AUF DER HEFT-CD

Demo-Programmierung unter Windows 95/NT

Am laufenden Band

Einfacher Text paßt kaum zu bunten Grafikeffekten. Deshalb animieren Sie Laufschriften in vielen Variationen.

CARSTEN DACHSBACHER/
NILS PIPENBRINCK

Laufschriften – englisch Scroller genannt – kennen Sie aus dem alltäglichen Leben. Im Abspann von Kinofilmen, in News-Tickern und auf Werbetafeln ziehen wandernde Zeichen Ihre Aufmerksamkeit auf sich. Selbst Informationen, die eigentlich nicht auf die Anzeigetafel oder den Bildschirm passen, schieben Sie mit dieser Methode in Lesegeschwindigkeit weiter.

Wir schreiben hier zunächst einen einfachen Lauftext, den wir dann um zusätzliche Gimmicks wie Bewegungen und Spiegelungen erweitern. Außerdem erfahren Sie, wie Sie den Effekt der legendären *Star-Wars*-Laufschrift mit sehr einfachen Mitteln nachbilden. Zum Abschluß dieser Ausgabe integrieren Sie diese Effekte in ein Programm, mit dem Sie animierte Texte als ausführbare *exe*-Datei weitergeben können.

IN DER DATEI *CHAR.BMP* legen Sie den Font als Bitmap ab.

Erste holprige Schritte

Die einfachsten Scroller sind wirklich primitiv: Sie schieben lediglich eine Reihe kleiner Bilder mit Buchstaben über den Bildschirm. Unter Windows erzeugen Sie mit TrueType-Fonts zwar schnell Schriften, als Grundlage für Demo-Scroller haben diese allerdings einen entscheidenden Nachteil: Sie sind alle einfarbig.

Aus diesem Grund stellen wir Ihnen zunächst eine kleine C++-Klasse vor, mit der Sie auf einfache Weise Texte, Buchstaben und Laufschrift in Ihre Demos einbauen. Die Klasse *Font*, die Sie aus *font.cpp* und *font.h* generieren, verwaltet Position und Größe einzelner Zeichen in einer Bitmap. Zudem stellt sie Funktionen zur Verfügung, mit denen Sie Zeichen oder auch ganze Zeichenketten schnell und flexibel darstellen.

Die Implementierung der Klasse selbst ist etwas kompliziert. Wenn Sie daran interessiert sind, sehen Sie sich den gut kommentierten Quellcode auf der Heft-CD an.

Einfacher und interessanter ist da schon die Benutzung:

```
Font *myFont =
  new Font(„chars.bmp“);
```

So initialisieren Sie eine Instanz der *Font*-Klasse. Als Parameter übergeben Sie den Namen der Bitmap, die die Buchstaben enthält. Die *Font*-Klasse kann nur mit 256-Farben-Bitmaps umgehen.

Pixel mit dem Wert *0* haben eine spezielle Bedeutung: Sie werden beim Zeichnen ausgelassen. Damit können Sie Bereiche der Buchstaben transparent gestalten.

Mit folgenden Zeilen setzen Sie den Clipping-Bereich:

```
myFont->SetClipping
  (0,0,SCREEN_X, SCREEN_Y);
```

Die Routinen zum Zeichnen von Texten verfügen alle über ein eingebautes Clipping. Sie brauchen sich also keine Sorgen zu machen, daß Texte, die länger als der Bildschirm sind, zu Fehlern führen.

Legen Sie nun eine Tabelle an, die die Plazierung der Buchstaben in der Bitmap beschreibt. Jede Zeile der Tabelle entspricht einer Zeile Zeichen in der Bitmap. Vergleichen Sie dazu das folgende Array mit der Abbildung links.

```
static char *fonttable[] =
{
  „abcdefghij“,
  „klmnopqrst“,
  „uvwxyz „,
  „0123456789“,
  „?’.,!“,
  NULL
}

myFont->SetFontMetric
  (fonttable,16,18);
```

Die Routine *SetFontMetric* erwartet als Parameter diese Tabelle sowie die Maße des Rasters, in dem die Buchstaben angeordnet sind. Möchten Sie keine festen Raster verwenden, definieren Sie die Zeichen einzeln. Dazu benutzen Sie die Funktion *SetCharMetric*. Wir empfehlen jedoch, von vornherein die Buchstaben im Raster zu plazieren. Das spart eine Menge Arbeit.

Wenn Sie soweit sind, können Sie losscrollen:

```
unsigned char *text =
  „Dies ist ein Lauftext“;

while (DemoRunning)
{
  //Hintergrund kopieren
  memcpy (screen,hintergrund2,
    SCREEN_Y*SCREEN_X*2);

  //Scroller darüber zeichnen
  //(30 Pixel/sec verschieben)

  int x = SCREEN_X-
    ((GetDemoTime()-StartZeit)
    *30)/1000;
  myFont->Print(screen,palette,
    SCREEN_X,text,x,120);

  //... und Bild darstellen
  BlitGraphic(screen);
}
```

Wenn Sie diesen Code ausführen, stellen Sie fest, daß die Laufschrift ruckelt. Das liegt an der ungleichmäßigen Verschiebung der Laufschrift. Eine gleichmäßige

Bewegung erhalten Sie nur, wenn Sie pro Bildaufbau die Laufschrift immer um den gleichen Betrag verschieben. Leider können Sie dagegen wenig machen, da Windows keine brauchbaren Methoden zur Synchronisation mit der Grafikkarte anbietet. Mit GDI (Graphics Device Interface) haben Sie keine Chance zu erfahren, wann die Videokarte das Bild neu aufgebaut hat.

Unter DirectDraw sieht es da etwas besser aus. Der Aufruf von *BlitGraphic* stellt sicher, daß Sie nicht mehr Bilder pro Sekunde darstellen, als die Bildwiederholfrequenz des Monitors zuläßt, der am Rechner angeschlossen ist.

Aber auch dies hilft Ihnen nur bedingt weiter: Sie wissen ja nicht, ob der Benutzer seinen Monitor mit niedrigen 50 oder mit 90 Hz oder mehr betreibt. Sie können nun zwar sicher sein, daß die Laufschrift nicht mehr ruckelt – dafür haben Sie aber keine Kontrolle mehr darüber, wie schnell der Scroller läuft.

Da Sie also nicht viel gegen dieses Manko ausrichten können, sorgen Sie am besten dafür, daß der Fehler nicht so auffällt. Dazu gestalten Sie die Bewegung der Buchstaben etwas komplizierter. Das Auge des Betrachters verliert dadurch die Orientierung und nimmt das Ruckeln weniger stark wahr.

Elegant hüpfende Zeichen

Diese Idee möchten wir Ihnen mit einem Scroller demonstrieren, der die Höhe der einzelnen Buchstaben anhand einer Tabelle verändert:

```
for (char *zeichen =
  aText; *zeichen; zeichen++)
{
  //Zeichen sichtbar?
  if ((x+(signed) aFont->info
    [*zeichen].w)>=0)
  {
    //Höhe berechnen
    int hoehe =
      y-bewegungs_tabelle
      [x & 1023];
    //Zeichen zeichnen
    aFont->DrawChar
      (dest,palette,SCREEN_X,
      *zeichen,x,hoehe);
  }
  //bis über den rechten Rand
  //hinaus fortfahren
  x+=aFont->info[*zeichen].w+2;
  if (x>SCREEN_X) return;
}
```

Dieser Code-Ausschnitt entspricht weitgehend der Methode *Print* der *Font*-Klasse. Allerdings ist die Höhe der Buchstaben nun abhängig von der x-Koordinate. Der Scroller zeichnet den Text also nicht mehr in eine Zeile, sondern versetzt jeden Buchstaben etwas in der Höhe. Dadurch vermeiden Sie zwar nicht das Ruckeln, aber es ist nicht mehr ganz so störend. Sie finden den kompletten Code zu den beweglichen Laufschriften in den Dateien *sinscrol.cpp* und *sinscrol.h*.

Sie können jetzt Ihrer Fantasie freien Lauf lassen und schöne Bewegungen und Erweiterungen programmieren. Als Anregung finden Sie bei den Quellcodes zum Artikel eine Laufschrift, die aussieht, als würde sie von einer spiegelglatten Oberfläche reflektiert.

Laufschrift mit Perspektive

Einen wahrlich klassischen Effekt bietet der sogenannte *Star-Wars*-Scroller. Wie im Vorspann der gleichnamigen Filme schiebt sich eine Laufschrift in den (Welt-)Raum und verschwindet dann langsam – immer dunkler werdend – im Nichts. Der *Star-Wars*-Scroller ist ein Paradebeispiel für die gekonnte Anwendung zweier Haupttechniken der Computergrafik: Tabellen und lineare Interpolation.

Dabei benutzen Sie eine Bitmap-Datei, die den gewünschten Text enthält. Entweder Sie füllen die Bitmap zur Laufzeit mit den Font-Routinen, oder Sie erledigen dies vorher mit einem Zeichenprogramm wie Paint Shop Pro.

Hauptaufgabe des Scrollers ist es, die einzelnen Zeilen der Bitmap-Textur auf unterschiedliche Breiten zu skalieren. Im Prinzip ist das nichts anderes als eine vereinfachte Form des Texture-Mappings (Abbildung einer Textur auf ein Objekt). Nur brauchen Sie sich hier lediglich um eine Achse – die x-Achse – zu kümmern.

Die nötigen Streckungen berechnen Sie sehr effizient mit Fixed-Point-Zahlen. Sind Sie mit dieser Methode nicht vertraut, erklären wir Ihnen hier kurz die Grundlagen: Nehmen Sie an, Sie haben eine Zeile Bilddaten in einem Array gespeichert. Diese Zeile sei 256 Pixel breit. Um sie auf eine Länge von 100 Pixeln zu verkleinern, lassen Sie einige Pixel aus.

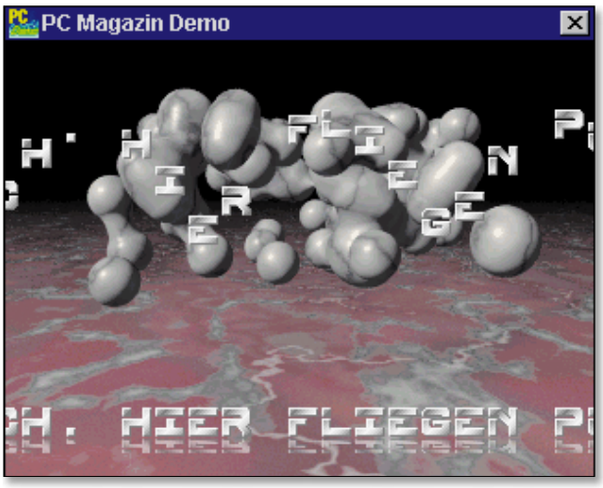
Hierfür benutzten Sie die Technik der Fixed-Point-Berechnung. Ein Code, der Ihr Problem löst, sieht so aus:

```
for (int x=0; x<100; i++)
  Zielbitmap[i] =
    Textur[(x * 256)/100];
```

Das funktioniert sehr gut, aber Sie haben pro Pixel eine Multiplikation und eine Division zu berechnen. Insbesondere Divisionen sind „sehr teuer“, was den Rechenaufwand und somit die Geschwindigkeit betrifft. Daher entfernen Sie die Division aus der Schleife:

```
int Steigung=100/256;
int Start = 0;
for (int x=0; x<100; i++)
{
  Zielbitmap[i]=Textur[Start];
  Start=Start+Steigung
}
```

Dieser Code wäre erheblich schneller – er funktioniert aber nicht, da die Variable *Steigung* ein Integer ist und der Wert der Division eine Fließkommazahl.



DIE AUF- UND ABBEWEGUNG der Buchstaben verschleiert das lästige Ruckeln.

Der Trick der Fixed-Point-Zahlen ist es, die Genauigkeit der Integer-Werte zu erhöhen, indem Sie einige Bits der Zahl für die Nachkommastellen nutzen. Die Umwandlung von Integer in das Fixed-Point-Format geschieht durch eine einfache Multiplikation und Division. Die Fixed-Point-Variante in unserem Beispiel sieht dann so aus:

```
int Steigung=(65535*100)/256;
int Start=0;
for (int x=0; x<100; i++)
{
  Zielbitmap[i] =
    Textur[Start/65536];
  Start=Start+Steigung.
}
```

Der Faktor 65 536 wurde mit Bedacht gewählt. Er entspricht dem Wert 2^{16}, ▶

denn Multiplikationen und Divisionen mit Potenzen von 2 führt die CPU durch einfache Bit-Shift-Befehle sehr schnell aus.

Zusätzlich teilt dieser Wert eine Integer-Zahl in genau zwei Hälften: 16 Bit für den ganzzahligen Teil und 16 Bit für die Nachkommastellen. Zugunsten der erhöhten Genauigkeit verlieren Sie 16 Bit im Wertebereich Ihres Integers.

Das Format von Fixed-Point-Zahlen wird gerne mit Doppelpunkten angegeben. Im Beispiel haben Sie es mit dem weit verbreiteten 16:16-Fixed-Point-Format zu tun. Aber auch andere Formate wie 8:24 sind häufig anzutreffen. Die Zahl vor dem Doppelpunkt gibt die Anzahl der Bits für den ganzzahligen Wert an, die dahinterstehende für die Genauigkeit-Bits. So viel zur Skalierung mit Hilfe von Fix-Point-Zahlen.

Auch die Breite der einzelnen Zeilen bekommen Sie ohne großen Aufwand: Es gibt einen Fluchtpunkt, in dem die Laufschrift verschwindet. Die Zeilen darunter werden zunehmend breiter. Alle diese Daten können Sie vor dem Zeichnen berechnen und in einer Tabelle speichern. Im Beispielprogramm erledigt dies die Funktion *calculate_scroller_table* in der Datei *StarScrol.cpp*.

Für die Darstellung auf dem Bildschirm bleibt nicht mehr viel zu rechnen übrig. Aus der Tabelle lesen Sie zeilenweise alle Informationen wie Breite, Startposition und die Position in der Textur aus und zeichnen den Text mit einer einfachen Schleife auf den Monitor. Der Scroll-Effekt kommt zustande, indem Sie bei jedem Bildaufbau die Textur etwas nach oben schieben. Im Beispielcode haben wir 32 Paletten berechnet und lassen damit den Scroller – je weiter er sich dem Fluchtpunkt nähert – dunkler werden. Da Sie die Scroller-Bitmap beim Zeichnen sowieso von 8 Bit in 16 Bit umwandeln, macht dies zeitlich kaum einen Unterschied.

Werfen Sie am besten einen Blick auf die Funktion *StarwarsScroller* im Modul *StarScrol.cpp*. Dort finden Sie noch einige Tricks, die den Scroller etwas schneller machen. So merkt sich die Routine in einer Tabelle, welche Zeilen der Textur schwarz sind und nicht interpoliert werden müssen.

BEIM STAR-WARS-SCROLLER verschwindet die Schrift perspektivisch im Nichts.

Bilineare Interpolation des Scrollers

Sehen Sie sich den *Star-Wars*-Scroller genau an: Im unteren Bildbereich werden die Buchstaben stark vergrößert. Dadurch ergeben sich unschöne Kanten, die den Eindruck stark beeinträchtigen. Diesem Problem begegnen Sie mit einer Technik, deren Name im Zeitalter der 3D-Grafikkarten jedem bekannt sein dürfte: der bilinearen Interpolation. Das Prinzip ist schnell programmiert: Beim Auslesen der Textur-Pixel mit der Fixed-Point-Berechnung kommt es immer vor, daß Sie ein Pixel nicht genau treffen, denn während der Umrechnung von Fixed-Point auf Integer werfen Sie die Information des Nachkommaanteils ja einfach weg.

In der Skizze auf der nächsten Seite ist das linke Pixel weiß, das rechte schwarz. Ein Punkt, der zwischen beide Pixel fällt, sollte vorzugsweise die Farbe Grau erhalten.

Mit ein wenig Mathematik ist dies kein Problem. Bei der Umwandlung in Integer verlieren Sie durch die Division einen Betrag, der im Prinzip die exakte Position zwischen den beiden Pixeln angibt. Diesen Wert extrahieren Sie aus der Fixed-Point-Zahl, indem Sie den ganzzahligen Teil ausmaskieren. Bei 16:16-Zahlen genügt eine *Und*-Verknüpfung mit der Maske 65 535.

Den interpolierten Wert erhalten Sie in Pseudo-Code dann wie folgt:

```
a = Wert des Pixels bei [x]
b = Wert des Pixels bei [x+1]
wert = Nachkomma-Anteil der
  Fixed-Point-Zahl
Punkt = a + ((b-a)*wert)/65536
```

Dies ist jedoch erst eine einfache lineare Interpolation. Für eine bilineare – also zweidimensionale – Interpolation, wie sie bei 3D-Karten üblich ist, bilden Sie zwischen vier Punkten einen Mittelwert.

Dazu führen Sie drei lineare Interpolationen durch, denn Sie suchen einen Wert, der sowohl zwischen zwei Texturzeilen als auch zwischen zwei Spalten liegt. Interpolieren Sie also beide Zeilen einzeln mit dem Nachkommaanteil der x-Koordinate. Die beiden Ergebnisse interpolieren Sie mit der y-Koordinate:

```
int xfixed=X-Koordinate
  im 16:16-Format
int yfixed=Y-Koordinate
  im 16:16-Format

int x=xfixed/65536;
int y=yfixed/65536;

a=texture[x][y] //oben links
b=texture[x+1][y] //oben rechts
c=texture[x][y+1] //unten links
d=texture[x+1][y+1]
//unten rechts

wert_oben =
  linear(a,b,xfixed & 65535);
wert_unten =
  linear(c,d,xfixed & 65535);

wert = linear(wert_oben,
  wert_unten,yfixed & 65536)
```

Drei lineare Interpolationen pro Pixel verbrauchen eine Menge Zeit, aber das Ergebnis ist deutlich besser: Der Scroller

STEUERCODES FÜR DEN LETTER-WRITER

Code	Parameter	Funktion
COLORx	–	Setzt vordefinierte Zeichenfarbe x (x = 1-8)
FLASHx	–	Setzt vordefinierte blinkende Zeichenfarbe x (x = 1-8)
FADE	Helligkeit (0-255)	Blendet auf bestimmte Helligkeit um
FSPEED	Geschwindigkeit	Setzt Geschwindigkeit folgender *FADE*-Aufrufe
SPEED	Geschwindigkeit (0-255)	Setzt Geschwindigkeit der Zeichenbewegung
DELAY	Dauer in msec (0-255)	Wartet zwischen Erscheinen zweier Zeichen
CLRSCR	–	Löscht Bildschirm und setzt Cursor links oben
MOVE	Anzahl der Zeichen	Bewegt Cursor um bestimmte Zeichenzahl weiter
MOVE1_2	–	Setzt Cursor ein halbes Zeichen nach rechts
NEWLINE	–	Setzt Cursor auf Beginn der nächsten Zeile
FEED1_2	–	Setzt Cursor eine halbe Zeile nach unten
PAUSEx	–	Wartet x Sekunden (x = 1-4)
EOT	–	„End of Text“, beendet Programm

sieht wesentlich runder aus, und die durch Interpolation entstandenen Kanten sind verschwunden.

Beachten Sie, daß Interpolation nur dort Sinn macht, wo Sie eine Textur vergrößern – bei unserem Scroller also nur im unteren Bildschirmbereich. Beim Verkleinern wenden Sie das Interpolieren besser nicht an. Statt dessen empfiehlt sich hier MIP-Mapping, um die Qualität – und auch die Geschwindigkeit – Ihrer Routinen zu erhöhen.

Fliegende Buchstaben

Früher war es Mode – vor allem auf dem Commodore 64 –, *readme*-Dateien und Nachrichten in Form eigener Programme zu verbreiten. Diese sogenannten Letter-Writer zeigten einen Text auf besondere Art und Weise an. So flogen die Buchstaben zunächst an den Ort, an dem sie stehen sollten, und der Text konnte blinken oder seine Farbe ändern.

Mit Ihrem Wissen über Laufschriften schreiben Sie nun auch so einen Textbetrachter. Bei *Flying Letters* – so der Name unseres Letter-Writers – erscheint der Text zunächst Buchstabe für Buchstabe. Danach fliegen die einzelnen Zeichen an ihren Platz.

Den genauen Ablauf beeinflussen Sie dabei über Steuercodes, die Sie in den Text einfügen. Sie können die Farben ändern, die Zeichen blinken lassen, die Fluggeschwindigkeit und die Zeit zwischen dem Erscheinen der neuen Zeichen festlegen, den Text genau positionieren oder das Bild ein- und ausblenden.

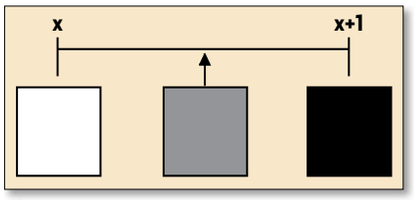
Um diese Funktionalität zu bieten, benötigen Sie Variablen für

- die Position des (unsichtbaren) Cursors auf dem Bildschirm,
- die aktuell gesetzte Zeichenfarbe,
- die verbleibende Zeit einer eventuellen Pause,
- die Verzögerung zwischen zwei Zeichen,
- die Geschwindigkeit der Zeichen,
- die gewünschte Helligkeit, auf die Sie umblenden möchten,
- die Geschwindigkeit des Umblendens sowie
- den aktuellen Helligkeitswert.

Weiterhin müssen Sie natürlich die auf dem Bildschirm sichtbaren Schriftzeichen verwalten. Dazu speichern Sie, um welches Zeichen es sich handelt, sowie dessen Farbe. Zudem benötigen Sie die Zielposition, die Bewegungsrichtung dorthin und natürlich die aktuelle Position.

Die dazugehörigen Strukturdefinitionen in C finden Sie im Listing am Ende des Artikels. In der Hauptschleife überprüft *Flying Letters* zuerst, ob eine Pause eingehalten werden soll. Ist das der Fall, zieht es so lange die seit dem letzten Schleifendurchlauf vergangene Zeit von der verbliebenen Pausenzeit ab, bis diese kleiner oder gleich Null ist.

Ist keine Pausenanweisung vorhanden, liest das Programm das jeweils nächste Zeichen und untersucht, ob es ein Textzeichen oder ein Steuercode ist. Dabei nutzt es die Tatsache, daß alle Buchstaben, Zahlen und die Sonderzeichen aus dem eigens erstellten Zeichensatz einen ASCII-Code kleiner als 128 besitzen. Die „oberen" 128 Zeichen dienen deshalb als Steuercodes. Die verschiedenen Codetypen (Ablaufsteuer-, Farb- und Pausencode) unterscheiden



DANK LINEARER INTERPOLATION erhält das gesuchte Pixel die Farbe Hellgrau.

Sie anhand vordefinierter Bitmasken.

Nachdem der Letter-Writer den nächsten Wert aus den Daten gelesen hat, prüft er zuerst, ob es sich um das *EOT*-Kommando handelt. In diesem Fall gibt er die angeforderten Ressourcen frei und beendet den Programmlauf. Alle anderen Steuercodes identifizieren Sie mit einem Ausdruck wie

```
if ((code & STEUERCODE) ==
  STEUERCODE)
```

Haben Sie den Code zum Beispiel als einen Ablaufsteuercode erkannt, können Sie mit folgender Konstruktion

```
switch (code) { case ... }
```

direkt die Auswirkungen programmieren. Bei einem Farb- oder Pausencode benötigen Sie keine spezielle Fallunterscheidung, wenn Sie die Konstanten so wie in unserem Beispiel definieren. In diesem Fall berechnen Sie die Farbnummer oder die Pausenlänge direkt aus dem Wert.

Konnten Sie keinen Steuercode ausmachen, handelt es sich sicher um ein Zeichen, das am Bildschirm erscheinen soll. In diesem Fall tragen Sie die entsprechenden Daten – Zielposition, Farbe, Bewegungsrichtung usw. – in das nächste freie Zeichenkonstrukt ein und erhöhen dessen Zähler. Für die Startposition des Zeichens verwendet das Beispielprogramm eine von der Zeit abhängige Sinus-/Cosinus-Funktion.

Nun bleiben noch zwei Aufgaben in der Hauptschleife des Letter-Writers, die auch ausgeführt werden, wenn eine Pause vorliegt. Die erste Aufgabe ist natürlich das Weiterbewegen der Zeichen und das Zeichnen an der neuen Position. Davor löschen Sie den Bildschirm.

Die Zeichen sollten sich – unabhängig von der Rechnergeschwindigkeit und eventueller Verzögerungen durch das Betriebssystem – gleichmäßig schnell bewegen. Deshalb berechnen Sie mit der Prozedur *GetDemoTime()* aus der verwendeten Grafikbibliothek *demosys.cpp* die verstrichene Zeit seit dem letzten Weiterbewegen der Zeichen. Dadurch erhalten Sie einen Faktor für den Richtungsvektor jedes Zeichens, den Sie dann auf die aktuelle Position aufsummieren. Hat ein Zeichen seine Zielposition erreicht, stellen Sie nur noch sicher, daß es jedesmal an der entsprechenden Stelle gezeichnet wird.

Die zweite Aufgabe ist das Ein- und Ausblenden der Helligkeit – das sogenannte Fading. Helligkeitsänderungen führen Sie geschickt durch, indem Sie die Einträge der Palette modifizieren, die Sie zum Zeichnen der Zeichen benutzen. Die Paletteneinträge sind – jeweils für Rot, Grün und Blau – das Produkt der aktuellen Helligkeit mit dem entsprechenden Wert in der zuvor im Programm definierten Farbtabelle.

Die momentane Helligkeit berechnen Sie einfach: Der Steuercode *FADE* informiert Sie über den angestrebten Helligkeitswert. Da Sie den aktuellen Wert ebenfalls kennen, wissen Sie auch, ob Sie ihn erniedrigen oder erhöhen müssen.

Sie führen also eine Addition bzw. Subtraktion der aktuellen *Fade*-Variablen mit dem durch *FSPEED* festgelegten Betrag durch, solange Sie die Zielhelligkeit nicht erreicht haben. Den Faktor für die Berechnung der Farbpalette erhalten Sie nach einer Division dieses *Fade*-Werts durch 256, da der Wertebereich aufgrund des *char*-Arrays von *0* bis *255* und nicht von *0.0* bis *1.0* reicht.

Kleiner Aufwand, große Wirkung

Um Ihren Text zu animieren, editieren Sie lediglich ein Konstantenfeld. Im Programm finden Sie das konstante Ar-

ray *„char *WRITE[]“*, in dem Sie sowohl Text als auch den Ablauf festlegen. Normalen Text geben Sie in Anführungszeichen an, die Konstanten der Steuercodes finden Sie in der Tabelle auf S. 262 mit einer Kurzbeschreibung aufgelistet.

Generell existieren zwei Arten von Steuercodes: Die einfachen Befehle mit festgelegter Wirkung fügen Sie wie gewöhnliche Zeichenketten ein. Die übrigen Kommandos verlangen einen Parameter, mit dem Sie bestimmte Eigenschaften wie die Geschwindigkeit beim Ein- und Ausblenden festlegen. Dieser Parameter folgt direkt auf den Steuerbefehl, eine Zahl wie 123 geben Sie dabei als *„(char)123“* an. Ein einfacher Text, der ein blinkendes *PC Underground* auf den Bildschirm fliegen läßt, sieht dann so aus:

```
char *WRITE[] =
{
//Zeichengeschwindigkeit = 80
  SPEED,(char*)80,
  //Zeichenverzögerung = 80
  DELAY,(char*)80,
  //2,5 Zeilen nach unten
  NEWLINE,NEWLINE,FEED1_2,
  //3 Zeichen nach rechts
  MOVE,(char*)3,
  //blinkende Farbe und Text
  FLASH1,“PC Underground“,
  //3 Sekunden Pause
  PAUSE3,
  //mit Geschwindigkeit 4
  FSPEED,(char*)4,
  //nach Schwarz (0) abblenden
  FADE,(char*)0,
  //dazu 3 Sekunden warten
  PAUSE3,
  //Writer beenden
  EOT
};
```

Um auf die einzelnen Zeichen zugreifen zu können, konvertiert der Letter-Writer dieses Array aus Zeigern vor dem Start in eine Zeichenkette

```
char *array
```

Durch diesen Umweg können Sie die Steuercodes direkt als Strings im Text eingeben und müssen nicht spezielle Sonderzeichen benutzen.

Ab der nächsten Ausgabe startet PC Magazin eine mehrteilige Serie zum Thema Spiele-Programmierung. Viele der hier gezeigten Effekte wenden Sie dort in einem richtigen Spiel an. PEI/BM

Die Quelltexte der Laufschriften, die zugrundeliegende Grafikbibliothek sowie das komplette Programm *Flying Letters* finden Sie auf unserer Heft-CD im Verzeichnis *praxis\underground* und im Internet-Angebot von *PC Magazin* unter

```
www.pc-magazin.de/magazin/
➥extras.htm
```

Klicken Sie unter *Online Extras* im Menü *Praxis* auf das entsprechende *Download*-Feld.

Steuercodes/Strukturdefinitionen in main.cpp

```
 1: //Bitmasken für Steuercodes
 2: #define STEUERCODE 128+64  //Ablauf-
    codes
 3: #define COLORCODE  128+32  //Farben-
    codes
 4: #define PAUSECODE  128+16  //Pausen-
    codes
 5:
 6: #define EOT (char*)255  //End of Text
 7:
 8: #define COLOR1 (char*)( 0|COLORCODE)
 9: #define COLOR2 (char*)( 1|COLORCODE)
10: #define COLOR3 (char*)( 2|COLORCODE)
11: #define COLOR4 (char*)( 3|COLORCODE)
12: #define COLOR5 (char*)( 4|COLORCODE)
13: #define COLOR6 (char*)( 5|COLORCODE)
14: #define COLOR7 (char*)( 6|COLORCODE)
15: #define COLOR8 (char*)( 7|COLORCODE)
16: #define FLASH1 (char*)( 8|COLORCODE)
17: #define FLASH2 (char*)( 9|COLORCODE)
18: #define FLASH3 (char*)(10|COLORCODE)
19: #define FLASH4 (char*)(11|COLORCODE)
20: #define FLASH5 (char*)(12|COLORCODE)
21: #define FLASH6 (char*)(13|COLORCODE)
22: #define FLASH7 (char*)(14|COLORCODE)
23: #define FLASH8 (char*)(15|COLORCODE)
24:
25: #define NEWLINE (char*)(0|STEUERCODE)
26: #define CLRSCR  (char*)(1|STEUERCODE)
27: #define MOVE1_2 (char*)(2|STEUERCODE)
28: #define FEED1_2 (char*)(3|STEUERCODE)
29: #define MOVE    (char*)(4|STEUERCODE)
30:
31: //+Zeichenzahl
32: #define FADE    (char*)(5|STEUERCODE)
33: //+Ziel-Helligkeit
34: #define FSPEED  (char*)(6|STEUERCODE)
35: //+Geschwindigkeit beim Umblenden
36: #define SPEED   (char*)(7|STEUERCODE)
37: //+Geschwindigkeit der Zeichenbewe-
    gung
38: #define DELAY   (char*)(8|STEUERCODE)
39: //+Dauer der Verzögerung
40:
41: #define PAUSE1 (char*)(0|PAUSECODE)
42: #define PAUSE2 (char*)(1|PAUSECODE)
43: #define PAUSE3 (char*)(2|PAUSECODE)
44: #define PAUSE4 (char*)(3|PAUSECODE)
45:
46: //Definition eines "fliegenden" Zei-
    chens
47:
48: typedef struct
49: {
50:   float x,y;    //aktuelle Position
51:   float zx,zy;  //Endposition
52:   float dx,dy;  //Richtung dorthin
53:   int color;    //Farbe (0-15)
54:   char c;       //Zeichen-Code
55: } ZEICHEN;
56:
57: //Definition der Variablen des Wri-
    ters
58:
59: typedef struct
60: {
61:   int pos;      //Position im Daten-
    array
62:   int color;    //aktuelle Zeichen-
    farbe
63:   int pause;    //Wartezeit
64:   int x,y;      //aktuelle Ziel-Posi-
    tion
65:   int delay;    //Verzögerung
66:   int speed;    //Zeichen-Geschwin-
    digkeit
67:   int fade_to,  //Fading-Werte
68:       fade,
69:       fade_speed;
70:   //Platz für die Zeichen am Bild-
    schirm
71:
72:   ZEICHEN zeichen[2000];
73:   //Anzahl benutzter ZEICHEN-Einträge
74:   int anz_zeichen;
75: } WRITER_STATUS;
76:
```

Die Belegung der Steuercodes und die Variablen-Strukturen definieren Sie im Hauptmodul *main.cpp* des Letter-Writers.