"
#include "libraries/diskfont.h"
#include "hardware/dmabits.h"

#define RP Screen->RastPort       /* Zugriff auf RastPort */

#define MAXBOBS 8                 /* Möwe in 8 Positionen */

#define MAXCOMPS (MAXBOBS*2-2)  /* Sequenz besteht aus 14 */
                                /* Bildern: 8 'hin' und   */
                                /* 6 'zurück'.            */
                                /* 'Amplitudenbilder'     */
                                /* kommen nur einmal vor. */

struct GfxBase *GfxBase;                  /* BasePointer */
struct IntuitionBase *IntuitionBase;

struct NewScreen NewScreen =              /* Unser Screen */
                    {
                      0,0,640,256,1,
                      1,0,
                      HIRES,
                      CUSTOMSCREEN,
                      NULL,
                      "",
                      NULL,NULL
                    };

struct Screen *Screen;

struct VSprite Anfang, Ende,          /* VSprites für GEL */
               BobsVSprite[MAXCOMPS]; /* Liste und Bobs   */

struct Bob Bobs[MAXCOMPS];                /* Unsere Bobs */
                    /* Achtung: 6 Bobs haben gleiches */
                    /* Image bzw. Aussehen!!!         */

UWORD *BobBuffer;
          /* MAXBOBS Bobs, mit 20 Linien a 3 WORDs für  */
          /* eine BitPlane                              */

                     /* Speicherbereich für SAVEBACK  */
                     /* In diesem Beispiel würde ein  */
                     /* Buffer reichen, aber wegen des*/
                     /* Prinzips und fürs einfache    */
                     /* Zufügen neuer, anderer Bobs,  */
                     /* hat jedes Bob einen           */
                     /* 'SaveBuffer'                  */
```

Außerdem wird die Position unter Berücksichtigung von RingXTrans, RingYtrans, XVel, YVel, XAccel, YAccel und XTrans sowie YTrans neu berechnet und an das zu zeichnende Bob übergeben.

Wenn Sie dann wie gwohnt SortGList und DrawGList aufrufen, werden Ihre AnimObs auf dem Bildschirm dargestellt.

18.3.1 Kollisionen mit AnimObs

Da ja die AnimObs auf unterster Stufe durch Bobs dargestellt werden, können Sie diese auch zur Kollisionskontrolle benutzen. Wenn Sie die 'HitMask' und 'MeMask' der VSprite-Struktur des Bobs wie gewohnt setzen (für alle Bobs in einem AnimComp-Ring möglichst gleich) und Ihre Routinen mit SetCollision festlegen, können Sie auch hier wie gewohnt auf Kollisionen testen.

Damit das alles jedoch nicht trockene Theorie bleibt, bieten wir Ihnen zum Abschluß der Animationen ein Programm, das eine fliegende Möwe mit Flügelschlag (in einem AnimOb) darstellt:

```
/*********************************************************/
/*                    LetsAnimate.c                      */
/*                                                       */
/* In diesem Programm werden mittels der Animations-     */
/* Routinen des Amiga AnimOb's (hier eine Möwe) und      */
/* AnimComp's (verschiedenene Flügelstellungen) erzeugt  */
/* und bewegt.                                           */
/*                                                       */
/* Compiled with: Lattice V3.10                          */
/*                                                       */
/* (c) Prgram by Bruno Jennrich, Idea and Artwork by my  */
/* little sister Ute.                                    */
/*********************************************************/

#include "exec/types.h"
#include "exec/memory.h"
#include "exec/devices.h"
#include "devices/keymap.h"
#include "graphics/gfx.h"
#include "graphics/gfxmacros.h"
#include "graphics/copper.h"
#include "graphics/gels.h"
#include "graphics/gfxbase.h"
#include "graphics/regions.h"
#include "hardware/blit.h"
#include "hardware/custom.h"
```

```
    {0x0000,0x1c38,0x0000},    /* Hier müßten   */
    {0x0000,0x0660,0x0000},    /* die Daten     */
    {0x0000,0x0180,0x0000},    /* für weitere   */
    {0x0000,0x0180,0x0000},    /* BitPlanes     */
    {0x0000,0x0180,0x0000}     /* folgen        */
},
{
    {0x0000,0x0000,0x0000},   /* Daten für      */
    {0x0000,0x0000,0x0000},   /* Bob2. Nur eine */
    {0x0200,0x0000,0x0040},   /* BitPlane.      */
    {0x0100,0x0000,0x0080},
    {0x0080,0x0000,0x0100},
    {0x0060,0x0000,0x0600},
    {0x0030,0x0000,0x0c00},
    {0x0018,0x0000,0x1800},
    {0x000c,0x0000,0x3000},
    {0x0006,0x0000,0x6000},
    {0x0003,0x0000,0xc000},
    {0x0001,0x8001,0x8000},
    {0x0000,0xe007,0x0000},
    {0x0000,0x300c,0x0000},
    {0x0000,0x1818,0x0000},
    {0x0000,0x0c30,0x0000},    /* Hier müßten   */
    {0x0000,0x0660,0x0000},    /* die Daten     */
    {0x0000,0x0180,0x0000},    /* für weitere   */
    {0x0000,0x0180,0x0000},    /* BitPlanes     */
    {0x0000,0x0180,0x0000}     /* folgen        */
},
{
    {0x0000,0x0000,0x0000},   /* Daten für      */
    {0x0000,0x0000,0x0000},   /* Bob3. Nur eine */
    {0x0000,0x0000,0x0000},   /* BitPlane.      */
    {0x0f00,0x0000,0x00f0},
    {0x00c0,0x0000,0x0300},
    {0x0060,0x0000,0x0600},
    {0x0018,0x0000,0x1800},
    {0x0006,0x0000,0x6000},
    {0x0003,0x0000,0xc000},
    {0x0001,0x8001,0x8000},
    {0x0000,0x4002,0x0000},
    {0x0000,0x2004,0x0000},
    {0x0000,0x1008,0x0000},
    {0x0000,0x0810,0x0000},
    {0x0000,0x0c30,0x0000},
    {0x0000,0x0660,0x0000},    /* Hier müßten   */
    {0x0000,0x0240,0x0000},    /* die Daten     */
    {0x0000,0x03c0,0x0000},    /* für weitere   */
    {0x0000,0x0180,0x0000},    /* BitPlanes     */
    {0x0000,0x0180,0x0000}     /* folgen        */
},
```

```
UWORD *BobMask;                    /* Bob Kollisions-Masken */
                                   /* Speicher              */
                                   /* (20 Linien * 3 Words) */

UWORD BobBorderLine [MAXCOMPS][3]; /* Bobs Borderline       */
                                   /* Speicher              */
                                   /* (logisches OR aller */
                                   /* Bob Zeilen in einer */
                                   /* Zeile (hier:3 Words)*/

extern struct Custom custom;       /* Zugriff auf Hardware */
                                   /* Register für Copper  */

struct UCopList *UCopList;         /* eigene Copper Liste */

struct AnimComp AnimComp[MAXCOMPS];
                          /* Möwe 'hin und zurück',     */
                          /* aber Start (Sequenz 1) und */
                          /* Endposition (Sequenz 8)    */
                          /* (s. Abbildung)             */
                          /* kommen in jeder Sequenz    */
                          /* nur einmal, nicht zweimal  */
                          /* wie die anderen Positionen */
                          /* der Möwe, vor !            */

struct AnimOb *HeadOb = 0,              /* Animations Key */
              Moewe;                    /* unsere Möwe    */

struct GelsInfo GelsInfo;  /* GelsInfo muss vor        */
                           /* Benutzung der Animation- */
                           /* Routinen ordnungsgemäß   */
                           /* initialisiert werden!    */
UWORD *Image,*Help;
UWORD BobImage[MAXBOBS][20][3] =
      {{
            {0x0080,0x0000,0x0100},   /* Daten für      */
            {0x0080,0x0000,0x0100},   /* Bob1. Nur eine */
            {0x00c0,0x0000,0x0300},   /* BitPlane pro   */
            {0x0040,0x0000,0x0200},   /* Bob            */
            {0x0060,0x0000,0x0600},
            {0x0030,0x0000,0x0c00},   /* Bitte benutzen */
            {0x0010,0x0000,0x0800},   /* Sie die Copy   */
            {0x0018,0x0000,0x1800},   /* Funktion Ihres */
            {0x000c,0x0000,0x3000},   /* Editors !!!    */
            {0x0004,0x0000,0x2000},   /* Erspart 'ne    */
            {0x0006,0x0000,0x6000},   /* Menge Arbeit!  */
            {0x0003,0x0000,0xc000},
            {0x0001,0x8001,0x8000},
            {0x0000,0xc003,0x0000},
            {0x0000,0x700e,0x0000},
```

```
    {0x0000,0x0000,0x0000},
    {0x0000,0x0000,0x0000},
    {0x07e0,0x0000,0x07e0},
    {0x3c3e,0x0000,0x7c3c},
    {0x4003,0x8001,0xc001},
    {0x0000,0x6006,0x0000},
    {0x0000,0x300c,0x0000},
    {0x0000,0x1818,0x0000},
    {0x0000,0x0420,0x0000},
    {0x0000,0x0660,0x0000},
    {0x0000,0x03c0,0x0000},    /* Hier müßten   */
    {0x0000,0x0180,0x0000},    /* die Daten     */
    {0x0000,0x0180,0x0000},    /* für weitere   */
    {0x0000,0x0180,0x0000},    /* BitPlanes     */
    {0x0000,0x0180,0x0000}     /* folgen        */
},
{
    {0x0000,0x0000,0x0000},   /* Daten für      */
    {0x0000,0x0000,0x0000},   /* Bob7. Nur eine */
    {0x0000,0x0000,0x0000},   /* BitPlane.      */
    {0x0000,0x0000,0x0000},
    {0x0000,0x0000,0x0000},
    {0x0000,0x0000,0x0000},
    {0x0000,0x0000,0x0000},
    {0x03fc,0x0000,0x3fc0},
    {0x0c0f,0x8001,0xe030},
    {0x1000,0xe007,0x0008},
    {0x2000,0x6003,0x0004},
    {0x2000,0x1818,0x0004},
    {0x0000,0x0c60,0x0000},
    {0x0000,0x0660,0x0000},
    {0x0000,0x0240,0x0000},
    {0x0000,0x0240,0x0000},    /* Hier müßten   */
    {0x0000,0x0180,0x0000},    /* die Daten     */
    {0x0000,0x0180,0x0000},    /* für weitere   */
    {0x0000,0x0180,0x0000},    /* BitPlanes     */
    {0x0000,0x0180,0x0000}     /* folgen        */
},
{
    {0x0000,0x0000,0x0000},   /* Daten für      */
    {0x0000,0x0000,0x0000},   /* Bob8. Nur eine */
    {0x0000,0x0000,0x0000},   /* BitPlane.      */
    {0x0000,0x0000,0x0000},
    {0x0000,0x0000,0x0000},
    {0x0000,0x0000,0x0000},
    {0x0000,0x0000,0x0000},
    {0x0000,0x0000,0x0000},
    {0x0000,0x0000,0x0000},
    {0x01fe,0x0000,0x7f80},
    {0x0301,0xc003,0x80c0},
```

```
{
    {0x0000,0x0000,0x0000},   /* Daten für      */
    {0x0000,0x0000,0x0000},   /* Bob4. Nur eine */
    {0x0000,0x0000,0x0000},   /* BitPlane.      */
    {0x0000,0x0000,0x0000},
    {0x3000,0x0000,0x000c},
    {0x0f80,0x0000,0x01f0},
    {0x0070,0x0000,0x0e00},
    {0x000c,0x0000,0x3000},
    {0x0003,0x0000,0xc000},
    {0x0001,0x8001,0x8000},
    {0x0000,0x6006,0x0000},
    {0x0000,0x300c,0x0000},
    {0x0000,0x1818,0x0000},
    {0x0000,0x0c30,0x0000},
    {0x0000,0x0420,0x0000},
    {0x0000,0x0240,0x0000},    /* Hier müßten   */
    {0x0000,0x03c0,0x0000},    /* die Daten     */
    {0x0000,0x0180,0x0000},    /* für weitere   */
    {0x0000,0x0180,0x0000},    /* BitPlanes     */
    {0x0000,0x0180,0x0000}     /* folgen        */
},
{
    {0x0000,0x0000,0x0000},   /* Daten für      */
    {0x0000,0x0000,0x0000},   /* Bob5. Nur eine */
    {0x0000,0x0000,0x0000},   /* BitPlane.      */
    {0x0000,0x0000,0x0000},
    {0x0000,0x0000,0x0000},
    {0x0000,0x0000,0x0000},
    {0x7ff0,0x0000,0x0ffe},
    {0x800f,0x0000,0x7001},
    {0x0003,0x8001,0xc000},
    {0x0000,0xc003,0x0000},
    {0x0000,0x2004,0x0000},
    {0x0000,0x1818,0x0000},
    {0x0000,0x0810,0x0000},
    {0x0000,0x0420,0x0000},
    {0x0000,0x0660,0x0000},
    {0x0000,0x0240,0x0000},    /* Hier müßten   */
    {0x0000,0x0180,0x0000},    /* die Daten     */
    {0x0000,0x0180,0x0000},    /* für weitere   */
    {0x0000,0x0180,0x0000},    /* BitPlanes     */
    {0x0000,0x0180,0x0000}     /* folgen        */
},
{
    {0x0000,0x0000,0x0000},   /* Daten für      */
    {0x0000,0x0000,0x0000},   /* Bob6. Nur eine */
    {0x0000,0x0000,0x0000},   /* BitPlane.      */
    {0x0000,0x0000,0x0000},
    {0x0000,0x0000,0x0000},
```

```
BobMask = (UWORD *)AllocMem(MAXCOMPS*20*3*sizeof(UWORD),
                            MEMF_CLEAR|MEMF_CHIP);

Image = (UWORD *)AllocMem(MAXBOBS*20*3*sizeof(UWORD),
                          MEMF_CLEAR|MEMF_CHIP);

if ((Image == 0) | (BobBuffer == 0) | (BobMask == 0))
   {
      printf ("No Chip Memory !!\n");
      goto cleanup4;
   }

Help = Image;

for (i=0;i<MAXBOBS;i++)
   for (j=0;j<20;j++)
      for (k=0;k<3;k++)
         {
            *Help = BobImage[i][j][k];
            Help++;
         }

SetRGB4 (&Screen->ViewPort,0,2,8,15);
SetRGB4 (&Screen->ViewPort,1,0,0,0);

SetRast (&RP,0);

BltClear (&Anfang, sizeof(struct VSprite),0);
BltClear (&Ende, sizeof(struct VSprite),0);
BltClear (&GelsInfo, sizeof(struct GelsInfo),0);
BltClear (BobsVSprite,
          sizeof(struct VSprite)*MAXCOMPS,0);
BltClear (Bobs,
          sizeof(struct Bob)*MAXCOMPS,0);

BltClear (AnimComp,
          sizeof(struct AnimComp)*MAXCOMPS,0);

BltClear (&Moewe, sizeof(struct AnimOb),0);
/*****************************************************/
/*            A little Copper - Power                */
/*****************************************************/

UCopList = (struct UCopList *)
   AllocMem (sizeof(struct UCopList),
             MEMF_CHIP | MEMF_CLEAR);

CWAIT (UCopList,70,0);
CMOVE (UCopList, custom.color[1], 0x0aaa);
                        /* Farbänderrung der Möve */
```

```
            {0x0400,0x6006,0x0020},
            {0x0800,0x1818,0x0010},
            {0x0800,0x0c30,0x0010},
            {0x1000,0x0420,0x0008},
            {0x1000,0x03c0,0x0008},    /* Hier müßten   */
            {0x1000,0x0180,0x0008},    /* die Daten     */
            {0x0000,0x0180,0x0000},    /* für weitere   */
            {0x0000,0x0180,0x0000},    /* BitPlanes     */
            {0x0000,0x0180,0x0000}     /* folgen        */
        }
    };


WORD MoveMoewe();                  /* Unsere AnimOb-Routine */
/* WORD Comp();                  Optionale AnimComp-Routine */

/*********************************************************/
/* Es geht los !                                         */
/*********************************************************/

main()
{
   int  i,j,k;
   char *LeftMouse = (char *) 0xBFE001;
                                   /* Linke Maustaste */

   if ((GfxBase = (struct GfxBase *)
           OpenLibrary("graphics.library",0)) == NULL)
           {
              printf ("No Graphics !!!\n");
              exit(0);
           }

   if ((IntuitionBase = (struct IntuitionBase *)
           OpenLibrary("intuition.library",0)) == NULL)
           {
              printf ("No Intuition !!!\n");
              goto cleanup2;
           }

   if ((Screen = (struct Screen *)
                 OpenScreen (&NewScreen)) == NULL)
        {
          printf ("No Screen !!!\n");
          goto cleanup3;
        }

   BobBuffer = (UWORD *)AllocMem(MAXCOMPS*20*3*sizeof(UWORD),
                           MEMF_CLEAR | MEMF_CHIP);
```

```
                                /* hinterher wieder    */
                                /* herstellen          */
        BobsVSprite[i].Depth = 1;
                               /* Nur eine Plane pro Bob */

                              /* BobsVSprite[i].ImageData */
                              /* wird in extra Schleife   */
                              /* initialisiert            */

        BobsVSprite[i].PlanePick = 1;
                             /* Auch im RastPort dann nur */
                             /* erste Plane beschreiben   */

        BobsVSprite[i].PlaneOnOff = 0;
                            /* Andere Planesbleiben auf 0 */

        BobsVSprite[i].CollMask = BobMask+i*20*3;
        BobsVSprite[i].BorderLine = &BobBorderLine[i][0];
                /* Speicher für CollMask und Borderline */
                /* zuweisen                             */

        Bobs[i].ImageShadow = BobMask+i*20*3;
                                    /* Shadow = CollMask */
        Bobs[i].Flags = BOBISCOMP;
                                      /* Bob ist Teil... */
        Bobs[i].BobComp = &AnimComp[i];
                                  /* ...dieses AnimComps */

        Bobs[i].SaveBuffer = BobBuffer+i*20*3;
                        /* Speicher für SAVEBACK Option */

        BobsVSprite[i].Y = 0;         /* Position wird von */
        BobsVSprite[i].X = 0;         /* Animationssystem  */
                                      /* berechnet !!!     */

        Bobs[i].Before = 0;          /* Prioritäten auch ! */
        Bobs[i].After = 0;
    }

for (i=0; i<MAXBOBS; i++)
    {
        BobsVSprite[i].ImageData = Image+i*20*3;
        InitMasks (&BobsVSprite[i]);
              /* Initialisiert CollMask und BorderLine. */
              /* (Speicher muß dafür zugewiesen         */
              /*  worden sein !!!)                      */
    }

for (i=MAXBOBS-2; i>0; i--)
    {
```

```
CWAIT (UCopList,90,0);
CMOVE (UCopList, custom.color[1], 0x0000);

CWAIT (UCopList, 150,0);
CMOVE (UCopList, custom.color[0], 0x000f);     /* Meer */
CEND  (UCopList);

Screen->ViewPort.UCopIns = UCopList;
                           /* Copper Liste einbinden */
RemakeDisplay();           /* und neu berechnen      */
/*******************************************************/
GelsInfo.sprRsrvd = 0xff; /* Alle Sprites für VSprites */

                  /* Speicher für GelInfo reservieren */

GelsInfo.nextLine = (WORD *)AllocMem(sizeof (WORD)*8,
                      MEMF_PUBLIC|MEMF_CLEAR);

GelsInfo.lastColor = (WORD **)AllocMem(sizeof (LONG)*8,
                         MEMF_PUBLIC|MEMF_CLEAR);

GelsInfo.collHandler = (struct collTable *)
                AllocMem(sizeof (struct collTable),
                         MEMF_PUBLIC|MEMF_CLEAR);

if ((GelsInfo.nextLine == 0)|(GelsInfo.lastColor == 0) |
    (GelsInfo.collHandler == 0))
      {
         printf (" No Memory for GelsInfo !!!\n");
         goto cleanup5;
      }

GelsInfo.leftmost = 0;          /* Boundary Collisions */
GelsInfo.rightmost = 640;       /* Rand-Kollisionen    */
GelsInfo.topmost = 0;
GelsInfo.bottommost = 200;

InitGels (&Anfang, &Ende, &GelsInfo);
                         /* GelsInfo initialisieren   */
RP.GelsInfo = &GelsInfo;  /* und in RastPort einbinden */

for (i=0; i<MAXCOMPS; i++)
   {
      Bobs[i].BobVSprite = &BobsVSprite[i];
      BobsVSprite[i].VSBob = &Bobs[i];
      BobsVSprite[i].Width = 3;       /* Alle Bobs sind */
      BobsVSprite[i].Height = 20;     /* gleichgross    */
      BobsVSprite[i].Flags = SAVEBACK;
                              /* Hintergrund retten  */
                              /* (in BobBuffer) und  */
```

```
      AnimComp[14-i].HeadOb = &Moewe;
   }                                        /* s. oben */

for (i=1; i<MAXCOMPS-1; i++)
   {
      AnimComp[i].NextSeq = &AnimComp[i+1];
      AnimComp[i].PrevSeq = &AnimComp[i-1];
           /* intitialisiere 'Ring' für Ring/Sequenz-  */
           /* Animation (PrevComp und NextComp bleiben */
           /* hier ohne Wirkung, und dürfen in diesem  */
           /* Fall nicht initialisiert, sonder auf 0   */
           /* gesetzt,  werden.                        */
   }

AnimComp[0].NextSeq = &AnimComp[1]; /* Schliese 'Ring' */
AnimComp[0].PrevSeq = &AnimComp[13];

AnimComp[MAXCOMPS-1].NextSeq = &AnimComp[0];
AnimComp[MAXCOMPS-1].PrevSeq = &AnimComp[12];


Moewe.HeadComp = &AnimComp[0];       /* AnimOb's erstes */
                                     /* AnimComp        */
Moewe.RingXTrans = 2*64;             /* X/Y Translation */
Moewe.RingYTrans = 1*64;             /* der Möwe        */

Moewe.AnX = 0;                      /* Ausgangsposition */
Moewe.AnY = 0;                     /* auf XTrans achten */
Moewe.XAccel = 0x0000;           /* keine Beschleunigung */
Moewe.YAccel = 0x0000;
Moewe.XVel = 0x0000;          /* keine 'Beschleunigungs- */
Moewe.YVel = 0x0000;          /* geschwindigkeit'        */
Moewe.AnimORoutine = MoveMoewe;
                          /* Unsere Komtroll/Routine */


AddAnimOb (&Moewe, &HeadOb, &RP);
                                     /* AnimOb in Liste */
while ((*LeftMouse & 0x40) == 0x40)
   {
      Animate(&HeadOb,&RP);                 /* Animieren */
      SortGList(&RP);                       /* Sortieren */
      WaitTOF();/* Um Blinken der Bobs zu verhindern ! */
      DrawGList(&RP,&Screen->ViewPort);     /* Zeichnen */
   }

                  /* GelsInfo's Speicher freigeben */
```

```
      BobsVSprite[MAXCOMPS-i].ImageData =
                                  Image+i*20*3;
                              /* Aussehen der Bobs für */
                              /* Sequenz-Animation     */
      InitMasks (&BobsVSprite[i]);
            /* Initialisiert CollMask und BorderLine. */
            /* (Speicher muß dafür zugewiesen         */
            /*  worden sein !!!)                      */
   }

for (i=0; i<MAXBOBS; i++)
   {
      AnimComp[i].AnimBob = &Bobs[i];
      AnimComp[i].PrevComp = 0;        /* kein weiteres */
      AnimComp[i].NextComp = 0;        /* AnimComp in   */
                                       /* AnimOb        */

      AnimComp[i].TimeSet = 3;    /* 3 mal Animate()    */
                                  /* bevor neue Sequenz */
                                  /* dargestellt wird.  */

      AnimComp[i].Flags = RINGTRIGGER;
                           /* Ring-Sequenz-Animation */

      AnimComp[i].XTrans = 128*64;
      AnimComp[i].YTrans = 0;
                          /* Offset zu AnX/AnY in Möwe */
                          /* Achtung: Festkomma !!!    */

      AnimComp[i].AnimCRoutine = NULL;          /* Comp; */
                         /* keine AnimComp Routine     */

      AnimComp[i].HeadOb = &Moewe;
                    /* HeadOb für AnimComp (für eigene */
                    /* Routine als Kreuzverweis !)     */
   }
/* Sequenz:          1 2 3 4 5 6 7 8 9 10 11 12 13 14   */
/* Bobs/AnimComp:    1 2 3 4 5 6 7 8 7 6  5  4  3  2    */

for (i=MAXBOBS-2; i>0; i--)
   {
      AnimComp[14-i].AnimBob = &Bobs[i];
      AnimComp[14-i].PrevComp = 0;
      AnimComp[14-i].NextComp = 0;
      AnimComp[14-i].TimeSet = 3;
      AnimComp[14-i].Flags = RINGTRIGGER;
      AnimComp[14-i].XTrans = 128*64;
      AnimComp[14-i].YTrans = 0;
      AnimComp[14-i].AnimCRoutine = NULL;     /* Comp; */
      Bobs[14-i].BobComp = &AnimComp[14-i];
```

```
/*********************************************************/
/* Diese Funktion würde jedesmal von Animate() aufgerufen */
/*-------------------------------------------------------*/
/* Eingabe-Parameter: AnimComp-Struktur, die von Animate()*/
/*                    gerade animiert wird. Dieser       */
/*                    Parameter wird von Animate()       */
/*                    übergeben.                         */
/*-------------------------------------------------------*/
/* Rückgabe-Werte: keine                                 */
/*********************************************************/

/* WORD Comp(Component)
struct AnimComp *Component;
{
   return(0);
} */
```

```
    cleanup5:
    if (GelsInfo.nextLine != 0)
       FreeMem (GelsInfo.nextLine, sizeof (WORD)*8);
    if (GelsInfo.lastColor != 0)
       FreeMem (GelsInfo.lastColor, sizeof (LONG)*8);
    if (GelsInfo.collHandler != 0)
       FreeMem (GelsInfo.collHandler,
                          sizeof(struct collTable));

    cleanup4:
    if (Image != 0)
        FreeMem(Image,MAXBOBS*20*3*sizeof(UWORD));
    if (BobBuffer != 0)
        FreeMem(BobBuffer,MAXCOMPS*20*3*sizeof(UWORD));
    if (BobMask != 0)
        FreeMem(BobMask,MAXCOMPS*20*3*sizeof(UWORD));

    cleanup3:   CloseScreen (Screen);
    cleanup2:   CloseLibrary(IntuitionBase);
    cleanup1:   CloseLibrary(GfxBase);
    return(0);
}

/*********************************************************/
/* Diese Funktion wird jedesmal von Animate() aufgerufen */
/*-------------------------------------------------------*/
/* Eingabe-Parameter: AnimOb-Struktur, die von Animate() */
/*                    gerade animiert wird. Dieser       */
/*                    Parameter wird von Animate()       */
/*                    übergeben.                         */
/*-------------------------------------------------------*/
/* Rückgabe-Werte: keine                                 */
/*********************************************************/

WORD MoveMoewe (Object)
struct AnimOb *Object;
{
    if ((Object->AnX < (-128*64)) ||   /* Ist unser Objekt */
        (Object->AnX > ((512-48)*64))) /* angeeckt ?       */
       Object->RingXTrans *= -1;

    if ((Object->AnY < (0)) ||
        (Object->AnY > (120*64)))
       Object->RingYTrans *= -1;
    return(0);
}
```

19. Die Copper-Programmierung

Wie Sie ja schon aus vorigen Kapiteln wissen, ist der Copper ein Coprozessor des Amiga (daher ja auch der Name). Er ist für den Bildschirmaufbau zuständig, das heißt, er bestimmt, was gerade an der Stelle, an der sich der Elektronenstrahl befindet, geschehen soll.

Das ist aber noch nicht alles. Mit seiner Hilfe werden nämlich auch die Sprites und VSprites dargestellt (s. Hardware-Register), und, das ist der Clou, der Benutzer kann ihn sogar selbst programmieren.

Dazu läßt man sich einfach einen Zeiger auf eine User-Copper-Liste ('struct UCopList *UserCopperListe') zuweisen, die man dann mittels der Copper-Instructions - so werden die Copper-Befehle genannt - programmieren kann.

Doch bevor wir diese benutzen können, müssen Sie dem Zeiger auf Ihre 'Benutzer-Copper-Liste' erst genügend Speicher zuweisen: 'UserCopperListe = (struct UCopList *) AllocMem(sizeof (struct UCopIns), MEMF_PUBLIC | MEMF_CLEAR)'.

Der Speicher für diese Struktur muß gelöscht werden - entweder direkt mit AllocMem durch die MEMF_CLEAR Option, oder z.B. nachträglich mit BltClear - damit erstens neue Copper Instructions aufgenommen werden können und zweitens MakeVPort überhaupt weiß, daß diese User-Copper-Liste noch leer ist, also noch keine Befehle enthält (sonst würde diese übergangen).

Doch nachdem die Voraussetzungen geklärt sind, kömmen wir zur Copper-Sprache. Diese ist extrem einfach. Sie besteht nämlich aus nur drei Befehlen: CMOVE, CWAIT, CEND. Allerdings reichen diese drei Befehle vollkommen aus, um z.B. die 'herunterziehbaren' Intuition-Screens zu programmieren.

Doch was bewirken die drei Copper-Befehle, und welches Format haben sie?

Nun, Der CMOVE-Befehl veranlaßt, daß ein bestimmter, von Ihnen festgelegter Wert in ein auch von Ihnen bestimmtes Hardware-Register (s. Anhang) geschrieben wird. Um Zugriff auf die Hardware-Register zu haben, benutzt man einfach die Custom-Struktur. Mit 'extern struct Custom custom' läßt man sich diese zuweisen und hat dann mit 'custom.<Registername>' Zugriff auf die einzelnen Hardware-Register.

Dazu übergeben Sie auch hier dem CWAIT-Befehl die 'User-Copper-Liste', in der der Befehl später stehen soll, sowie die X- und Y-Position des Elektronenstrahls, auf die gewartet werden soll. Allerdings ist hierbei zu beachten, daß nicht, wie zu erwarten gewesen wäre, zuerst die X- und dann die Y-Koordinate angegeben wird, sondern zuerst wird die Y- und dann die X-Koordinate angegeben. Wir möchten Sie deshalb bitten, besonders auf diesen Umstand zu achten, da er uns sehr viele Nerven gekostet hat, weil unsere eigenen Copper-Programme scheinbar nicht ordnungsgemäß liefen!

Mit 'CWAIT(&UserCopperListe,Y,X)' warten Sie also auf eine bestimmte Elektronenstrahlposition. Bitte beachten Sie, daß die Y-Position 263 und die X-Position 223 nicht überschreiten dürfen, wobei Sie weiterhin beachten sollten, daß die Y-Position relativ zur Oberkante des Viewports und die X-Koordinate relativ zur normalen Abtastposition angegeben wird. Das heißt, daß Sie bei der Positionierung in X-Richtung den 'Overscan' beachten müssen, denn der Elektronenstrahl tastet einen größeren Bereich als den tatsächlich sichtbaren ab. Ein guter Wert für die 'Positionierung' des Elektronenstrahls am linken Viewport-Rand liegt bei 'X=60±2' oder 'View.DxOffset/2±1'.

Und damit ein Copper-Programm auch ordnungsgemäß beendet werden kann, gibt es den CEND-Befehl. Er hat als einzigen Parameter die User-Copper-Liste, die 'beendet' werden soll. 'CEND (&UserCopperListe)' wartet dabei einfach, bis der Elektronenstrahl in der 10000. Zeile und 256. Spalte ist. Natürlich erreicht der Elektronenstrahl diese Position nie, so daß die User-Copper-Liste nicht weiter abgearbeitet wird.

Ist der Elektronenstrahl dann am unteren Ende des Bildschirms angelangt, wird er wieder an die oberste Zeile positioniert (Top Of Frame), und alle Copper-Listen des Viewports, und somit auch die User-Copper-Liste, werden von neuem abgearbeitet.

Insgesamt kann der Copper also als eine Art des Rasterzeilen-Interrupts, wie er z.B. vom C64 her bekannt ist, angesehen werden. Allerdings besteht hier die Möglichkeit, inmitten einer beliebigen Zeile z.B. die Farbe zu ändern (So etwas war beim C64 nicht möglich!). Bitte beachten Sie aber hierbei, daß nur eine maximale Auflösung von 8 Punkten pro CMOVE-Befehl möglich ist. Das heißt, daß bei normaler Auflösung maximal alle 8 Punkte ein CMOVE-Befehl ausgeführt werden kann. Oder anders gesagt: Zwischen zwei aufeinanderfolgenden CMOVEs vergehen 8 Punkte.

Diese werden nämlich dem CMOVE-Befehl als Parameter übergeben. Sie geben also, wie man leicht vermuten könnte, die absolute Adresse des zu beschreibenden Hardware-Registers an. Aber keine Angst. Da der Copper aber nur mit den Offsets der Register zu $DFF000 arbeitet, wird die absolute Adresse von CMOVE umgerechnet.

Damit der CMOVE-Befehl aber auch weiß, wo die User-Copper-Liste steht, muß man ihm vorher einen Zeiger auf diese Struktur übergeben.

Dann braucht man nur noch den 16-Bit-Wert (Word), der in das betreffende Hardware-Register hineingeschrieben werden soll, anzugeben.

Ein kompletter Aufruf des CMOVE-Befehls würde also so aussehen: CMOVE (UserCopperListe, Custom.<RegisterName>, Wert);

Leider müssen wir Sie hier auf eine kleine Einschränkung des CMOVE-Befehls aufmerksam machen. Generell dürfen Sie alle Hardware-Register mit einer Nummer größer als $20 (dskpt) ohne Einschränkung beschreiben. Die Register mit einer Nummer kleiner als $10 (adkconr) dürfen Sie auf gar keinen Fall mit dem Copper adressieren. Die Register dazwischen, also die Register $10 bis $20, dürfen Sie nur dann beschreiben, wenn Sie das Copper 'Danger Bit' des Register 'copcon' (Nr. $2E) gesetzt haben.

Wie Sie mit Hilfe des 68000er auf die Hardware-Register zugreifen, haben wir bei der Collision Detection der Sprites schon ausführlich erläutert, deshalb möchten wir Sie bitten, im Bedarfsfall dort nachzusehen.

Doch ist der CMOVE-Befehl nicht der einzige Copper-Befehl:

Mit dem CWAIT-Befehl können Sie nämlich auf eine spezielle Position des Elektronenstrahls warten. Die 'User-Copper-Liste' wird dann solange nicht weiter abgearbeitet, bis der Elektronenstrahl die festgelegte Position erreicht hat (Die Copper-Programmierung hat natürlich keinen Einfluß auf Ihr C-Programm, indem es dieses z.B. verzögern könnte). Sie können so z.B. auf den Anfang einer jeden Zeile warten und dann die Inhalte jedes einzelnen Farbregisters verändern. Sie können aber z.B. auch auf irgendeine beliebige Position mitten auf dem Monitor warten und ab dort mit der Darstellung eines eigens kontrollierten Sprites beginnen.

Zum Abschluß noch ein kleiner Tip: Wollen Sie Ihre Copper-Liste innerhalb Ihres Programms verändern, müssen Sie die selbst reservierte UCopList-Struktur löschen (am besten mit BltClear) und die gesamte Liste neu aufbauen.

```
/*********************************************************/
/*                      Copper.c                         */
/*                                                       */
/* Dieses Programm zeigt auf, wie man mit Hilfe des      */
/* Coppers auf die Hardware-Register des AMIGA zugreift  */
/*                                                       */
/* Compiled with: Lattice V3.10                          */
/*                                                       */
/* (c) Bruno Jennrich                                    */
/*********************************************************/

#include "exec/types.h"
#include "exec/nodes.h"
#include "exec/memory.h"
#include "graphics/gfx.h"
#include "graphics/gfxmacros.h"
#include "graphics/gfxbase.h"
#include "graphics/text.h"
#include "graphics/regions.h"
#include "graphics/clip.h"
#include "graphics/view.h"
#include "graphics/copper.h"
#include "graphics/gels.h"
#include "hardware/blit.h"
#include "hardware/custom.h"

#define WIDTH 320
#define HEIGHT 256
#define MODES 0

struct View View;
struct ViewPort ViewPort;
struct RasInfo RasInfo;
struct BitMap BitMap;
struct RastPort RastPort;

struct GfxBase *GfxBase;

struct View *oldview;

struct UCopList *UserCopperList;   /* unsere Copper Liste */
extern struct ColorMap *GetColorMap();
extern struct Custom custom;         /* Für Zugriff auf   */
                                     /* Hardware-Register */
```

Mit dem CWAIT-Befehl können Sie allerdings in einer Auflösung von 4 Punkten auf jede Elektronenstrahlposition warten. Allerdings vergehen bei einem nachfolgenden CMOVE wieder 8 Punkte!

Haben Sie also mit den oben erkärten Befehlen Ihre User-Copper-Liste aufgebaut, müssen Sie diese nur noch dem Viewport, in dem diese abgearbeitet werden soll, kenntlich machen. Dies geschieht mit 'ViewPort.UCopIns = UserCopperListe'. Bei Intuition-Screens müssen Sie dies natürlich wie folgt machen: 'Screen->ViewPort.UCopIns = &UserCopperList'.

Danach fahren Sie mit dem Öffnen des eigenen Bildschirms wie gewohnt fort, Sie rufen also MakeVPort und MrgCop auf.

Allerdings sieht die Sache bei Intuition-Screens etwas anders aus. Da nämlich OpenScreen dafür sorgt, daß die Copper-Listen für den Screen ordnungsgemäß berechnet werden, müssen Sie mit RemakeDisplay nachträglich dafür sorgen, daß die neue User-Copper-Liste mit in die globale View-Copper-Liste aufgenommen wird.

Den dynamisch reservierten Speicher der UCopList und die in Ihr enthaltenen Befehle, die jeweils 2 Words belegen, geben Sie dann mit FreeVPortCprList bzw. CloseScreen wieder frei. Sie brauchen also keine weiteren Maßnahmen zum Freigeben Ihrer User-Copper-Listen (z.B. FreeMem) zu treffen. Bitte beachten Sie aber, daß Sie die User-Copper-Liste als Pointer deklarieren und den benötigten Speicherplatz für die 'UCopList'-Struktur später innerhalb Ihres Programms reservieren. Denn FreeVPortCprList und CloseScreen 'befreien' auch den Speicher der UCopList-Struktur, nicht nur den durch die Befehle belegten.

Wenn Sie die 'UCopList' aber als 'normale' Struktur deklarieren würden, würde der Speicherplatz, den diese einnimmt, zweimal freigegeben werden: Das erste Mal durch 'FreeVPortCopLists' und das zweite Mal von Ihrem Programm, das bei Programmende dafür sorgt, daß der gesamte vom Programm belegte Speicher an das System zurückgegeben wird. Allerdings löst das zweimalige Freigeben eines Speicherbereichs, wie Sie vielleicht schon festgestellt haben, die heißgeliebten 'Guru Meditations' aus.

Also: Die eigene UCopList immer als Pointer deklarieren, dem der Speicherplatz nachträglich zugewiesen wird!

```
View.Modes = MODES;
View.ViewPort = &ViewPort;
ViewPort.DWidth = WIDTH;
ViewPort.DHeight = HEIGHT;
ViewPort.Modes = MODES;

RasInfo.RyOffset = 0;
RasInfo.RxOffset = 0;
RasInfo.Next = 0;

ViewPort.RasInfo = &RasInfo;
ViewPort.ColorMap = GetColorMap(16);

LoadRGB4(&ViewPort,&Colors,16);

InitBitMap(&BitMap,4,WIDTH,HEIGHT);
for (i=0; i<4; i++)
   {
      BitMap.Planes[i] = (PLANEPTR)
                         AllocRaster(WIDTH,HEIGHT);
      if (BitMap.Planes[i] == NULL)
         {
            printf("No BitMap - Space !!!\n");
            Exit(10);
         }
   }

InitRastPort(&RastPort);

RastPort.BitMap = &BitMap;
RasInfo.BitMap = &BitMap;

SetRast(&RastPort,0);

SetAPen(&RastPort,1);

for (i=0;i<17;i++)
   {
      Len = WIDTH/2-
        TextLength(&RastPort,Texts[i],strlen(Texts[i]))/2;

      Move(&RastPort,Len,i*9+63+RastPort.TxBaseline);

         /* Y-Koordinate sollte durch 9 teilbar sein ! */

      Text (&RastPort,Texts[i],strlen(Texts[i]));
   }
```

```
UWORD Colors[16] = {
                    0x000,0x0bbd,0x0f0,0xf00,
                    0x123,0x435,0x678,0x009,
                    0x123,0x435,0x678,0x009,
                    0x123,0x435,0x678,0x009
                   };
                                    /* Eigene ColorMap. */
                                    /* Farbreg. 1 wird  */
                                    /* vom Copper ver-  */
                                    /* ändert.          */
char *LeftMouse = (char *)0xbfe001;

char *Texts[17] = {"COPPER-Programmierung auf dem AMIGA",
                   "",
                   "Der SPECIAL-EFFECTS Prozessor in Action",
                   "",
                   "",
                   "Ein farbig unterlegter Text",
                   "(wie in vielen Spielen üblich)",
                   "und was dahinter steckt.",
                   "",
                   "",
                   "",
                   "Besonders bei sich bewegenden BOBS",
                   "erzeugt dieser Effekt STAUNEN !",
                   "",
                   "",
                   "",
                   "(MAUSTASTE)"};

/*********************************************************/
/* Es geht los !                                         */
/*********************************************************/
main()
{
   long i,Len;

   if ((GfxBase = (struct GfxBase *)
       OpenLibrary("graphics.library",0))==NULL)
      {
         printf (" No Graphics !!!\n");
         Exit(10);
      }

   oldview = GfxBase->ActiView;     /* Screen wie gewohnt */
   InitView(&View);                 /* aufbauen           */
   InitVPort(&ViewPort);
```

```
    FreeColorMap (ViewPort.ColorMap);

    FreeVPortCopLists(&ViewPort);
                /* UCopList wird automatisch geFREEt */

    FreeCprList(View.LOFCprList);
    FreeCprList(View.SHFCprList);
                        /* View-Copper Listen FREEn */

    CloseLibrary(GfxBase);
    return (0);
}
```

```
MakeVPort(&View,&ViewPort);     /* ViewPort darstellen. */
MrgCop(&View);                  /* So wird Unterschied  */
LoadView(&View);                /* erkennbar.           */

UserCopperList = (struct UCopList *)
          AllocMem(sizeof(struct UCopList),MEMF_CHIP);
                    /* Den nötigen Speicher für die   */
                    /* UCopList-Struktur reservieren  */

BltClear(UserCopperList,sizeof(struct UCopList),0);
                          /* UCopList-Struktur löschen */
   for (i=0; i<256; i+=9)
     {
        CWAIT (UserCopperList,i,View.DxOffset/2);
        CMOVE(UserCopperList,custom.color[1],0x0fff);
                                          /* weiß */

        CWAIT (UserCopperList,i+3,View.DxOffset/2);
        CMOVE(UserCopperList,custom.color[1],0x0bbd);
                                      /* hell lila */

        CWAIT (UserCopperList,i+5,View.DxOffset/2);
        CMOVE(UserCopperList,custom.color[1],0x088b);
                                          /* lila */

        CWAIT (UserCopperList,i+7,View.DxOffset/2);
        CMOVE(UserCopperList,custom.color[1],0x0558);
                                    /* dunkel lila */
     }
   CEND (UserCopperList);          /* UCopList beenden */

   Delay (100);                   /* 2 Sekunden warten */

   ViewPort.UCopIns = UserCopperList;
                 /* ViewPort und UCopList verbinden */

   MakeVPort(&View,&ViewPort);
                  /* Berechne ViewPort-Copper-Liste */
                  /* Beim Intuition Screen mit      */
                  /* RethinkDisplay()               */
   MrgCop(&View);
   LoadView(&View);

   while ((*LeftMouse & 0x40) == 0x40);
                             /* Warte auf Mausclick */
   LoadView(oldview);

   for (i=0; i<4; i++)
        FreeRaster(BitMap.Planes[i],WIDTH,HEIGHT);
```

Anhang A: Strukturen und Include-Files

In diesem Anhang finden Sie eine Liste aller symbolischen Konstanten ('#define xyz Numerischer Wert') und die Include-Files, in denen diese zu finden sind. Weiterhin sind alle benutzten Strukturen ('struct'), sowie alle benutzten C-Macros ('#define xyz()') hier aufgelistet. Weiter unten finden Sie dann noch die Funktion der Strukturen erklärt, wobei weiterhin die einzelnen Elemente der wichtigsten Strukturen beschrieben werden.

Deklaration	Include-File
struct AnimOb	"graphics/gels.h"
struct AnimComp	"graphics/gels.h"
struct AreaInfo	"graphics/rastport.h"
#define AREAOUTLINE	"graphics/rastport.h"
struct AvailFonts	"libraries/diskfont.h"
struct AvailFontsHeader	"libraries/diskfont.h"
struct BitMap	"graphics/gfx.h"
struct Bob	"graphics/gels.h"
#define BNDRYOFF()	"graphics/gfxmacros.h"
#define BOBISCOMP	"graphics/gels.h"
#define BOBSAWAY	"graphics/gels.h"
#define BOTTOMHIT	"graphics/collide.h"
#define CEND()	"graphics/gfxmacros.h"
#define CMOVE()	"graphics/gfxmacros.h"
struct ColorMap	"graphics/view.h"
#define COMPLEMENT	"graphics/rastport.h"
struct Custom	"hardware/custom.h"
#define CUSTOMBITMAP	"intuition/intuition.h"
#define CUSTOMSCREEN	"intuition/intuition.h"
#define CWAIT()	"graphics/gfxmacros.h"
struct DBuffPacket	"graphics/gels.h"
#define DUALPF	"graphics/view.h"
#define EXTRA_HALFBRITE	"graphics/view.h"
struct GelsInfo	"graphics/gels.h"
#define GELGONE	"graphics/gels.h"
struct GfxBase	"graphics/gfxbase.h"
#define HAM	"graphics/view.h"
#define HIRES	"graphics/view.h"
#define INVERSVID	"graphics/rastport.h"
struct IntuitionBase	"intuition/intuitionbase.h"
struct IntuiMessage	"intuition/intuition.h"

```
struct AnimComp
{
    WORD Flags;
            /* Mit dieser Variablen können Sie bestimmen, welcher Art die Animation
               sein soll. Setzen Sie das RINGTRIGGER-Flag, bedeutet das, daß Sie
               einen Ring von AnimComps aufgebaut haben, die durch
               'AnimComp.NextSeq' und 'AnimComp.PrevSeq' miteinander verbunden
               sind, und daß so verschiedene Sequenzen (z.B. verschiedene Flügel-
               stellungen eines Vogels) automatisch von Animate() dargestellt werden.

               Setzen Sie RINGTRIGGER nicht, werden Ihre Animationskomponenten
               nicht animiert. */

    WORD Timer;
            /* Diese Variable wird, wenn sie auf 0 heruntergezählt wurde, mit dem
               Wert aus TimeSet geladen und dann bei jedem Animate()-Aufruf herun-
               tergezählt. Ist 'Timer' dann wieder gleich 0, wird evtl., falls gewünscht
               (RINGTRIGGER), eine neue Sequenz dargestellt. */

    WORD TimeSet;
            /* Der Wert dieser Variablen wird in 'Time' geschrieben und dort bei jedem
               Animate()-Aufruf heruntergezählt. 'TimeSet' bestimmt, wie lange eine
               Sequenz aktiv sein soll, das heißt, beim wie vielten Animate()-Aufruf die
               nächste Sequenz dargestellt wird. */

    struct AnimComp *NextComp;
    struct AnimComp *PrevOb;
            /* Mit diesen beiden Variablen ist es möglich, innerhalb eines Animations-
               objektes (AnimOb) mehrere Komponenten (z.B. Arme, Beine und Kopf
               eines von Ihnen kreierten Männchens) miteinander zu verbinden, die
               dann, nach Animate(), dargestellt werden. Doch beachten Sie: Wollen
               Sie z.B. die Bewegung eines Armes beim Gehen mittels Sequenzen
               programmieren, dürfen Sie nicht mit PrevComp und NextComp diese
               AnimComp-Sequenzen miteinander verbinden. Dazu gibt es 'NextSeq'
               und 'PrevSeq'. */

    struct AnimComp *NextSeq;
    struct AnimComp *PrevSeq;
            /* Wenn Sie eine Animationskomponente immer wieder verändern wollen,
               bietet es sich an, die verschiedenen Darstellungen (Sequenzen) des
               Objektes (z.B ein Arm) zu definieren und die so entstandenen
               AnimComps mittels dieser beiden Zeiger zu verbinden. Animate() 'weiß'
               dann, daß dieser Arm zum Beispiel immer in anderen Stellungen, die
               Sie vorher programmiert haben, dargestellt werden soll, und tut dies
               auch. */
```

#define JAM1	"intuition/intuition.h"
#define JAM2	"intuition/intuition.h"
#define LACE	"graphics/view.h"
#define LEFTHIT	"graphics/collide.h"
struct NewScreen	"intuition/intuition.h"
struct NewWindow	"intuition/intuition.h"
#define MEMF_CHIP	"exec/memory.h"
#define MEMF_PUBLIC	"exec/memory.h"
struct Menu	"intuition/intuition.h"
struct MenuItem	"intuition/intuition.h"
#define OVERLAY	"graphics/gels.h"
#define PFBA	"graphics/view.h"
#define RASSIZE()	"graphics/gfx.h"
struct RasInfo	"graphics/view.h"
struct RastPort	"graphics/rastport.h"
#define RemBob()	"graphics/gels.h"
#define RINGTRIGGER	"graphics/gels.h"
#define RIGHTHIT	"graphics/collide.h"
struct Screen	"intuition/intuition.h"
#define SAVEBACK	"graphics/gels.h"
#define SAVEBOB	"graphics/gels.h"
#define SUSERFLAGS	"graphics/gels.h"
#define SELECTDOWN	"intuition/intuition.h"
#define SELECTUP	"intuition/intuition.h
#define SetOPen()	"graphics/gfxmacros.h"
#define SetDrPt()	"graphics/gfxmacros.h"
#define SetWrMsk()	"graphics/gfxmacros.h"
#define SetAfPt()	"graphics/gfxmacros.h"
struct SimpleSprite	"graphics/sprite.h"
#define SPRITE_ATTACHED	"graphics/sprite.h"
#define SPRITES	"graphics/view.h"
struct TextAttr	"graphics/text.h"
struct TextFont	"graphics/text.h"
struct TmpRas	"graphics/rastport.h"
#define TOPHIT	"graphics/collide.h"
struct UCopList	"graphics/copper.h"
typedef ULONG	"exec/types.h"
typedef UWORD	"exec/types.h"
struct View	"graphics/view.h"
struct ViewPort	"graphics/view.h"
#define VP_HIDE	"graphics/view.h"
struct VSprite	"graphics/gels.h"
#define WBENCHSCREEN	"intuition/intuition.h"
struct Window	"intuition/intuition.h"

Deshalb wird die Position gerettet, und erst nach sämtlichen Animate()-Aufrufen wird dann aus der alten Position und aus den folgenden Variablen die neue, aktuelle Position errechnet. */

WORD AnX, AnY;

/* In diesen beiden Variablen steht die aktuelle Position des AnimObs. Doch wird diese nicht, wie man meinen könnte, in Punkten bzw. Zeilen innerhalb des Rastports angegeben, sondern, aufgrund der Geschwindigkeits- und Beschleunigungsvariablen, in 64stel Punkten bzw. Linien.

Das heißt, daß man 'AnX = Breite/2 * 64' und 'AnY = Höhe/2 * 64' angeben muß, um ein AnimOb in die Mitte des Bildschirms zu bewegen. Nun stellt sich aber das Problem, wie man bei einem Wertebereich von ±32768 horizontale Positionen über 512 (= 32768/64) darstellen kann (vertikal wird man wohl kaum über 512 Linien kommen wollen)? Nun, dazu behilft man sich mit folgendem kleinen Trick: Die Variable 'XTrans' (und YTrans) in der AnimComp-Struktur gibt die Position der AnimComps relativ zum übergeordneten AnimOb an. Wenn man 'XTrans' einfach mit 128*64 initialisiert, kann man das Objekt auch im Hi-Res-Modus über den ganzen Bilschirm bewegen (512+128 = 640). Für horizontale Positionen kleiner 128 gibt man dann einfach negative Werte in AnX an. */

WORD YVel, XVel;

/* Diese beiden Variablen enthalten die Geschwindigkeit des AnimObs. Die Werte aus 'XVel' und 'YVel' werden nach jedem Animate()-Aufruf zu 'Anx' und 'AnY' addiert. Da aber Animate() normalerweise mehrmals pro Sekunde aufgerufen wird, könnte es passieren, daß Ihr Objekt in 'einem Affenzahn' über den Bilschirm 'fegt'. Um dem vorzubeugen, werden Geschwindigkeit (Velocity) und Beschleunigung (Acceleration) in vierundsechzigstel Animate()-Aufrufen angegeben (deshalb auch die eigentümliche Angabe der Werte für 'AnX' und 'AnY'). Das heißt, daß bei einem XVel-Wert von 1, 64 Animate()-Aufrufe nötig sind, um das Objekt z.B. um einen Punkt nach links zu bewegen. */

WORD YAccel, XAccel;

/* Diese beiden Variablen bestimmen die Beschleunigung des AnimObs. Auch hier wird der Wert in 64stel angegeben. Nur wird der Wert aus 'XAccel' und 'YAccel' zu 'XVel' bzw. 'YVel' addiert. */

WORD RingYTrans, RingXTrans;

/* Diese beiden Variablen geben auch die Geschwindigkeit des AnimObs an. Diese wird aber direkt zu AnX und AnY addiert. Beschleunigungen existieren hierfür nicht. */

WORD (*AnimORoutine)();

/* Die Routine, deren Adresse hier angegeben wird, wird bei jedem Animate()-Aufruf einmal angesprungen. Der Routine wird die gerade aktuelle AnimOb-Struktur übergeben, so daß Sie z.B. die aktuelle Posi-

```
    WORD (*AnimCRoutine)();
            /* Dieser Zeiger zeigt auf eine von Ihnen definierte Funktion (oder auf
               NULL), die bei jedem Aufruf von Animate(), wenn diese Komponente
               gerade dargestellt wird, aufgerufen wird. Dieser Routine wird dann die
               gerade bearbeitete AnimComp-Struktur übergeben. */

    WORD YTrans, XTrans;
            /* In diesen beiden Variablen steht die Position des AnimComps, relativ
               zum übergeordneten AnimOb. (Bitte beachten Sie die eigenartige 'Fest-
               komma-Arithmetik' dieser Variablen). */

    struct AnimOb *HeadOb;
            /* Dieser Zeiger zeigt auf das übergeordnete AnimOb, von dem dieses
               AnimComp ein Teil ist. */

    struct Bob *AnimBob;
            /* Natürlich muß Animate() auch wissen, was überhaupt dargestellt
               werden soll. Deshalb enthält die AnimComp-Struktur einen Zeiger auf
               das Bob, das mit diesem AnimComp assoziiert werden soll. (Bitte be-
               achten Sie, daß das Bob weiterin eine eigene VSprite-Struktur 'haben'
               muß). */
}
```

Das AnimComp oder die Animationskomponente stellt die Verbindung zwischen Bob und Animationsobjekt (AnimOb) her. Besonders bei der Sequenz-Animation (RINGTRIGGER) ist sie von großer Wichtigkeit, denn sie verbindet einzelne Komponenten in einem Ring miteinander und bestimmt, wie lange jede einzelne Sequenz aktiv sein soll.

---

struct AnimOb

```
{
    struct AnimOb *NextOb, *PrevOb;
            /* Mit diesen Variablen ist es möglich, mehrere Animationsobjekte mitein-
               ander zu verbinden, die dann mit Animate() animiert und nach
               SortGList(), DrawGList() etc. alle auf einmal auf den Bildschirm 'gemalt'
               werden (z.B. mehrere Männchen). */

    LONG Clock;
            /* Diese Variable enthält die Anzahl der Animate()-Aufrufe, die dieses
               AnimOb schon 'erlebt' hat. */

    WORD AnOldY, AnOldX;
            /* Diese Variablen enthalten die alte Position des Animationsobjekes. Doch
               warum speichert man die Position des Objekts? Nun, die alte Position
               des AnimObs wird deshalb gespeichert, da die tatsächliche Position
               (AnX, AnY) erst dann verändert wird, wenn der Timer des gerade aktu-
               ellen AnimComps heruntergezählt wurde. Der Benutzer könnte aber
               zwischendurch diese Position ändern, was aber zur Folge hätte, daß
               bestimmte Komponenten nicht am richtigen Platz dargestellt werden.
```

'AvailFontsHeader' in den der AvailFonts()-Routine angegebenen Speicherbereich geschrieben wurden.

---

struct BitMap

```
{
   UWORD BytePerRow;
            /* Diese Variable enthält die Anzahl der Bytes, die für eine Bitmap-Zeile
               benötigt werden ('BytesPerRow = Breite/8') */

   UWORD Rows;
            /* In dieser Variablen steht die Anzahl der Linien der Bitmap ('Höhe'). */

   UBYTE Flags;
            /* Diese Variable wird nur vom System benutzt. */

   UBYTE Depth;
            /* In dieser Variablen steht die Anzahl der Bitplanes der Bitmap. Bitte
               beachten Sie, daß die 'Tiefe' die Anzahl der Darstellbaren Farben beein-
               flußt (Farben = 2^Anz_BitPlanes). */

   UWORD pad;
            /* Dies ist ein Füll-WORD, damit die folgenden Zeiger mit einer LONG-
               WORD-Adresse beginnen. */

   PLANEPTR Planes[8];
            /* Diese 8 Zeiger enthalten die Adressen der einzelnen Bitplanes der
               Bitmap. Obwohl im Moment nur maximal 6 der 8 Zeiger benutzt
               werden, hat man hier schon für Erweiterungsmöglichkeiten gesorgt. */
}
```

Die BitMap-Struktur enthält die Adressen auf die einzelnen Speicherbereiche, in denen die Grafiken abgespeichert werden. Außerdem sind auch Informationen über die Höhe, Breite und Tiefe der BitMap enthalten.

---

struct Bob

```
{
   WORD Flags;
            /* Mit dieser Variablen bestimmt der Benutzer, wie das Bob vom System
               behandelt werden soll. Das Flag SAVEBOB z.B. bestimmt, daß das Bob,
               einmal gezeichnet, nicht wieder vom Bildschirm bzw. vom Rastport ent-
               fernt wird (Pinselfunktion). Mit BOBISCOMP bestimmen Sie, daß das
               Bob Teil einer Animationskomponente ist. Bitte beachten Sie, daß dann
               auch der Zeiger 'BobComp' auf die zugehörige AnimComp-Struktur
               zeigen muß.

               Jedoch ist es so, daß nicht nur der Benutzer Bob-Flags setzen kann,
               sondern daß auch das System sogenannte Status-Flags setzt, die Aus-
```

```
                tion des gerade dargestellten AnimObs kontrollieren und gegebenen-
                falls darauf reagieren können. */

    struct AnimComp *HeadComp;
                /* Dieser Zeiger zeigt auf die erste Animationskomponente des AnimObs.
                */

    AUserStuff AUserExt;
                /* Auch hier kann der Benutzer eigene Anhängsel an die Struktur kreieren
                (s. VSprite.VUserStuff). */
}
```

Die AnimOb-Struktur enthält ein Animationsobjekt in seiner Gesamtheit. Diese Objekt wird mit AddAnimOb() dem System zugänglich gemacht und kann mit Animate() animiert werden.

---

struct AvailFonts

```
{
    UWORD af_Type;
                /* In dieser Variablen 'steht drin', ob durch die unten angegebene TextAttr-
                Struktur ein Font im Speicher (AFF_MEMORY) oder ein Font auf Dis-
                kette im "SYS:fonts" Directory (AFF_DISK) beschrieben wird. Dies ist
                deshalb wichtig, damit man die richtige Routine zum Öffnen eines
                Fonts, OpenFont() oder OpenDiskFont(), anwenden kann. */

    struct TextAttr af_Attr;
                /* Dies ist die TextAttr-Struktur, die von AvailFonts() zurückgegeben wird.
                */
}
```

Diese Struktur wird nur von der Routine AvailFonts() benutzt (und von Ihnen, natürlich). Sie wird nach dem AvailFontsHeader in den Speicherbereich geschrieben, der AvailFonts() zur Verfügung gestellt wurde.

---

struct AreaInfo

Diese Struktur wird für die 'Area...'-Befehle benötigt. Sie muß mit InitArea initialisiert werden, und dient dann dazu, die Koordinaten der Polygon-Stützpunkte aufzunehmen.

---

struct AvailFontsHeader

Diese Struktur wird nur durch die Routine AvailFonts() angelegt und enthält nur eine Variable: 'afh_NumEntries'. Diese Variable enthält die Anzahl der AvailFonts-Strukturen, die nach dem

```
    struct DBufPacket *DBuffer;
                /* Wenn Sie Bobs in einer doppelt gepufferten Bitmap darstellen wollen,
                   wobei in beiden Bitmaps der Hintergrund gerettet wird, müssen Sie
                   diesen Zeiger auf das 'Double Buffer Packet' initialisieren, damit die
                   GEL-Software die Bobs ordnungsgemäß und ohne großen Aufwand für
                   Sie in beiden Bitmaps darstellen kann. (Chip Memory!) */

    BUserStuff BUserExt;
                /* Hier kann der Benutzer seine eigene 'Extensions' (Anhängsel) an die
                   Bob-Struktur anhängen. Dazu definiert er in seinem Programm einfach
                   mit '#define BUserStuff' die Art dieses Anhängsels und kann auf diese
                   dann z.B. in den 'AnimCRoutinen' oder 'AnimORoutinen' oder in den
                   Kollisionsroutinen zugreifen. Definiert der Benutzer diese nicht um, so
                   wird automatisch 'BUserStuff' als 'SHORT'-Variable definiert. */
}
```

Die Bob-Struktur beschreibt das Bob (Blitter Object). Bobs können beliebig groß sein und so viele Farben enthalten, wie der Rastport, in dem sie erscheinen sollen, zur Verfügung stellt.

---

struct ColorMap

Diese Struktur dient dazu, die Farben eines Viewports in sich aufzunehmen, und kann mit LoadRGB4(), GetRGB4() und SetRGB4() nachträglich bearbeitet werden.

---

struct Custom

Diese Struktur gibt Ihnen ein Abbild der Hardware-Register, damit Sie auch von 'C' aus auf elegante Art und Weise auf diese zugreifen können. Der Anhang über die Hardware-Register beschreibt alle in dieser Struktur enthaltenen 'Symbol-Register' und wie Sie auf diese zugreifen können. (Chip-Memory!)

---

struct DBuffPacket

Diese Struktur ist für die Benutzung von Bobs in doppelt gepufferten Bitmaps gedacht. Dort muß nämlich der Hintergrund, den ein Bob überschreibt, für beide Bitmaps gerettet werden (wenn Sie das wollen). Setzen Sie aber nur das SAVEBACK-Flag in der VSprite-Struktur des Bobs, dann wird der Hintergrund einer Bitmap gerettet. Installieren Sie aber für jedes Bob ein DBufPacket (Bob.DBuffer = &DBufPacket), dann wird der Hintergrund auch noch für die zweite Bitmap gerettet. Allerdings müssen entweder alle oder keine Bobs einer GEL-Liste dieses DBuffPacket enthalten.

kunft über den augenblicklichen Zustand des Bobs geben. Das Flag BOBNIX z.B. sagt aus, daß das Bob vollkommen vom Rastport verschwunden ist, der Hintergrund, falls gerettet, wieder zurückgeschrieben wurde, und daß das Bob aus der GEL-Liste entfernt wurde. */

```
WORD *SaveBuffer;
```
/* Dieser Zeiger zeigt auf einen von Ihnen angelegten Speicherplatz, in dem der Hintergrund abgespeichert wird, wenn ein Bob gezeichnet wird. Da Bobs nämlich in die Bitmap hineingeschrieben werden, wird normalerweise der Hintergrund, auf dem das Bob erscheint, zerstört. Setzen Sie aber das SAVEBACK-Flag in der VSPRITE-Struktur des Bobs, dann wird in diesem Speicher der Hintergrund gerettet. Der Speicher muß dafür mindestens so breit und so hoch sein wie das Bob, das gezeichnet werden soll, wobei zu beachten ist, daß für jede zu beschreibende Bitplane des Rastports (s. PlanePick und PlaneOnOff in der VSprite-Struktur) solch ein Speicher im Chip-Memory (untere 512 KByte) reserviert werden muß (der Hintergrund wird dann in diesem Speicher gerettet). */

```
WORD *ImageShadow;
```
/* Wie Sie ja wissen, definieren Sie ein Bob Bitplane für Bitplane, bzw. bestimmen mit 'PlanePick' und 'PlaneOnOff', welche Bitplanes wie beim Zeichnen des Bobs betroffen werden. 'ImageShadow' zeigt nun auf einen Speicherbereich, der genügend groß ist, um eine 'Bitplane' des Bobs aufnehmen zu können. Doch was enthält 'ImageShadow'? Nun, dort stehen alle gesetzten Punkte der einzelnen Bitplane-Definitionen für das Bob, oder anders gesagt: Alle Bob-Planes werden geORt und in 'ImageShadow' abgespeichert. (Chip-Memory!) */

```
struct Bob *Before;
struct Bob *After;
```
/* Mit diesen beiden Zeigern bestimmen Sie, in welcher Reihenfolge die Bobs gezeichnet werden sollen. Sie können somit den GEL-Routinen eine Bob-Zeichen-Reihenfolge aufzwingen. Allerdings muß dies nach AddBob() geschehen, da diese Funktion diese Zeiger löscht.*/

```
struct VSprite *BobVSprite;
```
/* Wie Sie wissen, braucht jedes Bob seine VSprite-Struktur, wie der Kaffee die Milch oder die Suppe das Salz, denn die Bob-Struktur enthält z.B. keine Variablen zur Positionierung, und außerdem besteht die GEL-Liste nur aus VSprite-Strukturen, und irgendwie muß ein Bob ja in diese Liste 'hinein'. Damit das alles funktioniert, zeigt dieser Zeiger also auf die VSprite-Struktur des Bobs (jedes Bob braucht eine eigene!) */

```
struct AnimComp *BobComp;
```
/* Dieser Zeiger zeigt auf die AnimComp-Struktur, zu der das Bob gehört. Dieser Zeiger muß allerdings nur dann gesetzt werden, wenn Sie das Bob-Flag BOBISCOMP gesetzt haben. */

```
    UBYTE Flags;
            /* Diese Variable wird nur vom System benutzt */

    struct VSprite *gelHead, *gelTail;
            /* Wie Sie wissen, werden die GELs (VSprites und Bobs) in einer sogenannten GEL-Liste organisiert. Diese beiden Zeiger geben nun den Anfang bzw. das Ende diese Liste an (sie müssen mit InitGels() initialisiert werden). */

    WORD *nextLine;
            /* Dieser Zeiger zeigt auf einen Speicherbereich von 8 Words und enthält Informationen über die hächste vertikale Position, an der ein Hardware-Sprite wieder - mittels der VSprite-Software - dargestellt werden soll. */

    WORD **lastColor;
            /* Um den Copper bei der Darstellung der VSprites ein wenig zu entlasten, wird in diesem Array von 8 Zeigern immer die Adresse der letzten Farbdefinition (VSprite.SprColors) der Hardware-Sprites abgespeichert. Wird nun festgestellt, daß diese 'alte' Farb-Adresse und die Adresse der Farbtabelle des neu darzustellenden VSprites identisch sind, wird darauf verzichtet, eine Farbänderung mittels des Coppers zu generieren, da ja diese Farben schon eingestellt sind. Zeigen alle 8 Zeiger auf die gleiche VSprite-Farbtabelle, können 8 anstatt 4 VSprites auf einer Rasterzeile dargestellt wrden. */

    struct collTable *collHandler;
            /* Hier werden die Adressen der verschiedenen Kollisionsroutinen abgespeichert, die mit SetCollision() festgelegt werden können, und je nach dem, welche GELs mit welchen Kollisionsmasken (MeMask und HitMask) zusammenstoßen, aufgerufen werden. */

    short leftmost, rightmost, topmost, bottommost;
            /* Mit diesen vier Variablen bestimmen Sie die Größe des Rechtecks, in dem sich die GELs aufhalten können, ohne daß eine 'Border-Collision', also eine Kollision mit dem Rand, gemeldet wird. Überschreitet ein GEL diese Grenzen, dann wird die Kollisionsroutine 0 aufgerufen. Allerdings muß dazu in dem angegebenen GEL das Bit 0 der 'HitMask' gesetzt sein. */

    APTR firstBlissObj,lastBlissObj;
            /* Diese beiden Zeiger werden einzig und allein vom Betriebssystem benutzt. */
}
```

Die GelsInfo-Struktur enthält wichtige Vaiablen und Zeiger für die GELs (Grafische ELemente) und muß, bevor die GEL-Routinen (AddBob(), AddVSprite(), Animate(), AddAnimOb()) benutzt werden können, mit InitGels() initialisiert werden. Diese Struktur muß nach

Dazu müssen Sie jedoch auch der DBufPacket-Struktur die Adresse eines weiteren Speicherbereichs im Chip-Memory übergeben, der die gleiche Größe wie der 'Bob.SaveBuffer' hat (DBufPacket.BufBuffer = &Speicher). Um die anderen Variablen des 'DBufPackets' brauchen Sie sich nicht zu kümmern (Das übernimmt die GEL-Software).

---

struct GfxBase

'GfxBase' ist Ihr Zeiger auf die Grafikbefehls-Bibliothek, den Sie mit 'GfxBase = OpenLibrary ("graphics.library", VERSIONS_NUMMER)' initialisieren. Danach haben Sie Zugriff auf alle Grafikbefehle, die Ihnen der Amiga zur Verfügung stellt.

Weiterhin enthält 'GfxBase' noch einen Zeiger auf den gerade aktuellen View. Wenn Ihre Programme einen eigenen View unterstützen, also nicht auf Intuition zurückgreifen, sollten Sie diesen GfxBase-Zeiger (OldView = GfxBase->ActiView) retten, damit Sie nach dem Ende Ihres Programms wieder den Intuition-View und somit den Workbench-Screen darstellen können.

Eine weitere Variable der GfxBase-Struktur - 'GfxBase-> SpriteReserved' - gibt Auskunft darüber, welche Hardware-Sprites gerade zur alleinigen Benutzung herangezogen werden.

Außerdem sind noch ein paar andere Zeiger in 'GfxBase' enthalten - z.B. für die Systemfontliste etc. -, die allerdings fast ausschließlich vom System benutzt werden oder auf einfache Weise über die Grafikbefehle beeinflußt werden können.

---

struct GelsInfo

```
{
    BYTE sprRsrvd;
            /* Diese Variable gibt Auskunft über die Sprites, die dem 'VSprite-Generator' zur Verfügung stehen. Wenn Sie in Ihren Programmen keine Hardware-Sprites verwenden wollen, brauchen Sie nur alle Bits in 'sprRsrvd' zu setzen, und schon weiß Ihr Amiga, daß er alle Hardware-Sprites für die VSprites benutzen kann. Wenn Sie aber neben den VSprites auch Hardware-Sprites benutzen wollen, müssen Sie, wenn Sie sicher gehen wollen, daß Ihr Hardware-Sprite auch korrekt dargestellt wird, dafür sorgen, daß es nicht für die VSprites benutzt wird. Dazu löschen Sie einfach das zugehörige Bit in 'sprRsvrd'. (Bit 0 für Sprite 0, Bit 1 für Sprite 1 usw.) */
```

```
struct Gadget *FirstGadget;
        /* Dieser Zeiger zeigt auf das erste von Ihnen erstellte Gadget. Gadget?
           Was sind Gadgets? Nun, Gadgets sind die kleinen 'Dinger', die Sie z.B.
           links oben an den System-Windows erkennen können, und bei deren
           Anklicken das Window im Hintergrund verschwindet. */

struct Image *CheckMark;
        /* Dieser Zeiger zeigt auf die Image-Struktur, die das Aussehen des Häk-
           chens bestimmt, das zur Kennzeichnung 'statischer' Menüpunkte
           benutzt wird. Wenn Sie hier NULL angeben, wird das Default-Häckchen
           verwendet. */

UBYTE *Title;
        /* Wenn Sie mehrere Windows darstellen wollen, erweist es sich als ange-
           nehm, diesen Namen zu geben. Dieser Name wird dann in der obersten
           Zeile des Windows als Überschrift dargestellt. 'Title' ist dabei der Zeiger
           auf das erste Zeichen eines Strings. (Das System-Window 'heißt' z.B.
           "AmigaDOS"). */

struct Screen *Screen;
        /* Damit ein Window überhaupt existieren kann, braucht es einen Screen.
           Denn irgendwo muß das Window ja auch dargestellt werden können.
           Dieser Zeiger zeigt dafür auf den vorher geöffneten Screen, in dem das
           Window später erscheint. */

struct BitMap *BitMap;
        /* Wenn Sie in der Flags-Variablen das 'SUPER_BITMAP'-Flag angegeben
           haben, muß dieser Zeiger auf Ihre eigens angelegte Bitmap zeigen.
           Dann haben Sie nämlich die Möglichkeit, eine Bitmap mit bis zu
           1024x1024 Punkten darzustellen, von der aber auf dem Monitor immer
           nur ein Ausschnitt erscheint, der so groß wie das Window ist. */

SHORT MinWidth, MinHeight;
SHORT MaxWidth, MaxHeight;
        /* Wenn Sie in der Flags-Variablen das WINDOWSIZING-Flag (und dazu
           das SIZEBRIGHT (Size Border Right) oder SIZEBBOTTOM (Size Border
           Bottom) gesetzt haben, dann erscheint in der rechten unteren Ecke das
           'Sizing'-Gadget, das es Ihnen erlaubt, die Größe des Windows zu verän-
           dern. Die Variablen MinHeight, MinWidth, MaxHeight und MaxWidth
           geben dabei die Grenzen an, in denen das Window 'größenverschieb-
           lich' ist. */

USHORT Type;
        /* Mit dieser Variablen legen Sie fest, ob das Window in der Workbench-
           Umgebung erscheint (WBENCHSCREEN), wobei Sie den Zeiger auf die
           Screen-Struktur auf NULL setzen müssen, oder ob das Window in einem
           eigenen Screen erscheinen soll (CUSTOMSCREEN). Hier müssen Sie
```

der Initialisierung dem Rastport zugewiesen werden (RastPort.GelsInfo = &GelsInfo).

---

struct IntuiMessage

Mit Hilfe dieser Struktur können Sie eine Nachricht, die von Intuition bzw. von einem Window, gesandt wurde, empfangen und weiter untersuchen. So können Sie mit deren Hilfe z.B. feststellen, ob ein Menüpunkt angeklickt wurde, ob die Maus bewegt oder der Mausknopf gedrückt wurde.

---

struct IntuitionBase

Wie jede Library hat auch die Intuition-Library einen 'BasePointer'. Dieser 'BasePointer' dient - wie bei allen Libraries - zur Bestimmung der Einsprungadressen der Library-Befehle.

---

struct NewWindow

```
{
    SHORT LeftEdge, TopEdge;
    SHORT Width, Height;
            /* Diese vier Variablen bestimmen die Position eines Windows (LeftEdge
               (X-Koordinate), TopEdge (Y-Koordinate)) sowie dessen Breite (Width)
               und Höhe (Height). */

    UBYTE DetailPen, BlockPen;
            /* Mit diesen beiden Variablen bestimmen Sie, in welcher Farbe der obere
               Balken (BlockPen) und in welcher Farbe die in ihm enthaltene Über-
               schrift (Title) erscheinen soll. Die Werte, die Sie hier angeben, sind die-
               selben wie bei SetAPen() etc. (die Nummer des Farbregisters). */

    ULONG IDCMPFlags;
            /* Die IDCMPFlags (Intuition Direct Communication Message Ports)
               bestimmen die Art der Kommunikation zwischen Benutzer und Intuition.
               Über diese Variable bestimmen Sie, welche Nachrichten von Intuition an
               Ihr Programm gesendet werden dürfen. So können Sie z.B. festlegen,
               daß nur Mausklicke (MOUSEBUTTON) oder Tastendrücke (RAWKEY /
               VANILLAKEY) gesendet werden. Eine genaue Aufschlüsselung der
               IDCMPs würde aber an dieser Stelle zu weit führen. */

    ULONG Flags;
            /* Mit der Flags-Variablen können Sie das Window näher beschreiben. So
               können Sie z.B. mit 'BORDERLESS' festlegen, daß um das Window kein
               Rahmen gezeichnet wird. Mit 'ACTIVATE' können Sie weiterhin bestim-
               men, daß das Window sofort nach dem Öffnen aktiv, das heißt, das
               aktuelle Ein/Ausgabe-Fenster ist. Doch auch hier gilt: Eine Beschrei-
               bung aller Möglichkeiten würde an dieser Stelle zu weit führen. */
```

```
    struct Gadget *Gadgets;
            /* Dieser Zeiger wird zur Zeit noch nicht benutzt und sollte daher, um Auf-
               wärtskompatibilität zu garantieren, auf 0 gesetzt werden. */

    struct BitMap *CustomBitMap;
            /* Dieser Zeiger zeigt auf die eigene Bitmap, falls in 'Type' das
               CUSTOMBITMAP-Flag gesetzt wurde. */
}
```

Die NewScreen-Struktur dient zur Beschreibung eines Screens. Dieser wird mit 'Screen = OpenScreen (&NewScreen)' geöffnet.

---

struct RastPort

```
{
    struct Layer *Layer;
            /* Dieser Zeiger zeigt auf die Layer-Struktur des Rastports. Doch was sind
               Layer? Layer sind Datenstrukturen, die helfen, die Windows zu steuern,
               also dafür sorgen, daß ein Window nicht von einem anderen so ohne
               weiteres überschrieben wird etc. */

    struct BitMap *BitMap;
            /* Das ist der Zeiger auf die Bitmap, die der Rastport 'vertritt'. Er muß bei
               nicht Intuition-Screens später initialisiert werden. */

    USHORT *AreaPtrn;
            /* Dieser Zeiger zeigt auf das Füllmuster des Rastports. Normalerweise
               werden Flächen ohne spezielles Muster gefüllt, aber mit dem Macro
               SetAfPt() kann man dieses Füllmuster ändern. */

    struct TmpRas *TmpRas;
            /* Dieser Zeiger zeigt auf einen zusätzlichen Speicherbereich, der für die
               Füllbefehle Area...() und Flood gebraucht wird. Dieser Speicher muß
               mindestens so groß sein, um den zu füllenden Bereich vollständig auf-
               nehmen zu können. */

    struct AreaInfo *AreaInfo;
            /* Dieser Zeiger wird nur von den Area...()-Befehlen gebraucht, denn
               irgendwo müssen ja die Punkte des Polygons, das mit AreaDraw() und
               AreaMove() festgelegt wird, abgespeichert sein. Dazu wird mit InitArea()
               eine AreaInfo-Struktur initialisiert bzw. genügend Speicherplatz an sie
               zugewiesen (pro Koordinate 5 Bytes) und dann in den Rastport, in dem
               geAREAt werden soll, eingebunden (RastPort.AreaInfo = &AreaInfo;). */

    struct GelsInfo *GelsInfo;
            /* Um in einem Rastport VSprites und Bobs darstellen zu können, wird
               diese Struktur benötigt. Sie enthält die VSprites aller grafischen Ele-
               mente verkettet in einer Liste (GEL). Diese Liste kann mit SortGList()
               und DrawGList() sortiert bzw. zum Zeichnen vorbereitet werden. */
```

```
                der NewWindow-Struktur allerdings die Adresse des Screens, in dem
                das Window erscheinen soll, mitteilen. */
}
```

Mit dieser Struktur können Sie ein Window beschreiben. Nachdem Sie alle Variablen und Zeiger der NewWindow-Struktur angegeben haben, rufen Sie 'Window = OpenWindow (&NewWindow)' auf und können Ihr Window z.B. zur grafischen Ausgabe benutzen.

---

struct NewScreen

```
{
    SHORT LeftEdge, TopEdge, Width, Height, Depth;
                /* Mit diesen Variablen können Sie die Position, Größe und Tiefe (Anzahl
                der Bitplanes) des Screens festlegen. Dabei müssen Sie aber beachten,
                daß ein Screen (noch) nicht in der horizontalen, also in X-Richtung, ver-
                schieblich ist. Werte für LeftEdge, die ungleich 0 sind, haben deshalb
                keine andere Wirkung als der Wert 0 selbst. */

    UBYTE DetailPen, BlockPen;
                /* Auch hier wird, wie beim Window, die Farbe des Textes (DetailPen) und
                die Farbe des Balkens (BlockPen) in der obersten Zeile des Screens
                festgelegt. */

    USHORT ViewModes;
                /* In dieser Variablen geben Sie den gewünschten Darstellungsmodus, in
                dem der Screen dargestellt werden soll, an. */

    USHORT Type;
                /* Diese Variable bestimmt den Typ des Screens. Dabei muß immer
                'CUSTOMSCREEN' angegeben werden. Wenn Sie allerdings eine
                eigene Bitmap benutzen wollen, die Sie selbst initialisiert haben, dann
                müssen Sie außerdem noch das 'CUSTOMBITMAP' Flag setzen. */

    strut TextAttr *Font;
                /* Wenn Sie einen anderen Zeichensatz als den Standard-Font in diesem
                Screen verwenden wollen, können Sie dies schon jetzt festlegen. Dazu
                definieren Sie einfach eine TextAttr-Struktur, die den neuen Font
                beschreibt, und übergeben hier die Adresse dieser Struktur. Dieser neue
                Font wird dann in diesem Screen und in allen in ihm enthaltenenen
                Windows benutzt. Wenn Sie den 'Default'-Font verwenden wollen
                ("topaz"), dann brauchen Sie hier nur 0 zu übergeben. */

    UBYTE *DefaultTitle;
                /* Dieser Zeiger zeigt, ähnlich wie bei den Windows, auf den String, der in
                der obersten Zeile des Screens ausgegeben werden soll (Der Work-
                bench-Screen hat den Titel "Workbench Screen"). Wollen Sie keinen
                Titel in Ihrem Screen, übergeben Sie hier einfach 0. */
```

```
UBYTE minterms[8];
        /* Wozu diese und die nächsten beiden Variablen benötigt werden, konnten
           wir nicht in Erfahrung bringen, da uns erstens die uns zur Verfügung
           stehende Literatur 'im Stich ließ' und zweitens eine Änderung dieser
           Parameter keine Resultate zeigte. */

SHORT PenWidth;     /* Siehe minterms */
SHORT PenHeight;    /* Siehe minterms */

struct TextFont *Font;
        /* Dieser Zeiger zeigt auf den gerade benutzten Font bzw. auf dessen
           TextFont-Struktur. Ist Ihnen der 'Normal-Font' ("topaz.font") zu 'langweilig',
           können Sie diesen mit OpenFont() und SetFont() ändern. */

UBYTE AlgoStyle;
        /* Diese Variable enthält die gerade aktuelle, mit SetSoftStyle() eingestellte,
           algorithmisch generierte Schriftart. */

UBYTE TxFlags;
        /* Diese Variable enthält die Flags, die den Font des Rastports näher
           kennzeichnen. So kann man mit diesen z.B. feststellen, ob der Font
           Proportionalschrift unterstützt (TxFlags = = FPF_PROPORTIONAL), oder
           ob er z.B. aus dem RAM (FPF_ROMFONT) oder von Diskette
           (FPF_DISKFONT) 'geladen' wurde. */

UWORD TxHeight;
        /* Diese Variable gibt die Höhe der Zeichen im gerade aktuellen Rastport-
           Font an. */

UWORD TxWidth;
        /* Hier steht die nominelle Breite der einzelnen Zeichen. */

UWORD TxBaseline;
        /* Diese Variable enthält die Position der Baseline des Fonts. Diese Baseline
           wird bei der Schriftart FPF_UNDERLINED (Unterstrichen) bei jedem
           Zeichen voll durchgezogen. Was aber viel wichtiger ist, ist die Tatsache,
           daß diese Baseline auch zur Positionierung von Zeichen, die mit Text()
           ausgegeben werden, herangezogen wird. Denn Strings werden vertikal
           nicht so positioniert, daß die oberste Linie der Zeichen mit der Y-Position,
           die mit Move() eingestellt wurde, übereinstimmt, sondern so, daß
           die Y-Position des Grafik-Cursors und diese Baseline übereinstimmen.
           */

WORD TxSpacing;
        /* Diese Variable gibt an, wie viele Punkte jedes einzelne Zeichen einnimmt,
           das heißt, wie breit jedes Zeichen ist, wenn es einzeln, also nicht
           im String, ausgegeben wird. */
```

```
UBYTE Mask;
        /* In dieser Variablen steht, welche Bitplanes des Rastports bzw. seiner
           Bitmap von den Grafik-Operationen betroffen werden. Normalerweise
           steht hier 0xff, was bedeutet, daß jede Bitplane betroffen wird (jedes
           gesetzte Bit steht für eine 'angeschaltete' Bitplane). Mit SetWrMsk
           (&RastPort, Maske) können Sie dies aber nach Belieben ändern.*/

BYTE FgPen;
        /* Diese Variable enthält die Nummer des Farbregisters, das die Farbe des
           APens bestimmt, oder anders gesagt: FgPen = = APen (Der APen
           wurde früher ForegroundPen genannt). */

BYTE BgPen;
        /* BgPen = = BPen. (BPen hieß mal BackgroundPen). */

BYTE AOlPen;
        /* AOlPen (AreaoutlinePen) = = OPen */

BYTE DrawMode;
        /* Diese Variable enthält den aktuellen Zeichenmodus, der mit dem Macro
           SetDrMd() festgesetzt wird.*/

BYTE AreaPtSz;
        /* Hier 'steht drin', wie viele Zeilen das Füllmuster, das mit SetAfPt() geän-
           dert werden kann, enthält. Dabei ist zu beachten, daß die Höhe als Po-
           tenz von 2 angegeben wird. */

BYTE linpatcnt;
        /* Diese Variable wird zum Ziehen von Linien gebraucht und dient als
           Hilfsvariable, deren Inhalt für den Benutzer nicht von Belang ist. */

BYTE dummy;
USHORT Flags;
        /* Diese Variable enthält einige Flags, die z.B. bestimmen, ob der erste
           Punkt einer Linie gezeichnet werden soll (Flags | = FRST_DOT), ob nur
           ein Punkt pro Rasterzeile gezeichnet werden soll (ONE_DOT), ob die mit
           Area...() festgelegten Flächen mit Linien der Farbe des OPens umrandet
           werden sollen (Flags | = AREAOUTLINE). */

USHORT LinePtrn;
        /* Diese Variable enthält das 16-Bit-Linienmuster, das man mit dem Macro
           SetDrPt() festlegen kann. */

SHORT cp_x, cp_y;
        /* Diese beiden Variablen enthalten die X- und Y-Position des Grafik-Cur-
           sors, der mit Move() auf eine neue Position innerhalb der Bitmap gesetzt
           werden kann. */
```

struct Screen

Diese Struktur (ähnlich wie die Window-Struktur) erlaubt den Zugriff auf einen vorher geöffneten Screen (Screen = OpenScreen (&NewScreen)). Denn alle System-Strukturen, die z.B. für die Grafik-Ausgabe benötigt werden, stehen Ihnen über diese Struktur zur Verfügung (Screen->RastPort.xxx, Screen->ViewPort.xxx, etc).

---

struct SimpleSprite

```
{
  UWORD *posctldata;
        /* Dieser Zeiger zeigt auf einen Speicherbereich der Form:

             struct SpriteDaten
                  {
                     UWORD posctl[2];
                        /* Repräsentant der Hardware-Register
                           'spr[x].pos' und 'spr[x].ctl' } */

                     UWORD Aussehen[Höhe*2];
                        /* In diesem Array wird das Aussehen des
                           Sprites Zeile für Zeile in jeweils
                           zwei UWORDs definiert. */
                     UWORD Reserviert[2] = {0,0};
                  }

           Allerdings müssen Sie selber dafür sorgen, daß die 'SpriteDaten'-Struktur
           angelegt wird, da sie nicht in einem Include-File existiert. */

  UWORD height;
        /* Diese Variable enthält die Höhe des Sprites. Sprites sind immer 16
           Punkte (ein WORD) breit. */

  UWORD x,y;
        /* Diese beiden Variablen enthalten die augenblickliche Position des
           Sprites. */

  UWORD num;
        /* Diese Variable enthält die Nummer des Hardware-Sprites (0-7), das
           durch diese Simple-Sprite-Struktur beschrieben wird und verändert
           werden kann. */
}
```

Die SimpleSprite-Struktur sorgt dafür, daß ein Hardware-Sprite benutzt werden kann. Sprites sind immer 16 Punkte breit, können aber beliebig hoch sein.

```
    APTR *RP_User;
                /* Diese Variable ist allein für den Benutzer reserviert, das heißt, daß der
                   Benutzer mit dieser Variablen zum Beispiel eigene Datenstrukturen für
                   irgendwelche Spezialanwendungen mit dem Rastport verbinden kann.
                   */

    UWORD wordreserved[7];      /* Reserviert */
    ULONG longreserved[2];      /* Reserviert */
    UBYTE reserverd[8];    /* Reserviert */
}
```

Die Rastport-Struktur ist die wichtigste Struktur in bezug auf die Veränderung einer Bitmap, denn die meisten Grafikbefehle 'verlangen' eine Rastport-Struktur, da diese z.B. den aktuellen Wert des Vordergrundstiftes und vieles mehr zur Verfügung stellt.
Nach der Initialisierung mit 'InitRastPort (&RastPort)' muß nur noch der Zeiger auf die Bitmap initialisiert werden (RstPort.BitMap = &BitMap).

---

struct RasInfo

```
{
    struct RasInfo *Next;
                /* Wenn Sie im Viewport den Darstellungsmodus DUALPF (Dual Playfields)
                   eingestellt haben, müssen Sie zwei Bitmaps bestimmen, die sich im
                   Viewport überlagern sollen. Dazu verbinden Sie zwei RasInfo-Strukturen,
                   die jeweils auf eine der beiden Bitmaps zeigen. Dies geschieht durch
                   'RasInfo1.Next = &RasInfo2'. Die 'erste' der beiden RasInfo-Strukturen
                   muß dann nur noch dem Viewport zugänglich gemacht werden
                   (ViewPort.RasInfo = &RasInfo1;). Der 'Rest' geschieht dann bei der
                   gewohnten Öffnung des Views und Viewports. */

    struct BitMap *BitMap;
                /* Dies ist der Zeiger auf die Bitmap des RasInfos und letztlich die, die im
                   Viewport dargestellt werden soll. */

    SHORT RxOffset, RyOffset;
                /* Diese beiden Variablen bestimmen, welcher Punkt der Bitmap mit der
                   linken oberen Ecke des Viewports zusammenfallen soll. Normalerweise
                   sind sie gleich 0, was bedeutet, daß die linke obere Ecke der Bitmap mit
                   der linken oberen Ecke des Viewports zusammenfällt. Durch Ändern
                   dieser Werte und anschließendem Neuberechnen der Copper-Listen
                   kann man ein 'Scrolling' der Bitmap erreichen. */
}
```

Die RasInfo-Struktur ist der Mittler zwischen Viewport und Bitmap.

---

füllenden Elements aufzunehmmen. Dies ist aufgrund des rekursiven Füllalgorithmus nötig.

```
struct View
  {
   struct ViewPort *ViewPort;
                /* Das ist der Zeiger auf den ersten Viewport des Views */

   struct cprlist *LOFCprList;
                /* Das ist der Zeiger auf die Copper-Liste, die mit MrgCop() erzeugt wird.
                   */

   struct cprlist *SHFCprList;
                /* Dies ist auch ein Zeiger auf eine Copper-Liste, nur daß diese nur
                   während des Interlaced-Betriebs benutzt wird, während LOFCprList
                   immer gebraucht wird. */

   short DyOffset, DxOffset;
                /* Diese beiden Variablen bestimmen die Position des Views auf dem
                   Monitor und werden von InitView() automatisch richtig gesetzt, so daß
                   sich der Benutzer nicht mehr um diese zu sorgen braucht. */

   UWORD Modes;
                /* Diese Variable enthält die Auflösungsmodi des Views. Wollen Sie in
                   irgendeinem Viewport des Views z.B. den Interlaced-Modus benutzen,
                   müssen Sie diesen schon hier setzen. */
  }
```

Die View-Struktur ist der Chef der grafischen Darstellung und stellt die wichtigste Verbindung zwischen System und Benutzer dar.

```
struct ViewPort
  {
   struct ViewPort *Next;
                /* Dies ist der Zeiger auf den nächsten Viewport, denn es ist möglich,
                   mehr als einen Viewport in einem View darzustellen. Doch dazu müssen
                   die Viewports eben miteinander verbunden werden. Nach InitVPort()
                   zeigt dieser Zeiger ins Nichts, was heißt, daß kein weiterer Viewport
                   folgt.*/

   struct ColorMap *ColorMap;
                /* Dieser Zeiger bestimmt die Colormap des Viewports. Da jeder Viewport
                   seine eigene Colormap hat, ist es möglich, jedem Viewport innerhalb
                   eines Views andere Farben zu geben. */
```

Die Struktur, die das Aussehen des Sprites bestimmt, ('SpriteDaten') muß vom Programmierer selbst angelegt werden, und wird mit 'SimpleSprite.posctldata = (UWORD *) &SpriteDaten' an die SimpleSprite-Struktur übergeben.

---

struct TextAttr

```
{
    STRPTR ta_Name;
            /* Dieser Zeiger zeigt auf den Namen des Fonts, den Sie mit OpenFont()
               bzw. OpenDiskFont() laden wollen ("name.font"). */

    UWORD ta_YSize;
            /* Hier steht die Höhe (in Zeilen) des gewünschten Fonts. */

    UBYTE ta_Style;
            /* Mit dieser Variablen bestimmen Sie, welche Schriftarten der oben spezi-
               fizierte Font von vorneherein haben soll. */

    UBYTE ta_Flags;
            /* In dieser Variablen steht, ob der Font z.B. Proportionalschrift erlaubt etc.
               */
}
```

Diese Struktur wird von den Befehlen OpenFont() und OpenDiskFont() benötigt, die den mit 'ta_Name' und 'ta_YSize' näher bestimmten Font versuchen zu laden und dabei den Font öffnen, der am besten 'paßt'.

---

struct TextFont

Diese Struktur wird benutzt, um auf einen mittels OpenFont() bzw. OpenDiskFont() geöffneten Font zugreifen zu können. Somit kann er dann in einen Rastport eingebunden werden (mit SetFont()) oder in die Systemfontliste eingetragen und gelöscht werden (AddFont(), RemFont()).

---

struct TmpRas

Die TmpRas-Struktur wird von den Füllbefehlen Flood und 'Area...' benötigt. Sie muß mittels 'InitTmpRas' initialisiert werden. Danach muß sie an einen Rastport übergeben werden. In diesem Rastport können dann die 'Area...'-Befehle und der Flood-Befehl verwendet werden.

Die TmpRas-Struktur dient dazu, einen Speicherbereich zur Verfügung zu stellen, der genügend groß ist, eine Bitplane des größten zu

```
WORD OldY, OldX;
        /* Wenn Sie das Flag SAVEBACK im VSprite eines Bobs gesetzt haben,
           wird der Hintergrund, der beim Zeichnen eines Bobs unweigerlich über-
           schrieben wird, in einem von Ihnen bereitgestellten Puffer gerettet.
           Wenn das Bob von seiner Position wegbewegt wird und DrawGList()
           aufgerufen wird, tritt der alte Hintergrund wieder an seine alte Stelle.
           Damit der Rechner weiß, wohin der Hintergrund gehört, wird die alte
           Position des Bobs in 'OldX' und 'OldY' gerettet. */

WORD Flags;
        /* Diese Variable bestimmt, wie das VSprite vom System zu behandeln ist,
           denn die VSprite-Struktur dient zur Erzeugung der VSprites selbst (dann
           ist Flags = VSPRITE), aber auch Bobs benutzen diese Struktur. */

WORD Y,X;
        /* Diese beiden Variablen bestimmen die Position des VSprites oder Bobs
           auf dem Bildschirm. */

WORD Height;
        /* Diese Variable gibt die Anzahl der Linien eines VSprites bzw. eines Bobs
           an. */

WORD Width;
        /* Diese Variable bestimmt, wie viele Words zur Darstellung einer Linie
           eines Bobs herangezogen werden. Bei Verwendung in Verbindung mit
           VSprites muß hier immer der Wert 1 stehen, da VSprites nicht breiter als
           16 Punkte (= 16 Bit = 1 Word) sein können. */

WORD Depth;
        /* Hiermit geben Sie an, wie viele Bitplanes Ihr Bob 'tief' ist, das heißt, wie
           viele Bitplanes Sie für Ihr Bob definiert haben (Bitte beachten Sie, daß
           Ihr Bob nicht 'tiefer' sein kann als der Rastport, aber durchaus weniger
           Bitplanes als der Rastport enthalten kann). */

WORD MeMask;
WORD HitMask;
        /* Mit diesen beiden Variablen bestimmen Sie, welche Kollisionsroutine
           (wenn überhaupt) bei der Kollision mit einem anderen GEL ausgeführt
           wird. Die 'MeMask' des einen und die 'HitMask' des anderen GELs
           werden miteinander geANDet, und das Ergebnis-Bit bestimmt die Kolli-
           sionsroutine, die angesprungen wird. (Bit 1 => Routine1, Bit2 =>
           Routine2, ... Bit15 => Routine15).
           Wenn Sie in der HitMask das Bit 0 gesetzt haben, wird bei einer GEL-
           Rand-Kollision Routine-0 aufgerufen. */

WORD *ImageData;
        /* Dieser Zeiger zeigt auf Ihre Bob/VSprite-Daten, die das Ausehen Ihres
           GELs bestimmen. Diese müssen im Chip-Memory liegen. */
```

```
    struct CopList *DspIns; /* Display Instructions */
    struct CopList *SprIns; /* Sprite Instructions */
    struct CopList *ClrIns; /* Sprite Instructions */
    struct CopList *UCopList *UCopIns;
                /* Dies sind die Zeiger auf die Zwischen- oder Viewport-Copper-Listen, die
                   mit MakeVPort() erzeugt werden. */

    SHORT DWidth, DHeight;
                /* Diese beiden Variablen bestimmen die Höhe und die Breite des View-
                   ports in Punkten bzw. Linien des für diesen Viewport eingestellten Auf-
                   lösungsmodus (Hi-Res oder Lace) */

    SHORT DxOffset, DyOffset;
                /* Diese beiden Variablen bestimmen die Position des Viewports innerhalb
                   des übergeordneten Views. InitVPort() setzt diese beiden Variablen auf
                   0, deshalb müssen andere Positionen von Ihnen eingestellt werden. */

    UWORD Modes;
                /* Hier legen Sie den Darstellungsmodus des Viewports fest. In der Wahl
                   der Modi HAM, Extra Halfbrite, Sprites etc. sind Ihnen keine Beschrän-
                   kungen auferlegt. Aufpassen müssen Sie darauf, daß der höchste mög-
                   liche Auflösungsmodus vom View festgelegt wird. */

    UWORD reserved;
    struct RasInfo *RasInfo;
                /* Dieser Zeiger stellt die Verbindung zwischen Viewport und Bitmap über
                   die RasInfo-Struktur her. */
}
```

Der Viewport, der mittels der ViewPort-Struktur beschrieben wird, ist das 'Fenster', durch das Sie Ihre Bitmap auf dem Monitor sehen können.

---

struct VSprite

```
{
    struct VSprite *NextVSprite;
    struct VSprite *PrevVSprite;
                /* Diese beiden Zeiger dienen zur Erstellung der sogenannten 'GEL-Liste',
                   die mit SortGList() nach den 'Y'- und 'X'-Werten (weiter unten) fürs
                   Zeichnen sortiert werden. */

    struct VSptite *DrawPath;
    struct VSprite *ClearPath;
                /* Diese beiden Zeiger werden für das ordnungsgemäße Zeichnen des
                   Bobs gebraucht. Mit 'DrawPath' (Path = Weg) werden die Bobs in den
                   Rastport geschrieben, und mit 'ClearPath', der aus 'DrawPath' errechnet
                   wird, werden, falls erwünscht, die Bobs wieder gelöscht und evtl. der
                   Hintergrund wieder dargestellt. */
```

Außerdem könen Sie über das Window Nachrichten von Intuition erhalten. Doch eine detaillierte Beschreibung würde den Rahmen dieses Anhangs sprengen.

```
    WORD *BorderLine;
    WORD *CollMask;
            /* Diese beiden Zeiger zeigen auf die von Ihnen angelegten Speicher-
               plätze für die 'BorderLine' und die 'CollisionMask'. Diese werden zur
               Kollisionsentdeckung benötigt.
               Die 'BorderLine' enthält so viele Words, wie das Bob oder VSprite breit
               ist (VSprites sind immer 1 Word breit!) und enthält das logische OR aller
               Zeilen eines GELS. Die 'CollMask' hat exakt dieselbe Größe wie das
               GEL, zu dem sie gehört, nur daß sie 'einplane-ig' ist. Sie enthält das
               logische OR aller Plane-Daten. (Diese beiden Speicher, die von Ihnen
               zur Verfügung gestellt werden müssen, werden mit InitMasks() initiali-
               siert.) Auch diese beiden Masken müssen im Chip-Memory liegen. */

    WORD *SprColors;
            /* Dieser Zeiger zeigt auf einen drei UWORD großen Speicherbereich, der
               die Farben des VSprites enthält. */

    struct Bob *VSBob;
            /* Wenn Sie in der VSprite.Flags-Variablen nicht das VSPRITE-Flag gesetzt
               haben, nimmt das System an, daß die VSprite-Struktur zur näheren
               Beschreibung eines Bobs benutzt wird. Dieser Zeiger zeigt dabei auf die
               Bob-Struktur des VSprites. */

    BYTE PlanePick;
    BYTE PlaneOnOff;
            /* Diese beiden Variablen bestimmen, welche Bitplanes für die Darstellung
               von Bobs 'angeschaltet' sind (PlanePick) und welche von den 'passiven'
               mit dem 'ImageShadow' des Bobs beschrieben werden. */

    VUserStuff VUserExt;
            /* Mit Hilfe dieser 'Extension' kann der Benutzer eigene Daten in das
               VSprite einbinden. Dazu definiert er nach Belieben, bevor die
               #INCLUDE-Anweisungen ausgeführt werden, den 'VUserStuff' (z.B.
               #define VUserStuff struct Geschwindigkeit {vx, vy};). */
};
```

Die VSprite-Struktur wird für beide Arten der GELs benötigt. Sowohl das VSprite selbst als auch das Bob machen von dieser Struktur Gebrauch.

---

struct Window

Wenn Sie mit 'Window = OpenWindow (NewWindow)' ein neues Window geöffnet haben, können Sie über die Window-Struktur in das Window hineinschreiben. Denn diese stellt Ihnen einen Rastport zur Verfügung (Window->RPort).

AddVSprite (&VSprite, &RastPort)

Mit dieser Routine wird das durch die angegebene VSprite-Struktur definierte VSprite (virtuelles Sprite) in die GEL-Liste des angegebenen Rastports eingebunden. Dies ist nötig, damit das VSprite später mit DrawGList() ordnungsgemäß gezeichnet werden kann.

Siehe: AddBob(), DrawGList(), SortGList(), InitGList()

Pointer = **AllocRaster** (Breite, Höhe)

Mit diesem Befehl können Sie sich Speicherplatz für eine Bitplane der Größe von 'Breite' x 'Höhe' Punkten zuweisen lassen. Nach Aufruf dieser Routine steht in 'Pointer' die Adresse des ersten zugewiesenen Words.

Konnte nicht genügend Speicherplatz reserviert werden, steht in 'Pointer' der Wert 0.

Siehe: FreeRaster()

Animate (&AnimOb,&RastPort)

Mit dieser Routine werden für alle Animationsobjekte, die mit dem AnimOb verbunden sind, bzw. für die in ihnen enthaltenen Bobs die Positionen neu berechnet sowie die 'Timer' der Animationskomponenten heruntergezählt. Ist der 'Timer' einer Animationskomponente gleich 0, wird bei einer Ring-Animation (AnimComp.Flags = RINGTRIGGER) die darauffolgende Sequenz aktiviert.

Siehe: AddAnimOb()

AreaDraw (&RastPort, x, y)

Dieser Befehl fügt einen weiteren Polygonpunkt in die AreaInfo-Struktur des angegeben Rastports ein. X und Y bestimmen die Koordinate dieses Punktes in der Bitmap.

Siehe: AreaMove(), AreaEnd(), InitArea()

Anhang B: Die Library-Funktionen

Dieser Anhang gibt dem C-Programmierer Aufschluß über die von uns verwendeten Library-Routinen.

Nartürlich müssen Sie, bevor Sie die Library-Routinen benutzen können, die zugehörigen Bibliotheken öffnen (mit OpenLibrary()).

Im folgenden finden Sie die Routinen, nach Libraries getrennt, alphabetisch aufgelistet:

Zuerst die GfxBase-Routinen:

AddAnimOb (&AnimOb, &AnimKey, &RastPort)

Mit dieser Routine werden alle Bobs eines Animationsobjektes bzw. die Bobs aus dessen Animationskomponenten in die GEL-Liste des angegebenen Rastports eingereiht, um hinterher ordnungsgemäß mit DrawGList() gezeichnet werden zu können. Der 'AnimKey' zeigt dabei auf das zuletzt 'eingebettete' AnimOb. Bei ersten AddAnimOb()-Aufruf muß dieser allerdings gleich NULL sein (struct AnimOb *AnimKey = 0).

AddBob (&Bob, &RastPort)

Mit dieser Routine wird das durch die angegebene Bob-Struktur definierte Bob (Blitter Object) in die GEL-Liste des angegebenen Rastports eingebunden. Dies ist nötig, damit das Bob später mit DrawGList() ordnungsgemäß gezeichnet werden kann.

Siehe: AddVSprite(), DrawGList(), SortGList(), InitGList()

AddFont (&TextFont)

Diese Funktion 'linkt' den mit der angegebenen TextFont-Struktur definierten Font in die Systemfontliste ein.

Danach ist der Font jedem Programm, das ihn eventuell anfordert, zugänglich.

Siehe: RemFont()

AreaMove (&RastPort, x, y)

Mit dieser Routine veranlassen Sie, daß vorher mit AreaDraw() aufgebaute Polygone geschlossen werden. Außerdem wird veranlaßt, daß folgende AreaDraw()-Befehle ein neues Polygon bestimmen, dessen erster Punkt durch die hier angegebenen Koordinaten bestimmt wird.

Siehe: AreaDraw(), AreaEnd(), InitArea()

AskFont (&RastPort, &TextAttr)

Mit diesem Befehl wird die angegebene TextAttr-Struktur mit den Werten des aktuellen Fonts im angegebenen Rastport initialisiert. Somit könnten Sie untersuchen, welcher Font vom Rastport gerade benutzt wird.

Siehe: SetFont()

Style = **AskSoftStyle** (&RastPort)

Diese Funktion liefert die möglichen Schriftarten des Fonts im angegebenen Rastport, die mittels SetSoftStyle gesetzt werden können. Jedes gesetzte Bit in Style steht dabei für eine Schriftart:

FSF_UNDERLINED	= Bit 0 = 1	(Unterstrichen)
FSF_BOLD	= Bit 1 = 2	(Fett)
FSF_ITALIC	= Bit 2 = 4	(Kursiv)
FSF_NORMAL	= kein Bit gesetzt = 0	

Siehe: SetSoftStyle()

BitPlanes = **BltBitMap** (&QuellBitMap, X1, Y1, &ZielBitMap, X2, Y2, Breite, Höhe, Minterm, Maske, Buffer)

Diese Funktion 'blittet' ein Rechteck aus einer Bitmap in eine andere Bitmap. Dabei müssen Sie die Position des Quellrechtecks innerhalb der QuellBitMap angeben (X1, Y1) sowie die Position, an der das Rechteck in der ZielBitMap erscheinen soll (X2, Y2). 'Höhe' und 'Breite' sind für beide Rechtecke natürlich gleich und brauchen deshalb nur einmal angegeben zu werden. Natürlich geben Sie auch die Quell- und ZielBitMap an, wobei diese selbstverständlich auch gleich sein können.

AreaEnd (&RastPort)

Dieser Befehl veranlaßt, daß das mit AreaMove() und AreaDraw() vorher bestimmte Polygon in dem angegebenen Rastport gezeichnet und mit dem aktuellen Füllmuster unter Beachtung des aktuellen APens, BPens, OPens und Draw Modes gefüllt wird.

Der APen und BPen bestimmen die Farben des Füllmusters, während der OPen die Umrandungsfarbe bestimmt.

Bitte beachten Sie, daß das Füllen von Flächen mittels der 'Area...'-Befehle nicht funktioniert, wenn nicht vorher eine Area- und eine TmpRas-Struktur in dem Rastport, in dem die Polygone gezeichnet werden sollen, initialisiert wurden.

Siehe: AreaDraw(), AreaMove(), InitArea(), InitTmpRas()

---

error = **AreaEllipse** (&RastPort, XMitte, YMitte, XRadius, YRadius)

Mit diesem Befehl können Sie eine Ellipse in den angegebenen Rastport zeichnen. Der Mittelpunkt liegt im Punkt ('XMitte', 'YMitte'). 'XRadius' und 'YRadius' geben die Radien der Ellipse an. (Gilt 'XRadius' = 'YRadius', so wird ein ausgefüllter Kreis gezeichnet. 'XRadius' und 'YRadius' müssen größer als 0 sein!)

Dieser Befehl ist ein weiterer Vertreter der 'Area...'-Befehle. Das heißt, daß eine im aktuellen Füllmuster ausgefüllte Ellipse gezeichnet wird. Das bedeutet aber auch, daß eine AreaInfo-Struktur und eine TmpRas-Struktur im angegebenen Rastport vorhanden sein müssen.

Die Variable 'error' gibt Ihnen Auskunft darüber, ob die Vektortabelle der AreaInfo-Struktur, die für die übrigen 'Area...'-Befehle die Koordinaten der Polygonstützpunkte enthält, noch genügend Platz hatte (mindestens (2+1)*5 Bytes), um die Daten für eine Ellipse aufnehmen zu können (error = 0). Ist nicht mehr genug Platz vorhanden, hat 'error' den Wert -1.

Um zu veranlassen, daß die Ellipse tatsächlich gezeichnet wird, muß AreaEnd() aufgerufen werden.

Siehe: AreaDraw(), AreaEnd(), AreaMove(), InitArea(), InitTmpRas()

Als Ergebnis wird der Erfolg dieses Befehls zurückgegeben (TRUE = erfolgreich, FALSE = Fehler).

Siehe: BltBitMap()

BltClear (&Speicher, AnzBytes, Flags)

Diese Funktion löscht, wie sich unschwer aus dem Namen erkennen läßt, einen beliebigen Speicherbereich. Dazu wird einfach die Adresse des zu löschenden Speichers übergeben.

Weiterhin übergeben Sie einen Flag-Parameter, von dem abhängt, wie 'AnzBytes' interpretiert wird:

Ist Bit 1 des Flag-Parameters gesetzt, bestimmen:

- die oberen 16 Bits von 'AnzBytes' die Anzahl der zu löschenden Linien
- die unteren 16 Bits von 'AnzBytes' die Anzahl der zu löschenden Bytes pro Linie

Ist Bit 1 gelöscht, so gibt 'AnzBytes' nur die (gerade) Anzahl zu löschender Bytes an.

Bit 0 des Flags-Parameters bestimmt, ob die Funktion sofort zum Programm 'zurückkehren' soll, oder ob solange gewartet wird, bis der Blitter alles gelöscht hat.

BltTemplate (&Quelle, BitPosition, Modulo, &RastPort, X, Y, Breite, Höhe)

Diese Funktion transferiert mittels des Blitters Daten aus einem gepackten Array in einen Rastport. Dazu übergeben Sie die Adresse dieses 'gepackten' Datenarrays und die BitPosition innerhalb dieses Arrays, ab der mit dem Lesen der Daten angefangen werden soll.

Der 'Modulo' gibt die Anzahl der Bytes einer Linie innerhalb des Daten-Arrays an. 'X', 'Y', 'Breite' und 'Höhe' geben an, wo und wie viele Daten aus dem Array auf dem angegebenen Rastport erscheinen sollen.

Weiterhin geben Sie den Minterm an. Dieser bestimmt, wie das Quell- und das Zielrechteck miteinander logisch verknüpft werden. Welche es gibt und was sie bewirken, können Sie im Kapitel 16 erfahren.

Der Parameter 'Maske' bestimmt, welche Bitplanes überhaupt geblittet werden sollen. Normalerweise steht hier $ff, was bedeutet, daß alle Bitplanes angesprochen werden. Sie können aber auch Bitplanes ausblenden, indem Sie in Maske einfach das zugehörige Bit löschen (Bit 0 für Bitplane 0 usw.).

'Buffer' zeigt auf einen Speicherbereich, der dann gebraucht wird, wenn sich Zielrechteck und Quellrechteck in einer Bitmap überschneiden. Dieser Puffer muß dann genügend Speicher enthalten, um eine Zeile des Rechtecks beinhalten zu können. Sind Sie sicher, daß sich Ziel- und Quellrechteck nicht überschneiden, können Sie hier 0 angeben.

Das Ergebnis, das diese Funktion zurückgibt, sind die tatsächlich von dem 'Blit' betroffenen Bitplanes (s. Maske).

Bitte beachten Sie, daß bei diesem Befehl nicht getestet wird, ob z.B. das Zielrechteck völlig innerhalb einer Bitmap liegt. Somit wäre es möglich, über die Grenzen einer Bitmap hinaus zu 'blitten'. Dies könnte allerdings Fehler verursachen, was daran festzustellen ist, daß weniger Bitplanes betroffen wurden als erwartet. Bei ganz schweren Fehlern meldet sich Ihr Amiga mit den heißgeliebten 'Guru Meditations'.

Siehe: ClipBlit()

BltBitMapRastPort (&QuellBitMap, X1, Y1, &ZielRastPort, X2, Y2, Breite, Höhe, Minterm)

Diese Funktion führt exakt dieselbe Operation durch wie BltBitMap(), nur daß hier ein Rechteck aus einer Bitmap in einen Rastport hinein geblittet wird.

Allerdings werden hier der 'Maske'- und 'Buffer'-Parameter nicht gebraucht.

ClearScreen (&RastPort)

Im Gegensatz zum ClearEOL()-Befehl, der nur eine einzige Zeile löscht, löscht ClearScreen(), wie der Name schon vermuten läßt, den gesamten Rastport ab der augenblicklichen Grafik-Cursor-Position.

Siehe: ClearEOL(), Move()

ClipBlit (&QuellRastPort, X1, Y1, &ZielRastPort, X2, Y2, Breite, Höhe, Minterm)

Dieser Befehl führt exakt dieselbe Funktion wie BltBitMap() durch. Die meisten Parameter haben auch dieselbe Bedeutung, nur wird hier von einem Rastport in den anderen geblittet und weiterhin von selbst auf eventuelle Überschneidungen geachtet.

Was diesen Befehl für die Anwendung so bevorzugt macht, ist die Tatsache, daß er erstens leicht zu benutzen (man braucht nicht extra einen Puffer zu reservieren) und zweitens überhaupt nicht fehleranfällig ist. Das 'Blitten' über die Grenzen eines Rastports hinaus nimmt er nicht so übel wie BltBitMap().

Siehe: BltBitMap(), BltBitMapRastPort()

CloseFont (&TextFont)

Wenn Sie eine Font mit 'OpenFont()' oder 'OpenDiskFont()' geöffnet haben, dann müssen Sie dafür sorgen, daß dieser spätestens am Ende Ihres Programms auch wieder geschlossen wird. Sonst kann der Speicherplatz, den der Font einnimmt, nämlich nicht freigegeben werden.

Siehe: OpenFont()

DisownBlitter ()

Diese Funktion 'befreit' den Blitter aus Ihren 'Klauen', d.h., daß andere Programme den Blitter wieder benutzen können.

Siehe: OwnBlitter()

ChangeSprite (&ViewPort, &Sprite, &SpriteDaten)

Diese Funktion verändert das Aussehen des angegebenen Hardware-Sprites. Zu beachten ist hierbei, daß die Sprite-Struktur schon vorher initialisiert worden sein muß, damit die Höhe und die Position auf dem Bildschirm schon vorher bestimmt sind und damit das richtige Hardware-Sprite (SimpleSprite.num) betroffen wird.

Die Daten, die für diese Funktion benötigt werden, haben folgendes Format:

```
struct SpriteDaten
       {
          UWORD posctl[2];
          UWORD Daten[Höhe][2];
          UWORD Reserviert[2]; /* = 0,0 */
       }
```

wobei 'posctl' stellvertretend für die Sprite-Hardware-Register 'spr[x].pos' und 'spr[x].ctl' stehen. Somit kann sich der Benutzer z.B. ein geradzahliges Sprite und das dazugehörige ungeradzahlige Sprite mit GetSprite() reservieren lassen und dann das SPRITE_ATTACAED-Bit in 'posctl[1]' setzen, um so sogenannte 'verbundene' Sprites zu bekommen. Bei diesen werden dann die Bitmuster beider Sprites, wenn sie sich an irgendeiner Stelle überlappen, zusammengenommen, um so 15 anstatt nur 3 Farben (plus transparent) darstellen zu können.

Siehe: GetSprite(), FreeSprite(), MoveSprite()

ClearEOL (&RastPort)

Dieser Befehl löscht im angegebenen Rastport eine Zeile, die die Höhe der Zeichen des gerade aktuellen Fonts hat. Mit dem Löschen wird ab der aktuellen Grafik-Cursor-Position (RastPort.cp_x, RastPort.cp_y) begonnen.

Siehe: ClearScreen(), Move()

Flood (&RastPort, Modus, x, y)

Mit diesem Befehl ist es möglich, zusammenhängende Flächen innerhalb eines Rastports zu füllen. Dazu werden ab der angegebenen X/Y-Position alle im Umkreis liegenden Punkte getestet, ob sie entweder die Farbe des OPens haben (Modus = 0), oder ob sie die Farbe des Startpunktes (Modus = 1) haben.

Alle im Umkreis liegenden Punkte, die keine der beiden Bedingungen erfüllen, werden unter Berücksichtigung des aktuellen Füllmusters gesetzt. Beim Feststellen einer solchen Bedingung wird der getestete Punkt nicht mehr verändert.

Sind alle Punkte innerhalb einer umschlossenen Linie, die eine der Bedingungen erfüllt, schon 'gefüllt', wird mit dem Füllen aufgehört.

Bitte beachten Sie aber, daß für den Flood()-Befehl eine sogenannte 'TmpRas'-Struktur initialisiert werden muß, die den für den Flood()-Befehl benötigten Speicherplatz enthält.

Siehe: InitTmpRas()

FreeColorMap (&ColorMap)

Mit dieser Funktion geben Sie den Speicherplatz für die Colormap-Struktur, die Sie sich vorher mit GetColorMap() zuweisen lassen haben, wieder an das System zurück.

Siehe: GetColorMap()

FreeCopList (&CopList)

Diese Routine wird aufgerufen, um eine einzige Zwischen-Copper-Liste eines Viewports (struct CopList), die mit MakeVPort() erstellt wurde, wieder freizugeben. 'FreeVPortCopList()' macht von dieser Routine Gebrauch, indem sie alle Zwischen-Listen eines Viewports mit Hilfe dieser Funktion wieder freigibt.

Siehe: FreeVPortCopList()

DoCollision (&RastPort)

Mit dieser Routine werden alle GELs der GEL-Liste des Rastports auf Kollisionen getestet, und gegebenfalls werden Ihre vorher mit SetCollision() festgelegten Kollisionsroutinen angesprungen.

Leider müssen wir Sie darauf hinweisen, daß DoCollision() nicht 100%ig funktioniert!

Siehe: SetCollision()

Draw (&RastPort, x, y)

Dies ist wohl einer der wichtigsten Grafikbefehle überhaupt. Mit ihm ist es nämlich möglich, eine Linie von der augenblicklichen Position des Grafik-Cursors zur angegebenen X/Y-Koordinate in den angegebenen Rastport zu zeichnen.

Siehe: Move()

DrawEllipse (&RastPort, XMitte, YMitte, XRadius, YRadius)

Mit diesem Befehl können Sie Ellipsen, bei XRadius = YRadius Kreise, zeichnen. Diese werden allerdings nicht wie bei AreaEllipse, ausgefüllt. Es wird nur die Peripherie gezeichnet.

Zu beachten ist, daß XRadius und YRadius auch hier Werte größer als 0 annehmen müssen.

DrawGList (&RastPort, &ViewPort)

DrawGList() veranlaßt, daß erstens die Bobs der GEL-Liste in den angegebenen Rastport gezeichnet werden und zweitens, daß die VSprites in die Viewport-Copper-Listen des angegebenen Viewports eingebunden werden.

VSprites werden nach DrawGList() aber noch nicht gezeichnet. Dazu muß erst mittels MakeVPort() oder MakeScreen() die neue Copper-Liste vorberechnet werden, die dann mit MrgCop() und LoadView() oder RethinkDisplay() dargestellt wird.

Siehe: MakeVPort(), MrgCop(), LoadView(), MakeScreen(), RethinkDisplay()

FreeVPortCopLists (&ViewPort)

Mit diesem Befehl geben Sie alle Copper-Listen - für die Farben, für die Darstellung, für die Sprites etc. - des angegebenen Viewports, die vorher mit MakeVPort() generiert wurden, wieder frei.

Siehe: MakeVPort(), FreeCopList()

---

*ColorMap = **GetColorMap** (AnzahlFarben)

Mit dieser Funktion wird eine komplette ColorMap-Struktur angelegt. Dabei werden die erforderlichen Variablen der ColorMap-Struktur initialisiert, auch soviel Speicherplatz wird reserviert, wie für 'AnzahlFarben' Farbeinträge gebraucht wird. Konnte nicht genügend Speicher für die neue ColorMap-Struktur oder den Farbspeicher reserviert werden, wird NULL (0) als Ergebnis zurückgegeben.

Diese ColorMap-Struktur muß von Ihnen in den Viewport mit 'ViewPort.ColorMap = ColorMap' eingebunden werden (bevor mit MakeVPort() die Copper-Listen für den Viewport berechnet werden).

Siehe: FreeColorMap(), LoadRGB4(), GetRGB4(), SetRGB4()

Farbe = **GetRGB4** (&ColorMap, Farbregister)

Mit dieser Funktion bekommen Sie das UWORD geliefert, das die Farbe des angegebenen Farbregisters bestimmt. Die 16 Bit des zurückgegebenen UWORDs sind dabei so aufgeteilt, daß Bit 0-3 die Blau-, Bit 4-7 die Grün- und Bit 8-11 die Rotkomponente des angegebenen Farbregisters enthalten.

Siehe: SetRGB4(), LoadRGB4()

SprNummer = **GetSprite** (&SimpleSprite, gewSprNummer)

Mit dieser Funktion können Sie jedes beliebige Hardware-Sprite für Ihren Eigengebrauch deklarieren. Dazu übergeben Sie der Funktion Ihre SimpleSprite-Struktur, die mit dem Sprite assoziiert werden soll. Weiterhin übergeben Sie die gewünschte Nummer (0-7) des zu belegenden Sprites, oder -1, wenn es Ihnen egal ist, welches Sprite Ihnen 'gehören' soll.

FreeCprList (&cprlist)

Mit dieser Funktion geben Sie den Speicherplatz frei, der von der Hardware-Copper-Liste des Views (View.LOFCprList, View.SHFCprList) belegt wurde. Diese Hardware-Copper-Liste wird aus den Copper-Listen der einzelnen Viewports eines Views mittels MrgCop() berechnet.

Siehe: MrgCop()

FreeRaster (&BitPlane, Breite, Höhe)

Diese Funktion ist das Gegenstück zu AllocRaster(). Sie gibt nämlich den mit AllocRaster() für Bitplanes reservierten Speicherplatz wieder an das System zurück, der somit wieder für andere Programme benutzbar wird.

'BitPlane' zeigt dabei auf den Speicherplatz, den Sie sich mit AllocRaster zuweisen lassen haben. 'Breite' und 'Höhe' müssen dabei für Free- und AllocRaster() immer gleich sein, da es sonst passieren könnte, daß zuviel oder zuwenig reservierter Speicher freigegeben wird.

Siehe: AllocRaster()

FreeSprite (SprNummer)

Mit FreeSprite() geben Sie das vorher mit GetSprite() nur für Ihren Gebrauch deklarierte Sprite wieder an das System zurück. Somit steht es dem gesamten System wieder zur Verfügung (vor allen Dingen den VSprites).

Dazu übergeben Sie diesem Befehl einfach die Nummer des Sprites, das Sie sich mit GetSprite() zuweisen lassen haben.

Siehe: GetSprite(), ChangeSprite(), MoveSprite()

= &GelsInfo). Danach können Sie mit dem Zeichnen der Bobs und VSprites loslegen.

Siehe: AddVSprite(), AddBob()

InitMasks (&VSprite)

Mit dieser Routine initialisieren Sie die 'BorderLine' und die 'CollMask' der angegebenen VSprite-Struktur. Der Speicher für diese beiden Masken muß reserviert und an die Variablen 'VSprite.BorderLine' und 'VSprite.CollMask' zugewiesen worden sein.

InitRastPort (&RastPort)

Mit dieser Routine initialisieren Sie die angegebene Rastport-Struktur. Diese Rastport-Struktur ist wohl die wichtigste Schnittstelle zwischen Benutzer und Bitmap.

Der Rastport enthält nämlich z.B. die Farbe des aktuellen Vordergrundstiftes (APen) sowie einen Zeiger auf die Bitmap, in der die Grafikbefehle 'wirken' sollen, sowie weitere wichtige Variablen und Zeiger, die von den meisten Grafikbefehlen benötigt werden.

TmpRas = **InitTmpRas** (&TmpRas, &Buffer, Buffergröße)

Wenn Sie innerhalb eines Rastports die Area...()-Befehle oder den Flood()-Befehl benutzen wollen, benötigen Sie einen extra Speicherplatz, der von diesen Befehlen benutzt wird. Dieser Speicherplatz ('Buffer') sollte mindestens so groß sein wie das größte zu füllende Objekt, denn aufgrund des rekursiven Algorithmus des Füllens von Flächen werden Speicherplätze zum Arbeiten benötigt.

Allerdings brauchen Sie in diesem Speicherplatz nur eine Bitplane von der Größe des größten Elements zu reservieren. Wollen Sie auf 'Nummer Sicher' gehen, dann reservieren Sie eine komplette Bitplane als 'temporäres Raster'.

Die initialisierte TmpRas-Struktur brauchen Sie dann nur noch dem Rastport zuzuweisen. Dazu gibt es zwei Möglichkeiten: Erstens können Sie das Ergebnis dieser Funktion benutzen, das die initialisierte TmpRas-Struktur zurückgibt (RastPort.TmpRas = InitTmpRas (...)), oder Sie weisen dem Rastport die zu initialisierte TmpRas-Struktur selbst zu (RastPort.TmpRas = &TmpRas).

In 'SprNummer' steht dann die Nummer des Sprites, das Sie bekommen haben. Waren aber schon alle Sprites vergeben, so steht in 'SprNummer' -1.

Siehe: FreeSprite(), MoveSprite(), FreeSprite()

InitArea (&AreaInfo, &Buffer, AnzKoord)

Mit dieser Funktion initialisieren Sie die angegebene AreaInfo-Struktur. Ihr wird der angegebene Speicher ('Buffer') zugewiesen, der 'AnzKoord'+1 Koordinaten für AreaDraw() und AreaMove() enthalten soll. Bitte beachten Sie, daß mindestens 5mal soviele Bytes in diesem Puffer enthalten sein müssen, wie Sie Koordinaten für die Polygone mit AreaMove() und AreaDraw() festlegen wollen.

Am besten ist es, wenn Sie den Speicher als ein Array von 'char' Elementen deklarieren. Dann brauchen Sie sich nämlich nicht extra darum zu kümmern, daß der für die AreaInfo-Struktur benutzte Speicher wieder freigegeben werden muß, da Arrays automatisch vom Programm, das der Compiler erzeugt, wieder freigegebeben werden.

Siehe: AreaDraw(), AreaMove(), AreaEnd()

InitBitMap (&BitMap, Tiefe, Breite, Höhe)

Mit dieser Routine wird die angegebene BitMap-Struktur initialisiert. Auf einfache Art und Weise werden die 'Tiefe' (Anzahl der Bitplanes), die 'Breite' (in Punkten) und die 'Höhe' (in Zeilen) festgelegt.

Siehe: AllocRaster()

InitGels (&Anfang, &Ende, &GelsInfo)

Mit dieser Funktion initialisieren Sie die angegebene GelsInfo-Struktur. Diese wird gebraucht, um alle VSprites und Bobs ordnungsgemäß zeichnen zu können.

Hier werden nämlich in einer verketteten Liste alle grafischen Elemente (= GEL) durch VSprite-Strukturen (&Anfang, &Ende) miteinander verbunden. &Anfang und &Ende markieren dabei den Beginn und das Ende dieser GEL-Liste.

Haben Sie mittels InitGels() die GelsInfo-Struktur initialisiert, müssen Sie diese nur noch dem Rastport zugänglich machen (RastPort.GelsInfo

LoadView (&View)

Nach dem Aufruf dieser Funktion wird der angegebene View und alle in ihm enthaltenen Viewports auf dem Bildschirm dargestellt. Dazu müssen vorher mit MakeVPort() für jeden Viewport, der in diesem View dargestellt werden soll, die Viewport-Copper-Listen erstellt werden, die dann mit MrgCop() zu einer einzigen View-Copper-Liste zusammengeschmolzen werden.

Die Anfangsadresse dieser View-Copper-Liste wird dann mit LoadView() in die zugehörigen Hardware-Register (cop1lc, cop2lc) geschrieben.

Siehe: InitView(), InitVPort(), MakeVPort(), MrgCop()

MakeVPort (&View, &ViewPort)

Mit dieser Routine werden die Zwischen- oder Viewport-Copper-Listen des angegebenen Viewports unter Einbeziehung des übergeordneten Views (zur Berechnung der Position der Viewports) berechnet. So existieren z.B. Viewport-Copper-Listen für die Farbdarstellung, in der alle Farbwerte mittels des Coppers in die dazugehörigen Hardware-Register geschrieben werden. Diese Farb-Copper-Liste wird aus der Colormap des Viewports berechnet. Ist diese aber gleich 0, wird die 'Default'-Colormap zur Berechnung der Farb-Copper-Liste des Viewports herangezogen.

Außerdem werden in diesen Listen auch die verschiedenen Darstellungsmodi der Viewports eingestellt (mehrere Viewports eines Views können unterschiedliche Darstellungsmodi haben).

Siehe: MrgCop(), FreeVPortCopLists()

Move (&RastPort, X, Y)

Mit dieser Funktion setzen Sie den Grafik-Cursor des angegebenen Rastports auf die angegebene Position (RastPort.cp_x = x; RastPort.cp_y = y;).

An dieser Position kann dann z.B. Text ausgegeben oder eine Linie zu einem anderen Punkt gezogen werden.

InitView (&View)

Mit dieser Funktion, wie könnte es anders sein, initialisieren Sie die angegebene View-Struktur. Dazu werden zuerst alle Variablen und Zeiger der View-Struktur auf 0 gesetzt. Dann werden die Variablen DxOffset und DyOffset in der View-Struktur so besetzt, daß der View ca. 2 cm vom oberen und rechten Monitorrand entfernt positioniert wird. Voraussetzung dafür ist natürlich, daß der Monitor richtig eingestellt ist.

Siehe: MrgCop(), MakeVPort()

InitVPort (&ViewPort)

Mit dieser Funktion werden alle Variablen und Zeiger der angegebenen ViewPort-Struktur gelöscht. Dies ist deshalb wichtig, damit hinterher die einzelnen Copper-Listen für diesen Viewport von MakeVPort() ordnungsgemäß berechnet werden können. Denn ist ein Zeiger auf eine Viewport-Copper-Liste ungleich 0, heißt das für MakeVPort(), daß diese Viewport-Copper-Liste schon existiert und deshalb nicht mehr neu berechnet zu werden braucht.

Außerdem könnten so 'undefinierte' Copper-Listen zur Abarbeitung gelangen.

Siehe: MakeVPort()

LoadRGB4 (&ViewPort, &Farbpalette[0], Farbeinträge)

Mit dieser Routine werden 'Farbeinträge' neue Farbwerte aus der 'Farbpalette' (UWORD Farbpalette[Farbeinträge];) in die Colormap des angegebenen Viewports umgeschaufelt.

So ist es auf einfache Art und Weise möglich, einem Viewport mit einem Schlag neue Farben zu geben. Allerdings werden diese nicht sofort dargestellt, da erst die Farb-Copper-Liste des Viewports neu berechnet werden muß. Das geschieht mit MakeVPort(), MrgCop() und einem anschließenden LoadView() oder, wenn Sie einen Intuition-Screen benutzen, mit RemakeDisplay().

Siehe: SetRGB4(), GetRGB4()

---

nämlich niemand mehr auf den Font zu, kann dieser mit 'RemFont()' aus dem Speicher entfernt werden.

Siehe: Closefont(), AddFont(), RemFont(), OpenDiskFont(), AvailFonts()

OwnBlitter ()

Mit dieser Funktion 'sagen' Sie dem Blitter, daß er nur noch für Sie arbeiten darf. Andere Tasks, sprich gleichzeitig ablaufende Programme, können dann den Blitter nicht mehr benutzen.

Nach OwnBlitter() sollten Sie aber mit WaitBlit() darauf warten, daß der Blitter einen vielleicht vor kurzem begonnenen 'Blit' vollendet.

Siehe: DisownBlitter(), WaitBlit()

PolyDraw (&RastPort, AnzKoords, &PunkteArray)

Die Punkte, deren Koordinaten Sie vorher in einem Array abgespeichert haben, können Sie mit dieser Routine in den angegebenen Rastport zeichnen. Dabei werden alle aufeinanderfolgenden Punkte miteinander durch eine Linie verbunden.

Abgespeichert werden die Punkte in einem int-Array (int PunkteArray[AnzKoords][2]), wobei eine Koordinate immer aus 2 Array-Elementen besteht. Im ersten Element wird die X-, im zweiten die Y-Koordinate des Polygon-Punktes gespeichert.

'AnzKoords' gibt an, wie viele Punkte des Punkte-Arrays gezeichnet werden sollen.

Farbregister = **ReadPixel** (&RastPort, X, Y)

Mit dieser Funktion können Sie sich das Farbregister zurückgeben lassen, mit dessen Farbe der Punkt an der angegebenen Koordinate dargestellt wird.

Liegt die Koordinate außerhalb des Rastports, so hat 'Farbregister' den Wert -1.

Siehe: WritePixel(), SetAPen()

MoveSprite (&ViewPort, &SimpleSprite, X, Y)

Mit dieser Funktion positionieren Sie das angegebene Hardware-Sprite innerhalb des angegebenen Viewports, falls 'ViewPort' ungleich 0 ist - sonst wird das Sprite relativ zum übergeordneten View positioniert.

X und Y geben die neue Position an. Beachten Sie dabei bitte, daß beim Positionieren in Hi-Res- oder Lace-Viewports immer um 2 anstatt um 1 positioniert werden muß, damit sich das Sprite tatsächlich bewegt.

Siehe: ChangeSprite(), GetSprite(), FreeSprite()

MrgCop (&View)

Dieser Befehl berechnet die Hardware-Copper-Liste des Views aus den Viewport-Copper-Listen, die vorher mit MakeVPort() erzeugt wurden. Diese wird dann nach LoadView() von der Hardware abgearbeitet (cop1lc, cop2lc).

Bei jeder Änderung in der Farbtabelle oder der Position der Hardware-Sprites muß eine neue Hardware-Copper-Liste berechnet werden. Bei der Bewegung der Hardware-Sprites wird das automatisch erledigt, während bei einer Farbpalettenänderung mittels LoadRGB4() der Benutzer aktiv werden muß.

Siehe: MakeVPort(), FreeCprList()

---

*TextFont = **OpenFont** (&TextAttr)

Diese Funktion durchsucht die Systemfontliste nach dem mittels der TextAttr-Struktur näher bestimmten Font. Dabei wird, wenn ein Font mit dem in der TextAttr-Struktur spezifizierten Namen nicht gefunden werden konnte, 0 zurückgegeben. Konnte aber ein Font mit dem festgelegten Namen, aber unterschiedlichen Schriftarten oder Größen, gefunden werden, so wird derjenige Font zurückgegeben, der die gestellten Anforderungen am besten erfüllt.

OpenFont() erhöht die Anzahl der Benutzer, die auf diesen Systemfont zugreifen. Deshalb müssen sich OpenFont() und CloseFont() die Waage halten, um eine 'saubere' Speicherverwaltung zu garantieren. Greift

RemVSprite (&VSprite)

Diese Routine löscht das angegebene VSprite aus der gerade aktuellen GEL-Liste. Vom Bildschirm verschwindet es allerdings noch nicht. Das geschieht erst nach einem folgenden Aufruf von 'SortGList(), DrawGList(), MakeVPort(), MrgCop(), LoadView()'.

Siehe: DrawGList(), SortGList()

ScrollRaster (&RastPort, DeltaX, DeltaY, X1, Y1, X2, Y2)

Diese Funktion 'scrollt' das mit X1, Y1, X2 und Y2 (s. RectFill) angegebene Rechteck um 'DeltaX' Punkte und 'DeltaY' Linien. Dabei wird bei positivem Vorzeichen der 'Delta' hoch bzw. nach links 'gescrollt'. Negative 'Delta' veranlassen, daß um 'Delta' Punkte nach rechts bzw. 'Delta' Linien nach unten 'gescrollt' wird.

Die durch das Verschieben frei gewordenen Flächen werden mit der Farbe des BPens gefüllt.

Allerdings sollten Sie beachten, daß diese Routine nicht besonders schnell ist und es so zu einem Flackern während des Scrollens kommen kann!

ScrollVPort (&ViewPort)

Nachdem der Benutzer die Werte 'RxOffset' und 'RyOffset' in der RasInfo-Struktur des Viewports auf neue Werte gesetzt hat, wird nach ScrollVPort() die Bitmap 'von einer anderen Seite' gezeigt. ScrollVPort() veranlaßt nämlich, daß die Copper-Listen für die neue Position der Bitmap berechnet wird.

Leider ist die Routine nicht genügend schnell, um ein gewisses Flackern zu verhindern. Deshalb sollten Sie die Berechnung der Copper-Listen selbst 'in die Hand nehmen', da Sie im 'Hintergrund' rechnen können.

RectFill (&RastPort, X1, Y1, X2, Y2)

Mit dieser Funktion können Sie beliebige rechteckige Flächen innerhalb des angegebenen Rastports unter Berücksichtigung des aktuellen Füllmusters sowie des aktuellen DrawModes, APens etc., ausfüllen.

Dazu übergeben Sie einfach die linke obere Ecke (X1, Y1) und die rechte untere Ecke des zu füllenden Rechtecks. Bitte beachten Sie dabei, daß die rechte untere Ecke tatsächlich rechts und unterhalb von der linken oberen Ecke liegt, da sonst der Rechner abstürzen könnte, weil unkontrolliert im Speicher 'herumgefüllt' wird.

Dieser Befehl benötigt keine TmpRas-Struktur, wie der Flood()- und die Area...()-Befehle.

RemFont (&TextFont)

Diese Funktion löscht den angegebenen Font aus der Systemfontliste, wobei natürlich den Programmen, die zur Zeit auf diesen Font zugreifen, der Zugriff weiterhin erlaubt bleibt. Erst wenn alle Programme mittels CloseFont() signalisiert haben, daß sie diesen Font nicht mehr benötigen, wird er auch physisch aus dem Speicher entfernt.

Nach RemFont() wird Programmen ein neuer Zugriff auf den Font nicht mehr erlaubt.

Siehe: AddFont(), OpenFont(), CloseFont()

RemIBob (&Bob, &RastPort, &ViewPort)

Mit dieser Routine wird das angegebenen Bob sofort (immediate) aus der GEL-Liste des Rastports und aus dem angegebenen Rastport selber gelöscht.

Siehe: DrawGList(), SortGList(), InitGList()

SetDrMd (&RastPort, Modus)

Mit diesem Befehl wird der Zeichenmodus (DrawMode) des Rastports, in dem Linien gezeichnet, Punkte gesetzt werden und Text ausgegeben wird, festgelegt. Dabei existieren folgende Zeichenmodi:

JAM1 (0): Es wird nur in der Farbe des APens 'gemalt'.

JAM2 (1): Es wird weiterhin mit der Farbe des BPens unterlegt.

COMPLEMENT (2): Die Punkte werden, bevor sie gesetzt werden, mit den schon vorher gesetzten Punkte geXORt (Bitplane-weise).

INVERSVID (4): Die Punkte werden, bevor sie ausgegeben werden, invertiert, d.h. daß Punkte, die eigentlich gesetzt werden sollten, nicht gesetzt werden und umgekehrt (Dies geschieht auch Bitplane-weise).

Die Modi COMPLEMENT und INVERSEVID funktionieren nur in Verbindung mit JAM1 oder JAM2.

SetFont (&RastPort, &Font)

SetFont() macht einen vorher mit OpenFont() oder OpenDiskFont() geöffneten Font für den angegebenen Rastport zugänglich, das heißt, daß die Zeichen, die dann mit Text() ausgegeben werden, ein neues Aussehen bekommen.

Siehe: OpenFont(), CloseFont()

SetRast (&RastPort, Farbregister)

Mit diesem Befehl werden alle Punkte der Bitmap des angegebenen Rastports mit der Farbe des angegebenen Farbregisters der Colormap des Viewports, in dem der Rastport erscheinen soll, gesetzt.

Ist 'Farbregister' z.B. gleich 0, so werden alle Punkte gelöscht bzw. mit der Farbe des 0. Farbregisters (= Hintergrundfarbe) der Colormap des Viewports dargestellt.

SetRG4CM (&ColorMap, Farbregister, Rot, Grün, Blau)

Dieser Befehl wirkt ähnlich wie SetRGB4. Nur wird hier die Farbänderung nicht sofort sichtbar. Dies geschieht erst, wenn die Copper-Listen z.B. nach RemakeDisplay neu berechnet und dargestellt wurden.

SetAPen (&RastPort, Farbregister)

Mit dieser Funktion legen Sie die Farbe des APens im angegebenen Rastport neu fest. 'Farbregister' gibt dabei die Nummer des Farbregisters in der Colormap des Viewports an.

Siehe: SetBPen()

SetBPen (&RastPort, Farbregister)

Auch hier wird - wie bei SetAPen() - die Farbe des BPens im angegebenen Rastport neu festgelegt. Auch hier wird wieder die Nummer des Farbregisters, mit dem von nun an die Punkte dargestellt werden sollen, mit 'Farbregister'angegeben.

Jedoch unterscheiden sich APen und BPen in ihrer Funktion, je nachdem, welcher Zeichenmodus eingestellt wurde. Im JAM2-Modus hat der BPen die Funktion des Hintergrundfarbstiftes. Texte werden dann z.B. mit der Farbe des BPens unterlegt. In den anderen DrawModes ist der BPen nicht mehr von Bedeutung.

Siehe: SetAPen()

SetCollision (Nummer, Routine, &GelsInfo)

Dieser Befehl legt die Kollisionsroutine fest, die beim Zusammenstoß zweier GELs bzw. eines GELs mit dem Rand nach Untersuchung der 'MeMask' und 'HitMask' ausgeführt werden soll.

Dabei ist zu beachten, daß bei GEL-GEL-Kollisionen Ihrer Routine die beiden VSprite-Strukturen der an der Kollision beteiligten GELs übergeben werden. Die erste dieser VSprite-Strukturen repräsentiert dabei das weiter links oben liegende GEL (Bob oder VSprite).

Bei Rand-Kollisionen wird Ihrer Routine die beteiligte VSprite-Struktur, sowie ein Flag übergeben. Dieses Flag enthält den Code für den Rand (TOPHIT, BOTTOMHIT, LEFTHIT, RIGHTHIT), mit dem das GEL zusammengestoßen ist.

Siehe: DoCollision()

SortGList (&RastPort)

Diese Routine veranlaßt, daß die GEL-Liste des angegebenen Rastports sortiert wird. Dazu werden die in ihr enthaltenen VSprite-Strukturen, die sowohl von den VSprites selber als auch von den Bobs zur Positionierung und Darstellung benutzt werden, nach ihren Koordinaten sortiert, wobei die Y-Koordinaten stärker ins Gewicht fallen als die X-Koordinaten (Bei gleichen Y-Koordinaten wird nach X-Koordinaten sortiert).

Nach SortGList() kann dann die Liste mittels DrawGList() neu gezeichnet werden.

Siehe: DrawGList()

Text (&RastPort, &"String", AnzZeichen)

Mit dieser Funktion können Sie einen beliebigen Text an der aktuellen Grafik-Cursor-Position in dem angegebenen Rastport ausgeben. Dazu übergeben Sie einfach die Anfangsadresse des auszugebenden Strings und die Anzahl der Zeichen, die dieser String enthält bzw. die ausgegeben werden sollen.

Siehe: TextLength()

Länge = **TextLength** (&RastPort, &"String", AnzZeichen)

Diese Funktion hat dieselben Parameter wie Text(), nur wird die Länge, d.h. die Anzahl der Punkte, die der auszugebende Text breit ist, zurückgegeben. Dabei wird der augenblicklich im Rastport aktuelle Font zu Rate gezogen.

Siehe: Text(), SetFont()

Position = **VBeamPos** ()

Diese Funktion liefert in 'Position' die augenblickliche, vertikale Position des Elektronenstrahls. Zurückgegeben werden Werte zwischen 0 und 255. Deshalb, und wegen der Tatsache, daß der Elektronenstrahl tatsächlich 262 Linien abtasten kann, werden Elektronenstrahl-Positionen zwischen 256 und 262 als Werte zwischen 0 und 6 zurückgegeben.

Leider ist es aber so, daß der Wert, der von VBeamPos() zurückgegeben wird, nicht mehr aktuell zu sein braucht, wenn Ihr Programm ihn

Die Farbänderung geschieht erst in der angegebenen Colormap, bevor Sie veranlassen, daß diese sichtbar wird.

Dieser Befehl eignet sich ähnlich wie LoadRGB4 dazu, eine ColorMap im Hintergrund mit neuen Werten zu initialisieren.

Siehe: LoadRGB4()

SetRGB4 (&ViewPort, Farbregister, Rot, Grün, Blau)

Mit diesem Befehl können Sie die Farbe eines Farbregisters in dem angegebenen Viewport verändern. Dazu übergeben Sie die Nummer des zu verändernden Farbregisters sowie die neue Rot-, Grün- und Blaukomponente (0-15) der neuen Farbe, die dieses Farbregister enthalten soll.

SetRGB4() verändert auch den Farbeintrag sowohl in der ColorMap-Struktur des Viewports als auch in der Copper-Liste, so daß die Farbänderung sofort zu erkennen ist.

Siehe: LoadRGB4(), GetRGB4()

NeuerStyle = **SetSoftStyle** (&RastPort, Schriftart, Style)

Diese Funktion setzt die gewünschte, algorithmisch zu generierende Schriftart des Fonts im Rastport. Dazu setzt man in 'Schriftart' das gewünschte Schriftarten-Bit (s. AskSoftStyle) und in 'Style' den Wert, den AskSoftStyle() zurückgegeben hat (= die Schriftarten, die mittels SetSoftStyle() noch gesetzt werden können). Denn es ist z.B. möglich, daß ein Font schon von vorneherein aus kursiven Zeichen besteht. Somit würde es keinen Sinn machen, mit SetSoftStyle() nochmals Kursivschrift algorithmisch generieren zu wollen.

SetSoftStyle() verändert die Schriftart also nur dann, wenn die Schriftarten-Bits auch in 'Style' gesetzt sind. 'NewStyle' enthält dann die Schriftarten, die nach SetSoftStyle() tatsächlich generiert werden.

Siehe: AskSoftStyle()

Nun zu den beiden DiskFontBase-Befehlen (zuerst muß dafür natürlich die Library "diskfont.library" geöffnet werden):

Fehler = **AvailFonts** (&Buffer, BufferGröße, Typ)

Mit dieser Funktion können Sie sich eine Liste aller dem gesamten System zugänglichen Fonts ausgeben lassen. Diese Liste wird in den von Ihnen angegebenen Speicherbereich (&Buffer), der 'BufferGröße' Bytes enthält, geschrieben.

Das erste, was in diesem 'Buffer' abgespeichert wird, ist der 'AvailFontsHeader'. In ihm ist die Anzahl der Font-Einträge abgespeichert (in 'AvailFontsHeader.afh_NumEntries). Danach folgen die verschiedenen 'AvailFonts'-Einträge, die sowohl den Typ des Fonts - ob er in der Systemfontliste (AFF_MEMORY) oder auf der Diskette (AFF_DISK) zu finden ist - als auch die den Font näher spezifizierende TextAttr-Struktur enthalten.

Diese TextAttr-Struktur kann dann, je nach Typ (AFF_MEMORY / AFF_DISK), für das Öffnen eines Fonts mittels OpenDiskFont() oder OpenFont() benutzt werden.

Der AvailFonts()-Parameter 'Typ' bestimmt, ob die Systemfontliste (AFF_MEMORY) oder das 'SYS:Fonts'-Unterverzeichnis (AFF_DISK), oder ob beides (AFF_MEMORY|AFF_DISK) nach den verfügbaren Fonts durchsucht werden soll.

Beim Öffnen eines Fonts mittels OpenDiskFont(), mit der von AvailFonts() zurückgelieferten TextAttr-Struktur, sollten Sie darauf achten, daß AvailFonts die aus dem 'SYS:Fonts'-Unterverzeichnis extrahierte TextAttr-Struktur nicht auf ihren Wahrheitsgehalt untersucht. OpenDiskFont() kann also eventuell nicht das gewünschte Ergebnis liefern bzw. dazu führen, daß der Rechner abstürzt.

Das Ergebnis, das AvailFonts() liefert, gibt Auskunft darüber, ob genügend Speicher zur Verfügung stand, um alle 'AvailFonts'-Strukturen beinhalten zu können. Bei 'Fehler' = 0 ist dies der Fall. Ist 'Fehler' allerdings ungleich 0, dann gibt 'Fehler' die Anzahl der Bytes an, die gefehlt haben, um alle 'AvailFonts'-Strukturen beinhalten zu können.

bekommen hat, denn die Multitasking-Fähigkeit des Amiga bringt nicht nur Vorteile, sondern kann Ihre Programme auch entscheidend verzögern, so daß zwischen dem Aufruf von VBeamPos() und dem Zuweisen des Rückgabewertes an die Variable die entscheidenden Millisekunden für ein anderes Programm benötigt werden.

WaitBlit ()

Diese Funktion hält Ihr Programm solange an, bis der Blitter mit seiner Aufgabe fertig ist. Doch Vorsicht: Aufgrund eines Prozessorfehlers kann es passieren, daß WaitBlit() früher 'zurückkommt' als erwartet, d.h. daß der Blitter noch gar nicht angefangen hat, seine Aufgabe zu bearbeiten, Ihr Programm aber trotzdem schon weiterläuft. Dies ist besonders dann der Fall, wenn Sie Ihr System in Hi-Res mit 4 Bitplanes laufen lassen.

WaitBOVP (&ViewPort)

Diese Funktion hält Ihr Programm solange an, bis der Elektronenstrahl, der das Bild aufbaut, die letzte Zeile des angegebenen Viewports erreicht hat (Bottom Of ViewPort).

Siehe: WaitTOF()

WaitTOF ()

Dieser Befehl funktioniert ähnlich wie WaitBOVP(), nur wird hier nicht auf das 'Ende' eines Viewports gewartet, sondern solange, bis der Elektronenstrahl den 'Vertikal Blank' (Top Of Frame), also den vertikalen Strahlrücklauf durchgeführt und die damit in Verbindung stehenden, zyklisch ausgeführten Routinen (Interrupts) abgearbeitet hat.

Siehe: WaitBOVP()

WritePixel (&RastPort, X, Y)

Diese Funktion setzt einen Punkt in der Farbe des APens im angegebenen Rastport an der Koordinate X,Y. Dabei wird natürlich der aktuelle Zeichenmodus des Rastports berücksichtigt.

Siehe: ReadPixel()

Jetzt folgen die IntuitionBase-Routinen (IntuitionBase ist der Zeiger auf die Intuition-Library, die mit 'IntuitionBase = (struct IntuitionBase *) OpenLibrary ("intuition.library",0)' geöffnet wird).

CloseScreen (&Screen)

Mit dieser Funktion schließen Sie einen vorher mit OpenScreen() geöffneten Sreen. Dabei werden der für die Bitmap belegte Speicher sowie alle für den Intuition-Screen benötigten Strukturen (Viewport, Rastport, etc.) wieder freigegeben.

Wird mit CloseScreen() der letzte vom Benutzer geöffnete Screen geschlossen, wird automatisch der Workbench-Screen wieder aktiv, wobei CloseScreen() nicht darauf achtet, ob alle Windows des Screens geschlossen wurden. Ist dies nicht der Fall, verursachen nachträglich 'CloseWindow()'-Aufrufe Systemabstürze.

Siehe: OpenSreen()

CloseWindow (&Window)

Hiermit schließen Sie ein vorher mit 'OpenWindow()' geöffnetes Window. Bitte beachten Sie, daß Sie alle Windows eines Screen schließen, bevor Sie dazu übergehen, den Screen des Windows zu schließen.

Siehe: OpenWindow()

DisplayBeep (&Screen)

Da der Amiga keine eingebaute Möglichkeit besitzt, um auf kleinere Fehler hinzuweisen, läßt man dazu einfach einen Sreen aufblitzen, indem die Hintergrundfarbe des Screens kurzzeitig verändert wird.

Geben Sie allerdings 0 anstatt der Adresse des Sreens an, werden alle Intuition-Screens 'geblitzt'. Dies sollten Sie sich aber für schwerwiegendere Fehler vorbehalten.

MakeScreen (&Screen)

Diese Funktion führt MakeVPort() für den Viewport des angegebenen Screens durch. Somit können nachträgliche Änderungen in den Viewports der Intuition-Sreens (z.B. Farbänderungen mit LoadRGB4()) auch nachträglich dem Copper 'mitgeteilt' werden.

*TextFont = **OpenDiskFont** (&TextAttr)

Diese Funktion öffnet den mit der TextAttr-Struktur näher spezifizierten Font. Dabei wird das 'SYS:Fonts'-Directory nach dem Font durchsucht und derjenige zurückgegeben, der der TextAttr-Struktur in SChriftart und Größe am besten entspricht. Konnte kein Font mit dem in der TextAttr-Struktur festgelegten Namen gefunden werden, wird 0 zurückgegeben.

Siehe: OpenFont()

Hier nun die 'DosBase'-Routinen (Bitte beachten Sie, daß die 'DosBase' sowie die 'ExecBase' nicht extra geöffnet zu werden brauchen, da diese in C-Programmen immer geöffnet sind):

Delay (Zeit)

Diese Routine verzögert Ihr Programm um 'Zeit' * 1/50 Sekunden. Dabei werden allerdings nicht die anderen, gleichzeitig im Speicher ablaufenden Programme (Tasks) angehalten, wie es z.B. bei einer Verzögerung mit 'for()' der Fall wäre.

Exit (ReturnWert)

Mit diesem Befehl sind Sie in der Lage, ein Programm zu jedem Zeitpunkt, auch aus Rekursionsschleifen, zu verlassen. Wenn Sie Ihr Programm dabei als 'Overlay-Prozess' ablaufen lassen haben, wird 'ReturnWert' an Ihren 'Haupt-Prozess' übergeben. Allerdings haben wir die Technik des Programm-Overlays in unseren Beispielen nicht benutzt, so daß der angegebene Rückgabewert mehr Symbolcharakter hat.

Siehe: MakeVPort()

ModifyIDCMP (&Window, IDCMPFlags)

Mit dieser Funktion verändern Sie die IDCMPFlags (Intuition Direct Communication Message Ports) eines Windows. Diese sind für die verschiedenen Nachrichten, die von Intuition an Ihr Window gesendet werden, von Bedeutung. Doch alle Möglichkeiten dieser IDCMPs hier zu erläutern, würde den Rahmen dieses Anhangs sprengen.

OpenScreen (&NewScreen)

Diese Funktion öffnet einen Intuition-Screen, der mit der angegebenen NewScreen-Struktur näher bestimmt wurde. Dieser Befehl legt dafür einen Viewport sowie alle anderen zur Ausgabe benötigten Strukturen (Bitmap, Rastport, etc,) an. Dieser neue Screen wird dann dem übergeordneten Intuition-View hinzugefügt und dargestellt.

Siehe: CloseScreen()

RemakeDisplay ()

Diese Funktion berechnet für alle Intuition-Screens die Copper-Listen neu. Das heißt, daß für jeden Screen MakeScreen() und danach RethinkDisplay() aufgerufen wird. Allerdings sollten Sie beachten, daß bei Ausführung des Befehls für kurze Zeit das Multitasking abgeschaltet wird.

Siehe: MakeScreen(), RethinkDisplay()

RethinkDisplay ()

Diese Funktion führt für den Intuition-View 'MrgCop()' und 'LoadView()' aus. Allerdings wird hierfür kurzeitig das Multitasking abgeschaltet.

Siehe: MakeScreen(), RemakeDisplay(), MrgCop(), LoadView()

Message = **GetMsg** (&Port)

Mit diesem Befehl können Sie eine Nachricht (Message) von Intuition empfangen. Dabei wird solange gewartet, bis tatsächlich eine Nachricht gesendet wurde.

Siehe: ReplyMsg()

OpenLibrary (&LibName, Versionsnummer)

Mit diesem Befehl öffnen Sie die angegebene Library, mit der Sie dann Zugriff auf bestimmte Betriebssystem-Routinen haben.

Wollen Sie z.B. die Grafikbefehle benutzen, müssen Sie mit 'GfxBase = (struct GfxBase *)OpenLibrary ("graphics.library",0)' die Grafik-Bibliothek öffnen. 'GfxBase' ist dabei der Zeiger auf die zugehörige Bibliothek, der zu Ende Ihres Programms an CloseLibrary() übergeben werden muß.

Wenn Sie die aktuelle Versionsnummer der Library nicht kennen, brauchen Sie nur 0 anzugeben. Dann wird nämlich die gerade aktuelle Bibliothek benutzt. Wenn aber z.B. ein Befehl nur in einer bestimmten Version einer Bibliothek zu finden ist, sollten Sie deren Versionsnummer angeben und evtl. frühzeitig das Programm beenden, wenn diese Bibliothek nicht gefunden werden konnte.

ReplyMsg (&Port)

Mit diesem Befehl 'sagen' Sie z.B. Intuition, daß Sie die von ihm gesandte Nachricht empfangen und untersucht haben.

Nun folgen die 'ExecBase'-Routinen:

*Speicher = **AllocMem** (AnzBytes, Anforderung)

Mit dieser Routine können Sie 'AnzBytes' Bytes aus dem Speicher Ihres Rechners für den Eigengebrauch deklarieren. Allerdings müssen Sie bestimmen, wozu Sie den Speicher 'gebrauchen wollen'. Wenn Sie nämlich als Anforderung 'MEMF_CHIP' (MEMF = MemoryFlag) angeben, wird der Speicher aus den unteren 512 KByte 'abgezogen'. Dies ist besonders dann wichtig, wenn dieser Speicher von allen Prozessoren des Amiga angesprochen werden muß (z.B. für Bitplanes etc.).

Bei 'MEMF_PUBLIC' als Anforderung wird der Speicherbereich 'von irgendwoher' genommen. Er kann also auch im Bereich über 512 KByte liegen (natürlich nur dann, wenn Sie tatsächlich mehr als 512 KByte besitzen).

Mit 'MEMF_CLEAR' können Sie dann noch bestimmen, daß der Speicher sofort gelöscht werden soll. Sie bekommen also in 'Speicher' die Adresse eines gelöschten Speicherbereichs.

Siehe: FreeMem()

CloseLibrary (&BasePointer)

Mit diesem Befehl schließen Sie die vorher mit 'OpenLibrary()' geöffnete Library wieder. Dabei übergeben Sie 'CloseLibrary()' den Zeiger, den Sie von 'OpenLibrary()' geliefert bekommen haben.

Bitte achten Sie darauf, daß alle Libraries geschlossen werden, bevor SIe ein Programm verlassen!

Siehe: OpenLibrary()

FreeMem (&Speicher, Größe)

Diese Routine gibt einen vorher reservierten Speicherplatz wieder frei. Dazu übergeben Sie einfach die Anfangsadresse dieses Speicherbereichs und die Anzahl der Bytes, die belegt wurden.

Siehe: AllocMem()

---

in das Register geschrieben werden. Die gelöschten Bits dieses Words lassen die Bits des Registers dagegen unangetastet.

Ist dieses Bit 15 beim Schreiben gelöscht, heißt das, daß alle anderen Bits, die gesetzt sind, im Register gelöscht werden. Die Bits, die gleich 0 sind, bleiben auch hier im Register unangetastet.

Um das an einem Beispiel klar zu machen: Beim Schreiben von %1111111111111110 werden alle Bits außer Bit 0 des Registers gesetzt. Bei %0111111111111111 werden alle Bits außer Bit 0 gelöscht. Diese Art der Registerkontrolle wird uns noch häufiger begegnen, also versuchen Sie bitte jetzt schon, sich dieses ein wenig 'unorthodoxe' Vorgehen zu merken. Überall, wo ein CLR/SET-Bit auftaucht, wird wie oben erläutert vorgegangen.

Bit 14, 13 PRECOMP1/2: Diese Bits legen den Precomp für Diskettenoperationen fest. Mit dem Wert %00 in beiden Registern wählt man 0 ns. Der Wert %01 legt 140 ns, %10 legt 280 ns und %11 legt 560 ns fest.

Bit 12 MFMPREC: Mit diesem Bit läßt sich das Aufzeichnungs- bzw. Leseformat der Diskette festlegen. Ist das Bit gesetzt, so wird das für den Amiga übliche MFM-Aufzeichnungsformat gewählt. Bei gelöschtem MFMPREC-Bit wird das für Apple Computer übliche Aufzeichnungsformat (GCR) benutzt.

Bit 11 UARTBRK: Ist dieses Bit gesetzt, so werden alle Leitungen der RS-232-Schnittstelle auf 0 gesetzt (Universal Asynchronus Receiver/Transmitter Break).

Bit 10 WORDSYNC: Ist dieses Bit gesetzt, so wird die Datenübertragung über den DMA-Kanal gestartet und jedes gelesene Word mit dem Word in dsksync synchronisiert.

Bit 9 MSBSYNC: Dieses Bit bestimmt, ob das höchstwertigste Bit eines Words im GCR-Format synchronisiert werden soll.

Bit 8 FAST: Mit diesem Bit bestimmt man die Zeit für das Lesen eines Bits von Diskette. Ist es gesetzt, so werden 2 µs für eine Bitzelle festgesetzt. Diese Zeit wird für das MFM-Aufzeichnungsformat benötigt. Der Wert 0 dieses Bits legt 4 µs für eine Bitzelle fest, was für das Lesen von GCR (Apple-Format) Disketten gebraucht wird.

Anhang C: Die Hardware-Register

In diesem Anhang wollen wir Ihnen einen Überblick über die Hardware-Register des Amiga geben. Diese sind insbesondere bei der Copper-Programmierung von Nutzen, denn sie können mit dessen Hilfe leicht beschrieben werden.

Wie bekannt, wird dazu im CMOVE()-Befehl die Nummer des zu beeinflussenden Registers angegeben. Wollen Sie aber mit dem 68000er auf diese Register zugreifen, müssen Sie beachten, daß Sie eigentlich auf bestimmte Speicherstellen des Amiga zugreifen.

Die Hardware-Register befinden sich im Speicherbereich ab $DFF000. Das bedeutet, daß Sie diesen Wert zum angegebenen Offset der einzelnen Register addieren müssen, um mit dem 68000er auf das Register zugreifen zu können.

Wollen Sie von 'C' aus auf die Hardware-Register zugreifen, hilft Ihnen die 'Custom'-Struktur, die mit 'extern struct Custom custom' angelegt werden muß. Auf die einzelnen Register greifen Sie mit 'custom.NAME' sowohl mit dem Copper als auch mit dem 68000er zu.

Im folgenden werden wir diese Registernamen (Wie sie im Include-File 'hardware/custom.h' definiert sind) sowie die Offsets der einzelnen Register zu $DFF000 angeben. So hat der 'C'- und auch der Maschinensprache-Programmierer ein komplettes Nachschlagewerk über die Register und weiß, wie er darauf zugreifen kann.

---

adkcon	$09E	Audio und Disk Kontrolle (Schreiben)
adkconr	$010	Audio und Disk Kontrolle (Lesen)

Dieses Register ist, wie Sie sehen können, in zweifacher Ausführung vorhanden. Einmal kann es nur gelesen werden (adkconR), das heißt, ein Beschreiben hat keinen Sinn, und ein zweites Mal kann es nur beschrieben werden (adkcon).

Die Bitbelegung ist bei beiden Registern gleich, zu beachten ist aber, daß Bit 15, das beim Lesen keinen Sinn hat, beim Schreiben von enormer Bedeutung ist.

Bit 15 SET/CLR: Ist dieses Bit im zu schreibenden Word nämlich gesetzt, heißt das, daß alle anderen gesetzten Bits (14-0) dieses Words

aud[X].ac_len $0A4 Audio Kanal X Länge

Dieses Register enthält die Anzahl der Wörter, die über den DMA-Kanal zur Soundgenerierung geschickt werden sollen. Die Anfangsadresse dieser Wörter legen Sie mit oberem Register fest.

---

aud[X].ac_per $0A6 Audio Kanal X Sampling Frequenz

Mit diesem Register bestimmen Sie, wie viele Bytes pro Sekunde zur Soundgenerierung herangezogen werden. Der Sound-DMA-Kanal ist mit dem Bildschirm-DMA-Kanal synchronisiert und kann deshalb nur 2 Bytes pro Rasterzeile hörbar machen. So wären theoretisch 2 (Bytes) * 262.5 (Rasterzeilen) * 59.94 (Bildschirmaufbauten) = 31468.5 Bytes pro Sekunde hörbar zu machen.

Da der DMA-Controller noch andere Aufgaben außer der Soundgenerierung erledigen muß, sind 'nur' 28867 Bytes/s für das Sampling zu ermöglichen.

Das heißt, es ist möglich, in 1/28867 s = 34.642 µs ein Byte über den Soundkanal auszugeben. So ergibt sich für den Wert dieses Registers: ac_per = 34.642 µs / Anz_Mikrosekunden für eine Abtastung.

Da der Hardware die Begrenzung von einer Abtastung pro Taktzyklus auferlegt ist, ein Taktzyklus aber 280 ns = 0.28 µs dauert, bedeutet das, daß der Wert 124 (34.643 µs/280 ns) der kleinstmögliche Wert für dieses Register ist. Wählen Sie einen kleineren Wert, so kann der DMA-Kanal nicht mehr richtig arbeiten.

---

aud[X].ac_vol $0A8 Audio Kanal X Lautstärke

Mit diesem Register legen Sie die Lautstärke für den Audio-Kanal X fest. Es sind aber nur die untersten 5 Bits für die Lautstärke von Bedeutung.

So können Sie 64 als maximalen Lautstärkewert setzen, während bei dem Wert 0 nichts mehr zu hören ist.

---

aud[X].ac_dat $0AA Audio Kanal X Data

Dieses Register ist der Audio-DMA-Datenpuffer. Es enthält 2 Bytes im 2er-Komplement, die sequentiell über die Sound-Hardware ausgegeben werden.

Bit 7-4 ATPER3-0: Mit diesen Bits kann man die Abtastrate (Sampling) für die einzelnen Audio-Kanäle modulieren. Ist ATPER3 gesetzt, wird die Ausgabe über diesen Sound-Kanal gestoppt. Ist ATPER2 gesetzt, wird die Abtastrate von Audio-Kanal 3 mit der von Audio-Kanal 2 moduliert. Ist ATPER1 gesetzt, wird die Abtastrate von Audio-Kanal 2 mit der von Audio-Kanal 1 moduliert. ATPER0 setzt die Modulation der Abtastrate von Audio-Kanal 1 mit der von Audio-Kanal 0.

Bit 3-0 ATVOL3-0: Diese Bits bestimmen die Lautstärken-Modulation der einzelnen Audio-Kanäle. Das Muster der Modulation ist wie bei ATPER3-0, nur daß hier die Lautstärke moduliert wird.

---

aud[X].ac_ptr $0A0 Audio Kannal X Startadresse

Hier wird die Startadresse der Daten, die über den Sound-DMA-Kanal (Direct memory acces = Speicherzugriff ohne Hilfe des 68000ers) übertragen werden sollen, festgelegt.

Eigentlich ist dies ein Registerpaar, das in $0A2 die unteren 15 Bits und in $0A0 die oberen 3 Bits der Startadresse enthält. Aber aufgrund der 'Custom' Deklaration im Include-File wird für den Pointer *ac_ptr vom Compiler aus schon ein Long-Word reserviert.

Wollen Sie auf dieses Register(paar) zugreifen, müssen Sie in der Sprache 'C' als Index eine Zahl zwischen 0 und 3 für den jeweiligen Audio-Kanal (aud[X]...) angeben.

Machinensprache-Programmierer müssen jedoch immer X*12 Words addieren, um an die Register für den jeweiligen Soundkanal heranzukommen, denn diese beiden bilden mit den folgenden 5 Registern einen Block, der in vierfacher Ausführung hintereinander angelegt worden ist. Im Anschluß hieran finden Sie jedoch die vollständige Belegung der Register-relevanten Speicherstellen ab $DFF000 siehe Tabelle am Ende dieses Anhangs).

Bitte beachten Sie, daß hier nur 18 Bits für die Adresse ausschlaggebend sind, da nur die untersten 512 KByte vom Sound-Chip wie auch vom Blitter adressiert werden können.

Dazu gibt man für jede zu bearbeitende Bitmap die Breite in Bytes als Modulo-Parameter an. So werden die zu 'blittenden' Rechtecke aus der einen Bitmap 1:1 in eine andere übertragen.

Bitte beachten Sie, daß der Blitter nur Bitplane-weise die Daten verarbeiten kann. Das heißt also, wenn Sie selber den Blitter ansteuern und einen rechteckigen Bereich aus einer Bitmap in eine andere Bitmap übertragen wollen, müssen Sie für jede Bitplane die Position des Rechtecks mit 'bltxpt' angeben und Bitplane-weise 'blitten'. Der Modulo wird dann zum letzten gelesenen und geschriebenen Wort einer Zeile addiert, um an den Anfang der nächsten Zeile zu gelangen.

Aber warum sollten Sie es sich unnötig schwer machen? Die Leute von Commodore haben nämlich leistungsfähige Befehle (BltBitMap(), BltTemPlate(), ScrollRaster()) zur Ausnutzung des Blitters entwickelt.

Dasselbe gilt für das Ziehen von Linien, denn auch dabei haben die Modulo-Register eine bestimmte Bedeutung, aber wie gesagt, der Draw()-Befehl nimmt uns die ganze Arbeit ab.

---

bltafwm $044 BlitterFirst-Word Maske für Quelle A
bltalwm $046 BlitterLast-Word Maske für Quelle A

Wie kann man man aber Rechtecke blitten, die nicht an einer Word-Adresse, sondern mitten innerhalb eines Words beginnen? Nun, dazu wird einfach mit Hilfe dieses Registers das erste und das letzte Wort einer zu blittenden Zeile (der Quelle A) mit einem bestimmten Wert geANDet. Die Bits, die dann noch übrigbleiben, werden verändert und dann bearbeitet und ins Ziel geschrieben.

---

bltXdat $074 Blitter X Datenregister

In diese Register werden die einzelnen Wörter, ähnlich wie bei der Soundgenerierung, aus Quelle a, b, c, bevor sie miteinander verknüpft werden, gelesen. Dies übernimmt der DMA-Kanal. Nach dem Blit dieser 3 Words wird dann das resultierende Word in bltddat geschrieben und danach vom DAM-Kanal ans Ziel geschrieben. Wie Sie sehen, wird der ganze Blitvorgang auf das Blitten dreier Words reduziert.

Für das Ziehen von Linien mit dem Blitter gilt das Übliche.

Der DMA-Controller schreibt automatisch die Werte, die er über 'ac_ptr' liest, in dieses Register. Man kann aber auch 'von Hand', also mit dem 68000er, Werte in dieses Register schreiben und so Sound ohne die Benutzung eines DMA-Kanals generieren. Hierbei sollten Sie allerdings beachten, daß Sie den DMA-Zugriff nicht ermöglichen dürfen, da sonst das Interrupt-Timing durcheinander kommt.

---

bltXpt $050 Blitter Pointer

Diese Pointer (Registerpaare) enthalten einen 18-Bit-Zeiger auf die DMA-Daten, die der Blitter bearbeiten soll. X steht dabei für a, b, c und d, was die drei Quellen (a, b, c) und das Ziel (d) der Blitter-Operationen (Blits) symbolisieren soll.

Aus bltapt, bltbpt und bltcpt werden also die Daten für einen 'Blit' in den Blitter gelesen, bearbeitet und nach 'bltdpt' abgespeichert.

Hat der Blitter seine Aufgabe erledigt, enthalten die Pointer die letzte Daten-Adresse (plus Inkrement und Modulo), an der Daten geschrieben bzw. gelesen wurden. Wollen Sie das Ziel (d) mit in den 'Blit' einbeziehen, muß ein Zeiger der Quellen a, b oder c auf dieselbe Adresse wie das Ziel d zeigen.

Der Blitter ist aber nicht nur für das 'Blitten', sprich Manipulieren des Speichers, zuständig. Er hat auch die Aufgabe, Linien zu ziehen. Da diese Aufgabe extra in dem Prozessor integriert wurde, versteht sich von selbst, daß der Amiga beim Ziehen von Linien unschlagbar schnell ist.

Wie die Register 'bltapt', 'bltbpt', 'bltcpt' und 'bltdpt' das Ziehen von Linien beeinflussen, soll uns jedoch nicht weiter interessieren, denn der Draw()-Befehl wurde ja extra zu dessen Ansteuerung entwickelt. Warum Sie also mit unnötiger Theorie belasten, da Sie das Ziehen von Linein dadurch keineswegs beschleunigen könnten?

---

bltXmod $064 Blitter Modulo X

Dieses Register enthält den Modulo für die Blitter Quellen a, b und c und das Blitter Ziel d. Aufgrund der Tatsache, daß verschieden große Bitmaps existieren können, zwischen denen hin und her 'geblittet' wird - ein 'Blit' ist nämlich nicht nur auf eine einzige Bitmap beschränkt - muß man wissen, wie groß jede Bitmap, die vom Blitter bearbeitet werden soll, ist.

Bits: LF	7	6	5	4	3	2	1	0
Minterm:	ABC	$AB\overline{C}$	$A\overline{B}C$	$A\overline{B}\overline{C}$	$\overline{A}BC$	$\overline{A}B\overline{C}$	$\overline{A}\overline{B}C$	$\overline{A}\overline{B}\overline{C}$

ABC: D = A and B and C

$AB\overline{C}$: D = A and B and !C

$A\overline{B}C$: D = A and !B and C

$A\overline{B}\overline{C}$: D = A and !B and !C

$\overline{A}BC$: D = !A and B and C

⋮

bltcon1 hat im Area-Modus folgende Bitbelegung:

Bit 15-12 BSH3-0: Diese Bits haben dieselbe Bedeutung wie ASH3-0 bei bltcon0. Nur wird hier Quelle b 'geshiftet'.

Bit 11-5: Diesen Bits haben die Leute von Commodore (noch) keine Bedeutung zukommen lassen.

Bit 4, 3 EFE/IFE: Daß Sie mit dem Blitter auch Flächen füllen können, haben wir an anderer Stelle schon erwähnt. Diese beiden Bits bestimmen den Füllmodus, indem ein vorher festgesetzter Bereich vom Blitter gefüllt wird: Ist das Bit EFE gesetzt (Exclusive Fill Enable), dann werden die Begrenzungslinien der Fläche, die links vom Rand des Füllbereichs liegen, nach dem Füllen gelöscht, während bei gelöschtem EFE-Bit und bei gesetztem IFE (Inclusive Fill Enable)-Bit die Fläche ganz normal gefüllt wird. Damit die Fläche 'sauber' gefüllt wird, müssen die Begrenzungslinien eine bestimmte Voraussetzung erfüllen: Auf jeder Horizontallinie darf nur ein einziges Pixel gesetzt sein (s. SING).

Bit 2 FCI: Dieses Bit ist der Startwert des 'Füll-Flip-Flops'. Wie bei einem elektronischen Flip-Flop schaltet auch hier der Blitter beim Füllen zwischen zwei Zuständen hin und her, wenn ein bestimmtes äußeres Ereignis eintritt, und behält diesen Zustand solange bei, bis ein erneutes Ereignis derselben Art eintritt.

Für den Blitter sieht solch ein Ereignis wie folgt aus: Entdeckt er ein gesetztes Bit innerhalb einer Bitplane-Zeile, wird der Status des 'Füll Flip-Flops' gewechselt. Alle weiteren Bits, die gelöscht sind, werden dann mit dem Wert des FCI beschrieben. Somit kann man also bestimmen, ob Flächen 'innen' oder 'außen' gefüllt werden.

bltcon0	$040	Blitter Kontrolregister 0
bltcon1	$042	Blitter Kontrolregister 1

Diese beiden Register werden zur Kontrolle der Blitter-Operationen ('Blits') benutzt. Dabei gibt es zwei Modi, Area und Line, die durch Bit 0 des Register BLTCON1 ausgewählt werden:

Der *Area Modus*

bltcon0

Bit 15-12 ASH3-0: Haben Sie schon einmal ein Bob Pixelweise bewegt? Wenn ja, haben Sie sich sicher auch schon gefragt, wie das möglich ist, da der Blitter ja nur auf Word-Adresssen zugreifen kann.

Aber mit Hilfe der Bits ASH3-0 können Sie bestimmen, um wie viele Punkte die Daten aus Quelle A rotiert werden sollen, bevor sie ans Ziel geschrieben werden.

Bit 11-8 USEx: Diese Bits bestimmen, welche Quellen der Blitter ansprechen kann. Haben Sie zum Beipiel nur 2 Quellen, ist es doch vollkommen sinnlos, trotzdem auf alle zugreifen zu wollen, da dort vielleicht gar keine sinnvollen Daten stehen, oder?

Doch mit den Bits USEx, wobei das X stellvertretend für a, b, c und d steht, können Sie eine Auswahl der Quellen treffen. Beachten sollten Sie aber, daß Sie das Ziel immer aktivieren sollten, denn wozu denn überhaupt einen 'Blit' durchführen, wenn uns das Ergebnis verlorengeht?

Bit 7-0 LF7-0: Mit diesen 8 Bits bestimmen Sie die Art, wie der Blitter die Daten aus den Quellen miteinander verknüpft. Die Bitmuster, die Sie in diese Bits hineinschreiben können, nennt man auch 'Minterms'. Diese haben Sie schon im Zusammenhang mit den Blitter-Befehlen kennengelernt, allerdings wurden immer nur zwei Quellen (b und c) betrachtet.

Doch es stehen Ihnen drei Quellen zur Verfügung. Was mit welcher Quelle passieren soll, bestimmen die einzelnen Bits LF7-0:

Bit 4-2 SUD, SUL, AUL: Auch hier konnten wir nicht genügend Informationen zusammentragen. Diese Bits werden für das 'superschnelle' Ziehen der Linien verwendet. Diese superschnelle Methode basiert auf den Symmetrie-Eigenschaften einer Linie, die vom Blitter (auch auf irgendeine verzwickte Art und Weise) ausgenutzt werden (sollen?). Mit diesen 3 Bits, die voll ausgeschrieben die Namen 'Sometimes Up or Down', 'Sometimes Up or Left' und 'Always Up or Left' haben, steuert man das Zeichnen einer Linie. Aber ob sie vom Benutzer oder gar vom Blitter selbst (vielleicht nur als Flags) gesetzt werden, konnten wir nicht in Erfahrung bringen.

Bit 1 SING: Wenn dieses Bit gesetzt ist, dann wird jede Linie nur noch durch jeweils einen Punkt auf jeder Bitplane-Zeile dargestellt. Das ist besonders beim Füllen von Flächen mit dem Blitter wichtig.

Bit 0 LINE: (=1) Dieses Bit bestimmt den Modus des Blitters. Ist es gesetzt, arbeitet der Blitter im LINE-Modus.

---

bltsize $058 Blitter Start und Größe (Breite, Höhe)

Dieses Register enthält die Höhe und Breite des Blitter-Operations-Bereichs. In diesem definierten Window werden alle Operationen getätigt. Die Anfangsadresse wird mit bltXpt festgelegt, die Operationen mit bltcon0/1 und anderen Registern.

Wenn Sie auf dieses Register zugreifen (mit dem Copper oder 68000er), egal ob Sie es bloß lesen wollen oder beschreiben, legt der Blitter mit seinem 'Blit' los.

Deshalb sollte man dieses Register erst dann beschreiben, wenn alle anderen Register (BLTAF/LWM, BLTxDAT, BLTCON0/1, BLTxMOD etc.) initialisiert worden sind. Auch beim Ziehen von Linien wird dann erst 'gestartet'.

Bit 15-6 H9-H0: Diese 10 Bits, die die Werte 0 - 1024 festlegen können, geben die Anzahl der Zeilen des Blitter-Operations-Bereichs an.

Bit 5-0 W5-W0: Diese Bits geben die Breite des Blitter-Operations-Bereichs in Words (64 * 16 Bits = 1024) an. Wie Sie sehen, kann der Blitter also theoretisch einen Speicherbereich von der Größe 1024 x 1024 Punkten (Superbitmap) bearbeiten.

Bit 1 DESC: Dieses Bit gibt die Richtung an, in der der Blitter den angegebenen Datenbereich (Quelle a, b, c, d) bearbeitet. Ist DESC gesetzt, wird von der gegebenen Adresse abwärts gearbeitet. Ist es gelöscht, arbeitet sich der Blitter 'hoch'. Dieses Bit ist besonders dann wichtig, wenn sich Ziel und Quelle überschneiden.

Bit 0 LINE: (=0) Ist dieses Bit gelöscht, dann arbeitet der Blitter im Area-Modus.

Beim *Ziehen von Linien* bekommen alle diese Bits eine neue Bedeutung:

bltcon0

Bit 15-12 START3-0: Diese Bits geben den Code für die horizontale Position des ersten Pixels der Linie an. Wie dieser genau auszusehen hat, wurde uns von Commodore leider nicht 'überliefert' (das ist einer der Gründe, warum wir nicht so weitgehend auf das Thema 'Linien ziehen mit dem Blitter' eingehen wollen, da wir sonst vielleicht nicht vermeiden könnten, Unwahrheiten zu berichten).

Bit 11, 10, 9, 8 %1011: Diese Bits müssen mit dem Wert 1011 initialisiert werden. Warum? Nur Vater Commodore weiß Bescheid!

Bit 7-0 LF7-LF0: Die Blitter-Quellen werden auf irgendeine verschwommene Art und Weise zum Zeichnen von gemusterten Linien 'mißbraucht', aber wie schon gesagt: Auf diesem Gebiet konnten wir nicht genügend Informationen zusammentragen.

bltcon1

Bit 15-12 TEXTURE3-0: Der Wert (0-15) in diesen 4 Bits gibt an, mit welchem Bit die Musterung der Linie begonnen werden soll!?!

Bit 11-7 %00000: Diese Bits müssen alle gelöscht werden.

Bit 6 SIGN: Dieses Bit muß das Vorzeichen der Steigung der Linie enthalten.

Bit 5 %0: Dieses Bit ist für einen neuen Modus reserviert und sollte deshalb, um die Aufwärtskompatibilität der Programme zu anderen Maschinen (Amiga 2000 etc.) zu garantieren, gelöscht werden.

bpl1mod	$108	Bit Plane Modulo (ungerade Planes)
bpl2mod	$10A	Bit Plane Modulo (gerade Planes)

Diese Register enthalten den Modulo (die Breite) für die geraden (2, 4, 6) und ungeraden (1, 3, 5) Bitplanes (s. DBLPF und bplXmod). Diese Modulos sind für die richtige Bildschirmdarstellung besonders wichtig.

---

bplcon0	$100	Bit Plane Kontroll Register
bplcon1	$102	Bit Plane Kontroll Register
bplcon2	$104	Bit Plane Kontroll Register

Natürlich können Sie so, wie Sie den Blitter kontrollieren konnten, auch den Videoshifter kontrollieren. Dazu dienen diese 3 Register. Sie sind das Herz der gesamten Grafik-Darstellung.

bplcon0

Bit 15 HIRES: Ist dieses Bit gesetzt, erfolgt die Darstellung im hochauflösenden Modus (640 * ... Punkte). In diesem Modus sind aber nur 16 Farben möglich (4 Bitplanes). Diese Farben werden über die untersten 16 Farbregister (s. COLORxx) bestimmt.

Bit 14-12 BPU2-0: Diese Bits geben an, wie viele Bitplanes von Ihnen benutzt werden (0-6). Sechs Bitplanes werden nur für Dual Playfields (DBLPF) und im Hold-and-Modify- (HAM) und Extra-Halfbrite-Modus benötigt. Die Anzahl der Bitplanes bestimmt die Anzahl der Farben: Farben = 2^{BPU}. So ist es möglich, ohne Hold-and-Modify, mit 5 Bitplanes maximal 32 Farben darzustellen. Setzen Sie alle diese Bits auf 0, dann wird nur die Hintergrundfarbe (COLOR00) dargestellt.

Bit 11 HAM: Mit diesem Bit wird der Hold-and-Modify-Modus eingeschaltet.

Bit 10 DBLPF: Mit diesem Bit wird der Dual-Playfield-Modus eingeschaltet.

Bit 9 COLOR: Bei Setzen dieses Bits bestimmen Sie, daß die Informationen eines Bildes für einen Composite-Monitor 'rausgeschickt' werden. Das heißt, daß nun nicht mehr wie üblich drei Farbleitungen (RGB), sondern nur noch eine Leitung benutzt wird.

Alle älteren PAL-Fernseher sind solche Composite-'Monitoren'. Natürlich ist das Bild eines Composite-Monitors wesentlich schlechter als

Auch hier hat das Register eine andere Bedeutung beim Ziehen von Linien: BLTSIZE kontrolliert die Länge der Linie, und der Zugriff auf dieses Register veranlaßt den Blitter, diese zu zeichnen. Die Bits H9-H0 kontrollieren die Linienlänge (bis zu 1024 Pixel pro Linie), und die Bits W5-W0 müssen mit %00010 initialisiert werden.

Abschließend zu den Blitter-Registern möchten wir noch einmal darauf hinweisen, daß diese nur der Vollständigkeit halber erwähnt wurden. Zur Benutzung raten wird Ihnen, auch dem Maschinensprache-Programmierer, den Weg über die Library- bzw. BASIC-Befehle.

---

bplpt[6] $0E0 Bit Plane X Zeiger

Dieses Registerpaar (s. ac_ptr) enthält den 18-Bit-Zeiger auf die Anfangsadresse der Bitplane x (x = 1, 2, 3, 4, 5, 6) DMA-Daten. Dieses Register muß nach jedem Strahlrücklauf durch den 68000er oder den Copper, was üblicher ist, neu initialisiert werden. Es ist nämlich dafür zuständig, daß Sie überhaupt etwas von Ihren Grafiken sehen können.

Da ja nur ein wenig mehr als 500 Zeilen auf den Bildschirm gebracht werden können (im Interlaced-Modus und mit 'Overscan'), könnten Sie durch zeilenweises Verschieben dieser Register eine Art Scrolling erreichen. Aber warum sich anstrengen und in der Hardware 'rumpopeln'? Wozu gibt es denn erstens die Variablen 'RxOffset' und 'RyOffset' in der RasInfo-Struktur und zweitens den Library-Befehl ScrollVPort(), mit dem Sie auf einfache Art und Weise die gesamte Bitmap eines Viewports scrollen können?

---

bpldat[6] $110 Bit Plane x Daten

Wie beim DMA-Betrieb üblich, werden Register zur Zwischenspeicherung benötigt. Nachdem der DAM-Controller diese sechs Register nämlich mit Daten gefüttert hat, werden diese über die Video-Hardware (Videoshifter) verarbeitet.

Denn es ist wohl möglich, mit 'bplpt[x]' ein gewisses, Word-weises Scrolling zu erreichen, aber für den, der mehr, sprich feinere Abstufungen, will, gibt es diese Bits.

Doch wir haben ja weiterhin 'ScrollVPort()' und 'RasInfo.RxOffset'. Also brauchen wir uns um dieses Register nicht zu sorgen.

bplcon2

Bit 15-7 Unbenutzt

Bit 6 PF2PRI: Ist dieses Bit gesetzt, dann hat Playfield 2 volle Videopriorität über Playfield 1. Ist es gelöscht, so ist Playfield 1 im Vordergrund (PFBA).

Bit 5-3 PF2P2-0
Bit 2-0 PF1P2-0: Die Bitkombination in diesen Bits setzt die Videoprioritäten zwischen Playfield 1 und den Sprites sowie zwischen Playfield 2 und den Sprites:

Wert	Prioritäten
000	PF1/2 > SP0/1 > SP2/3 > SP4/5 > SP6/7
001	SP0/1 > PF1/2 > SP2/3 > SP4/5 > SP6/7
010	SP0/1 > SP2/3 > PF1/2 > SP4/5 > SP6/7
011	SP0/1 > SP2/3 > SP4/5 > PF1/2 > SP6/7
100	SP0/1 > SP2/3 > SP4/5 > SP6/7 > PF1/2

('>' bedeutet: hat Priorität über)

Bei Sprites muß die Videopriorität nicht extra angegeben werden, da hardwaremäßig festgelegt ist, daß Sprites mit einer niedrigeren Nummer volle Videopriorität über Sprites mit höheren Nummern haben (So hat z.B. der Mauszeiger, der mit Sprite 0 dargestellt wird, volle Videopriorität über alle anderen Sprites).

Wenn verschiedene Viewports dargestellt werden, werden diese Register unter Umständen mehrmals pro Bildschirmaufbau geändert. Bei nur einem Viewport ändert der Copper diese nur jede 60stel Sekunde.

das eines RGB-Monitors, da erst die Farbwerte aus der einen Informationsleitung rückcodiert werden müssen.

Bit 8 GENLOCK: Mit dem Genlock-Interface ist es möglich, Bilder von einem Videorecorder, einer Videokamera oder einem Laser-Disk-Player anstelle der Hintergrundfarbe auf dem Bildschirm darzustellen. So ist es z.B. möglich, Videofilme mit Untertiteln o.ä. zu versehen. Wenn Sie solch ein Interface benutzen wollen, muß die beiliegende Software dafür sorgen, daß dieses Bit gesetzt wird.

Bit 7 EXTRA_HALFBRITE: Mit diesem Bit schalten Sie den Halfbrite-Modus ein.

Bit 6-4 Unbenutzt

Bit 3 LPEN: Wenn Sie einen Lichtgriffel benutzen wollen, müssen Sie dieses Bit setzen, damit der Amiga 'weiß', daß er die Lichtgriffel-Position über den Control-Port erfragen soll. Lichtgriffel können nur über Control-Port 1 (linker Port) gelesen werden. (s. VPOSR)

Bit 2 LACE: Durch das Setzen dieses Bits wird der Interlaced-Modus gesetzt. Dieser Modus verdoppelt die Bildschirmauflösung in horizontaler Richtung. Wo vorher 200 Zeilen waren, sind nun 400 Zeilen. Im Interlaced-Modus wird der Bildschirm zweimal aufgebaut: Das erste Mal werden die geradzahligen Zeilen und das zweite Mal die ungeradzahligen Zeilen der Bitmap dargestellt.

Bit 1 ERSY: Dieses Bit ermöglicht die externe Synchronisation des Elektronenstrahls.

Bit 0 Unbenutzt

bplcon1

Bit 15-8 Unbenutzt

Bit 7-4 PF2H3-0,
Bit 3-0 PF1H3-0: Diese Bits bestimmen die Anzahl der Pixel, um die ein Playfield bei der Darstellung horizontal gescrollt werden soll. Der Wert kann zwischen 0 und 15 Pixeln liegen.

Bit 6	Playfield 2 mit Sprite 2 (oder 3)
Bit 5	Playfield 2 mit Sprite 0 (oder 1)
Bit 4	Playfield 1 mit Sprite 6 (oder 7)
Bit 3	Playfield 1 mit Sprite 4 (oder 5)
Bit 2	Playfield 1 mit Sprite 2 (oder 3)
Bit 1	Playfield 1 mit Sprite 0 (oder 1)
Bit 0	Playfield 1 mit Playfield 2

(Playfield 1 sind die ungeraden Bitplanes (1, 3, 5) und Playfield 2 die geraden (2, 4, 6))

Wie Sie sehen, bietet dieses Register ausreichende Möglichkeiten, Sprite-Kollisionen festzustellen. Mit Bitplane-Kollisionen sieht es da schon etwas weniger erfreulich aus, was eine differenzierte Abfrage ziemlich unmöglich macht.

Bei Sprite-Kollisionen sollten Sie darauf achten, daß diese nicht, wie man meinen könnte, dann gemeldet werden, wenn sich schon die Außenkanten der Sprites berühren, sondern erst dann, wenn sie sich um eine Zeile bzw. Punktspalte überlappen.

Außerdem sollten Sie darauf achten, daß das Register, nachdem Sie es ausgelesen haben, sofort gelöscht wird und erst beim nächsten Vertikal Blank (Strahlrücklauf) erneut beschrieben wird.

---

color[32] $180 Color Register

Diese Register (es gibt 32 Stück davon ($180 - $1BE)) enthalten den 12-Bit-Farbcode, für jede durch die Bitplanes zustande gekommene Farbe.

Das Farbregister, aus dem die Farbinformation für einen bestimmten Punkt geholt wird, wird errechnet, indem man den Wert aller Bitplanes für jeweils einen einzigen Punkt miteinander ORt.

Der Farbcode setzt sich aus 12 Bit zusammen, was 4096 verschiedene Farben ermöglicht. Man kann also maximal 32 Farben aus 4096 möglichen wählen, um ein Bild zu kolorieren (Ausnahme: Hold-and-Modify und Extra Halfbrite). Der Punkt wird dann mit der Farbe des Farbregisters, dessen Wert errechnet wurde, dargestellt. Hier die Verwendung der Bits eines jeden Farbregisters:

clxcon $098 Kollisionskontrolle

Dieses Register bestimmt, ob Kollisionen zwischen Bitplanes, zwischen Sprites und Bitplanes oder zwischen Sprites und Sprites gemeldet werden sollen. Welche Objekte nun genau zusammengestoßen sind, kann man aus den Werten in clxdat erfahren. Die einzelnen Bits für clxcon haben folgende Funktion:

Bit 15-12 ENSP7, 3, 5, 1: Das Setzen eines dieser Bits erlaubt es Ihnen, auch die ungeraden Sprites (7, 5, 3, 1) in die Kollisionskontrolle miteinzubeziehen. Denn die geradzahligen Sprites (0, 2, 4, 6) werden immer auf Kollisionen getestet, die ungeradzahligen nicht.

Allerdings kann man aus CLXDAT nicht erfahren, ob das geradzahlige oder das ungeradzahlige Sprite mit einem Objekt kollidiert ist, es sei denn, es handelt sich hier um 'attached' (verbundene) Sprites, die an der gleichen Position stehen.

Bit 11-6 ENBPx: Mit diesen Bits bestimmen Sie, welche Bitplanes (x = 6-1) auf Kollisionen getestet werden sollen. Wenn Sie alle Bits löschen, wird immer eine Bitplane-Kollision aufgezeigt.

Bit 5-0 MVBP6-1: Diese Bits bestimmen, ob Bitplane-Kollisionen von Sprites mit gesetzten oder gelöschten Bits gemeldet werden sollen, das heißt, daß zum Beispiel nur dann eine Bitplane-Kollision gemeldet wird, wenn in Bitplane 1 ein Punkt gesetzt wird, in Bitplane 2 an dieser Stelle aber keiner, usw. Das ermöglicht in mehr oder weniger großen Grenzen die Abfrage, ob bestimmte Farbpunkte mit Sprites zusammengestoßen sind.

---

clxdat $00E Kollisions Daten Register

Mit Hilfe dieses Registers können Sie erfahren, was mit wem zusammengestoßen (kollidiert) ist.

Bit 15	unbenutzt (aber meistens gleich 1)
Bit 14	Sprite 4 (oder 5) mit Sprite 6 (oder 7)
Bit 13	Sprite 2 (oder 3) mit Sprite 6 (oder 7)
Bit 12	Sprite 2 (oder 3) mit Sprite 4 (oder 5)
Bit 11	Sprite 0 (oder 1) mit Sprite 6 (oder 7)
Bit 10	Sprite 0 (oder 5) mit Sprite 4 (oder 5)
Bit 9	Sprite 0 (oder 1) mit Sprite 2 (oder 3)
Bit 8	Playfield 2 mit Sprite 6 (oder 7)
Bit 7	Playfield 2 mit Sprite 4 (oder 5)

cop1lc	$080/$082	Adresse der ersten Copper Liste
cop2lc	$082/$084	Adresse der zweiten Copper Liste

Diese Register enthalten die Adresse der Copper-Liste, die ausgeführt wird, wenn auf das Register copjmp1 bzw. copjmp2 zugegriffen wird.

In diese Register wird durch LoadView() die Adresse der Hardware-Copper-Listen geladen, wobei cop2lc nur im Interlaced-Modus für den zweiten Durchlauf benutzt wird. Nach Ende des ersten Durchlaufs wird diese zweite Copper-Liste dann von der ersten gestartet.

---

copins $08C Copper Instruction Register

Wie für die DMA-Kanäle immer ein Zwischenpuffer existiert, gibt es solch einen Puffer auch für den Copper. Dieser 'zieht' sich die zwei 16-Bit-Befehlswörter in dieses Register und arbeitet sie der Reihe nach ab.

Dazu werden im ersten Befehlswort die drei möglichen Copper-Befehle kodiert sowie das zu betreffende Hardware-Register bzw. die Position des Elektronenstarahls, auf die gewartet werden soll, übergeben.

Doch die drei möglichen Befehle sind nicht nur, wie der C-Programmierer vermuten könnte, der CMOVE()-, CWAIT()- und CEND()-Befehl, sondern es existieren die Befehle Move, Wait und Skip.

Dies kommt daher, daß CEND() nur eine Unterart des CWAIT()-Befehls ist und man den Skip-Befehl nicht für würdig genug erachtet hat, um in das C-Repertoir aufgenommen zu werden.

Move und Wait haben, wie könnte es anders sein, die gleiche Bedeutung wie der CMOVE()- und CWAIT()-Befehl.

Doch Skip überspringt den folgenden Copper-Befehl, wenn der Elektronenstrahl die angegebene Position oder eine Position, die darüber hinaus liegt, erreicht hat.

Wie oben schon gesagt, besteht ein vollständiger Copper-Befehl aus zwei 16-Bit-Wörtern, die nacheinander in dieses Register geladen und abgearbeitet werden.

Bit 15-12		Unbenutzt
Bit 14-8	Rot3-0	Rotkomponente der Farbe
Bit 7-4	Grün3-0	Grünkomponente der Farbe
Bit 3-0	Blau3-0	Blaukomponente der Farbe

Jeweils 4 Bit bestimmen die Intensität einer Farbkomponente (Rot, Grün, Blau). Durch Mischung dieser 3 verschiedenen Farben und deren Intensitäten (0 = Dunkel, $0F = Hell) ist es möglich, fast jeden Farbton darzustellen.

---

copcon $02E Copper Kontroll Register

Dieses Register ist nur ein 1-Bit-Register. Das einzige Bit, das benutzt wird, ist das Bit 1, alle anderen Bits werden folglich nicht benutzt.

Es wird unter 'Freaks' ehrfurchtsvoll das 'Copper Danger Bit' genannt. Die Register 0 (bltddat) bis einschließlich 30 (strlong) sind grundsätzlich von der Manipulation des Coppers ausgeschlossen.

Auf die Register 31 (bltcon0) bis einschließlich 49 (dsksync) darf der Copper normalerweise auch nicht zugreifen. Dies wird aber möglich, wenn das 'Danger Bit' gesetzt ist. Nach jedem Reset wird dieses Bit aber gelöscht.

---

copjmp1	$088	(Strobe) Copper Neustart der ersten Liste
copjmp2	$08A	(Strobe) Copper Neustart der zweiten Liste

Wenn Sie auf eines dieser beiden Register zugreifen, egal ob mit dem Copper oder 68000er, beginnt der Copper mit der Ausführung einer Copper-Liste, deren Anfangsadresse in cop1lc bzw. in cop2lc steht (hoffentlich!), d.h. der Copper-Program-Counter (PC) wird auf diese Adresse gelenkt.

Die Copper-Liste, deren Anfangsadresse in cop1lc steht, wird außerdem bei jedem Strahlrücklauf (Vertial Blank) ausgeführt.

Aber Vorsicht: Für eigene Experimente mit Copper-Listen ist es sicherer, die User-Copper-Listen zu benutzen, denn schließlich wird das gesamte Display über den Copper gesteuert, und Eingriffe Ihrerseits könnten böse Auswirkungen haben, wenn Sie nicht die Steuerung des gesamten Bildschirms übernehmen.

Copper nur 113 Positionen abfragen. Dies entspricht einer Auflösung von 4 Low-Resolution (HIRES = 0) bzw. 8 High-Resolution (HIRES = 1) Pixels.

Somit können Sie zwar auf jede '4 Punkte'-Position warten, das heißt aber nicht, daß Sie alle 4 Punkte eine Änderung irgendeines Registers vornehmen können. Wenn Sie zwei Moves 'hintereinanderschalten' vergehen zwischen beiden Änderungen immer 8 Low-Resolution-Punkte (Im Mode Hi-Res verdoppelt sich natürlich alles).

Bit 0 1: Dieses Bit und das Bit 0 des zweiten Befehlswortes dienen zur Erkennung der Copper-Befehle Wait und Skip. Ist das Bit 0 des ersten Wortes gelöscht, dann weiß der Copper, daß es sich um den Move-Befehl handelt, und testet das zweite Befehlswort nicht mehr, das ja vollständig in ein angegebenes Register geschrieben wird.

Ist es gesetzt, weiß der Copper, daß geWAITet oder geSKIPt werden soll. Was von beiden tatsächlich 'angesagt' ist, legt man mit dem zweiten Befehlswort fest.

2. Befehlswort

Bit 15 BFD: Setzen Sie dieses Bit, dann wird dem Wait-Befehl erst dann erlaubt fortzufahren, wenn der Blitter sein 'OK' gegeben hat (Bliter Finished Disable).

Bit 14-8 VE6-VE0: VE (Vertical Comparison Enable) und HP (Horizontal Comparison Enable) legen fest, welche Bits zur Überprüfung der Elektronenstrahl-Position herangezogen werden.

Bit 7-1 HE8-HE2: Was VE für die vertikale Position ist, ist HE für die horizontale Position.

Bit 0 %0: Hieran erkennt der Copper den Wait-Befehl.

SKIP

Die Bitbelegungen der Befehlswörter beim Skip- und beim Wait-Befehl sind fast identisch. Der einzige Unterschied zwischen beiden besteht darin, daß Bit 0 des zweiten Befehlswortes gesetzt ist. Die Bedeutung aller anderen Bits bleibt gleich.

Befehl:

MOVE

1. Befehlswort

Bit 15-9 Unbenutzt

Bit 8-1 DA8-1: DAx (Destination Adress) wird nur beim Move-Befehl gebraucht und gibt an, in welches Register der Wert des zweiten Befehlswortes geschrieben werden soll. DA wird als Offset von $DFF000 angegeben, um die jeweiligen Hardware-Register anzusprechen.

Bit 0 0: Dieses Bit dient zur Erkennung des Copper-Move-Befehls und muß beim Move-Befehl immer 0 sein (Das erste Bit des zweiten Befehlswortes wird zur Decodierung dieses Befehls nicht gebraucht!).

2. Befehlswort

Dieses Wort enthält den 16-Bit-Wert, der in das Register, das mit DA festgelegt wird, geschrieben werden soll.

WAIT

1. Befehlswort

Bit 15-9 VP7-VP0: VP (Vertical Position) enthält die vertikale (Y) Position des Elektronenstrahls, auf die gewartet werden soll.

Da es 262 verschiedene vertikale Strahlpositionen gibt, auf die man warten kann, VP nur eine Datenbreite von 8 Bit hat, muß man sich eines Tricks bedienen, um an die Positionen 256-262 'heranzukommen':

Man wartet auf normalem Weg auf die vertikale Position 255. Dann wartet man mit einem zweiten Wait-Befehl auf eine Position zwischen 0 und 6.

Bit 8-1 HP8-HP1: HP (horizontal position) enthält die horizontale (X) Position des Elektronenstrahls, auf die gewartet werden soll. Er kann die Positionen 0 bis 226 einnehmen. Da aber nur eine Datenbreite von 7 Bits für die horizontale Position zur Verfügung steht, kann der

Nun, zur X-Endkoordinate werden grundsätzlich $100 addiert, um die tatsächliche Endposition zu erhalten.

Bei der Berechnung der tatsächlichen vertikalen Endkoordinaten spielt Bit VSTOP7 eine große Rolle. Das imaginäre, tatsächlich nicht existierende Bit VSTOP8 ist nämlich das Einerkomplement von VSTOP7. Das heißt also: Ist VSTOP7 gelöscht, ist VSTOP8 gesetzt. Sie müßten dann also zu dem Wert in VSTOP7-0 noch 2^8 = 256 addieren, um die tatsächliche vertikale Endposition zu erhalten.

Bei gesetztem VSTOP7-Bit brauchen Sie sich um VSTOP8 nicht mehr zu sorgen, da es ja sowieso 0 ist und keine 'aktive' Zweierpotenz darstellt.

Wenn Sie diese Register beschreiben wollen, sollten Sie den sogenannten 'Vertikal Blank'-Bereich beachten, denn die oberste Zeile auf dem Monitor, die Sie sehen können, ist nicht die nullte Rasterzeile! Ebenso ist die erste Spalte nicht die erste Spalte.

Das rührt daher, daß der Elektronenstrahl ein wenig Zeit zum 'Verschnaufen' braucht, wenn er am unteren Bildschirmrand angelangt ist, so wie er auch ein wenig Zeit zur Erholung nach jeder dargestellten Zeile braucht.

Normale Werte für 'diwstrt' und 'diwstop' wären somit:

```
diwstrt = $2c81
diwstop = $f4c1
```

Das ergibt für die Größe des so definierten Bildschirmfensters:

```
HSTOP+$100-HSTART = $c1+$100-$81 = $140 = 320 Punkte
VSTOP+(VSTOP8 * $100)-VSTART = $F4( = %11110100)+(0 *
$100) - $2c = 200 Normal-Rasterzeilen
```

Noch etwas zur Beachtung: Da beim Bildschirmaufbau ein Bereich für den Strahlrücklauf (Vertikal Blank) gebraucht wird, sollte man die Startwerte des Bildschirmfensters vertikal nicht kleiner als $20 wählen - man würde sowieso nichts sehen - weil man dem Elektronenstrahl sonst nicht genügend Zeit ließe.

diwstrt	$08E	Display Window Start (obere, linke Position)
diwstop	$090	Display Window Stop (untere, rechte Position)

Mit diesen beiden Registern kann man die tatsächliche Größe des Bildschirms (Viewport) festsetzen. Die Werte in diesen beiden Registern bestimmen die linke obere Ecke sowie die rechte untere Ecke des Bildschirmfensters. Außerhalb dieses Bildschirmfensters kann nichts anderes außer der Hintergrundfarbe (in color[0]) dargestellt werden.

Die Positionen der Ecken werden in 'normalen', also nicht interlaceten, Rasterzeilen und in Low-Resolution-Pixel angegeben. Sie brauchen diese Werte also nicht zu verändern, wenn Sie den Interlaced- oder den Hi-Res-Modus benutzen wollen.

Die Bitbelegung der beiden Register ist wie folgt:

diwstrt

Bit	Funktion	
15-8	VSTART	vertikale Startposition
7-0	HSTART	horizontale Startposition

Mit diesem Register legt man die Startposition des Bildschirmfensters fest. Aber wie Sie sicher schon bemerkt haben, können Sie nicht davon ausgehen, daß Sie den Start eines Viewports beliebig festlegen können.

Aufgrund der Datenbreite von 8 Bits sowohl für die vertikale als auch für die horizontale Position können Sie diesen nur innerhalb der obereren 256 'normalen' Rasterzeilen und innerhalb der 257 rechten Lo-Res-Punkte festlegen.

diwstop

Bit	Funktion	
15-8	VSTOP	vertikale Startposition
7-0	HSTOP	horizontale Startposition

Aber wie können Sie dann z.B. Bildschirmfenster darstellen, die mehr als 256 Punkte breit sind, denn diwstop hat ja die gleiche Bitbelegung und somit auch nur 8 Bits für die X- und Y-Position zur Verfügung.

lung von maximal 376 Low-Resolution-Pixel (= 752 High-Resolution-Pixels). Setzen Sie für DDFSTRT einen Wert kleiner als $38, dann kann es passieren, daß einige Sprites nicht mehr dargestellt werden.

---

dmacon	$096	DMA Kontrollregister (Schreiben)
dmaconr	$002	DMA Kontrollregister (Lesen)

Diese Register sind für den DMA-Controller zuständig, der die verschiedenen Prozessoren über DMA-Kanäle mit wichtigen Daten 'füttert'.

Mit dmacon können Sie festlegen, welche DMA-Kanäle geöffnet werden sollen.

Doch hier die Bitbelegung, die für beide Register übrigens wieder gleich ist:

Bit 15 SET/CLR (s. ADKCON)

Bit 14 BBUSY: Dieses Bit zeigt den Status des Blitters an. Ist er in 'action', ist es gesetzt.

Bit 13 BZERO: Dieses Bit ist dann gesetzt, wenn alle Ergebnis-Bits eines 'Blits' (Bildschirmbereiche kopieren etc.) gleich 0 waren.

Bit 12, 11 Unbenutzt

Bit 10 BLTPRI: Dieses Bit bestimmt die Prioritäten zwischen Blitter und 68000er. Denn die interne Taktfrequenz von ca. 14 MHz wird zwischen beiden Prozessoren aufgeteilt. Wie, das bestimmt dieses Register.

Ist es gesetzt, nimmt der Blitter jeden Taktzyklus, den er kriegen kann, auch die, die eigentlich dem 68000er 'gehören'. Ist es gelöscht, so gesteht er dem 68000er bei jedem dritten Memory-Refresh-Zyklus einen Taktzyklus zu.

Bit 9 DMAEN: Bei gesetztem DMAEN-Bit ist es möglich, alle DMA-Kanäle zu aktivieren. Ist es gelöscht, so haben die Werte in den folgenden Bits keinen Einfluß auf den Status des jeweiligen DMA-Kanals, den sie aktivieren. DMAEN ist somit der 'Obermacker' des DMA-Verkehrs.

ddfstrt	$092	Bildschirmdaten hol Start
ddfstop	$094	Bildschirmdaten hol Stop

Bitte machen Sie uns nicht für diese etwas unglückliche Namensgebung für diese beiden Register verantwortlich, aber übersetzt (Data fetch start, Data fetch stop) würden sie tatsächlich so heißen!

Doch zur Erläuterung dieser beiden Register:

Sie bestimmen, wann damit angefangen wird, die Daten aus dem Speicher auf dem Monitor darzustellen. Ja, Sie haben richtig gelesen. Diese Register geben ein Zeitverhältnis an.

Denn um zwei Punkte darstellen zu können, wird ein Taktzyklus benötigt. Um also festzulegen, wann Daten dargestellt und wann damit wieder aufgehört werden soll, brauchen Sie nur die Werte aus den Registern 'diwstrt' und 'diwstop' zu halbieren und evt. noch einen kleinen Offset abzuziehen, und schon haben Sie die gewünschten Werte für ddfstrt und ddfstop.

Die Offsets sind aber für den Lo-Res- und Hi-Res-Modus unterschiedlich. Im Lo-Res-Modus beträgt er nämlich 8.5 und im Hi-Res-Modus 4.5.

Für die Startposition ergibt sich:

ddfstrt = ($81/2 - 8.5) = $38
(Hi-Res: ddfstrt = ($81/2 - 4.5) = $3C)

Die Werte für ddfstop errechnen sich wie folgt:

ddfstop = ddfstrt + (16/2 *(Anzahl der Words pro Linie -1))
(Low-Resolution)
ddfstop = ddfstrt + (16/4 *(Anzahl der Words pro Linie -2))
(High-Resolution)

Wie Sie sehen, wird hier auch wieder die Anzahl der darzustellenden Punkte (1 Word = 16 Punkte) halbiert bzw. geviertelt. Die Viertelung im Hi-Res-Modus kommt natürlich daher, daß im Hi-Res-Modus doppelt soviele Punkte in der gleichen Zeit dargestellt werden müssen.

Leider sind auch diese Werte nicht frei wählbar, d.h. sie sind von der Hardwareseite her beschränkt. DDFSTRT darf nicht kleiner als $18 sein und DDFSTOP höchstens $D8. Diese Werte erlauben die Darstel-

Diese Register wollen wir jedoch in diesem Buch nicht erklären, da diese eher zum Thema 'Amiga Intern' passen würden.

Bit 15 DMAEN: Wenn dieses Bit gesetzt ist, kann die Datenübertragung über den DMA-Kanal stattfinden.

Bit 14 WRITE: Dieses Bit bestimmt die Schreib-/Leserichtung. Wenn es gesetzt ist, werden Daten vom RAM über den DMA-Kanal auf Diskette geschrieben. Ist es gelöscht, werden Daten von Diskette gelesen.

Bit 13-0 LENGTH: Diese Bits enthalten die Anzahl der Words, die gelesen oder geschrieben werden sollen.

---

dskdat $026 Disk DMA Daten (Schreiben)
dskdatr $008 Disk DMA Daten (Lesen)

Diese Register enthalten die Daten, die entweder vom Disk-DMA-Kanal gelesen wurden oder die über ihn geschrieben werden sollen.

Bei dskdatr sollten Sie beachten, daß das Register vor Ihrem Copper-Zugriff 'geschützt' ist.

---

dskbytr $01A Disk Datenbyte und Status

Dieses Register ist ein Datenpuffer, der direkt mit dem Disk-Microprozessor verbunden ist. Von diesem Register aus werden die Daten zum DMA-Controller oder zum Disk-Controller geschickt. Außerdem enthält es ein paar Status-Bits.

Bit 15 DSKBYT: Wenn Bytes von der Diskette gelesen werden, wird dieses Bit gesetzt, wenn das Byte vollständig gelesen wurde.

Bit 14 DMAON: Dieses Bit hat den gleichen Wert wie DMAEN in dsklen und wird mit Bit DMAEN aus dmacon geANDet. Ist das Ergebnis 1, dann kann auf die Diskettenstation mittels der DMA zugegriffen werden.

Bit 13 DISKWRITE: Dieses Bit hat denselben Wert und dieselbe Bedeutung wie WRITE in dsklen.

Bit 12 WOREQUAL: Dieses Bit ist nur dann gesetzt, wenn im Register dsksync die Byte-Daten zu Word-Daten synchronisiert werden.

Bit 8 BPLEN: Ist dieses Bit sowie DMAEN gesetzt, so wird der DMA-Kanal für die Bilddatenübertragung aus den Bitplanes aktiviert.

Bit 7 COPEN: Dieses Bit bestimmt den Status des Coppers. Ist es gesetzt, so kann der Copper arbeiten, ist es gelöscht, so ist der Copper abgeschaltet.

Bit 6 BLTEN: Was oben für den Copper gilt, gilt hier für den Blitter.

Bit 5 SPREN: Ist dieses Bit gesetzt, so können Daten für die Darstellung von Sprites über den DMA-Kanal geschickt werden.

Bit 4 DSKEN: Dieses Bit bestimmt den Status des Disk-DMA-Kanals. Ist es gelöscht, so ist keine Datenübertragung zwischen Rechner und Floppy mehr möglich.

Bit 3-0 AUD3-0EN: Diese Bits bestimmen den Status für den jeweiligen Sound-DMA-Kanal. Sind sie gelöscht, so kann man keine Töne mit Hilfe des DMA-Kanals hervorzaubern.

---

dskpt $020 Disk Pointer

Dieses Registerpaar (es ist ja eigentlich ein Zeiger mit 18 Bits) bestimmt die Adresse, an der die gelesenen Disk-DMA-Daten abgespeichert werden bzw. wo die auf Diskette zu schreibenden Bytes zu finden sind.

---

dsklen $024 Disk-Daten Länge

Dieses Register enthält die Anzahl der Words, die über den Disk-DMA-Kanal gesendet oder gelesen werden sollen. Außerdem enthält es ein Bit, das die Übertragungsrichtung (RAM -> Disk oder Disk -> RAM) bestimmt. Ein weiteres Bit bestimmt, ob auf den DMA-Kanal überhaupt zugegriffen werden kann.

Außerdem wird mit dem Zugriff auf dieses Register die Datenübertragung gestartet. Dazu sollte dieses Register zweimal mit dem gleichen Wert beschrieben werden. Sind alle Daten übertragen, wird ein 'Disk Block Interrupt' (s. INTREQ) gesendet, der die Datenübertragung stoppt.

Um eine vollständige Diskoperation, z.B. das Laden eines Programms, durchzuführen, genügt es natürlich nicht, nur die DMA-Register zu setzen. Dafür müßen Sie auch ein paar I/O Register der CIAs setzen.

Bit 11 RBF (receive buffer full): Hier wird der Interrupt für den seriellen Port getestet. Er wird dann ausgeführt, wenn der serielle Eingabepuffer voll ist und vom Benutzer ausgelesen werden kann (Ebene 5).

Bit 10-7 AUD3-0: Hier wird dann ein Interrupt ausgelöst, wenn die Sound-Daten über den DMA-Kanal abgearbeitet worden sind oder wenn, im manuellen Modus, die Audio-Datenregister wieder bereit sind, neue Daten zu verarbeiten (Ebene 4).

Bit 6 BLIT (Blitter finished): Wenn dieses Bit gesetzt ist, heißt das, daß der Blitter mit seiner Arbeit fertig ist (Ebene 3).

Bit 5 VERTB: Bei jedem Strahlrücklauf wird eine Interrupt-Routine abgearbeitet, die die Rücksetzung vieler Pointer und die Abarbeitung anderer Systemaufgaben verursacht.

Bit 4 COPER: COPER zeigt an, daß ein Interrupt vom Copper ausgelöst wurde. Der Copper kann jedes dieser Bits verändern, so wie er fast jedes Register verändern kann. Der Copper-Interrupt tritt dann auf, wenn das gesamte Bild dargestellt wurde, d.h. wenn der Elektonenstrahl die Position DIWSTOP erreicht hat. So ist es z.B. möglich, den 68000er während des Strahlrücklaufs spezielle Aufgaben erfüllen zu lassen.

Bit 3 PORTS: PORTS zeigt einen Interrupt an, der durch das Low-legen der Prozessor-Leitung INT2 hervorgerufen wird (Ebene 2).

Bit 2 SOFT: Diese Leitung ist für Software-Interrupts reserviert.

Bit 1 DSKBLK (Disk block finished): DSKBLK zeigt an, daß die Datenübertragung über den Disk-DMA-Kanal vollendet worden ist (Ebene 1).

Bit 0 TBE (transmit buffer empty): TBE zeigt an, daß der Output-Buffer von UART (Universal Asynchronus Receiver/Transmitter) Daten aufnehmen kann und daß in diesen Puffer hinein geschrieben werden kann (Ebene 1).

Noch ein Wort zu den Ebenen eines Interrupts: Grundsätzlich können über die Prozessor-Leitungen INT0, INT1 und INT2 sieben verschiedene Interrupts durch Low-Legen dieser Leitungen hervorgerufen werden. Dies wären dann jedoch nur System-Interrupts.

Bit 11-8 Unbenutzt

Bit 7-0: In diesen Bits ist das Datenbyte enthalten.

---

dsksync $07E Disk Synchronisations Register

Dieses Register enthält den Synchronisationscode für Disketten-Operationen.

---

intreq	$09C	Interrupt Request Bits (Schreiben)
intreqr	$01E	Interrupt Request Bits (Lesen)
intena	$09A	Interrupt Enable Bits (Schreiben)
intenar	$01C	Interrupt Enable Bits (Lesen)

Dies sind Interrupt Request und Enable oder Mask Bits. Sie bestimmen, welche Interrupts (zyklische Unterbrechungen) stattfinden können.

Hierbei ist zwischen 'Request' (Anforderungen) und 'Enabling' (Zulassung) zu unterscheiden. Interrupt-Anforderungen können nur dann ausgeführt werden, wenn die entsprechenden Zulassungsbits gesetzt sind.

Somit erklärt sich schon fast von selbst, daß diese vier Register dieselbe Bitbelegung haben. Die 'Request'- und 'Enable'-Register sind auch hier in zweifacher Ausführung zu finden. Einmal nur zum Beschreiben, ein zweites Mal nur zum Lesen.

Doch hier die Bitbelegung:

Bit 15 SET/CLR (s. ADKCON)

Bit 14 INTEN (Master Interrupt enable): Ist dieses Bit gelöscht, kann kein Interrupt-Ereignis mehr abgearbeitet werden.

Bit 13 EXTER: Über dieses Bit wird der externe Interrupt, über CIAB, generiert (Ebene 6).

Bit 12 DSKSYN: Dieser Interrupt wird ausgeführt, wenn Daten mit dem DSKSYNC-Register synchronisiert worden sind (Ebene 5).

Der Joystick:

Die Abfrage der Joystick-Stellungen gestaltet sich etwas einfacher als die der Mausbewegung. Sind die Bits 1 oder 9 von joy0/1dat gesetzt, so heißt das, daß der Joystick nach rechts bzw. nach links gedrückt wurde. Um die Stellung oben und unten abzufragen, ist es nötig, die Bits 9 und 8 bzw. 1 und 0 zu XORen. Der Feuerknopf des Joysticks am linken Port wird genauso abgefragt wie der linke Mausknopf. Der Feuerknopf des rechten Joystickknopfes wird über Bit 5 des Registers $BFE001 abgefragt.

Die Abfrage von Maus und Joystick sollte der 'normale' Programmierer nicht von Hand vornehmen, da die Betriebssystem-Software dies schon regelt, aber wem dies zu umständlich ist, oder wessen Anforderungen die Betriebsystem-Software nicht genügt, der mag sich daran versuchen.

---

joytest $036 Beschreibe alle Mauszähler

Der Wert, den Sie in dieses Register hineinschreiben, wird in die beiden Register joy0dat und joy1dat übertragen. So können Sie diese mit 'einem Schlag' initialisieren.

---

pot0dat $012 Poti Zähler links (vert, horiz).
pot1dat $014 Poti Zähler rechts (vert, horiz).

Wie bei jedem anständigen Computer kann man auch beim Amiga sogenannte Paddles anschließen (da, wo sonst Joysticks oder die Mäuse angeschlossen sind). Diese sind aber nichts anderes als regelbare Widerstände (Potentiometer), deren augenblicklicher Widerstandswert zyklisch, d.h. alle 60stel Sekunde, abgefragt wird.

Setzt man das Startbit in potgo, wird Strom durch den Widerstand über einen Kondensator geleitet. Der Kondensator lädt sich eine kurze Zeit auf (ca. 8 Rasterzeilen), dann wird der Kondensator entladen. Der jeweilige Ladungszustand des Kondenstors wird mit einem festen Wert verglichen.

Ist die augenblickliche Belastung höher als der festgesetzte Wert, so wird ein vorher gelöschter Zähler inkrementiert, ist die Belastung kleiner, so wird der Vergleich abgebrochen, und ein dem Widerstand proportionaler Wert steht in diesen Registern. An jeden Control Port können zwei dieser Paddles angeschlossen werden, was die Realisierung von Proportional Joysticks erlaubt. Die Knöpfe der Paddles fragt

Der Amiga aber spezifiziert diese System-Interrupts innerhalb der Interrupt-Handler noch weiter, so daß mehr als 7 Interrupt-Ebenen möglich werden.

---

joy0dat	$00A	Joystick/Maus 1 (linker Port)
joa1dat	$00C	Joystick/Maus 2 (rechter Port)

Diese beiden Register geben Auskunft über den Status der Eingabegeräte (Joystick, Maus, Lichtgriffel, Potentiometer etc.) an dem jeweiligen Control Port. Für die folgenden Eingabegeräte gilt:

Maus bzw. Trackball:

Für die Maus und den Trackball, der eigentlich eine auf dem Rücken liegende Maus ist, enthalten die Register jeweils einen Wert, der direkt proportional zu ihrer tatsächlichen Bewegung ist. Diese Bewegung wird in eine horizontale (X-Richtung) und in eine vertikale (Y-Richtung) Bewegung aufgespalten. Die Bits 15-8 enthalten die vertikale Bewegungskomponente, während die Bits 7-0 der Register die horizontale Komponente enthalten. Der Wert in diesen beiden Registern entsteht durch Drehung der Lochrasterscheiben, die sich im Inneren der Maus zwischen Lichtschranken bewegen.

Einmal die Lichtschranke unterbrechen und wieder freigeben bedeutet einen Impuls. Die Amiga Maus sendet 200 solcher Impulse pro Inch (2,54 cm). Diese beiden Bytes sind also eigentlich Zähler, die Unterbrechungen der Lichtschranken zählen.

Diese beiden Zähler werden hochgezählt, wenn Sie die Maus nach links oder runter (zu sich hin) bewegen, und sie werden heruntergezählt, wenn Sie die Maus nach rechts oder von sich weg bewegen. Die Richtung der Bewegung können Sie aus dem Vorzeichen der Differenz des aktuellen und des vorigen Zählerwertes - für jede Bewegungskomponente - bestimmen. Ist diese Differenz negativ, so wurde die Maus nach unten oder nach rechts bewegt.

Nach jedem Vertikal Blank wird dieses Register gelesen und gelöscht, um die Position der Maus für Intuition zugänglich zu machen.

Der linke Mausknopf wird über Bit 6 des CIAA-Registers $BFE001 abgefragt. Der rechte Mausknopf wird aus dem gleichen Register wie der Widerstandswert für das Potentiometer (pot0dat) gelesen.

Dieses Register sollte niemals vom 68000er oder Copper angesprochen werden, da dadurch eventuell das gesamte interne Timing durcheinander kommt und so Daten verlorengehen könnten.

---

serdat $030 Serieller Port: Daten und Stop Bits (Schreiben)

Die Daten, die in dieses Register hineingeschrieben wurden, werden in ein Verschieberegister (Shifter) geschrieben, das das Byte Bitweise auf einen Puffer und dann an die serielle Schnittstelle ausgibt. Ist dieser Puffer empfangsbereit, sendet er einen TBE (Transmit Buffer Empty) -Interrupt (s. INTREQ). Die Anzahl der Datenbits (8 oder 9) hängt vom LONG-Bit aus serper ab.

Bit 15-10 0

Bit 9 Stopbit

Bit 8-0 Datenbits

---

serdatr $018 Serieller Port: Daten und Status (Lesen)

Dieses Register ist das Gegenstück zu serdat. Es enthält die Datenbits, die vom Puffer über ein Verschieberegister empfangen wurden. Verschiedene Interrupt Request (INTREQ) Bits sind auch in diesem Register enthalten.

Bit 15 OVERRUN: Dieses Bit ist gesetzt, wenn der Empfangspuffer übergelaufen ist. Durch Löschen von RBF in INTREQ wird dieses Bit wieder gelöscht.

Bit 14 RBF: s. INTREQ

Bit 13 TBE: s. INTREQ

Bit 12 TSRE: Ist dieses Bit gesetzt, heißt das, daß das Verschieberegister leer ist.

Bit 11 RXD: Der RXD-Prozessor-Pin erhält die Daten direkt von UART. Dieses Bit kann dann direkt vom 68000er getestet werden.

Bit 10 0

Bit 9 Stopbit

man normalerweise über die linke und rechte Position des Joysticks ab. Nun die Bitbelegung der beiden Register:

Bit 15-8 Y7-Y0: Potentiometerstellung des Potis an Pin 9

Bit 7-0 X7-X0: Potentiometerstellung des Potis an Pin 5

Der Strom für diese Widerstände, die mindestens 470 kΩ ±10% groß sein müssen, wird über Pin 7 (+5 V, 125 mA) abgegeben.

---

potinp	$034	Poti Port Daten schreiben und starten
potgo	$016	Poti Port Daten lesen

Die Control Ports fungieren auch noch als 4-Bit Bidirektionaler I/O Port. Zur Kontrolle dieses Ports sind diese Register da:

Bit 15 OUTRY (rechter Port, Pin 9)
Bit 14 DATRY
Bit 13 OUTRX (rechter Port, Pin 5)
Bit 12 DATRX
Bit 11 OUTLR (linker Port, Pin 9)
Bit 10 DATLR
Bit 9 OUTLX (linker Port, Pin 5)
Bit 8 DATLX

Die Bits OUTxx setzen den Status der Leitungen. Sind sie gesetzt, werden die Daten DATxx ausgegeben. Sind sie gelöscht, werden die Daten DATxx gelesen.

Bit 7-0 0 Reserviert

Bit 0 START: Wird dieses Bit gesetzt (potgo), dann werden die Kondensatoren aufgeladen, und die Zähler fangen an zu zählen.

---

refptr	$028	Write Refresh Pointer

Dieser Pointer wird als RAM Refresh Pointer benutzt. Die Speicherbits stehen ja nicht konstant unter Strom, sondern werden alle 280 ns (= ein Memory Cycle) wieder aufgeladen, d.h. ihr alter Wert wird wieder aufgefrischt. Durch diese Technik ist es möglich, daß die meisten ICs nur eine Spannung von 5 Volt benötigen.

Sprites x. Wie Sie sehen, stehen nur 8 Bits für jede Koordinate der Position zur Verfügung. Deshalb ist in spr[x].ctl das höchstwertige Bit SV8 und das niederwertigste Bit SH0 enthalten. Somit können Sprites Positionen zwischen 0 und 512 in horizontaler wie in vertikaler Richtung einnehmen. Die Bewegung von Sprites kann nur in Low-Resolution-Punkten und in normalen Rasterzeilen geschehen.

ctl

Bit 15-8 EV7-EV0: In diesen Bits ist die vertikale (Y) Endposition des Sprites x enthalten. Dies erklärt die Tasache, daß Sprites beliebig hoch sein können. (Sprites sind immer 16 Punkte breit) Bitte beachten Sie, daß das höchstwertigste Bit (EV8) Bit 1 ist.

Bit 7 ATT: Ist dieses Bit gesetzt, heißt das, daß jeweils zwei Sprites (0/1, 2/3, 4/5, 6/7) zusammengezogen werden, um eine größere Farbenvielfalt, d.h. 15 anstatt 3 Farben, zu erhalten. Die Sprites eines Paares können jedoch frei, d.h. unabhängig voneinander bewegt werden. Nur wenn sie sich ganz oder teilweise überschneiden, werden ihre Bitmuster zur Darstellung von 15 Farben herangezogen.

Bit 6-4 Unbenutzt

Bit 2 SV8: Höchstwertigstes Bit zu SV7-SV0 in spr[x].pos

Bit 1 EV8: s. EV7-EV0

Bit 0 SH0: Höchstwertigstes Bit zu SH7-SH0 in spr[x].pos

---

spr[8].dataa	$144	Sprite X Bilddatenregister A
spr[8].datab	$146	Sprite X Bilddatenregister B

Diese Register enthalten jeweils die Zeile eines Sprites, die gerade dargestellt werden soll. dataa enthält das erste Wort und datab das zweite Wort eines jeden Sprites. Wenn in das Register dataa geschrieben wird, wird die aktuelle Strahlposition mit dem Wert in SH8-SH0 verglichen, und wenn beide Werte gleich sind, wird die Sprite-Zeile dargestellt.

Das Verändern von spr[x].ctl unterbindet diesen Vergleich. Erst wenn auf dataa zugegriffen wird, wird auch die Position des Elektronenstrahls verglichen und eventuell das Sprite dargestellt.

Bit 8 Stopbit oder D8

Bit 7-0 D7-D0 (Datenbits)

---

serper $032 Serieller Port: Rate und Kontrolle

Mit diesem Register wird die Datenbreite (8 oder 9 Bits) des seriell zu empfangenden oder zu sendenden Worts festgelegt. Außerdem setzt man hiermit auch die Übertragungsgeschwindigkeit.

Bit 15 LONG: Ist dieses Bit gesetzt, so ist das Wort, das über den seriellen Port gelesen oder gesendet wird, 9 Bits breit.

14-0 RATE: Gibt die Zeit an, in der ein Bit gesendet oder gelesen wird. Die Zeit errechnet sich wie folgt: (Wert in den unteren 14 Bits + 1) * .2794 μs.

---

sprpt[8] $120 Sprite x Pointer

Dieses Registerpaar enthält die Adresse, an der die Daten für jedes Sprite (x = 0, 1, 2, 3, 4, 5, 6, 7) stehen. Diese Register müssen, wenn Sprites dargestellt werden sollen, vom Copper oder vom 68000er nach jedem Strahlrücklauf neu gesetzt werden. Zur Darstellung von Sprites muß auch der DMA-Kanal (dmacon/SPREN) aktiviert werden.

---

spr[8].pos $140 Sprite X vert. u. horiz. Startposition
spr[8].ctl $142 Sprite X vert. Stopposition

Diese beiden Register arbeiten zusammen, indem sie die Größe, Position und andere Funktionen in Verbindung mit den Hardware-Sprites kontrollieren. Sie werden meistens während des horizontalen Strahlrücklaufs vom DMA-Controller geladen.

Natürlich werden die Positionen auch wieder in Rasterzeilen bzw. 'normalen' Punkten angegeben.

pos

Bit 15-8 SV7-SV0
Bit 7-0 SH8-S8H1

SV7-SV0 stellt die vertikale (Y-Richtung) Startposition des Sprites x dar. SH7-SH0 enthält die horizontale (X-Richtung) Startposition des

Die Bitbelegung ist wie folgt:

Bit 15-8 V7-V0: Diese Bits enthalten die niederwertigen Bits der vertikalen Strahlposition. Das höchstwertigste Bit steht in VPOSR.

Bit 7-0 H8-H1: In diesen Bits steht die aktuelle horizontale Position des Elektronenstrahls. Das Auflösungsvermögen beträgt ein 1/160 der Bildschirmbreite.

strequ	$038	Strobe für horiz. Synchronisation mit VB und EQU
strvbl	$03A	Strobe für horiz. Synchronisation mit VB
strhor	$03C	Strobe für horiz. Synchronisation
strlong	$03E	Strobe für Identifikation einer langen horizontal Linie.

Diese Register werden für interne Zwecke für den Viodeoshifter benötigt. Welche Aufgabe die einzelnen Register genau haben, konnten wir leider nicht in Erfahrung bringen, können Ihnen aber versichern, daß der 'normale' Programmierer diese nie gebrauchen wird.

---

vposr	$004	Höchstwertigstes Vertikalpositionsbit (Lesen)
vposw	$004	Höchstwertigstes vertikalpositionsbit (Schreiben)

Diese Register werden dazu gebraucht, um Aktionen des 68000 mit dem Elektronenstrahl zu synchronisieren. Somit können Sie z.B. mit dem 68000er auf die 0. Y-Koordinate des Elektronenstrahls warten und dann erst mit dem Programm fortfahren (s. WaitTOF()).

Diese Register enthalten jedoch nur das höchstwertige Bit der vertikalen Position und ein 'Interlaced'-Flag.

Bit 15 LOF: Dieses Bit zeigt an, daß der Interlaced-Modus angeschaltet wurde.

Bit 14-0 Unbenutzt

Bit 0 V8: Dies ist das höchstwertigste Bit der vertikalen Elektronenstrahlposition. Mit ihm können Sie auch Zeilen über 256 testen (313 für PAL, und 262 für NTSC).

---

vhposr	$006	Vertikale und horizontale Elektronenstrahlposition (Lesen)
vhposw	$02C	Vertikale and horizontael Elektronenstrahlposition (Schreiben)

Diese Register werden benötigt, um die aktuelle Position des Elektronenstrahls zu lesen oder um diese Position neu zu setzen, was bei Tests (im Normalfall aber nicht) vorkommen kann.

038	bltdpt	$054
039	bltsize	$058
	-------	$05a
	-------	$05c
	-------	$05e
040	bltcmod	$060
041	bltbmod	$062
042	bltamod	$064
044	bltdmod	$066
	-------	$068
	-------	$06a
	-------	$06c
	-------	$06e
045	bltcdat	$070
046	bltbdat	$072
048	bltadat	$074
	-------	$076
	-------	$078
	-------	$07a
	-------	$07c
049	dsksync	$07e
050	cop1lc	$080
051	cop2lc	$084
052	copjmp1	$088
053	copjmp2	$08a
054	copins	$08c
055	diwstrt	$08e
056	diwstop	$090
057	ddfstrt	$092
058	ddfstop	$094
059	dmacon	$096
060	clxcon	$098
061	intena	$09a
062	intreq	$09c
063	adkcon	$09e

aud[0]

064	ac_ptr	$0a0
065	ac_len	$0a4
066	ac_per	$0a6
067	ac_vol	$0a8
068	ac_dat	$0aa
	------	$0ac

Im folgenden eine Liste der Register, die nicht alphabetisch, sondern ihren Adressen (bitte den Offset von $DFF000 beachten) bzw. Nummern nach geordnet sind:

Nr.	Name	Adresse
000	bltddat	$000
001	dmaconr	$002
002	vposr	$004
003	vhposr	$006
004	dskdatr	$008
005	joy0dat	$00a
006	joy1dat	$00c
007	clxdat	$00e
008	adkconr	$010
009	pot0dat	$012
010	pot1dat	$014
011	potinp	$016
012	serdatr	$018
013	dskbytr	$01a
014	intenar	$01c
015	intreqr	$01e
016	dskpt	$020
017	dsklen	$024
018	dskdat	$026
019	refptr	$028
020	vposw	$02a
021	vhposw	$02c
022	copcon	$02e
023	serdat	$030
024	serper	$032
025	potgo	$034
026	joytest	$036
027	strequ	$038
028	strvbl	$03a
029	strhor	$03c
030	strlong	$03e
031	bltcon0	$040
032	bltcon1	$042
033	bltafwm	$044
034	bltalwm	$046
035	bltcpt	$048
036	bltbpt	$04c
037	bltapt	$050

	-------	$106
093	bpl1mod	$108
094	bpl2mod	$10a
	-------	$10c
	-------	$10e
095	bpl1dat	$110
096	bpl2dat	$112
097	bpl3dat	$114
098	bpl4dat	$116
099	bpl5dat	$118
100	bpl6dat	$11a
	-------	$11c
	-------	$11e
101	sprpt[0]	$120
102	sprpt[1]	$124
103	sprpt[2]	$128
104	sprpt[3]	$12e
105	sprpt[4]	$130
106	sprpt[5]	$134
107	sprpt[6]	$138
108	sprpt[7]	$13c

spr[0]

109	pos	$140
110	ctl	$142
111	dataa	$144
112	datab	$146

spr[1]

113	pos	$148
114	ctl	$14a
115	dataa	$14c
116	datab	$14e

spr[2]

117	pos	$150
118	ctl	$152
119	dataa	$154
120	datab	$156

```
      ------      $0ae
aud[1]

069   ac_ptr      $0b0
070   ac_len      $0b4
071   ac_per      $0b6
072   ac_vol      $0b8
073   ac_dat      $0ba
      ------      $0bc
      ------      $0be

aud[2]

074   ac_ptr      $0c0
075   ac_len      $0c4
076   ac_per      $0c6
077   ac_vol      $0c8
078   ac_dat      $0ca
      ------      $0cc
      ------      $0ce

aud[3]

079   ac_ptr      $0d0
080   ac_len      $0d4
081   ac_per      $0d6
082   ac_vol      $0d8
083   ac_dat      $0da
      ------      $0dc
      ------      $0de
084   bltpt[0]    $0e0
085   bltpt[1]    $0e4
086   bltpt[2]    $0e8
087   bltpt[3]    $0ec
088   bltpt[4]    $0f0
089   bltpt[5]    $0f4
      --------    $0f8
      --------    $0fa
      --------    $0fc
      --------    $0fe
090   bplcon0     $100
091   bplcon1     $102
092   bplcon2     $104
```

150	color09	$192
151	color10	$194
152	color11	$196
153	color12	$198
154	color13	$19a
155	color14	$19c
156	color15	$19e
157	color16	$1a0
158	color17	$1a2
159	color18	$1a4
160	color19	$1a6
161	color20	$1a8
162	color21	$1aa
163	color22	$1ac
164	color23	$1ae
165	color24	$1b0
166	color25	$1b2
167	color26	$1b4
168	color27	$1b6
169	color28	$1b8
170	color29	$1ba
171	color30	$1bc
172	color31	$1be
...		
xxx	NO-OP	$1fe

spr[3]

121	pos	$158
122	ctl	$15a
123	dataa	$15c
124	datab	$15e

spr[4]

125	pos	$160
126	ctl	$162
127	dataa	$164
128	datab	$166

spr[5]

129	pos	$168
130	ctl	$16a
131	dataa	$16c
132	datab	$16e

spr[6]

133	pos	$170
134	ctl	$172
135	dataa	$174
136	datab	$176

spr[7]

137	pos	$178
138	ctl	$17a
139	dataa	$17c
140	datab	$17e

141	color00	$180
142	color01	$182
143	color02	$184
144	color03	$186
145	color04	$188
146	color05	$18a
147	color06	$18c
148	color07	$18e
149	color08	$190

Anhang E: Hinweise zur beiliegenden Diskette

Im Umschlag dieses Buches finden Sie eine Diskette, auf der alle in diesem Buch veröffentlichten Programme enthalten sind. Damit erübrigt sich das zeitintensive und oft mit Flüchtigkeitsfehlern versehene Abtippen aus dem Buch. Um einen von Anfang an problemlosen Umgang mit dieser Diskette und den auf ihr enthaltenen Programmen zu gewährleisten, finden Sie an dieser Stelle einige wichtige Hinweise.

Das Supergrafik-Buch ist in drei Hauptbereiche unterteilt:

a) Anfängerteil BASIC (Kapitel 1)
b) Fortgeschrittenes Programmieren BASIC (Kapitel 2 bis 8)
c) Programmieren in "C" (Kapitel 9 bis 19)

Ganz analog dazu finden Sie auf der beiliegenden Diskette drei Unterverzeichnisse mit den Namen:

a) Trapp
b) Weltner
c) Jennrich

Die Programme des Anfängerteils (Trapp) besitzen, im Gegensatz zu den Programmen in den anderen Schubladen, zudem noch .info-Dateien, können also auch via Maus geladen und gestartet werden. Dazu müssen Sie zunächst das Disketten-Icon und anschließend die Schublade mit dem Namen "Trapp" doppelt anklicken. Um nun problemlos mit den dann auf dem Bildschirm angezeigten Programmen arbeiten zu können, solltem Sie wie folgt vorgehen.

Vor jeder Nutzung der Supergrafikdiskette sollten Sie unbedingt eine Sicherheitskopie anfertigen und nur noch mit dieser arbeiten.

Nehmen Sie sich eine leere, formatierte Diskette und kopieren Sie das AmigaBASIC von Ihrer ExtrasD-Diskette auf diese leere Diskette. Danach sollten Sie die von den Programmen benötigten .bmap-Files auch auf diese Diskette kopieren. Sie finden diese .bmap-Files teilweise auf der ExtrasD-Diskette in der Schublade BasicDemos oder auf der Supergrafikdiskette. Wenn Sie nun die Diskette soweit vorbereitet haben, dann können Sie ein oder mehrere Programme aus der Trapp-Schublade von der Supergrafikdiskette auf diese Diskette kopieren. Anschließend legen Sie die Supergrafikdiskette weg, da Sie nun mit der neu angefertigten Diskette weiterarbeiten können. Um nun ein

Anhang D: Literaturverzeichnis

Braun

3-D-Grafik-Programmierung (ST)
Data-Becker Verlag 1986

ROM Kernel Reference Manuals

Libraries and Devices
Commodore Business Machines, Inc.
Addison - Wesley Publishing Company, Inc.

Exec
Commodore Business Machines, Inc.
Addison - Wesley Publishing Company, Inc

Hardware Reference Manual

Commodore Business Machines, Inc.
Addison - Wesley Publishing Company, Inc

Intuition Reference Manual

Commodore Business Machines, Inc.
Addison - Wesley Publishing Company, Inc

Rom Kernel Manual Volume 1

Commodore Amiga Inc.

Rom Kernel Manual Volume 2

Commodore Amiga Inc.

Anhang F: Stichwortverzeichnis

AllocMem siehe Amiga-Magazin 12/88 S134

Programm zu starten, brauchen Sie nur noch das entsprechende Programm-Icon doppelt anzuklicken.

Mit den Programmen aus der Weltner-Schublade können Sie prinzipiell genauso verfahren, allerdings muß hierbei das Kopieren über das CLI geschehen. Auch das Laden und Starten der Programme muß hier per Tastatur im Direktmodus des AmigaBASIC-Interpreters gemacht werden.

Die "C"-Programmierer arbeiten zweckmäßigerweise mit dem CLI und bekommen Zugriff auf ihre Programme mittels:

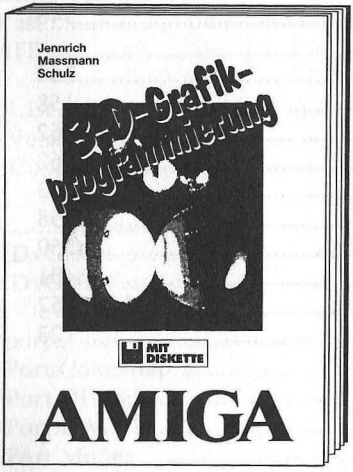
```
cd "df0:Jennrich"
```

Wie gewohnt tragen die Sources jeweils die Endung .c, und die Dateien ohne diese Endung sind direkt vom CLI aus startbar.

Viel Spaß mit dem Buch und der Diskette!

Bücher zum Amiga

Der Amiga hat ausgezeichnete Grafikfähigkeiten, aber vielen Amiga-Besitzern fällt es schwer, diese Fähigkeiten auch in eigenen Programmen auszuschöpfen – besonders, wenn es um dreidimensionale Darstellungen geht. Mit diesem Buch wird das einfach, weil nicht nur die entsprechenden Algorithmen ausführlich erklärt, sondern auch gleich die fertigen Lösungen angeboten werden.



Aus dem Inhalt:

- Grundlagen des Ray-Tracings
- Eingabe der dreidimensionalen Objekte über komfortablen Objekteditor
- Materialbestimmung über Materialeditor
- Automatische Berechnung in verschiedenen Auflösungen
- Alle Amiga-Auflösungen können genutzt werden (Lowres, Hires, Interlace, Hold and Modify)
- Verschiedene Lichtquellen, beliebiger Blickpunkt
- Grafiken können anschließend im IFF-Format gespeichert und dadurch in die gängigen Malprogramme eingelesen werden
- Mathematische Grundlagen auch für Nichtmathematiker
- Die Diskette im Buch enthält nicht nur alle Routinen als editierbare BASIC-Versionen, sondern auch die compilierten Fassungen, die deutlich schneller sind

Jennrich, Massmann, Schulz
3-D-Grafikprogrammierung
284 Seiten, DM 59,–
ISBN 3-89011-174-2