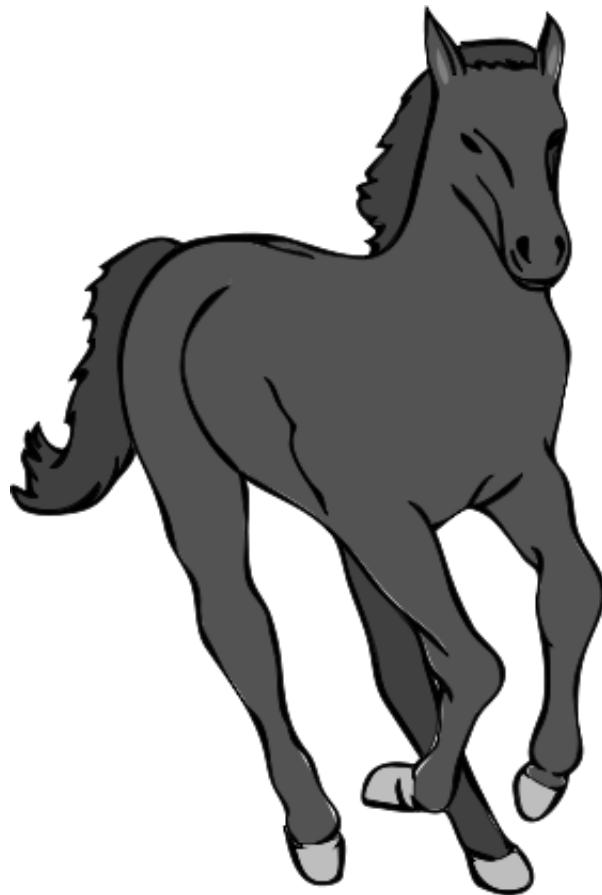


2D-Spieleprogrammierung in FreeBASIC

Ein anwendungsorientierter Leitfaden



von Stephan Markthaler

Stand: 13. Dezember 2011

Einleitung

1. Über dieses Buch

Dieses Buch behandelt die 2D-Spieleprogrammierung in FreeBASIC. Neben einigen grundsätzlichen Überlegungen über Struktur und Aufbau eines Spiels will es Grundlagen zur Spielsteuerung, Speicherung der Daten sowie für den Einsatz von Grafik und Soundeffekten vermitteln.

2. Für wen ist dieses Buch gedacht?

Das Buch ist für FreeBASIC-Programmierer geschrieben, die sich bereits mit den ersten Grundlagen der Sprache vertraut gemacht haben und einen Leitfaden für anwendungsorientierte Spieleprogrammierung suchen. Um vom Buch profitieren zu können, sollten Ihnen die allgemeinen Kontrollstrukturen, Umgang mit Variablen und Arrays sowie Prozeduren vertraut sein.

Auch wenn einige allgemeine Überlegungen zur Spieleprogrammierung auch für andere BASIC-Dialekte und weitere Programmiersprachen gelten, ist der größte Teil doch speziell auf FreeBASIC zugeschnitten.

3. Was sollten Sie von diesem Buch nicht erwarten?

Wie bereits erwähnt, eignet sich das Buch nicht, um das Programmieren von Grund auf zu erlernen. Gewisse Kenntnisse in der Programmierung werden bereits vorausgesetzt.

Ebenso wenig dürfen Sie erwarten, im Laufe des Buches ein komplettes (und möglicherweise auch noch geniales) Spiel vorgesetzt zu bekommen. Alle enthaltenen Beispiele dienen dazu, Ihnen Anwendungsmöglichkeiten vorzustellen. Die Aufgabe, aus diesen Inhalten ein (möglicherweise geniales) Programm zu erstellen, liegt bei Ihnen. Bedenken Sie jedoch: Programmierung ist eine Angelegenheit, die viel Übung und Erfahrung erfordert; deshalb lassen Sie sich nicht entmutigen, wenn es eine Weile dauert, bis Ihr Programm Ihren Vorstellungen entspricht. Windows wurde schließlich auch nicht an einem Tag programmiert.

4. Welcher Compiler wird benötigt?

Die Inhalte des Buches bauen auf dem FreeBASIC-Compiler fbc v0.23 auf. Die meisten Aussagen sind auch für frühere und (vermutlich) für spätere Versionen des Compilers gültig, doch kann dafür keine Garantie übernommen werden. Den aktuellen Compiler sowie umfangreiche Informationen zu FreeBASIC erhalten Sie u. a. unter der Adresse <http://freebasic-portal.de>

5. Warum 2D-Programmierung?

FreeBASIC stellt von Haus aus eine leicht zu bedienende Bibliothek zur Ausgabe von Grafik zur Verfügung. Damit können ohne allzu großen Aufwand 2D-Anwendungen allein mit „Bordmitteln“ erstellt werden. 3D-Anwendungen sind zwar mit FreeBASIC-Bordmitteln ebenfalls machbar, aber ungleich aufwändiger. Sollten Sie vorhaben, ein 3D-Spiel zu erstellen, dann sollten Sie ernsthaft in Erwägung ziehen, eine geeignete Grafikkbibliothek wie z. B. OpenGL zu nutzen.

6. Rechtliches

Das Dokument unterliegt der Lizenz „Creative Commons Namensnennung – Nicht-kommerziell – Weitergabe unter gleichen Bedingungen 3.0 Deutschland“ (CC BY-NC-SA 3.0). Sie sind berechtigt, das Werk zu vervielfältigen, verbreiten und öffentlich zugänglich zu machen sowie Abwandlungen und Bearbeitungen des Werkes anzufertigen, sofern dabei

- der Name des Autors genannt wird
- das Werk nicht für kommerzielle Nutzung verwendet wird und
- eine Bearbeitung des Werkes unter Verwendung von Lizenzbedingungen weitergegeben wird, die mit denen dieses Lizenzvertrages identisch oder vergleichbar sind.

Abwandlungen des Werkes müssen als solche erkennbar gemacht werden. Einen vollständigen Lizenztext erhalten Sie unter

<http://creativecommons.org/licenses/by-nc-sa/3.0/de/legalcode>

Unabhängig von oben genannten Lizenzbedingungen gestattet der Autor eine kommerzielle Nutzung der Print-Version dieses Dokuments im Rahmen des FreeBASIC-Portals Deutschland (<http://freebasic-portal.de>), sofern die erzielten Einnahmen der FreeBASIC-Community zugute kommen.

Die im Buch enthaltenen Quelltexte dürfen, auch zu kommerziellem Zweck, frei verwendet werden. Bei aufwändigeren Quelltexten freut sich der Autor über eine namentliche Nennung.

7. Weitere Informationen

Sämtliche längeren Quelltexte stehen auf der Projektseite

<http://freebasic-portal.de/projekte/59>

zum Download zur Verfügung. Dort erhalten Sie gegebenenfalls auch weitere Informationen.

8. Danksagung

Mein Dank gilt allen FreeBASICern, die direkt oder indirekt an der Entstehung des Buches mitgewirkt haben. Besonders erwähnen möchte ich

- Sebastian für die Bereitstellung des FreeBASIC-Portals
- MOD für das Korrekturlesen und den ganzen Rest
- ThePuppetMaster und Volta für ihre Beiträge zur FreeBASIC-Gemeinde, die zum Teil in diesem Buch Erwähnung finden

Inhaltsverzeichnis

Einleitung	ii
1. Über dieses Buch	ii
2. Für wen ist dieses Buch gedacht?	ii
3. Was sollten Sie von diesem Buch nicht erwarten?	ii
4. Welcher Compiler wird benötigt?	iii
5. Warum 2D-Programmierung?	iii
6. Rechtliches	iii
7. Weitere Informationen	iv
8. Danksagung	iv
I. Das Labyrinth	1
1. Vorüberlegungen	2
1.1. Anforderungen des Programms	2
1.2. Realistische Selbsteinschätzung	2
1.3. Erweiterbarkeit	2
1.4. Wartbarkeit des Codes	3
2. Das Spielfeld	4
2.1. Interne Speicherung der Leveldaten	4
2.2. Leveldaten im Quelltext	6
2.3. Leveldaten in einer externen Datei	8
2.3.1. ASCII-Speicherung	8
2.3.2. binäre Speicherung	9
2.3.3. Sprechende Werte	10
2.3.4. Vergleich: ASCII vs. binär	11
3. Steuerung	12
3.1. Tastatursteuerung	12
3.1.1. Tastaturabfrage mit INKEY	12

3.1.2. Tastendruck über MULTIKEY	14
3.2. Joysticksteuerung	14
3.3. Maussteuerung	16
3.4. Beschleunigung	17
4. Grafik	19
4.1. Initialisierung des Grafikfensters	19
4.2. Grundlegende Grafikroutinen	20
4.3. Zeichnen in den Grafikpuffer	20
4.4. Hintergrundgrafik sichern	21
4.5. Externe Grafiken einbinden	24
4.6. Textausgabe	26
4.7. Double Buffering	28
5. Spielelemente	30
5.1. Spielobjekte	30
5.2. Untergrund	31
5.3. Eigener Datentyp	31
5.4. Verknüpfung von Spielobjekten	32
5.5. Zeitgesteuerte Ereignisse	33
6. Anwendung: ein Minensuchspiel	35
6.1. Spielelemente und Bombenverteilung	35
6.2. Felder aufdecken	36
6.3. Hauptprogramm	38
6.4. Zusammenfassung	39
II. Anbauten	40
7. Highscore	41
7.1. Highscore einlesen	41
7.2. Highscore schreiben	42
7.3. Highscore bearbeiten	43
8. Nebenläufigkeit	44
8.1. Aufbau	44
8.2. Optimierung für Multicore-Prozessoren	46
8.3. Fehlerquellen	48

9. Externe Bibliotheken	50
9.1. Einbindung externer Bibliotheken	50
9.2. Sound und Musik	51
9.2.1. Wiedergabe von Ogg-Vorbis-Dateien	51
9.2.2. Wiedergabe von Trackermoduldateien	52
9.3. Rotation und Skalierung	53
9.4. Weitere Bibliotheken	54
10. Erweiterte Grafikprogrammierung	56
10.1. Animation	56
10.2. Scrolling	57
10.3. Parallax-Scrolling	59
11. Kollisionskontrolle	61
11.1. Kontrolle anhand von Farbwerten	61
11.2. Erstellung einer Maske	62
11.3. Kontrolle anhand der Objektposition	64
III. Die Gegenspieler	65
12. Einfache Computersteuerung	66
12.1. Vordefinierte Computerzüge	66
12.2. Zufällige Computerzüge	67
12.3. Verfolgungsjagd	69
12.4. Computereinsatz in Strategiespielen	70
13. Menschliche Gegner	74
13.1. Mehrere Spieler an einem Computer	74
13.2. Netzwerkspiel	74
IV. Anhang	78
A. ASCII-Zeichentabelle	79
B. MULTIKEY-Scancodes	80
C. Ereignisse von SCREENEVENT	81

D. Modi für SCREENRES und SCREEN	82
Index	83
Liste der Quelltexte	86

Teil I.

Das Labyrinth

1. Vorüberlegungen

Bevor Sie Ihr Projekt beginnen, sollten Sie sich einige Gedanken über Ihre Ziele machen. Welche Art von Spiel soll erstellt werden? Welche besonderen Anforderungen stellt es an den Programmierer? Welche Inhalte sollen umgesetzt werden?

1.1. Anforderungen des Programms

Je nach Spieltyp kommen auf den Programmierer verschiedene Aufgaben zu, die bewältigt werden müssen. Für ein komplexes Strategiespiel gegen den Computer werden Sie sich Gedanken über eine gute – oder zumindest funktionierende – KI machen müssen. Jump 'n' Runs und Shoot 'em ups leben vielfach von scrollbaren Leveln und ansprechender Grafik. Rollenspiele wiederum sollten eine gute Hintergrundgeschichte beinhalten und erfordern eine hohe Aufmerksamkeit gegenüber der Vermeidung von logischen Fehlern.

1.2. Realistische Selbsteinschätzung

Prüfen Sie vor Beginn eines größeren Projekts, inwieweit Sie in der Lage sind, Ihr Vorhaben umzusetzen. Die schönste Idee bringt nichts, wenn sie nach einigen Wochen oder Monaten frustriert fallen gelassen wird, weil sie sich als nicht durchführbar erweist. Im schlimmsten Fall haben Sie neben einer Menge Zeit auch die Lust verloren, weiter zu programmieren. Im günstigsten Fall können Sie Ihre Erwartungen noch herunterschrauben und eine abgespeckte Version Ihres ursprünglichen Plans angehen. Grundsätzlich gilt: Beginnen Sie (gerade in der Anfangsphase der Programmierlaufbahn) mit einfachen, überschaubaren Projekten. Ein kleines, aber fertiges Programm wird Ihnen mehr Freude bringen als ein riesiges Projekt, das unvollendet aufgegeben wurde.

1.3. Erweiterbarkeit

Besser ist es, ein Projekt klein zu beginnen und gleichzeitig auf eine mögliche Erweiterbarkeit zu achten. Starten Sie zunächst mit einem überschaubaren, für sich funktionierenden Teil des Programms und erweitern Sie es dann nach und nach um zusätzliche Elemente.

Das ist einfacher gesagt als getan – Sie müssen dazu nämlich schon frühzeitig bedenken, wie Sie Ihr Projekt erweiterbar halten. Ob es sich um eine spätere Änderung der Levelgröße oder das Hinzufügen von Animationen handelt; mit je weniger Aufwand Sie Erweiterungen einbauen können, desto besser.

1.4. Wartbarkeit des Codes

Größere Projekte wachsen über einen längeren Zeitraum hinweg. Während Sie zu Beginn der Arbeit Ihren Code wahrscheinlich sofort überblicken und verstehen werden, wird es mit zunehmender Programmierdauer und Codelänge immer schwerer. Denken Sie auch daran, dass Sie möglicherweise in einem Jahr oder später den Code wieder auskramen wollen, um ihn zu verbessern oder zu erweitern. Programmieren Sie daher so, dass der Code auch nach längerer Zeit verständlich bleibt. Dazu gehört eine aussagekräftige Bezeichnung der Variablen und Prozeduren genauso wie eine gute Programmstruktur, Einrückungen usw. Kommentieren Sie außerdem Ihre Arbeit ausreichend.

2. Das Spielfeld

Im Sinne dieser Vorüberlegungen beginnen wir mit einem sehr einfachen Spiel, dessen einziges Ziel es ist, ein Männchen durch ein Labyrinth zum Ausgang zu steuern. Dabei werden wir uns im Laufe der nächsten Kapitel um folgende Dinge Gedanken machen:

- Wie „merkt“ sich der Computer den Aufbau des Spielfeldes?
- Wie organisiere ich mehrere Spiellevel?
- Wie steuere ich die Spielfigur? Wie prüfe ich auf Kollision?
- Wie überprüfe ich, dass ein Level gelöst oder das Spiel beendet wurde?

Die „Grafik“ beschränkt sich vorerst auf einzelne ASCII-Zeichen in der Konsole. Der Einbau von „echter“ Grafik wird dann im [Kapitel 4](#) behandelt. Das Spielfeld soll zunächst einmal folgendermaßen aussehen:

```
#####  
#S  #  #      #      #  
###  # ##### #  
# ### #      #      #  
# #      # # ## # #  
# #      # # # #  
# # ##### # # #####  
#      # # #      #  
##### #####  
#      #      A  
#####
```

Der Spieler startet an der mit S gekennzeichneten Stelle links oben; er soll seine Figur zum Ausgang A bewegen. Nun müssen wir zuerst einmal dafür sorgen, dass auch der Computer den Aufbau des Levels kennt.

2.1. Interne Speicherung der Leveldaten

Das erste zu klärende Problem ist, wie man dem Programm die Leveldaten mitteilen kann. Entweder müssen die Daten bereits im Programm vorhanden sein oder sie müssen

2. Das Spielfeld

als externe Datei vorliegen und eingelesen werden. Doch zunächst wollen wir uns um die interne Speicherung der Leveldaten kümmern.

Da das Spielfeld in 11 Zeilen aus je 21 Zeichen aufgeteilt ist, ist es sinnvoll, die Daten ebenfalls in ein Array mit 21x11 Einträgen zu speichern. Was ist unter einem solchen zweidimensionalen Array zu verstehen?

Ähnlich wie bei einem Schachbrett ist das Feld in mehrere Reihen und Spalten aufgeteilt. Um bei einer Schachpartie eindeutig angeben zu können, mit welcher Figur gezogen wurde, werden die senkrechten Linien mit den Buchstaben a-h und die waagrechten Reihen mit den Zahlen 1-8 bezeichnet. Der einzige Unterschied bei einem Array ist, dass hier keine Buchstaben verwendet werden, sondern, wie bei einem Koordinatensystem, in jede Richtung mit Zahlen gearbeitet wird. Um z. B. ein Feld mit 6 Spalten und 4 Reihen zu speichern, kann man folgendermaßen vorgehen:

DIM felddata(5, 3) **AS** datentyp

Beachten Sie, dass Array-Indizes standardmäßig von 0 ab gezählt werden. In unserem Beispiel gibt es keinen triftigen Grund, davon abzuweichen. Damit werden die Spalten von 0 bis 5 gezählt, und das Feld in der 2. Spalte der 3. Reihe trägt die Nummer (2,3) usw.

	Spalte 0	Spalte 1	Spalte 2	Spalte 3	Spalte 4	Spalte 5
Reihe 0	(0,0)	(1,0)	(2,0)	(3,0)	(4,0)	(5,0)
Reihe 1	(0,1)	(1,1)	(2,1)	(3,1)	(4,1)	(5,1)
Reihe 2	(0,2)	(1,2)	(2,2)	(3,2)	(4,2)	(5,2)
Reihe 3	(0,3)	(1,3)	(2,3)	(3,3)	(4,3)	(5,3)

Die Daten unseres Spielfeldes werden ebenso gespeichert, nur dass es nun 21 Spalten und 11 Reihen sind.

DIM AS STRING*1 felddata(20, 10) ' Spielfeld mit 21x11 Feldern

Jeder einzelne Eintrag im Array *felddata* gibt an, was für eine Art von Feld an der entsprechenden Stelle vorliegt. Soll überprüft werden, ob das Feld 13 in der Zeile 3 begehbar ist, dann wird getestet, ob sich an dieser Stelle ein Leerzeichen befindet.

IF felddata(13, 3) = " " **THEN** ' Feld ist begehbar

Wäre an der gewünschten Stelle eine Wand, dann müsste der String statt eines Leerzeichens das Rautenzeichen # beinhalten.

Achtung:

So lange sich Ihr Programm im Entwicklungsstadium befindet, ist es hilfreich, mit der Compileroption `-exx` zu compilieren. Gerade im Umgang mit Arrays entdecken Sie dadurch einen Zugriff auf ungültige Bereiche viel schneller. Die Version, die Sie dann veröffentlichen, sollten Sie jedoch ohne die Option `-exx` compilieren, weil die zusätzliche Überprüfung eine Menge zusätzlichen Code in Ihr Programm einfügen muss!

2.2. Leveldaten im Quelltext

Die Leveldaten müssen dem Programm in irgendeiner Weise zugänglich gemacht werden. Eine Möglichkeit dazu ist die direkte Eingabe in das Array im Quelltext. Dies kann bereits beim Anlegen des Arrays geschehen.

Etwas flexibler ist der Einsatz von **DATA**-Zeilen. Mit **RESTORE** können dann die Zeilen angesteuert werden, die für das Level verwendet werden sollen. **DATA**-Zeilen sind fest im Programmcode integriert, was – gerade für Levelinformationen – auch Nachteile mit sich bringt. Dennoch wird die Methode hier der Vollständigkeit halber vorgestellt.

Quelltext 2.1 liest die Daten zeilenweise aus den **DATA**-Zeilen und zerlegt sie anschließend in die einzelnen Zeichen. Die mit 'S' gekennzeichnete Stelle ist eigentlich ein leeres Feld, welches die zusätzliche Information der Spieler-Startposition enthält. Entdeckt das Programm diese Stelle, dann müssen entsprechende Modifikationen vorgenommen werden: Das Programm setzt die Spielfigur an die gefundene Stelle (hierzu werden die Variablen *sx* und *sy* verwendet) und ändert das Feld anschließend in ein leeres Feld. Der Ausgang wird ebenfalls gespeichert (in den Variablen *ax* und *ay*), um später eine einfachere Abfrage durchführen zu können, ob der Spieler das Ziel erreicht hat. Allerdings kann es auf diese Weise im Level auch nur einen möglichen Ausgang geben.

Achtung:

Die folgenden Beispiele verwenden für die Leveldaten unterschiedliche Speicherformate, die auch unterschiedlich ausgewertet werden müssen! Verinnerlichen Sie sich die Bedeutung der einzelnen Programmschnipsel, bevor Sie sie verwenden. Wenn Sie Code aus verschiedenen Schnipseln kombinieren, ohne verstanden zu haben, was die einzelnen Teile tun, werden Sie mit hoher Wahrscheinlichkeit kein funktionierendes Programm erhalten!

Quelltext 2.1: Leveldaten über DATA-Zeilen einlesen

```

5  DIM AS STRING*1 felddata(20, 10) ' Daten des Spielfeldes
    DIM AS STRING text              ' eingelesene DATA-Zeile
    DIM AS INTEGER sx, sy            ' Position der Spielfigur
    DIM AS INTEGER ax, ay            ' Position des Ausgangs
    RESTORE level1

    FOR zeile AS INTEGER = 0 TO 10
        READ text
        FOR spalte AS INTEGER = 0 TO 20
10         ' einzelnes Zeichen extrahieren
            felddata(spalte, zeile) = MID(text, spalte+1, 1)
            IF felddata(spalte, zeile) = "S" THEN
                ' Startposition entdeckt; Feld wird angepasst
                felddata(spalte, zeile) = " "
15                 sx = spalte
                 sy = zeile
            END IF
            IF felddata(spalte, zeile) = "A" THEN
                ' Ausgang entdeckt; Feld wird angepasst
20                 felddata(spalte, zeile) = " "
                 ax = spalte
                 ay = zeile
            END IF
        NEXT
    NEXT
25

    level1:
    DATA "##### "
    DATA "#S  #  #  #  #"
30    DATA "###  # ##### #"
    DATA "# ### #  #  # #"
    DATA "# #  #  # # # #"
    DATA "# #  #  # # # #"
    DATA "# # ##### # # ##### "
35    DATA "#  #  #  #  #"
    DATA "##### ##### #"
    DATA "#  #  A"
    DATA "##### "

40    level2:
    ' DATA-Zeilen des zweiten Levels
    ' ...

```

Nachteil dieser Methode ist, dass Sie das Level nicht mehr verändern können, ohne das Programm neu zu compilieren. Außerdem ist auch die exakte Länge und Breite des Spielfeldes bereits im Quellcode fest verankert. Hier ließe sich das Programm noch flexibler gestalten. **MID** ist übrigens eine ziemlich langsame Methode, um ein einzelnes Zeichen aus einem String zu extrahieren. Schneller geht es mit direktem Zugriff mittels String-Indizierung, die im nächsten Beispiel angewandt wird.

2.3. Leveldaten in einer externen Datei

Wollen Sie die Level später verändern, ohne dabei in den Quellcode einzugreifen, müssen Sie die Daten in einer externen Datei speichern. Die erste vorgestellte Methode läuft ähnlich wie beim Einlesen der **DATA**-Zeilen.

2.3.1. ASCII-Speicherung

Der Text, der zuvor in den **DATA**-Zeilen stand, wird nun in einer eigenen Datei namens *level1.dat* gespeichert. (Die Datei kann natürlich auch anders heißen, jedoch wird im folgenden Quelltext auf *level1.dat* Bezug genommen.) Ansonsten funktioniert das Prinzip ähnlich wie im [Quelltext 2.1](#): Die Daten werden (nun mittels **LINE INPUT**) zeilenweise aus der Datei gelesen und dann zerlegt.

Der Zugriff auf die einzelnen Zeichen des Strings wird diesmal über die wesentlich effektivere String-Indizierung vorgenommen. Die Rückgabe ist jedoch kein String der Länge 1, sondern der ASCII-Code des gewünschten Zeichens. Deshalb müssen im Programm ein paar kleine Änderungen vorgenommen werden. Beachten Sie auch, dass das erste Zeichen eines Strings nicht mit dem Wert 1, sondern mit 0 indiziert ist.

```
wert = stringvariable[index-1]
```

liefert damit dieselbe Rückgabe wie

```
wert = ASC(MID(stringvariable, index, 1))
```

Das Programm speichert nun diesen ASCII-Wert als **INTEGER** statt, wie zuvor, den String. Das bringt gewisse Vorteile mit sich: zum einen benötigen die Einzelstrings mehr Speicherplatz, zum anderen laufen Berechnungsroutinen mit Strings verhältnismäßig langsam. Nachteil ist, dass die Bedeutung der Zahlen nicht mehr so offensichtlich ist. Statt eines Leerzeichens steht nun die eher nichtssagende Zahl 32, statt der Raute # die Zahl 35. Hier ist eine gute Kommentierung angeraten; [Kapitel 2.3.3](#) stellt noch eine verbesserte Möglichkeit vor, den Inhalt der Variablen ersichtlich zu machen.

Sofern keine wide-chars (**WSTRING**) verwendet werden, reicht zur Speicherung der ASCII-Zeichen auch der Datentyp **UBYTE** aus. Da der Compiler aber 32bit-Anwendungen erstellt, laufen Berechnungen im 32-Bit-Datentyp **INTEGER** wesentlich schneller. Daher sollten Sie andere Ganzzahl-Typen nur dann einsetzen, wenn die entstehende Speicherersparnis signifikant ist. [Quelltext 2.3](#) wird eine Speicherung in **UBYTE**s demonstrieren; auch dort wäre eine **INTEGER**-Speicherung denkbar.

Doch nun zum Programm:

Quelltext 2.2: Leveldaten über eine ASCII-Datei einlesen

```
5  DIM AS INTEGER felddata(20, 10), f = FREEFILE
    DIM AS STRING text
    DIM AS INTEGER sx, sy      ' Position der Spielfigur
    DIM AS INTEGER ax, ay     ' Position des Ausgangs
10  OPEN "level1.dat" FOR INPUT AS #f
    FOR zeile AS INTEGER = 0 TO 10
        LINE INPUT #f, text
        FOR spalte AS INTEGER = 0 TO 20
            ' Inhalt des aktuellen Feldes auslesen
            felddata(spalte, zeile) = text[spalte]
            IF felddata(spalte, zeile) = ASC("S") THEN ' Startposition
                felddata(spalte, zeile) = 32          ' Leerfeld setzen
                sx = spalte
                sy = zeile
            END IF
            ' usw.
        NEXT
    NEXT
20  CLOSE #f
```

2.3.2. binäre Speicherung

Möglich ist auch eine Speicherung der Level in binärem Format. Die Level werden dadurch etwas kompakter und es ist leichter, direkt auf bestimmte Einträge zuzugreifen. Zum Erstellen und Verändern der Level sollten Sie sich dann aber einen Leveleditor anlegen, da Sie (abgesehen von einem Hex-Editor) die Daten nicht direkt betrachten können. Auch eine spätere Änderung des Speicherformats ist schwerer umzusetzen. Machen Sie sich also zunächst intensiv Gedanken über den Aufbau Ihrer Daten und planen Sie bereits für eventuelle spätere Erweiterungen voraus.

Wie schon erwähnt, arbeitet das folgende Beispiel mit dem Datentyp **UBYTE**. Damit stehen Speicherplätze für 256 verschiedene Objekte zur Verfügung. Wenn Sie damit rechnen, dass Ihr Programm irgendwann einmal mehr Objekte verwalten muss, dann sollten Sie schon jetzt einen größeren Datentyp wie **(U)SHORT** oder **(U)INTEGER** verwenden. Die Leveldateien werden dann zwar doppelt bzw. viermal so groß, jedoch sparen Sie sich später möglicherweise eine aufwändige Anpassung der alten Level an ein neues Format.

Der Zugriff auf binäre Dateien läuft über den Dateimodus **BINARY**.

Quelltext 2.3: Leveldaten über eine Binär-Datei einlesen

```

DIM AS INTEGER f = FREEFILE
DIM AS UBYTE felddata(20, 10) ' Speicherung jetzt in UBYTES

OPEN "level1.dat" FOR BINARY AS #f
5 FOR zeile AS INTEGER = 0 TO 10
  FOR spalte AS INTEGER = 0 TO 20
    ' Inhalt des aktuellen Feldes auslesen
    GET #1,, felddata(spalte, zeile)
    ' weitere Auswertungen ...
10  NEXT
  NEXT
CLOSE #f
```

Der Nachteil, dass sich die Daten nicht ganz so leicht auslesen lassen wie in der ASCII-Speicherung, kann zugleich ein Vorteil sein. Wenn der Spieler daran gehindert werden soll, sich bereits im Vorfeld anhand der Leveldatei über den Spielverlauf zu informieren, können Sie die Daten auch noch verschlüsseln, etwa indem Sie sie in ungewöhnlicher Reihenfolge speichern oder nach einem bestimmten Muster Werte addieren oder subtrahieren. Einen Benutzer, der Ihre Level unbedingt entschlüsseln oder modifizieren will, wird das vermutlich nicht aufhalten; der Zugriff wird jedoch deutlich erschwert.

Hierzu noch ein kleiner Hinweis: Der Begriff „binäre Speicherung“ hat nichts damit zu tun, ob Sie die Zahlenwerte im Binär-, Dezimal- oder sonstigen System darstellen. Ob Sie z. B. die Dezimalzahl 19 lieber als Binärzahl `&b10011` oder als Hexadezimalzahl `&h13` schreiben wollen, macht technisch keinen Unterschied. „Binäre Speicherung“ bedeutet vielmehr, dass die Daten nicht primär als Text (ASCII-Zeichen) interpretiert werden.

2.3.3. Sprechende Werte

Um die Zahlenwerte, die bei der ASCII- bzw. Binärspeicherung auftreten, im Programmcode sofort verstehen zu können, sollte man sie mit sprechenden Namen ausstatten. Statt

```
IF felddata(13, 3) = 32 THEN ' Feld ist begehbar
```

lässt sich z. B. schreiben:

```
#DEFINE FeldLeer 32
...
IF felddata(13, 3) = FeldLeer THEN ' Feld ist begehbar
```

`#DEFINE` ist ein Metabefehl, der bereits beim Compilervorgang umgesetzt wird. Nach dem Befehl ersetzt der Compiler alle auftretenden *FeldLeer* durch die Zahl 32. Die Ersetzung findet, wie gesagt, bereits beim Compilervorgang statt und hat daher später keine negativen Auswirkungen auf die Geschwindigkeit des Programms.

Eine andere Möglichkeit ist die Listenerstellung mit **ENUM**. Diese Anweisung bietet sich vor allem dann an, wenn Sie Ihre Werte einfach der Reihe nach durchnummerieren wollen. Sie können der Liste einen Namen geben und diesen als Variablentypen bei **DIM**-Anweisungen verwenden.

```
ENUM Felderliste
    FeldLeer
    FeldWand
    ' ...
END ENUM
DIM felddata(20, 10) AS Felderliste

' ...

IF felddata(13, 3) = FeldLeer THEN ' Feld ist begehbar
```

Natürlich müssen Sie darauf achten, dass die Nummerierung in der Leveldatei mit der Nummerierung im Programm übereinstimmt.

2.3.4. Vergleich: ASCII vs. binär

Hier noch einmal die Vorteile der verschiedenen Speicherformate zusammengefasst:

1. Vorteile der ASCII-Speicherung:
 - anschauliche Darstellung der Daten
 - direkte Bearbeitung in einem Texteditor möglich
 - einfachere Veränderbarkeit des Levelformats
2. Vorteile der Binär-Speicherung:
 - kompaktes Speicherformat
 - leichter Zugriff auf Teildaten
 - Möglichkeit der Verschlüsselung

3. Steuerung

3.1. Tastatursteuerung

3.1.1. Tastaturabfrage mit INKEY

Eine gängige Methode zur Tastatursteuerung ist die regelmäßige Abfrage und Auswertung eines Zeichens aus dem Tastaturpuffer. Dazu eignet sich der Befehl **INKEY**, da hier nicht auf einen Tastendruck gewartet wird und damit das Spiel auch ohne Benutzereingaben nicht unterbrochen wird.

Für die Pfeiltasten gelten besondere Tastaturcodes:

Pfeiltaste oben:	CHR(255, 72);	Pfeiltaste unten:	CHR(255, 80)
Pfeiltaste links:	CHR(255, 75);	Pfeiltaste rechts:	CHR(255, 77)

Natürlich muss auch überprüft werden, ob die Spielfigur in die gewünschte Richtung bewegt werden kann, also ob das Zielfeld leer ist. Wir verwenden im folgenden Beispiel eine Datenspeicherung im Format von [Quelltext 2.2](#) bzw. [Quelltext 2.3](#).

Quelltext 3.1: Steuerung über Tastatur (INKEY)

```

DIM AS STRING taste
DO
  taste = INKEY
  SELECT CASE taste
5    CASE CHR(255, 72)
      ' Spielfigur nach oben bewegen
      IF felddata(sx, sy-1) = 32 THEN
        LOCATE sy, sx
        PRINT " ";      ' alte Position leeren
10       sy -= 1         ' Spielerposition aendern
        LOCATE sy, sx
        PRINT "S";      ' neue Position schreiben
      END IF
    CASE CHR(255, 75)
15       ' Spielfigur nach links bewegen
      , ...
  END SELECT
  SLEEP 1 ' Pause, um die Kontrolle an andere Prozesse zu uebergeben
LOOP UNTIL taste = CHR(27)
```

Der unter **CASE CHR(255, 72)** abgedruckte Teil müsste nun insgesamt viermal in sehr ähnlicher Form getippt werden – für jede Richtung einmal. Codeteile, die an mehreren verschiedenen Stellen in gleicher oder sehr ähnlicher Form auftauchen, sollten Sie in eine Prozedur auslagern – zum einen, weil Sie sich dadurch Tipparbeit sparen, zum anderen, weil dadurch eine spätere Änderung erleichtert wird. Im Augenblick müssten Sie die Änderung an vier verschiedenen Stellen durchführen, wodurch natürlich auch die Fehleranfälligkeit steigt. Ausgelagert in eine **FUNCTION** könnte der Code folgendermaßen aussehen:

Quelltext 3.2: Verbesserte Steuerung über Tastatur

```
DECLARE FUNCTION bewege(BYREF sx AS INTEGER, BYREF sy AS INTEGER, _
                        dx AS INTEGER, dy AS INTEGER) AS INTEGER
DIM AS STRING taste
DO
5   taste = INKEY
   SELECT CASE taste
     CASE CHR(255, 72)
       bewege(sx, sy, 0, -1) ' nach oben
     CASE CHR(255, 75)
10    bewege(sx, sy, -1, 0) ' nach links
     CASE CHR(255, 77)
       bewege(sx, sy, 1, 0) ' nach rechts
     CASE CHR(255, 80)
       bewege(sx, sy, 0, 1) ' nach unten
15  END SELECT
   SLEEP 1 ' Pause, um die Kontrolle an andere Prozesse zu uebergeben
LOOP UNTIL taste = CHR(27)

20  FUNCTION bewege(BYREF sx AS INTEGER, BYREF sy AS INTEGER, _
                  dx AS INTEGER, dy AS INTEGER) AS INTEGER
    ' sx, sy: Spielerposition
    ' dx, dy: Bewegung in x- bzw. y-Richtung
    ' Rueckgabe: -1 bei erfolgreicher Bewegung; sonst 0
    IF felddata(sx+dx, sy+dy) = 32 THEN
25      ' das Zielfeld ist leer; die Bewegung kann stattfinden
      LOCATE sy, sx ' alte Position leeren
      PRINT " ";
      sx += dx ' Spielerposition aendern
      sy += dy
30      LOCATE sy+1, sx+1 ' neue Position schreiben
      PRINT "S";
      RETURN -1 ' Bewegung erfolgreich
    END IF
    RETURN 0 ' keine Bewegung erfolgt
35  END FUNCTION
```

Wie Sie sehen (jedenfalls dann, wenn Sie in [Quelltext 3.1](#) alle vier Richtungsabfragen ausschreiben), wird der Quellcode ein gutes Stück kompakter. Der Rückgabewert wird zur Zeit noch nicht ausgewertet, aber es schadet nicht, sich hier eine Option offen zu

halten. Sollten Sie sich letzten Endes gegen eine Auswertung der Rückgabe entschließen, können Sie immer noch **SUB** statt **FUNCTION** verwenden.

3.1.2. Tastendruck über MULTIKEY

Für unser einfaches Beispiel mag die **INKEY**-Methode ausreichen. Sie ist jedoch von der eingestellten Tasten-Wiederholungsrate abhängig – wenn Sie eine Pfeiltaste gedrückt halten, wird die Spielerfigur vermutlich nach dem ersten Schritt eine kleine Pause einlegen, bevor sie dann im schnelleren Tempo weiterleitet. Der Grund dafür ist, dass bei einer gehaltenen Taste der Tastaturpuffer nur Schritt für Schritt gefüllt wird.

MULTIKEY arbeitet anders: Diese Funktion fragt nicht den Tastaturpuffer ab, sondern den Tastenstatus, also ob die gewünschte Taste gedrückt ist oder nicht. Sie werden mit **MULTIKEY** in aller Regel eine flüssigere Bedienung erzeugen können als mit **INKEY**. Allerdings müssen Sie nun zwischen den einzelnen Schritten selbst eine kleine Pause einbauen, damit die Spielerfigur nicht schon bei einem kurzen Tastendruck mehrere Felder zurücklegt.

Quelltext 3.3: Steuerung über Tastatur (MULTIKEY)

```
DO
  IF MULTIKEY(72) THEN
    ' Spielfigur nach oben bewegen
    ' ...
5  ELSEIF MULTIKEY(80) THEN
    ' Spielfigur nach unten bewegen
    ' ...
    ' ...
  END IF
10 SLEEP 200, 1
  LOOP UNTIL MULTIKEY(1) ' ESC-Taste
```

Bei den benötigten Tastencodes handelt es sich um die sogenannten *Scancodes*, die in [Anhang B](#) aufgelistet sind. Für die Pfeiltasten sind die Tastencodes dieselben wie bei **INKEY**, jedoch ohne das dort vorangestellte **CHR(255)**.

3.2. Joysticksteuerung

Mit **GETJOYSTICK** stellt FreeBASIC eine Funktion zur Abfrage von Joysticks und Gamepads zur Verfügung.

```
rueckgabe = GETJOYSTICK(id, buttons, x, y, z, r, u, v)
```

Am Rückgabewert lässt sich überprüfen, ob unter *id* ein Gerät angesprochen werden konnte. Ist der Wert 0, dann war die Abfrage erfolgreich. *buttons* enthält dann den Status

3. Steuerung

der Buttons und die anderen Variablen den Ausschlag diverser Achsen im Bereich von -1.0 bis +1.0

Im folgenden Beispiel verwenden wir die x - und y -Achse zur Steuerung. Mit der im [Quelltext 3.2](#) verwendeten Bewegungs-Funktion geht das sehr einfach.

```
5 DIM buttons AS INTEGER, x AS SINGLE, y AS SINGLE
DO
  IF GETJOYSTICK(0, buttons, x, y) = 0 THEN bewege INT(x), INT(y)
  SLEEP 1
LOOP UNTIL INKEY = CHR(27)
```

Solange der Joystick bei vollem Ausschlag den Wert 1 bzw. -1 zurückliefert, funktioniert das sehr gut. Wenn Sie sich jedoch nicht auf die exakten Werte 1 und -1 für einen vollen Ausschlag bzw. 0 für die Ruheposition verlassen wollen, können Sie der Funktion einen gewissen Spielraum einräumen, in dem der Achsenausschlag als Bewegung oder als Stillstand interpretiert wird.

Quelltext 3.4: Steuerung mit Joystick

```
5 DIM buttons AS INTEGER, x AS SINGLE, y AS SINGLE
  DECLARE FUNCTION bewege(BYREF sx AS INTEGER, BYREF sy AS INTEGER, _
    BYVAL dx AS SINGLE, BYVAL dy AS SINGLE) AS INTEGER
DO
  IF GETJOYSTICK(0, buttons, x, y) = 0 THEN
    IF bewege(sx, sy, x, y) THEN SLEEP 200, 1 ' kleine Bewegungspause
  END IF
  IF sx = ax AND sy = ay THEN END ' Ausgang erreicht
  SLEEP 1 ' Pause fuer den Prozessor
10 LOOP UNTIL INKEY = CHR(27)

FUNCTION bewege(BYREF sx AS INTEGER, BYREF sy AS INTEGER, _
  BYVAL dx AS SINGLE, BYVAL dy AS SINGLE) AS INTEGER
  ' sx, sy: Spielerposition
  ' dx, dy: Bewegung in x- bzw. y-Richtung
  ' Rueckgabe: -1 bei erfolgreicher Bewegung; sonst 0
  IF ABS(dx) < .7 THEN dx = 0 ' Bewegungsspielraum festlegen
  IF ABS(dy) < .7 THEN dy = 0
  IF felddata(sx+dx, sy+dy) = 32 THEN
20   LOCATE sy+1, sx+1 ' alte Position leeren
   PRINT " ";
   sx += SGN(dx) ' Spielerposition aendern
   sy += SGN(dy)
   LOCATE sy+1, sx+1 ' neue Position schreiben
25   PRINT "S";
   RETURN -1 ' Bewegung erfolgreich
  END IF
  RETURN 0 ' keine Bewegung erfolgt
END FUNCTION
```

Hinweise: dx und dy werden nun als SINGLE-Werte übergeben, damit die Funktion entscheiden kann, ob der Ausschlag stark genug für eine Bewegung ist. Das Schlüsselwort **BYVAL** wird verwendet, da die Werte der Variablen dx und dy innerhalb der Funktion verändert werden und sich dies nicht auf das Hauptprogramm auswirken soll. Die Verwendung der Vorzeichenfunktion **SGN** ist ein kleiner Trick, um aus den **DOUBLE**-Werten wieder die Richtungswerte 0, -1 bzw. 1 zu erhalten.

3.3. Maussteuerung

Für das hier vorgestellte Spiel ist eine Maussteuerung eher schwierig umzusetzen. Man könnte zwar die Mausklicks im Spielfeld auswerten und die Figur an die angeklickte Stelle bewegen, aber dazu müsste man entweder eine Routine zur Wegsuche implementieren (wobei eine Wegsuche ja eigentlich die Aufgabe des Spielers sein sollte), oder der Spieler kann immer nur ein Feld neben der aktuellen Position anklicken, womit die Mausbedienung recht unkomfortabel wird.

Komfortable Mausbedienung würde bedeuten, dass der Spieler seine Figur durch das Bewegen der Maus steuern kann. Dazu sollte die Positionierung der Figur pixelgenau erfolgen, womit eine ASCII-Grafik nicht mehr geeignet ist. Die folgenden Beispiele greifen daher auf die gfx-Grafikroutinen zurück (näheres dazu in [Kapitel 4](#)). Außerdem werden wir vorerst das Labyrinth beiseite lassen und uns einem anderen „Spiel“ zuwenden, bei dem einfach nur ein Kreis im Fenster bewegt werden muss.

```
SCREENRES 200, 200
DIM AS INTEGER mausX, mausY
SETMOUSE 100, 100, 0, 1 ' Maus im Fenster zentrieren und ausblenden
' ...
GETMOUSE mausX, mausY ' Mausposition abfragen
IF mausX > 100 THEN bewege_Figur_nach_rechts
IF mausX < 100 THEN bewege_Figur_nach_links
' usw.
```

Die Maus wurde durch den vierten Parameter von **SETMOUSE** im Fenster „eingesperrt“, trotzdem funktioniert die Steuerung nicht mehr, wenn sich die Maus am Fensterrand befindet. Daher wird die Maus nun nach der Abfrage immer wieder in die Ausgangsposition zurückgesetzt.

Achtung:

Beachten Sie, dass alle Parameter von **GETMOUSE** auf -1 gesetzt werden, wenn sich die Maus außerhalb des Grafikfensters befindet!

Quelltext 3.5: Steuerung mit Maus

```
5 SCREENRES 200, 200
  DIM AS INTEGER mausX, mausY, mausB, ballX = 100, ballY = 100
  SETMOUSE 100, 100, 0, 1      ' Maus im Fenster zentrieren und ausblenden
  CIRCLE (ballX, ballY), 5      ' Ball zeichnen
10 DO
  GETMOUSE mausX, mausY, , mausB      ' Maus abfragen ...
  SETMOUSE 100, 100                  ' ... und zuruecksetzen
  LINE (ballX-5,ballY-5)-STEP(10,10),0,BF ' Ball loeschen (uebermalen)
  ballX += mausX - 100                ' Ball um die Strecke bewegen, die
  ballY += mausY - 100                ' von der Maus zurueckgelegt wurde
  IF ballX < 5 THEN ballX = 5          ' evtl Ball ins Fenster zurueckholen
  IF ballX > 195 THEN ballX = 195
  IF ballY < 5 THEN ballY = 5
  IF ballY > 195 THEN ballY = 195
15 CIRCLE (ballX, ballY), 5          ' Ball zeichnen
  SLEEP 50
  LOOP UNTIL mausB <> 0
```

Um die alte Position des Kreises auf dem Bildschirm zu löschen, wird diese Stelle mit einem schwarzen Rechteck überzeichnet. Das funktioniert selbstverständlich nur, wenn der Hintergrund komplett schwarz ist. In [Kapitel 4.4](#) wird näher auf die Möglichkeiten eingegangen, den Hintergrund wiederherzustellen.

Alternativ zur Abfrage über **GETMOUSE** kann im Übrigen auch die Funktion **SCREENEVENT** mit dem Ereignis **EVENT_MOUSE_MOVE** verwendet werden. In [Anhang C](#) finden Sie dazu eine Zusammenstellung der abfragbaren Ereignisse. **SCREENEVENT** wird hier jedoch nicht ausführlicher behandelt.

3.4. Beschleunigung

Realistischer wird der Bewegungsablauf, wenn die Spielfigur nicht direkt gesteuert, sondern in die gewünschte Richtung beschleunigt wird und bei fehlender Eingabe allmählich wieder abbremst. Obwohl eine Beschleunigung der Spielfigur prinzipiell mit jedem Eingabegerät umgesetzt werden kann, wird im Folgenden nur die Mauseingabe behandelt. Wie in [Kapitel 3.3](#) wird die Mausbewegung durch die Differenz der neuen und alten Position berechnet, nun aber jeweils zu einem Geschwindigkeitsvektor addiert. Dieser wird nun wiederum zum x - bzw. y -Wert des Balles hinzugezählt. Kollidiert der Ball mit einer Fenstergrenze, dann dreht sich der zugehörige Geschwindigkeitsvektor um, d. h. der Ball wird in die entgegengesetzte Richtung zurückgeworfen. Der verwendete Faktor dient dazu, feinere Bewegungsabläufe zu ermöglichen.

Quelltext 3.6: Bewegung mit Beschleunigung

```
SCREENRES 200, 200
DIM AS INTEGER mausX, mausY, mausB, faktor = 100
DIM AS INTEGER ballX = 100*faktor, ballY = 100*faktor, vx = 0, vy = 0
SEIMOUSE 100, 100, 0, 1 ' Maus im Fenster zentrieren und ausblenden
5 CIRCLE (ballX/faktor, ballY/faktor), 5 ' Ball zeichnen

DO
    GETMOUSE mausX, mausY, , mausB ' Maus abfragen ...
    SETMOUSE 100, 100 ' ... und zuruecksetzen
    vx += mausX - 100 ' Geschwindigkeiten anpassen
    vy += mausY - 100
    vx -= SGN(vx) ' allgemeine Abbremsung
    vy -= SGN(vy)
    IF ABS(vx) > 5*faktor THEN vx = 5*faktor*SGN(vx) ' Hoechstgeschwindigkeit
    IF ABS(vy) > 5*faktor THEN vy = 5*faktor*SGN(vy)

    LINE (ballX/faktor-5, ballY/faktor-5)-STEP(10, 10), 0, BF
    ballX += vx ' neue Position berechnen
    ballY += vy
    IF ballX < 5*faktor THEN vx = ABS(vX) ' Ball am Fensterrand wird
    IF ballX > 195*faktor THEN vx = -ABS(vX) ' zurueckgeworfen
    IF ballY < 5*faktor THEN vy = ABS(vy)
    IF ballY > 195*faktor THEN vy = -ABS(vy)
    CIRCLE (ballX/faktor, ballY/faktor), 5 ' Ball zeichnen
    SLEEP 10
25 LOOP UNTIL mausB <> 0
```

Wenn Sie wollen, können Sie in das Programm noch eine allgemeine Schwerkraft einbauen, welche den Ball in Richtung Boden beschleunigt. Das lässt sich mit einer einzigen zusätzlichen Programmzeile bewerkstelligen.

4. Grafik

4.1. Initialisierung des Grafikfensters

Einer der großen Vorteile von FreeBASIC ist die gfx-Bibliothek, die es erlaubt, schnell und problemlos einfache Grafikbefehle zu verwenden. Die Entwickler der Bibliothek haben sich dabei das Ziel gesetzt, mit so wenig Abhängigkeiten auszukommen wie möglich, sodass Ihr kompiliertes Programm auch auf anderen Computern ohne zusätzliche Bibliotheken lauffähig ist. Genauer gesagt werden nur Bibliotheken benötigt, von denen man annehmen kann, dass sie auf jedem Computer verfügbar sind - unter Windows sind dies **user32.dll**, **ddraw.dll** und **dinput.dll**, unter Linux **libX11**, **libXext**, **libXxf86vm** und **libpthread**.

Um die Grafikausgabe nutzen zu können, muss zuerst ein Grafikfenster initialisiert werden. Dazu diente früher der Befehl **SCREEN**. Dieser ist jedoch recht eingeschränkt, da er nur Fenster in einigen vordefinierten Größen erzeugen kann. Flexibler ist der Befehl **SCREENRES**, auf den in diesem Abschnitt genauer eingegangen wird.

SCREENRES erfordert als Parameter mindestens die Breite und die Höhe des gewünschten Grafikfensters. Zusätzlich können die Farbtiefe, die Seitenzahl, die Bildwiederholrate und weitere Fensterflags angegeben werden.

SCREENRES Breite , Hoehe , Farbtiefe , Seitenzahl , Flags , Bildwiederholrate
--

Achtung:

Sie sollten auf jeden Fall vermeiden, ein Fenster zu öffnen, das größer ist als die aktuelle Bildschirmgröße. Die Bildschirmauflösung lässt sich mit **SCREENINFO** bestimmen.

Wenn Sie keine Farbtiefe angeben, wird ein Fenster mit der Farbtiefe 8 Bits per Pixel (8bpp, ergibt 256 indizierte Farben) geöffnet. Möglich sind auch z. B. die Angaben 16 und 32 für 16bpp (2^{16} Farben) bzw. 32bpp (2^{32} Farben). Die Farbtiefe kann dabei nicht die in der aktuellen Auflösung eingestellte Farbtiefe überschreiten. Im Bedarfsfall wird sie automatisch niedriger gesetzt.

Die Verwendung mehrerer Seiten ermöglicht es, *double buffering* umzusetzen. Damit können flimmerfreie Animationen erzeugt werden. Die Technik des *double buffering* wird

in [Kapitel 4.7](#) behandelt. Die möglichen Fensterflags können Sie in [Anhang D](#) nachlesen. Die Bildwiederholungsrate sollte im Allgemeinen nicht verändert werden.

4.2. Grundlegende Grafikroutinen

Ist das Grafikfenster initialisiert, können darauf die Grafikbefehle eingesetzt werden. Grundsätzliche Grafikbefehle sind das Zeichnen einzelner Punkte (**PSET** und **PRESET**), von Strecken und Rechtecken (**LINE**), Kreisen und Ellipsen (**CIRCLE**) sowie das Füllen eines Bereiches mit einer bestimmten Farbe (**PAINT**). Ebenfalls interessant ist der Befehl **DRAW**, der Steuerbefehle zum Zeichnen von Figuren bereitstellt. Mit diesen so genannten *drawing primitives* stellt FreeBASIC Funktionen zur Verfügung, mit denen man sehr schnell einfache Grafiken erzeugen kann. Für stark grafisch orientierte Spiele ist die reine Verwendung der *drawing primitives* aufwändig und auch recht langsam. Um auf die Schnelle eine Benutzeroberfläche zu erstellen, reichen sie jedoch aus.

Die *drawing primitives* werden hier nicht ausführlich behandelt, da der Umgang mit ihnen weitgehend selbsterklärend ist. [Quelltext 4.1](#) verwendet einige dieser Befehle zum Zeichnen eines Spielsteines.

4.3. Zeichnen in den Grafikpuffer

Soll eine bestimmte Grafik immer wieder gezeichnet werden, dann ist es wesentlich effektiver, sie in einem Grafikpuffer zu speichern und diesen komplett auf dem Bildschirm auszugeben. Mit **IMAGECREATE** stellt FreeBASIC einen eigenen Befehl zur Verfügung, um einen Grafikspeicher zu reservieren. Der Befehl funktioniert ähnlich wie **ALLOCATE**, nur dass zusätzlich gleich der Header für den Bildspeicher korrekt gesetzt wird. Sie können anschließend ein Bild vom Datenträger in diesen Speicher laden oder mit **GET** einen Bildschirmausschnitt hineinkopieren. Zudem können Sie auch direkt in den Puffer hineinzeichnen. Sämtliche *drawing primitives* unterstützen auch das Zeichnen in einen Puffer.

Achtung:

Denken Sie daran, dass ein mit **IMAGECREATE** reservierter Speicherbereich auch wieder mit **IMAGEDESTROY** freigegeben werden muss!

Das folgende Beispiel zeichnet eine Grafik direkt in den Puffer und gibt sie anschließend mehrmals auf dem Bildschirm aus. Bereits bei dieser noch recht einfachen Grafik ist die Ausgabe des Puffers mehr als zehnmals so schnell wie das Zeichnen auf den Bildschirm.

Der Hauptgrund dafür ist der zeitraubende Befehl **PAINT**. Doch auch ohne ihn ist die Ausgabe des Puffers immer noch etwa viermal so schnell wie die direkte Bildschirmausgabe. Solange die Grafikausgabe nicht zeitkritisch ist – also beispielsweise beim Zeichnen eines Spielfeldes – mag dies keine besonders große Rolle spielen. Beim Einsatz von Animationen o. ä. können dadurch aber erhebliche Performance-Unterschiede auftreten.

Quelltext 4.1: Arbeiten mit dem Grafikpuffer

	#DEFINE PI 3.141592653589793	' Kreiskonstante fuer die Ellipse
	SCREENRES 300, 200	' Grafikscreen, indizierte Farben
	DIM AS ANY PTR bild	
	DIM AS INTEGER farbe = 12	' Steinfarbe: Wert von 8 bis 15
5	' Bild in den Puffer schreiben	
	bild = IMAGECREATE (40, 40)	' Bildpuffer erstellen
	CIRCLE bild, (20, 25), 15, farbe, PI, 0, .6	' untere halbe Ellipse
10	LINE bild, (5, 20)– STEP (0, 5), farbe	' Verbindungsstrecken zwischen der
	LINE bild, (35, 20)– STEP (0, 5), farbe	' unteren und der oberen Ellipse
	CIRCLE bild, (20, 20), 15, farbe, , , .6	' obere Ellipse
	PAINT bild, (20, 30), farbe, farbe	' Flaechen ausfuellen
	PAINT bild, (20, 20), farbe–8, farbe	
15	' Puffer auf dem Bildschirm ausgeben	
	FOR i AS INTEGER = 1 TO 5	
	PUT (i*50–20, 80), bild	
	NEXT	
20	IMAGEDESTROY bild	' Bildpuffer freigeben
	GETKEY	' auf Tastendruck warten

4.4. Hintergrundgrafik sichern

Wenn ein Objekt über den Bildschirm bewegt werden soll, muss es immer wieder von der alten Position gelöscht werden. Eine sehr einfache Möglichkeit dafür ist das Übermalen mit der Hintergrundfarbe, wie es z. B. auch in [Quelltext 3.5](#) und [Quelltext 3.6](#) gemacht wurde: über die alte Position des Kreises wurde mit **LINE** ein mit der Hintergrundfarbe gefülltes Rechteck gezeichnet.

Diese Methode funktioniert leider nur, wenn der Hintergrund einfarbig ist. Probleme gibt es auch, wenn sich zwei bewegliche Objekte überlagern. Wird eines davon bewegt, dann würde es das andere teilweise überzeichnen.

Eine mögliche Lösung bietet das Aktionswort **XOR**. Aktionsworte können zusammen mit **PUT** oder **DRAW STRING** (dazu später mehr) eingesetzt werden, um das Zusammenspiel zwischen Bildinformation und Hintergrund zu regeln. Mit **XOR** werden die Pixel des Bildes und des Hintergrundes mittels „exclusive OR“ verknüpft. Bei indizierten Farbpaletten

können sich damit, gelinde gesagt, „interessante“ Effekte ergeben. Klarer Vorteil der **XOR**-Verknüpfung ist jedoch, dass sich zweimaliges Zeichnen an dieselbe Stelle gegenseitig aufhebt.

XOR ist das Standard-Aktionswort von **PUT**, aber es schadet nicht, es trotzdem der Deutlichkeit halber anzugeben.

Quelltext 4.2: PUT mit Aktionswort XOR

```

#DEFINE PI 3.141592653589793
SCREENRES 300, 200                                     ' Grafikscreen, indizierte Farben
DIM AS ANY PTR bild
DIM AS INTEGER farbe = 12

5  ' Bild in den Puffer schreiben
   bild = IMAGECREATE(40, 40)
   CIRCLE bild, (20, 25), 15, farbe, PI, 0, .6
   LINE bild, (5, 20)-STEP (0, 5), farbe
10  LINE bild, (35, 20)-STEP (0, 5), farbe
   CIRCLE bild, (20, 20), 15, farbe, , , .6
   PAINT bild, (20, 30), farbe, farbe
   PAINT bild, (20, 20), farbe-8, farbe

15  ' Hintergrund erstellen
   LINE (50, 50)-(250, 150), 2, BF                     ' gruenes Rechteck ...
   LINE (80, 80)-(220, 120), 3, BF                     ' ... und darin ein blaues

   DIM AS INTEGER mausX = 0, mausY = 0, mausB           ' Mausposition und Buttonstatus
20  SETMOUSE mausX, mausY, 0, 1                         ' Maus auf das Fenster beschraenken
   PUT (mausX, mausY), bild, XOR
   DO
       PUT (mausX, mausY), bild, XOR                   ' alte Position loeschen
       GETMOUSE mausX, mausY, , mausB                   ' neue Position ermitteln ...
25  IF mausX > 260 THEN mausX = 260                     ' ... und an den Grenzen anpassen
       IF mausY > 160 THEN mausY = 160
       PUT (mausX, mausY), bild, XOR                   ' neue Position zeichnen
       SLEEP 1
   LOOP UNTIL mausB > 0 OR INKEY = CHR(27)              ' bei Mausklick oder ESC beenden
30  IMAGEDESTROY bild                                   ' Bildpuffer freigeben

```

Hier wurde bewusst ein Beispiel gewählt, bei dem die **XOR**-Methode an ihre optischen Grenzen stößt. Es gibt durchaus Situationen, in denen sie vorteilhaft eingesetzt werden kann. Ein anderes Problem ist das gelegentlich auftretende Flackern. Dies kann leicht mit **SCREENLOCK** vermieden werden:

```

SCREENLOCK
PUT (mausX, mausY), bild, XOR                           ' alte Position loeschen
' ...
PUT (mausX, mausY), bild, XOR                           ' neue Position zeichnen
SCREENUNLOCK

```

Noch besser, gerade bei aufwändigeren Zeichenvorgängen, ist eine Überprüfung, ob die

Maus bewegt wurde. Das Zeichnen wird nur dann durchgeführt, wenn es notwendig ist.

Achtung:

Eine Sperrung mit **SCREENLOCK** sollte so kurz wie möglich sein und muss mit **SCREENUNLOCK** wieder aufgehoben werden. Insbesondere sollten Sie es vermeiden, im gesperrten Bildschirm ein weiteres **SCREENLOCK** zu verwenden oder Benutzereingaben wie **GETKEY** abzufragen. Dies wird mit großer Wahrscheinlichkeit zu einem Programmabsturz führen!

Wenn die Grafik ohne Farbverfälschung gezeichnet werden soll, können Sie den Hintergrund speichern und später wieder herstellen. Zum Zeichnen bietet sich das Aktionswort **PSET** an, wenn die Grafik den gesamten Bereich überdeckt, oder **ALPHA** bzw. **TRANS**, wenn der Hintergrund teilweise durchscheinen soll. Mit **ALPHA** können Sie Bilder mit Alphakanal einsetzen – FreeBASIC unterstützt auch Bitmaps mit Alphakanal – während **TRANS** eine Maskenfarbe verwendet.

Das folgende Beispiel arbeitet mit einer Farbtiefe von 32bit. Die Maskenfarbe ist Pink mit dem Hexadezimalwert &hFF00FF bzw. den RGB-Wert **RGB(255, 0, 255)**. Diese Maskenfarbe wird beim Erstellen des Grafikpuffers als Hintergrundfarbe eingesetzt. Außerdem wird nun geprüft, ob die Maus bewegt wurde und das Neuzeichnen nötig ist.

Bei der Verwendung von **GET** zum Speichern eines Bildschirmausschnitts müssen Sie darauf achten, nicht „über die Bildschirmgrenzen hinaus“ zu lesen, da eine solche Vorgehensweise schnell zu einem Programmabsturz wegen illegalem Speicherzugriff führt. Außerdem sollte der gespeicherte Ausschnitt niemals größer sein als der reservierte Grafikpuffer. Denken Sie daran, dass

GET (startX , startY)– STEP (breite , hoehe), puffer
--

eine insgesamte Breite und Höhe von **breite+1** bzw. **hoehe+1** benötigt! Der Startpunkt zählt in dieser Berechnung mit.

Quelltext 4.3: PUT mit Hintergrund-Speicherung

```

#DEFINE PI 3.141592653589793
SCREENRES 300, 200, 32          ' Grafikscreen mit 32bit Farbtiefe
DIM AS ANY PTR bild, hg
DIM AS INTEGER hell = RGB(255, 64, 64) ' heller Farbwert des Steins
5 DIM AS INTEGER dunkel = RGB(192, 0, 0) ' dunkler Farbwert des Steins

' Bild in den Puffer schreiben
bild = IMAGECREATE(40, 40)
hg = IMAGECREATE(40, 40)
10 CIRCLE bild, (20, 25), 15, dunkel, PI, 0, .6
LINE bild, (5, 20)-STEP (0, 5), dunkel
LINE bild, (35, 20)-STEP (0, 5), dunkel
CIRCLE bild, (20, 20), 15, dunkel, , , .6
PAINT bild, (20, 30), dunkel, dunkel
15 PAINT bild, (20, 20), hell, dunkel

' Hintergrund erstellen
LINE (50, 50)-(250, 150), RGB(0,255,0), BF ' gruenes Rechteck ...
LINE (80, 80)-(220, 120), RGB(0,0,255), BF ' ... und darin ein blaues

20 DIM AS INTEGER mausX = 0, mausY = 0, mausB ' Mausposition und Buttonstatus
DIM AS INTEGER altX = 0, altY = 0 ' zuletzt gemerkte Mausposition
SETMOUSE mausX, mausY, 0, 1 ' Maus auf das Fenster beschraenken
GET (altX, altY)-STEP(39, 39), hg ' Hintergrund speichern
25 PUT (mausX, mausY), bild, TRANS
DO
  GETMOUSE mausX, mausY, , mausB ' neue Position ermitteln ...
  IF mausX > 260 THEN mausX = 260 ' ... und an den Grenzen anpassen
  IF mausY > 160 THEN mausY = 160
  30 IF mausX <> altX OR mausY <> altY THEN ' Maus wurde bewegt
    SCREENLOCK
    PUT (altX, altY), hg, PSET ' alte Position wiederherstellen
    GET (mausX, mausY)-STEP(39, 39), hg ' Hintergrund speichern
    PUT (mausX, mausY), bild, TRANS ' neue Position zeichnen
  35 SCREENUNLOCK
    altX = mausX ' neue Position merken
    altY = mausY
  END IF
  SLEEP 1
40 LOOP UNTIL mausB > 0 OR INKEY = CHR(27) ' bei Mausklick oder ESC beenden
IMAGEDESTROY bild ' Bildpuffer freigeben
IMAGEDESTROY hg

```

4.5. Externe Grafiken einbinden

Mit **BSAVE** und **BLOAD** können Grafikpuffer gespeichert und geladen werden. Dabei verwendet FreeBASIC sein internes Grafikformat. Allerdings wird auch von Haus aus das Laden von BMP-Bildern (Windows Bitmap) unterstützt. Dazu muss der verwendete

Dateiname lediglich mit „.bmp“ enden. Wie bereits erwähnt, werden auch BMPs mit Alphakanal unterstützt; dies ist besonders dann interessant, wenn Sie mit Teiltransparenz arbeiten wollen. Bei der Verwendung von BMPs – genauer gesagt generell beim Laden von Grafikpuffern – müssen Sie darauf achten, dass die geladene Grafik dieselbe Farbtiefe besitzt wie der aktuell eingestellte Grafikmodus.

Solange Sie wissen, wie groß das einzubindende Bild ist, stellt das Laden kein großes Problem dar. Reservieren Sie dazu mit **IMAGECREATE** einen ausreichend großen Speicherplatz und laden Sie das Bild in diesen Speicher. Ist die Bildgröße nicht bekannt, dann kann sie aus der Datei ermittelt werden:

Quelltext 4.4: Bildgröße ermitteln

```

DIM bild AS ANY PTR
DIM AS INTEGER breit, hoch, dateinr
DIM AS STRING datei = "meinbild.bmp"
dateinr = FREEFILE ' freie Dateinummer ermitteln
5
' Bildgroesse (Breite und Hoehe) auslesen
OPEN datei FOR BINARY AS #dateinr
GET #dateinr, 19, breit
GET #dateinr, 23, hoch
10 CLOSE #dateinr

' Fenster und Bildpuffer erstellen
SCREENRES breit, hoch, 32
bild = IMAGECREATE(breit, hoch)
15
' Bild laden und ausgeben
BLOAD datei, bild
PUT (0,0), bild, PSET
IMAGEDESTROY bild
20 GETKEY
```

Selbstverständlich hätte man in diesem Beispiel das Bild auch direkt in den Bildschirm laden können, indem einfach die Zieladresse ausgelassen wird. Es sollte hier aber auch das Laden in einen Grafikpuffer demonstriert werden.

Um andere Grafiken außer BMP einzubinden, benötigen Sie eine externe Bibliothek. Zum Einbinden von JPEG- oder PNG-Grafiken stehen inzwischen einige Bibliotheken zur Verfügung. Eine davon ist die *FreeBASIC Extended Library*¹, kurz *fbext*. Sie bietet unter anderem die Möglichkeit, BMP-, PNG-, TGA- und JPG-Dateien zu laden, drehen, skalieren und bearbeiten. Dies ist jedoch nur ein sehr kleiner Teil der Funktionen, die *fbext* zu bieten hat.

¹ siehe [Seite 55](#), Fußnote 7

4.6. Textausgabe

Selbstverständlich kann auch im Grafikscreen eine Textausgabe mittels **PRINT** erfolgen. Die Position der Textausgabe lässt sich damit jedoch nur zeilen- bzw. spaltengenau festlegen. Der Befehl **DRAW STRING** ermöglicht dagegen eine pixelgenaue Positionierung des Textes. Die Koordinaten geben die linke obere Ecke des ausgegebenen Textes an.

```
DRAW STRING (150, 100), "Ein kleiner Teststring"
```

DRAW STRING funktioniert wie alle *drawing primitives*: es kann z. B. auf Grafikbuffer angewendet werden und Aktionswörter einsetzen. Die Aktionswörter haben allerdings nur bei der Verwendung eines benutzerdefinierten Fonts eine Auswirkung. Wird einer der FreeBASIC-eigenen Standard-Schriftsätze verwendet, dann lässt **DRAW STRING** den Hintergrund bestehen und zeichnet den Ausgabetext darüber. Insbesondere wird ein bereits bestehender Text nicht in dem Sinne „überschrieben“, dass er gelöscht wird. Stattdessen bleibt sowohl der alte als auch der neue Text übereinander bestehen. Im unten stehenden Beispiel überlagern sich die Texte „kurz“ und „lang“.

Quelltext 4.5: DRAW STRING

```
SCREENRES 300, 300, 32
LINE (100, 100)-(200, 200), &hffff00, bf
DRAW STRING (60, 145), "Das ist ein kurzer Text", &h0000ff
DRAW STRING (60, 145), "Das ist ein langer Text", &h0000ff
5 GETKEY
```

Wenn Sie einen bestehenden Text überschreiben wollen, müssen Sie mit einer der in [Kapitel 4.4](#) genannten Methoden zuerst den Hintergrund wiederherstellen und dann den neuen Text ausgeben. Speziell für **DRAW STRING** gibt es auch noch die Möglichkeit, die alte Ausgabe mit gefüllten Kästchen (**CHR(219)**) in der Hintergrundfarbe zu überschreiben. Das funktioniert aber nur, wenn die Hintergrundfarbe einheitlich ist.

```
SCREENRES 300, 300, 32
DIM AS STRING text = "Hallo Welt!"
DRAW STRING (60, 145), text, &h0000ff           ' Text schreiben
GETKEY
5 DRAW STRING (60, 145), STRING(LEN(text), 219), 0 ' Text loeschen
GETKEY
```

FreeBASIC stellt drei Schriftsätze mit den Größen 8×8 , 8×14 und 8×16 zur Verfügung. Der verwendete Schriftsatz kann über **WIDTH** eingestellt werden. Genauer gesagt wird mit **WIDTH** die Anzahl der Zeilen und Spalten angegeben. Nur wenn sich diese Angabe sinnvoll in eine der oben angegebenen Schriftgrößen umrechnen lässt, wird der Schriftsatz auch entsprechend umgestellt. Die Methode hat aber einen Nachteil: **WIDTH** setzt gleichzeitig den Bildschirm zurück, löscht also seinen Inhalt. Das bedeutet, dass Sie auf diese Weise

immer nur eine Schriftgröße zur gleichen Zeit einsetzen können. Für die gleichzeitige Verwendung mehrerer Größen müssen Sie etwas in die Trickkiste greifen. Volta hat dazu eine Funktion zusammengestellt:

Quelltext 4.6: Verschiedene Schriftgrößen gleichzeitig

```
'fb_font_x.bas by Volta
TYPE fb_font_x
  AS INTEGER breit, hoch
  AS ANY PTR start
5 END TYPE
EXTERN Font8 ALIAS "fb_font_8x8" AS fb_font_x
EXTERN Font14 ALIAS "fb_font_8x14" AS fb_font_x
EXTERN Font16 ALIAS "fb_font_8x16" AS fb_font_x

10 SUB DrawString(BYVAL buffer AS ANY PTR = 0, BYVAL xpos AS INTEGER, _
  BYVAL ypos AS INTEGER, BYREF text AS STRING, _
  BYVAL fgcol AS INTEGER = COLOR, BYREF f AS fb_font_x)
  DIM AS INTEGER l, bits, xend
  DIM row AS UBYTE PTR
  15 l = LEN(text)-1
  IF l<0 THEN EXIT SUB
  SCREENINFO xend
  SCREENLOCK
  FOR i AS INTEGER = 0 TO l
    row = text[i]*f.hoch+f.start
    20 FOR y AS INTEGER = ypos TO ypos+f.hoch-1
      bits = *row
      FOR x AS INTEGER = xpos TO xpos+7
        IF (bits AND 1) THEN
          25 IF (buffer = 0) THEN
            PSET (x,y),fgcol
          ELSE
            PSET buffer,(x,y),fgcol
          END IF
        30 END IF
        bits = bits SHR 1
      NEXT
      row +=1
    NEXT
    35 xpos +=f.breit
    IF (xpos-f.breit)>xend THEN EXIT FOR
  NEXT
  SCREENUNLOCK
END SUB

40 SCREENRES 300, 200, 32
  DrawString ,10, 10, "Schrifttyp 8x8 Font", &hff0000, Font8
  DrawString ,30, 30, "Schrifttyp 8x14 Font", &h00ff00, Font14
  DrawString ,60, 60, "Schrifttyp 8x16 Font", &h0000ff, Font16
45 GETKEY
```

4.7. Double Buffering

Unter *double buffering* versteht man das Konzept, den Grafikspeicher in zwei Bereiche zu teilen. Während der Aufbau der grafischen Ausgabe in einem Speicherbereich stattfindet, wird der andere Bereich angezeigt. Erst wenn die Grafik komplett aufgebaut wurde, wechselt die Anzeige vom alten Bild auf das neue. Dadurch entsteht kein Flackern während des Aufbaus, und der grafische Ablauf kann flüssiger gestaltet werden. FreeBASIC stellt zu diesem Zweck einige Befehle zur Verfügung.

Achtung:

FreeBASIC verwendet intern bereits *double buffering*, sodass es in der Regel reicht, den Bildschirm während der Grafikroutinen, die zu einem Flackern führen können, mit **SCREENLOCK** zu sperren (siehe [Kapitel 4.4](#)). Eine zusätzliche Implementierung des *double buffering* führt zu größerem Speicheraufwand und erhöhter Rechenzeit. Die hier vorgestellte Methode ist nur sinnvoll, wenn während des Bildschirmaufbaus größere Berechnungen durchgeführt werden müssen, bei denen **SCREENLOCK**/**SCREENUNLOCK** zu Problemen führen kann.

Mit **SCREEN** bzw. **SCREENRES** kann die Anzahl der verwendeten Bildschirmseiten angegeben werden. Zwei Bildschirmseiten benötigen natürlich doppelt so viel Speicherplatz wie eine. Für *double buffering* werden aber mindestens zwei Seiten gebraucht, die im Folgenden mit 'aktive Seite' und 'sichtbare Seite' bezeichnet werden: auf der aktiven Seite finden die Grafikausgaben statt, während die sichtbare Seite angezeigt wird. Welches die aktive und welches die sichtbare Seite ist, kann mit **SCREENSET** festgelegt werden. Soll anschließend die gezeichnete Seite angezeigt werden, dann funktioniert das mit dem Befehl **SCREENCOPY**. Damit wird ganz einfach der Inhalt der aktiven Seite auf die sichtbare Seite kopiert.

Hinweis: Es gibt noch zwei weitere Befehle, die sich weitgehend identisch zu **SCREENCOPY** verhalten: Der QBASIC-Befehl **PCOPY** steht im Sinne der Abwärtskompatibilität auch in FreeBASIC zur Verfügung, und **FLIP** besitzt lediglich in einem OpenGL-Fenster eine eigene Bedeutung.

Die erste Bildschirmseite erhält die Nummer 0, bei zwei Bildschirmseiten kann also die Seite 0 und die Seite 1 angesprochen werden. Außerdem ist noch zu beachten: Wird das Kopieren der Bildschirmseiten durchgeführt, während der Bildschirm gerade aktualisiert wird, kann es zum Flackern kommen. Um das zu verhindern, wartet der Befehl **SCREENWAIT** auf die Aktualisierung des Bildschirms.

Das folgende Beispiel ist absichtlich nicht besonders performant. **PAINT** ist, wie schon einmal erwähnt, ein sehr langsamer Befehl, und das damit erreichte Befüllen des Bildschirmhintergrunds wäre besser gleich zusammen mit dem **CLS** erledigt worden. Dafür

sehen Sie in dem Beispiel sehr schön, wie es ohne *double buffering* zum Flackern kommt. Setzen Sie dazu in Zeile 2 einfach einmal **SCREENSET 0, 0**.

Quelltext 4.7: Double Buffering

	SCREENRES 200, 200, 32, 2	' 32bit Farbtiefe , zwei Bildschirmseiten
	SCREENSET 0, 1	' aktive und sichtbare Seite setzen
	DO	
	COLOR &h0000ff, &hff0000	' blau auf rot
5	CLS	
	PAINT (100, 100), &h00ff00	' gruener Hintergrund
	LINE (50, 50)–(150, 150),, BF	' blaues Rechteck
	SCREENSYNC	
	SCREENCOPY	' jetzt das Gezeichnete anzeigen
10	SLEEP 50, 1	
	COLOR &hff0000, &h0000ff	' rot auf blau
	CLS	
	PAINT (100, 100), &h00ff00	' gruener Hintergrund
	CIRCLE (100, 100), 70,,,,, f	' roter Kreis
15	SCREENSYNC	
	SCREENCOPY	' jetzt das Gezeichnete anzeigen
	SLEEP 50, 1	
	LOOP UNTIL INKEY <> ""	

Wie Sie sehen, müssen Sie sich beim *double buffering* nicht allzu sehr den Kopf zerbrechen. Sind erst einmal die zwei Bildschirmseiten gesetzt und als aktive und sichtbare Seite aufgeteilt, dann genügt ein **SCREENCOPY** immer an den Stellen, an denen ein neues Bild angezeigt werden soll. Im Gegensatz zu **SCREENLOCK** kommt es hier bei einer falschen Verwendung nicht zu einem Absturz. Schlimmstenfalls sieht der Benutzer bei einer vergessenen Bildschirmaktualisierung keine Veränderungen mehr, aber das Programm läuft dennoch 'normal' weiter.

5. Spielelemente

Zurück zum Labyrinth: Zu Beginn des Buches wurde das Spielfeld in einzelne kleine Felder unterteilt, in denen festgelegt wurde, an welcher Stelle sich beispielsweise Wände oder begehbare Bereiche befinden. Nun werden wir uns mit folgenden Fragen beschäftigen:

- Wie können unterschiedliche Spielobjekte umgesetzt und dargestellt werden?
- Wie verwendet man verschiedene Untergründe?
- Wie können die Informationen über ein Feld sinnvoll gespeichert werden?
- Wie können verschiedene Objekte miteinander verknüpft werden?
- Wie setzt man zeitgesteuerte Ereignisse um?

Dieses Kapitel ist (vielleicht abgesehen von [Kapitel 5.3](#)) weniger als Anleitung zu programmiertechnischen Fragen zu sehen, sondern vielmehr als Ideensammlung für verschiedene Spielelemente.

5.1. Spielobjekte

Neben Wänden und freien Feldern gibt es noch weitere Objekte, die im Labyrinth auftreten können. Eine Möglichkeit sind Türen, die erst geöffnet und passiert werden können, wenn zuvor ein passender Schlüssel aufgesammelt oder ein bestimmter Schalter betätigt wurde. Während das primäre Spielziel bisher war, den Ausgang zu erreichen, können auch zusätzliche Bonusobjekte (Geldtruhen o. ä.) eingeführt werden, die der Spieler einsammeln kann oder sogar muss, bevor der Weg zum Ausgang frei wird. Wenn das Spiel eine Zeitbeschränkung besitzt, kann außerdem ein weiteres Objekt eingeführt werden, das die verbleibende Zeit erhöht.

Mit der in [Kapitel 4.5](#) vorgestellten Einbindung externer Grafiken lässt sich eine optische Umsetzung der Spielobjekte leicht realisieren. Für jede Art von Objekt (Wand, Tür, Schlüssel, ...) gibt es eine Grafik, die zu Spielbeginn in den Speicher geladen wird und dann nur noch an der gewünschten Stelle ausgegeben werden muss. Wurde das Spielfeld in ein Raster gleich großer Felder unterteilt, dann sind sinnvollerweise auch die

Objekte auf diese Größe zu beschränken. Das ganze Spielfeld entsteht dann gewissermaßen wie ein Mosaik durch das Aneinandersetzen mehrerer Kacheln, auch Tiles genannt. Auch wenn diese Tiles festdefinierte Maße besitzen, müssen Sie deshalb auf größere Objekte nicht verzichten – setzen Sie diese doch einfach aus mehreren Tiles zusammen!

Wichtig neben dem Aussehen der Objekte ist deren Interaktion mit der Spielfigur. Kann die Spielfigur ein Feld betreten, auf dem sich das Objekt befindet? Wenn ja, was passiert anschließend mit dem Objekt – wird es „eingesammelt“ oder bleibt es bestehen? Wirkt es sich anderweitig positiv oder negativ auf den Status der Spielfigur aus?

5.2. Untergrund

„Objekte“, die beim Betreten nicht verändert werden, können auch als Untergrund behandelt werden. Der Vorteil an der Verwendung eines Untergrundes ist, dass er mit anderen Objekten kombiniert werden kann. Soll beispielsweise ein Schlüssel als Objekt eingefügt werden, dann kann der dazugehörige Untergrund unabhängig gewählt werden. Der Untergrund kann rein optischer Natur sein, aber auch spieltechnische Bedeutung haben, z. B. weil sich dadurch die Bewegung der Spielfigur verändert oder beim Aufenthalt auf dem Feld die Lebensenergie verringert.

Grafisch wird der Untergrund am einfachsten umgesetzt, indem zuerst die Untergrund-Grafik gezeichnet wird und darauf (bei Bedarf) das Objekt. Dazu ist natürlich die Verwendung von Transparenz – entweder mithilfe der Transparenzfarbe oder des Alpha-kanaals – sinnvoll. Der Alphakanal ermöglicht auch Teiltransparenzen, womit sich schöne Effekte erzielen lassen.

5.3. Eigener Datentyp

Wie „merkt“ sich nun das Programm am einfachsten die vielen verschiedenen Daten? In [Kapitel 2](#) wurde die Speicherung in einem **INTEGER**-Array erläutert. Diese Werte repräsentierten das Objekt, das sich an dieser Stelle befand (wobei hier freie Felder der Einfachheit halber ebenfalls als Objekte angesehen werden). Um mehrere Informationen gleichzeitig zu speichern, bietet sich die Verwendung eines UDTs (*user defined type*) an.

```
TYPE feldtyp
  AS ANY PTR hintergrund , vordergrund
  AS INTEGER betretbar , bonus
  AS INTEGER bewegungsmodifikation , lebensmodifikation
  ' weitere Merkmale ...
END TYPE
```

Sie sehen die Vorteile einer guten Namensgebung: die Bedeutung der einzelnen Records ist ziemlich offensichtlich. *hintergrund* und *vordergrund* sind Zeiger auf den Grafikpuffer, der das Hintergrund- bzw. Vordergrundbild beinhaltet. *betretbar* gibt an, ob das Feld betreten werden kann, und *bonus* speichert den aufsammelbaren Bonus (z. B. Punktebonus oder Schlüssel für Türen). *bewegungsmodifikation* und *lebensmodifikation* schließlich regeln, inwieweit die Bewegung und die Lebensenergie der Spielfigur durch das Betreten des Feldes beeinflusst wird. Die verwendeten Merkmale dienen natürlich nur zur Anregung. Sie können (und sollen) nach Belieben angepasst werden.

Ob diese Art der Speicherung sinnvoll ist, hängt stark vom Einsatzbereich ab. Besser ist es vermutlich, für die verschiedenen verfügbaren Untergründe und Objekte jeweils ein eigenes **UDT** anzulegen und im Feld beide einzubinden. Dies hat den Vorteil, dass die Auswirkung eines Objekts nur einmal definiert werden muss und dieses Objekt dann mehrmals (mit identischen Auswirkungen) verwendet werden kann.

Quelltext 5.1: Feld-Daten als UDT

```
5  TYPE untergrundtyp
    AS ANY PTR grafik
    AS INTEGER bewegungsmodifikation , lebensmodifikation
    ' weitere Merkmale ...
END TYPE

10 TYPE objekttyp
    AS ANY PTR grafik
    AS INTEGER betretbar , bonus
    ' weitere Merkmale ...
END TYPE

15 TYPE feldtyp
    AS untergrundtyp untergrund
    AS objekttyp objekt
    ' evtl. spezifische Merkmale des Feldes ...
END TYPE
```

5.4. Verknüpfung von Spielobjekten

Gelegentlich löst eine Aktion an der einen Stelle des Levels eine Reaktion an einer ganz anderen Stelle aus. Man denke dabei an Schalter, bei deren Betätigung sich ein Durchgang öffnet, oder an Druckplatten, bei deren Berührung etwas Schönes oder Schreckliches passiert. Für die Umsetzung bietet sich die Erweiterung des **UDTs** *feldtyp* um die Koordinaten des verknüpften Feldes an. Der unten stehende Codeschnipsel setzt eine Druckplatte für das Öffnen und Schließen einer Tür. Beachten Sie bitte, dass die Funktionsweise nur angedeutet ist und für eine saubere Umsetzung noch einiges getan

werden muss.

Der Codeschnipsel verwendet für *untergrundtyp* und *objekttyp* noch einen weiteren Record *id*, der die besondere Art des Untergrunds bzw. des Objekts angibt. Damit ist es leichter herauszufinden, wie im aktuellen Fall reagiert werden muss. Die verschiedenen Untergrund- und Objektarten werden zur Verdeutlichung durch Variablen – in diesem Fall *druckplatte*, *tuerAuf* und *tuerZu* – repräsentiert, die natürlich noch deklariert werden müssen. Dafür kann z. B. **ENUM** eingesetzt werden; der unten stehende Code ignoriert dies jedoch und überlässt die Umsetzung dem Leser als Übung.

Quelltext 5.2: Feld-Daten als UDT (2)

```

5  TYPE untergrundtyp
    AS ANY PTR grafik
    AS INTEGER id, bewegungsmodifikation, lebensmodifikation
    ' weitere Merkmale ...
END TYPE

10 TYPE objekttyp
    AS ANY PTR grafik
    AS INTEGER id, betretbar, bonus
    ' weitere Merkmale ...
END TYPE

15 TYPE feldtyp
    AS untergrundtyp untergrund
    AS objekttyp objekt
    AS INTEGER zielX, zielY
END TYPE

20 ' ...

DIM AS feldtyp feld = felddata(sx, sy) ' Information des Spielerfeldes
IF feld.undergrund.id = druckplatte THEN
    DIM AS feldtyp ziel = felddata(feld.zielX, feld.zielY) ' Zielfeld der Aktion
    IF ziel.objekt.id = tuerZu THEN ziel.objekt.id = tuerAuf
    ' ...
25 END IF

```

5.5. Zeitgesteuerte Ereignisse

Für Objekte, die abhängig von der Zeit gesteuert werden – z. B. Türen, die sich alle fünf Sekunden automatisch öffnen bzw. schließen – gibt es zwei grundsätzliche Lösungsansätze. Eine häufig verwendete Methode, um eine vom Rest des Programms völlig unabhängige Steuerung zu erreichen, ist der Einsatz von Multithreading. Die Grundlagen dazu werden in [Kapitel 8](#) gesondert behandelt. Alternativ dazu kann man auch versuchen, die Zeitsteuerung direkt in die Hauptschleife des Spiels zu integrieren. Selbstverständlich kann

der Spielablauf nicht einfach so lange pausiert werden, bis das Ereignis – also z. B. das Öffnen der Tür – eintritt. Schließlich soll es ja während der Wartezeit weiterhin möglich sein, die Spielfigur zu steuern, und vielleicht gibt es auch noch weitere zeitgesteuerte Ereignisse, die währenddessen überprüft werden müssen.

Die Idee ist folgende: Mithilfe der Funktion **TIMER** wird in einer Variablen der Zeitpunkt festgehalten, an dem das Ereignis zuletzt ausgeführt wurde. Eine weitere Variable speichert, wie lange es bis zur nächsten Ausführung des Ereignisses dauert. **TIMER** gibt die Anzahl der vergangenen Sekunden seit dem Systemstart zurück² – der Wert erhöht sich also ständig. Es muss nun regelmäßig verglichen werden, ob bereits genug Zeit verstrichen ist, um das Ereignis auszulösen. Dieses Prinzip wird im [Quelltext 5.3](#) demonstriert. Das Programm gibt jede Sekunde einen Punkt aus; daneben ist es zu jedem Zeitpunkt möglich, eine Tastatureingabe zu machen, die ebenfalls ausgegeben wird. Beachten Sie, dass das Programm in einer Konsole ausgeführt werden muss.

Quelltext 5.3: Zeitsteuerung in der Hauptschleife

```
5  DIM AS DOUBLE letzteAusfuehrung = TIMER ' Zeitpunkt der letzten Ausfuehrung
    DIM AS DOUBLE naechsteAusfuehrung = 1.0 ' Sekunden zwischen zwei Ausfuehrungen
    DIM AS STRING taste ' Benutzereingabe
    DO
        taste = INKEY
        IF taste = CHR(27) THEN
            EXIT DO ' Programmende bei Eingabe von ESC
        ELSEIF taste <> "" THEN
            PRINT taste; ' Benutzereingabe anzeigen
        END IF
        IF TIMER > letzteAusfuehrung + naechsteAusfuehrung THEN
            ' Ausfuehrung des Ereignisses
            PRINT ".";
            letzteAusfuehrung = TIMER ' aktuellen Zeitpunkt speichern
        END IF
        SLEEP 1 ' Pause fuer den Prozessor
    LOOP
```

² Unter Windows und DOS; unter Linux und anderen unixartigen Betriebssystemen werden stattdessen die vergangenen Sekunden seit der Unix-Epoche (01.01.1970) zurückgegeben.

6. Anwendung: ein Minensuchspiel

Mithilfe der bisherigen Ergebnisse wollen wir nun ein kleines Spiel zusammenstellen, das bei Windows-Benutzern unter dem Namen *Minesweeper* bekannt ist. Auf dem Spielfeld werden nach Zufallsprinzip mehrere Bomben verteilt. Der Spieler hat die Aufgabe, alle Felder aufzudecken, ohne dabei eine der Bomben zu erwischen. Jedes aufgedeckte Feld zeigt an, wie viele Bomben waagrecht, senkrecht oder diagonal an ihm angrenzen.

6.1. Spielelemente und Bombenverteilung

Um die Spielfeldgröße flexibel zu halten, werden Breite und Höhe des Spielfeldes sowie die Größe der einzelnen Felder und Anzahl der Bomben in Variablen gespeichert. Zu Beginn könnte der Spieler dann die Feldgröße und die Bombenzahl einstellen. Im Augenblick werden die Parameter jedoch fest im Quellcode verankert.

Jedes Feld muss speichern, ob es eine Bombe enthält und ob es verdeckt, mit einer Fahne markiert oder aufgedeckt wurde. Diese Eigenschaften werden als Flags in einem gemeinsamen **INTEGER** gespeichert. Die Anzahl der Bomben, die sich um das Feld herum befinden, könnte ebenfalls gespeichert werden; man benötigt sie jedoch lediglich einmal an der Stelle, an der ein Feld aufgedeckt wird, weshalb sie ohne Aufwand direkt beim Aufdecken berechnet werden kann.

Quelltext 6.1: Minensuchspiel – Spielelemente

```
#DEFINE Offen 16      ' 5. Bit – die Zahlen 0–8 werden freigehalten
#DEFINE Fahne 32      ' 6. Bit
#DEFINE Bombe 64      ' 7. Bit
5 DIM SHARED AS INTEGER Feldbreite = 20, Feldhoehe = 20, Feldgroesse = 20
DIM SHARED AS INTEGER Bombenzahl = 50
DIM SHARED AS INTEGER feld(0 TO Feldbreite-1, 0 TO Feldhoehe-1)
DIM SHARED AS ANY PTR BildBombe, BildFahne
SCREENRES Feldbreite*Feldgroesse, Feldhoehe*Feldgroesse, 32
BildBombe = IMAGECREATE(Feldgroesse, Feldgroesse)
10 BildFahne = IMAGECREATE(Feldgroesse, Feldgroesse)
BLOAD "bombe.bmp", BildBombe
BLOAD "fahne.bmp", BildFahne
```

Der Code setzt voraus, dass zwei 32bit-BMPs namens *bombe.bmp* und *fahne.bmp* existieren, welche genau 20x20 Pixel groß sind.

Als nächstes werden die Bomben verteilt. Dazu sucht sich das Programm per Zufalls-generator einen x- und y-Wert. Sofern sich an dieser Stelle noch keine Bombe befindet, wird sie nun gesetzt. Achten Sie darauf, dass Sie die Bombenzahl nicht höher setzen als die Gesamtzahl der Felder – die Bombenverteilung wird sich sonst aufhängen, sobald keine freien Felder mehr verfügbar sind. Das müssen Sie vor allem dann beachten, wenn Sie den Spieler die Feldgröße und Bombenzahl frei einstellen lassen.

Quelltext 6.2: Minensuchspiel – Bomben verteilen

```
5  DIM AS INTEGER verteilt = 0, x, y
    RANDOMIZE
    DO
        x = INT(RND*Feldbreite)
        y = INT(RND*Feldhoehe)
        IF feld(x, y) = Bombe THEN CONTINUE DO
        feld(x, y) = Bombe
        verteilt += 1
    LOOP UNTIL verteilt = Bombenzahl
```

6.2. Felder aufdecken

Bevor der Programmablauf zusammengestellt wird, machen wir uns erst Gedanken über das Aufdecken und Anzeigen der einzelnen Felder. Das Spielprinzip wird sehr einfach gehalten: ein Rechtsklick auf ein Feld markiert dieses mit einer Fahne (oder löscht die Markierung wieder), ein Linksklick deckt es auf. Es ist sinnvoll, mit einer Fahne markierte Felder nicht aufzudecken. Klickt der Spieler dagegen auf ein Feld ohne benachbarter Bombe, dann sollen automatisch auch alle umliegenden Felder aufgedeckt werden. Das wird durch einen rekursiven Aufruf der Aufdeck-Funktion erreicht. Dazu ist es aber nötig, dass bereits offene Felder nicht erneut aufgedeckt werden; sonst würde sich die Rekursion in einer Endlosschleife verfangen (genauer gesagt: das Programm würde wegen der immer größeren Rekursionstiefe schnell den kompletten Stapelspeicher ausschöpfen und mit einem *segmentation fault* abbrechen).

Die Funktion *aufdecken* dient zur Anzeige eines einzelnen Feldes. Das Feld wird als aufgedeckt markiert und sein Inhalt – die Anzahl der angrenzenden Bomben oder das Bomben-BMP – angezeigt. Wenn sich keine Bombe in direkter Umgebung befindet, werden alle angrenzenden Felder rekursiv aufgedeckt. Der Rückgabewert der Funktion ist entweder die Anzahl der umliegenden Bomben oder eine der Werte *Offen*, *Fahne* oder *Bombe*, wenn versucht wurde, ein entsprechendes Feld aufzudecken.

Des Weiteren wird ein Unterprogramm namens *feldZeigen* bereitgestellt, das alle verbleibenden Felder aufdeckt. Es wird erst ganz am Ende des Programms benötigt, etwa wenn der Spieler eine Bombe anklickt. Es kann aber auch zu Testzwecken eingesetzt

werden, um zu überprüfen, ob die Verteilung der Bomben zu Beginn des Programms wie erwartet abgelaufen ist.

Quelltext 6.3: Minensuchspiel – Feld aufdecken

```

FUNCTION aufdecken(x AS INTEGER, y AS INTEGER) AS INTEGER
  ' offene sowie mit Fahne versehene Felder werden nicht aufgedeckt
  IF feld(x, y) AND Offen THEN RETURN Offen
  IF feld(x, y) AND Fahne THEN RETURN Fahne
5
  ' Aufdecken einer Bombe
  IF feld(x, y) AND Bombe THEN                                ' "Bombe"-Flag gesetzt?
    PUT (x*FeldGroesse, y*FeldGroesse), BildBombe
    feld(x, y) OR= Offen                                         ' das "Offen"-Flag setzen
10    RETURN Bombe
  END IF

  ' normales Feld: umgebende Bomben zaehlen
  DIM AS INTEGER zaehler = 0
15  FOR i AS INTEGER = x-1 TO x+1
    IF i < 0 OR i >= FeldBreite THEN CONTINUE FOR              ' ausserhalb des Spielfelds
    FOR k AS INTEGER = y-1 TO y+1
      IF k < 0 OR k >= FeldHoehe THEN CONTINUE FOR            ' ausserhalb des Spielfelds
      IF feld(i, k) AND Bombe THEN zaehler += 1
20      NEXT
    NEXT
  DRAW STRING ((x+.5)*FeldGroesse-4, (y+.5)*FeldGroesse-4), STR(zaehler)
  feld(x, y) OR= Offen                                         ' das "Offen"-Flag setzen
25
  ' rekursiver Aufruf, wenn keine Bomben in der Naehе sind
  IF zaehler = 0 THEN
    FOR i AS INTEGER = x-1 TO x+1
      IF i < 0 OR i >= FeldBreite THEN CONTINUE FOR
      FOR k AS INTEGER = y-1 TO y+1
30        IF k < 0 OR k >= FeldHoehe THEN CONTINUE FOR
        aufdecken i, k
      NEXT
    NEXT
  END IF
35
  RETURN zaehler
END FUNCTION

SUB feldZeigen
40  FOR x AS INTEGER = 0 TO FeldBreite-1
    FOR y AS INTEGER = 0 TO FeldHoehe-1
      aufdecken x, y
    NEXT
  NEXT
45 END SUB

```

6.3. Hauptprogramm

Nun kommen wir zum eigentlichen Programm. Da die Vorarbeit schon weitgehend erledigt ist, gibt es nicht mehr viel zu tun. Das Programm fragt die Maus-Eingabe ab und deckt das angeklickte Feld auf bzw. markiert es mit einer Fahne. Durch das Aktionswort **XOR** kann das Setzen und Entfernen der Fahne mit einer einzigen Anweisung erledigt werden – ist noch keine Fahne vorhanden, wird sie gezeichnet, sonst wird sie wieder gelöscht.

Beendet wird das Spiel mit der ESC-Taste oder wenn eine Bombe angeklickt wurde.

Quelltext 6.4: Minensuchspiel – Hauptprogramm

```

' Raster zeichnen
FOR i AS INTEGER = 0 TO Feldbreite-1
  FOR k AS INTEGER = 0 TO Feldhoehe-1
    LINE (i*Feldgroesse, k*Feldgroesse)-STEP(Feldgroesse-1, Feldgroesse-1),, B
5  NEXT
NEXT

DIM AS INTEGER mausX, mausY, mausB, mausX2, mausY2, mausB2, fx, fy, wert
DO
10  ' Maus abfragen
    GETMOUSE mausX, mausY,, mausB
    IF mausB THEN
        ' Position berechnen
        fx = mausX \ Feldgroesse
        fy = mausY \ Feldgroesse
15  ' auf das Loslassen der Maus warten
        DO
            GETMOUSE mausX2, mausY2,, mausB2
            SLEEP 1
20  LOOP UNTIL mausB2 = 0
        ' ueberpruefen, ob sich die Maus noch im selben Feld befindet
        IF mausX2\Feldgroesse <> fx OR mausY2\Feldgroesse <> fy THEN CONTINUE DO
        IF mausB = 1 THEN
            wert = aufdecken(fx, fy)
25  IF wert = Bombe THEN feldZeigen : GETKEY : EXIT DO
        ELSEIF mausB = 2 AND (feld(fx, fy) AND Offen) = 0 THEN
            ' Fahnenmarkierung setzen bzw. loeschen
            feld(fx, fy) XOR= Fahne
            PUT (fx*Feldgroesse, fy*Feldgroesse), BildFahne, XOR
30  END IF
        END IF
        SLEEP 1
    LOOP UNTIL INKEY = CHR(27)
    IMAGEDESTROY bildBombe
35  IMAGEDESTROY bildFahne
    ' Bildpuffer freigeben
```

6.4. Zusammenfassung

Wenn Sie die Überschrift einmal ganz wörtlich nehmen und alle Quellcodes dieses Kapitels zu einem Programm zusammenfassen, erhalten Sie bereits ein funktionierendes Spiel. Es gibt noch einige Stellen, an denen das Programm ausgebaut werden kann. Hierzu ein paar Anregungen:

- Zu Beginn des Programms könnte der Spieler die Spielfeldgröße und die Bombenzahl wählen. Achten Sie darauf, dass nicht mehr Bomben gewählt werden dürfen, als auf dem Spielfeld Platz haben!
- Beim Aufdecken des gesamten Feldes werden Fahnenmarkierungen bisher beibehalten. Sinnvoller wird es sein, diese Markierungen zu löschen und durch eine Bombe bzw. den Zahlenwert zu ersetzen.
- Wenn Sie während des Spiels die verstrichene Zeit anzeigen, erhält der Spieler eine Rückmeldung über seine Leistung. Diese kann auch in einem Highscore festgehalten werden.
- Das Spiel wird bisher noch nicht automatisch beendet, wenn alle Felder (außer den Bomben) aufgedeckt wurden.

Teil II.

Anbauten

7. Highscore

Eine Highscore-Tabelle kann über eine gewöhnliche Textdatei verwaltet werden. Normalerweise enthält ein Eintrag nur den Namen und die Punktzahl des entsprechenden Spielers. Die Angabe kann aber auch z. B. um das erreichte Level erweitert werden. Im Folgenden wird für jeden Highscore-Eintrag eine Zeile verwendet. Sie beginnt mit der Punktzahl, gefolgt von einem Leerzeichen und dem Spielernamen:

```
1032 Max
944 Moritz
716 Witwe Bolte
```

In dieser Reihenfolge ist die Eingabe am besten zu verarbeiten, und der Spielername kann auch Leerzeichen enthalten, ohne dass diese für Probleme sorgen.

7.1. Highscore einlesen

Zu Beginn muss der bereits bestehende Highscore eingelesen werden. Für den allerersten Start des Programms empfiehlt es sich, per Hand eine Highscore-Tabelle anzulegen und dort Werte einzutragen, die für einen durchschnittlichen Spieler gut erreichbar sind, aber doch eine gewisse Herausforderung darstellen. In der Regel werden Sie mit einer festen Anzahl an Einträgen arbeiten; wir gehen hier von zehn Einträgen aus, die von 0 bis 9 nummeriert sind.

Quelltext 7.1: Highscore-Type

```
TYPE highscore
  AS INTEGER punkte
  AS STRING spieler
END TYPE
5 DIM AS highscore highTab(9)
```

Zum Einlesen wird die Datei *highscore.txt* (sie könnte natürlich auch anders heißen) geöffnet. Die Daten werden zeilenweise eingelesen und anhand des ersten Leerzeichens in die Punktzahl und den Spielernamen zerlegt. Als Punktzahl muss nur die eingelesene Zeile mittels **VALINT** in einen **INTEGER** umgewandelt werden – **VALINT** bricht automatisch beim trennenden Leerzeichen ab. Für den Spielernamen wird mit **INSTR** nach dem ersten

Leerzeichen gesucht und dann alles hinter diesem Leerzeichen mit der Funktion **MID** ausgelesen.

Quelltext 7.2: Highscore einlesen

```
SUB highscoreLesen(highTab() AS highscore)
  DIM zeile AS STRING, such AS INTEGER, dateinr AS INTEGER = FREEFILE
  IF OPEN ("highscore.txt" FOR INPUT AS #dateinr) = 0 THEN
    ' erfolgreich geoeffnet - Daten koennen eingelesen werden
5    FOR i AS INTEGER = 0 TO UBOUND(highTab)
      LINE INPUT #dateinr, zeile
      highTab(i).punkte = VALINT(zeile)           ' Punktzahl einlesen
      such = INSTR(zeile, " ")                   ' erstes Leerzeichen suchen
      highTab(i).spieler = MID(zeile, such+1)     ' Name hinter dem Leerzeichen
10   NEXT
      CLOSE #dateinr
    END IF
  END SUB
```

Beachten Sie, dass ein Array immer **BYREF** übergeben wird und daher die Änderung, die innerhalb der Prozedur an der Highscore-Tabelle vorgenommen wird, auch im Hauptprogramm sichtbar ist.

Es wird davon ausgegangen, dass das Array der Highscore-Tabelle bei 0 beginnt. Die obere Grenze ist jedoch nicht festgelegt, sondern wird aus dem übergebenen Array ermittelt. Dadurch bleibt die Prozedur flexibler. Wenn Sie auch die untere Grenze variabel halten wollen, können Sie zusätzlich **LBOUND** einsetzen.

7.2. Highscore schreiben

Das Schreiben des Highscores geht noch einfacher, da die zu schreibenden Zeilen nur als Punktzahl + Leerzeichen + Spielername zusammengesetzt werden müssen. Damit kann dieser Abschnitt sehr kurz abgehandelt werden.

Quelltext 7.3: Highscore schreiben

```
SUB highscoreSchreiben(highTab() AS highscore)
  DIM zeile AS STRING, dateinr AS INTEGER = FREEFILE
  IF OPEN ("highscore.txt" FOR OUTPUT AS #dateinr) = 0 THEN
    ' erfolgreich geoeffnet - Daten koennen geschrieben werden
5    FOR i AS INTEGER = 0 TO UBOUND(highTab)
      ' Datenzeile schreiben
      PRINT #dateinr, highTab(i).punkte & " " & highTab(i).spieler
    NEXT
    CLOSE #dateinr
10   END IF
  END SUB
```

7.3. Highscore bearbeiten

Zu klären bleibt noch der Umgang mit neuen Highscore-Ergebnissen. Nach jedem Spiel muss die erreichte Punktzahl mit der Highscore-Tabelle abgeglichen und bei Bedarf an der richtigen Stelle eingefügt werden. Schlechtere Ergebnisse rutschen dabei um eine Position nach unten, der zuvor letzte Eintrag fällt aus der Tabelle heraus.

Quelltext 7.4 durchsucht den Highscore von hinten beginnend, bis er auf einen höheren Punktestand stößt. Während der Suche werden die überprüften Einträge um eine Position nach hinten verschoben – also fällt der zehnte Eintrag aus der Tabelle heraus, der neunte Eintrag kommt an die zehnte Stelle, der achte Eintrag an die neunte Stelle und so weiter. An der zuletzt frei werdenden Stelle wird der neue Eintrag vorgenommen.

Quelltext 7.4: Highscore bearbeiten

```
highscoreLesen highTab()

' Spielablauf – neuen Spielstand ermitteln
' ...

5  ' neuen Spielstand einreihen
  DIM AS INTEGER ub = UBOUND(highTab)
  IF highTab(ub).punkte < neuePunktzahl THEN      ' neuer Eintrag erforderlich
    FOR i AS INTEGER = ub-1 TO 0 STEP -1
      IF highTab(i).punkte < neuePunktzahl THEN    ' Eintrag nach unten schieben
        highTab(i+1) = highTab(i)
      ELSE
        highTab(i+1).punkte = neuePunktzahl      ' neuen Eintrag schreiben
        highTab(i+1).spieler = neuerSpielername
      EXIT FOR
    END IF
  NEXT
END IF

15
20 highscoreSchreiben highTab()
```

Die Abarbeitung von hinten nach vorn ist sinnvoll, weil es nach häufigerem Spielen immer schwerer wird, alte Ergebnisse zu überbieten. Die Schleife wird also früher verlassen. Noch entscheidender ist jedoch, dass auf diese Weise gleichzeitig die Verschiebung der schlechteren Ergebnisse durchgeführt werden kann.

Die sehr einfache Art der Speicherung hat natürlich den Nachteil, dass der Highscore auch sehr leicht von außen manipuliert werden kann. Wenn Sie es dem Benutzer nicht ganz so leicht machen wollen, den Highscore nach seinen Wünschen anzupassen, können Sie auch auf die binäre Speicherung aus [Kapitel 2.3.2](#) zurückgreifen und die Inhalte gegebenenfalls verschlüsseln. Beachten Sie jedoch, dass Sie damit nur die ganz offensichtliche Manipulation verhindern.

8. Nebenläufigkeit

8.1. Aufbau

Mithilfe von Threads ist es möglich, mehrere Programmteile unabhängig voneinander ablaufen zu lassen. Quelltext 5.3 gab jede Sekunde einen Punkt aus, während der Benutzer eine Eingabe tätigen konnte. Dieses Programm kann nun dahingehend geändert werden, dass die regelmäßige Ausgabe des Punktes völlig unabhängig vom Hauptprogramm geschieht.

Die Verwendung von Threads wirft jedoch einige Schwierigkeiten auf. Wenn zwei Threads (auch das Hauptprogramm gilt dabei als Thread) gleichzeitig auf dieselben Ressourcen zugreifen, kann das zu unerwarteten Ergebnissen führen; auch Programmabstürze sind keine Seltenheit. Um dies zu verhindern, werden Mutexe eingesetzt. Ein Mutex (**mutual exclusion**, zu deutsch „wechselseitiger Ausschluss“) dient dazu, einen exklusiven Zugriff auf bestimmte Ressourcen zu erlangen – der Thread sperrt den Mutex, führt den kritischen Abschnitt aus und gibt den Mutex anschließend wieder frei. Will währenddessen ein zweiter Thread den Mutex sperren, muss er erst solange warten, bis der Mutex vom ersten Thread entsperrt wurde. Damit ist sichergestellt, dass immer nur eine der kritischen Stellen gleichzeitig ausgeführt werden kann. Wichtig ist jedoch, die Freigabe des Mutex nicht zu vergessen, da das Programm sonst hängen bleibt.

Die für Threads benötigten Befehle sind **THREADCREATE** zum Erstellen und **THREADWAIT**, um auf das Beenden des Threads zu warten. Für die Mutexe werden die Befehle **MUTEXCREATE** zum Erstellen, **MUTEXLOCK** und **MUTEXUNLOCK** zum Sperren bzw. Entsperren sowie **MUTEXDESTROY** zum Zerstören des Mutex benötigt. Quelltext 8.1 erstellt eine regelmäßige, zeitgesteuerte Textausgabe innerhalb eines Threads, während das Hauptprogramm die Benutzereingabe erlaubt. Der gemeinsame Zugriff auf die Variable *terminate* wird durch ein Mutex geschützt. *terminate* wird als Abbruchbedingung für den Thread benötigt. Die Abfrage wird in Quelltext 8.1 durch die Hilfsvariable *endetest* gelöst, welche im mutexgeschützten Bereich den Wert von *terminate* aufnimmt und im weiteren Programmverlauf ohne Schwierigkeiten abgefragt werden kann.

Für eine einzige Variable ist der Einsatz einer lokalen Kopie nicht unbedingt nötig; man könnte nach dem Sperren des Mutex direkt *terminate* abfragen und gegebenenfalls den Mutex entsperren und die Schleife verlassen. Bei umfangreicherer Arbeit mit zu

schützenden Variablen ist (jedenfalls beim rein lesenden Zugriff) das anfängliche Anlegen einer Kopie dagegen durchaus sinnvoll. Das Sperren und Entsperren der Mutexe hält einige Fallstricke bereit, auf die Sie achten sollten; siehe dazu die Anmerkungen in [Kapitel 8.3](#).

Quelltext 8.1: Zeitsteuerung in einem Thread

```

DIM SHARED AS INTEGER terminate = 0
DIM SHARED mutex AS ANY PTR
mutex = MUTEXCREATE

5 SUB punktethread
  DIM AS INTEGER endetest
  DO                                     ' jede Sekunde einen Punkt ausgeben
    MUTEXLOCK mutex
    endetest = terminate
  10  MUTEXUNLOCK mutex
    IF endetest = 1 THEN EXIT DO      ' Programm wurde terminiert
    PRINT ".";
    SLEEP 1000, 1
  LOOP
15 END SUB

' Thread starten
DIM AS ANY PTR thread = THREADCREATE(CAST(ANY PTR, @punktethread))
DIM AS STRING taste

20 DO
  taste = INKEY
  IF taste = CHR(27) THEN
    EXIT DO
  25  ELSEIF taste <> "" THEN
    PRINT taste;
    END IF
    SLEEP 1
  LOOP

30  MUTEXLOCK mutex
    terminate = 1
  35  MUTEXUNLOCK mutex
    THREADWAIT thread
    MUTEXDESTROY mutex

```

Der Befehl **PRINT** ist im Konsolenfenster threadsicher, das bedeutet, dass sich zwei gleichzeitig in verschiedenen Threads stattfindende **PRINT**-Ausgaben nicht in die Quere kommen, da sie intern gekapselt werden. Bei grafischer Ausgabe sieht die Sache schon anders aus; hier müssen Sie selbst für die Threadsicherheit sorgen. Dazu gehört auch **PRINT** im Grafikfenster! Der Grund dafür ist der schreibende Zugriff auf den Speicherbereich des Fensters.

Generell sind alle schreibenden Zugriffe auf gemeinsam genutzten Speicher durch einen

Mutex zu schützen. Wenn alle Threads nur lesend auf eine Variable zugreifen, ist kein Mutex erforderlich. Bei gleichzeitigem Lesen und Schreiben derselben Variablen können, insbesondere bei **UDTs**, unter Umständen bereits Probleme auftreten. Lesen und Schreiben von lokalen Variablen ist dagegen unproblematisch, da diese nicht mit anderen Threads geteilt werden.

Achten Sie andererseits darauf, dass die Ausführungszeit des gesperrten Bereichs so kurz wie möglich ist, da Sie sonst die anderen Threads unnötig ausbremsen. Betrachten Sie die folgenden beiden Varianten:

```
' Variante 1: MUTEXLOCK ausserhalb der Schleife
MUTEXLOCK grafikMutex
FOR i AS INTEGER = 1 TO 100
    grafikbefehl 1
    grafikbefehl 2
    nochIrgendwas
NEXT
MUTEXUNLOCK grafikMutex

' Variante 2: MUTEXLOCK innerhalb der Schleife
FOR i AS INTEGER = 1 TO 100
    MUTEXLOCK grafikMutex
    grafikbefehl 1
    grafikbefehl 2
    MUTEXUNLOCK grafikMutex
    nochIrgendwas
NEXT
```

Welche der beiden Vorgehensweisen besser ist, lässt sich nicht verallgemeinern. Es hängt vom Aufruf *nochIrgendwas* ab. Benötigt dieser Aufruf viel Rechenzeit, dann ist die erste Variante eine schlechte Idee: Der Mutex wird während der Schleife komplett blockiert und andere Threads müssen lange auf die Freigabe warten. Bei einem sehr kurzen *nochIrgendwas* dagegen läuft die Schleife grundsätzlich schnell durch, wird aber durch das ständige Sperren und Entsperren stark ausgebremst.

Des Weiteren ist bei Prozedur-Aufrufen darauf zu achten, dass die Prozedur keinen bereits gesperrten Mutex sperrt. Das hätte sonst unweigerlich einen Stillstand des Threads zur Folge! Darauf wird in [Kapitel 8.3](#) genauer eingegangen.

8.2. Optimierung für Multicore-Prozessoren

Threads können auch gezielt dafür eingesetzt werden, um die Performance auf Multicore-Prozessoren zu steigern. Ein Programm, das aus einem einzelnen Thread besteht, wird auch nur von einem Kern verarbeitet. Wird der Programmablauf in zwei Threads aufgeteilt, können(!) beide Threads auf verschiedenen Kernen ausgeführt werden. Die Zuteilung

der Threads zu den Kernen übernimmt allerdings das Betriebssystem.

Wenn zwei Teilbereiche eines Programms von zwei verschiedenen Kernen ausgeführt werden statt nur von einem, kann dies zu deutlichen Geschwindigkeitsvorteilen führen. Doch Vorsicht: Eine Aufteilung sorgt nicht automatisch für mehr Geschwindigkeit. Es kann sogar der gegenteilige Effekt eintreten!

Schuld daran ist die Notwendigkeit, beim Zugriff auf gemeinsame Ressourcen alle beteiligten Kerne untereinander abzugleichen. Wird ein Mutex gesperrt, müssen alle Kerne gleichzeitig ihre Überprüfung durchführen, da es sonst passieren könnte, dass zwei Kerne gleichzeitig sperren. Der Abgleich zwischen den Kernen wird vom Betriebssystem übernommen; Sie müssen sich darüber also nicht den Kopf zerbrechen. Um jedoch einen Geschwindigkeitsvorteil zu erzielen, sollten Sie darauf achten, dass die einzelnen Threads ihre Arbeit so weit wie möglich unabhängig voneinander erledigen und ein Datenabgleich nur selten erforderlich ist.

Suchen Sie also Programmstränge, die weitgehend unabhängig voneinander sind. Denkbar ist z. B., dass ein Thread die nächste Animation berechnet, während ein anderer die physikalischen Berechnungen übernimmt. Größere Berechnungen können oft auch in zwei Blöcke geteilt werden, etwa die Darstellung der rechten und linken Hälfte eines Bildschirms oder auch nur eines bestimmten Objekts. Dabei ist es sinnvoll, wenn beide Threads in etwa gleich viel Rechenzeit beanspruchen – die Aufteilung in einen großen und viele sehr kleine Threads ist für die Performance ungünstig.

```
' weitere Berechnung auf zwei Threads aufteilen
DIM AS ANY PTR thread1 = THREADCREATE(CAST(ANY PTR, @linkeSeiteBerechnen))
DIM AS ANY PTR thread2 = THREADCREATE(CAST(ANY PTR, @rechteSeiteBerechnen))
' warten, bis beide Threads fertig gerechnet haben
THREADWAIT thread1
THREADWAIT thread2
```

Ansonsten können Threads auch einfach nur eingesetzt werden, um den Programmcode übersichtlicher zu gestalten. Dafür bieten sich alle zeitgesteuerten Elemente an wie das Herunterzählen der Spielzeit, automatische Türen und vieles mehr. Auch die computergesteuerte Bewegung anderer Figuren (siehe [Kapitel 12](#)) kann sehr gut in einen Thread ausgelagert werden. Mit einem gut durchdachten System lassen sich all diese Ereignisse auch direkt im Hauptprogramm steuern. Durch die Auslagerung in Threads wird das Hauptprogramm allerdings schlanker und damit übersichtlicher. Achten Sie nur darauf, dass sich die einzelnen Threads nicht zu oft untereinander „austauschen“ müssen.

8.3. Fehlerquellen

Der Nachteil an Multithreading ist, dass sich Fehler in der Programmierung oft nur sehr schwer aufspüren lassen. Der an sich offensichtlichste Fehler, der auch sehr schnell Auswirkungen zeigt, ist das doppelte Sperren eines Mutex bzw. ein vergessenes **MUTEXUNLOCK**. Bereits erwähnt wurde die Situation, dass innerhalb eines gesperrten Abschnitts eine Prozedur aufgerufen wird. Wenn diese – direkt oder indirekt – denselben Mutex sperren will, kommt es zu einem Konflikt. Der folgende rekursive Aufruf ist eine etwas umständlichere Art, $zahl \bmod param$ zu berechnen, wozu *param* jedoch während der Berechnung konstant bleiben muss. Das Problem der Funktion ist aber, dass beim zweiten Rekursionsschritt ein Mutex gesperrt werden soll, der noch vom ersten Schritt gesperrt ist.

Quelltext 8.2: Mutex-Fehler im rekursiven Aufruf

```
FUNCTION fehlerhafteRekursion(zahl AS INTEGER) AS INTEGER
  DIM AS INTEGER rueckgabe
  MUTEXLOCK einMutex           ' Schutz der Variablen 'param'
  IF zahl > param THEN
5    ' ACHTUNG: der naechste Aufruf wird fuer Probleme sorgen!
      rueckgabe = fehlerhafteRekursion(zahl - param)
  ELSE
      rueckgabe = zahl
  END IF
10  MUTEXUNLOCK einMutex
  RETURN rueckgabe
END FUNCTION
```

Manchmal wird das **MUTEXUNLOCK** auch einfach vergessen. Besonders in Prozeduren fällt das nicht immer sofort ins Auge:

Quelltext 8.3: Mutex-Fehler bei der Rückgabe

```
FUNCTION fehlerhafteFunktion AS INTEGER
  MUTEXLOCK einMutex           ' Schutz der Variablen 'param'
  IF param > 0 THEN
      param += 1
5    RETURN param               ' ACHTUNG: Fehler!
  END IF
  MUTEXUNLOCK einMutex
  RETURN 0
END FUNCTION
```

Ist *param* größer als 0, dann wird die Funktion beim ersten **RETURN** verlassen. Zu diesem Zeitpunkt ist der Mutex *einMutex* jedoch noch gesperrt. Spätestens wenn die Funktion das nächste Mal aufgerufen wird, blockiert der Thread, weil er auf eine Mutex-Freigabe wartet, die nie eintreten wird. Der Mutex muss also vor dem **RETURN** entsperrt werden.

Quelltext 8.4: Mutex-Fehler bei zu früher Freigabe

```

FUNCTION fehlerhafteFunktion AS INTEGER
  MUTEXLOCK einMutex           ' Schutz der Variablen 'param'
  IF param > 0 THEN
    param += 1
    MUTEXUNLOCK einMutex
  5   RETURN param               ' ACHTUNG: moeglicherweise auch schlecht
  END IF
  MUTEXUNLOCK einMutex
  RETURN 0
10 END FUNCTION

```

Das Problem im oben stehenden Code besteht in der Gefahr, dass *param* genau in dem Moment, bevor es zurückgegeben wird, von einem anderen Thread verändert wird. Um dies auszuschließen, können Sie eine Zwischenspeicherung verwenden:

Quelltext 8.5: Korrekte Mutex-Verwendung

```

FUNCTION richtigeFunktion AS INTEGER
  MUTEXLOCK einMutex           ' Schutz der Variablen 'param'
  IF param > 0 THEN
    param += 1
    5   DIM AS INTEGER temp = param
    MUTEXUNLOCK einMutex       ' Mutex korrekt entsperren
    RETURN temp
  END IF
  MUTEXUNLOCK einMutex
10 RETURN 0
END FUNCTION

```

Die Zwischenspeicherung birgt wiederum eine andere Gefahr. Nehmen wir an, Sie fragen die Anzahl bestimmter Elemente ab, um sie anschließend nacheinander zu bearbeiten. Wenn sich die Anzahl aber währenddessen ändert, greifen Sie möglicherweise auf Elemente zu, die bereits nicht mehr existieren.

Quelltext 8.6: Mutex-Fehler bei Zwischenspeicherung

```

MUTEXLOCK einMutex           ' Schutz des Arrays 'param'
DIM AS INTEGER temp = UBOUND(param)
MUTEXUNLOCK einMutex
  FOR i AS INTEGER = 1 TO temp   ' ACHTUNG: schlecht, wenn sich die Anzahl
5   ' bearbeite die Parameter      der Parameter inzwischen veraendert hat
  NEXT

```

Wie Sie sehen, hält die Arbeit mit Multithreading einige Fallen bereit, die zu unerwarteten Fehlern führen können. Der Umgang mit Threads will also gut durchdacht sein. Wenn Sie jedoch sorgfältig damit umgehen, bieten sie eine hervorragende Möglichkeit zur Optimierung Ihres Programms.

9. Externe Bibliotheken

9.1. Einbindung externer Bibliotheken

Für FreeBASIC stehen eine große Zahl externer Bibliotheken zur Verfügung. Bei einer Bibliothek handelt es sich, vereinfacht gesagt, um eine Sammlung verschiedener Prozeduren, welche nach der Einbindung im eigenen Programm eingesetzt werden können. Das Spektrum reicht von 2D- und 3D-Grafikroutinen über Sound, GUI-Entwicklung, mathematische und physikalische Berechnungen bis hin zum Zugriff auf die API-Funktionen des Betriebssystems.

Die Anzahl der Bibliotheken, die als FreeBASIC-Quellcode vorliegen, ist eher gering. Prinzipiell können jedoch alle Bibliotheken, die mit C verwendet werden können, auch in FreeBASIC eingesetzt werden. Dazu werden (neben der Bibliothek selbst) die Header-Dateien (im Folgenden nur *Header* genannt) benötigt, in denen sich die in der Bibliothek benötigten Deklarationen befinden. Unter FreeBASIC hat es sich eingebürgert, Header mit der Extension *.bi* zu versehen.

Externe Dateien – in diesem Fall die Header – werden mit dem Metabefehl **#INCLUDE** eingebunden. Die Stelle, an der dieser Befehl auftaucht, wird dann durch den Inhalt der eingebundenen Datei ersetzt. Um zu verhindern, dass eine Datei mehrfach eingebunden wird – z. B. weil man zwei Dateien benötigt, die beide wiederum eine gemeinsame dritte Datei einbinden – kann **#INCLUDE ONCE** verwendet werden.

```
#INCLUDE ONCE "externeLib.bi"
```

Achtung:

Achten Sie darauf, dass Dateinamen unter Linux und anderen unixartigen Betriebssystemen *case sensitive* behandelt werden. Unter Windows ist die Groß-/Kleinschreibung nicht relevant. Im Zuge der Portierbarkeit sollten Sie aber auch hier auf die korrekte Groß-/Kleinschreibung achten.

Wenn Sie eine fremde Bibliothek (oder überhaupt fremden Code) verwenden wollen, erkundigen Sie sich vorab, unter welchen Bedingungen Sie ihn bei sich einbinden können. Manche Bibliotheken dürfen nur zu nicht-kommerziellen Zwecken eingesetzt werden, andere erfordern, evtl. auch nur für den kommerziellen Einsatz, eine Lizenzgebühr.

Wieder andere dürfen für alle Zwecke frei verwendet werden. Sollten Unklarheiten darüber bestehen, ob und unter welchen Bedingungen Sie einen bestimmten Code verwenden dürfen oder nicht, fragen Sie lieber beim Entwickler nach.

9.2. Sound und Musik

Als Sound-Bibliothek werden in FreeBASIC häufig *FMOD*³ oder *BASS*⁴ eingesetzt. Bei beiden handelt es sich um proprietäre Software, die für den kommerziellen Einsatz kostenpflichtig ist. Welcher der beiden Bibliotheken Sie den Vorzug geben, ist größtenteils Geschmacksache; dieses Kapitel geht näher auf die Verwendung von BASS ein.

Ein Vorteil von BASS ist die hervorragende Unterstützung von Trackermodulen. Zunächst einmal soll aber die Wiedergabe einer ogg-Datei vorgestellt werden. Dazu gibt es zwei grundsätzliche Konzepte: Zum einen kann die Datei mit **BASS_SampleLoad** komplett in den Speicher geladen und immer bei Bedarf abgespielt werden, zum anderen besteht auch die Möglichkeit, die Datei mit **BASS_StreamCreateFile** als Stream zu öffnen. Komplett im Speicher gelagerte Dateien bieten sich vor allem bei kurzen Soundeffekten an, die öfter abgespielt werden sollen, während größere Musikdateien – etwa die Hintergrundmusik – besser gestreamt werden.

Achtung:

Mit BASS und FMOD können auch MP3-Dateien problemlos abgespielt werden, jedoch muss für jede Hard- und Software, die MP3 verwendet, eine Lizenzgebühr entrichtet werden. Diese fällt unabhängig von der Lizenz der Sound-Bibliothek an. Nicht nur aus diesem Grund wird vom Einsatz von MP3 abgeraten und eher Ogg Vorbis empfohlen.

9.2.1. Wiedergabe von Ogg-Vorbis-Dateien

Bevor die Soundwiedergabe genutzt werden kann, wird BASS mit dem Befehl **BASS_Init** eingerichtet. [Quelltext 9.1](#) initialisiert BASS auf dem Standardgerät mit 44100 Hz. Dann werden die Channels bereitgestellt. Alle Soundverarbeitungen laufen über solche Channels, egal ob es sich um einen Stream, einen in den Speicher geladenen Sound oder ein Trackermodul handelt; auch Aufnahmen werden über Channels umgesetzt. Mit **BASS_StreamCreateFile** wird ein Stream für die Hintergrundmusik erstellt, der mit **BASS_ChannelPlay** gestartet und erst bei Programmende gestoppt wird. Mit

³ <http://fmod.org>

⁴ <http://www.un4seen.com>

BASS_SampleLoad laden wir außerdem einen Soundeffekt in den Speicher, der später bei jedem Tastendruck wiedergegeben wird. Am Ende kann der Speicher mit **BASS_StreamFree** bzw. **BASS_SampleFree** wieder freigegeben werden, der Befehl **BASS_Free** gibt aber automatisch allen von BASS belegten Speicher frei.

Quelltext 9.1: Sound- und Musikausgabe

```
#INCLUDE "bass24.bi"

' BASS initialisieren
BASS_Init -1, 44100, 0, 0, 0

5
' Soundeffekt und Hintergrundmusik laden
DIM AS STRING musikname = "hintergrundmusik.ogg"
DIM AS HSTREAM musik = BASS_StreamCreateFile(0, STRPTR(musikname), 0, 0, 0)
DIM AS STRING soundname = "soundeffekt.ogg"
10 DIM AS HSAMPLE sound = BASS_SampleLoad(0, STRPTR(soundname), 0, 0, 16, 0)
DIM soundchannel AS HCHANNEL = BASS_SampleGetChannel(sound, 0)
BASS_ChannelPlay musik, 0 ' Musik abspielen

DO
15 IF GETKEY = 27 THEN EXIT DO ' ESC-Taste
BASS_ChannelPlay soundchannel, 0 ' Soundeffekt abspielen
LOOP
BASS_Free ' wichtig, um den belegten Speicher wieder freizugeben
```

9.2.2. Wiedergabe von Trackermoduldateien

Trackermodulformate speichern eine Auswahl von digitalen Samples, die als Instrumente des Musikstücks dienen. Die Tonfolgen werden durch eine Trackliste bestimmt, in der festgelegt wird, wann welche Noten auf welchem Instrument zu spielen sind. Geschickt aufgebaute Moduldateien können Musikstücke auf sehr platzsparende Weise speichern. Außerdem bieten sie eine einfache Möglichkeit, bestimmte Stellen im Stück direkt anzukommen.

Moduldateien müssen mit BASS ein wenig anders behandelt werden als z. B. ogg-Streams. Zunächst einmal werden sie mit **BASS_MusicLoad** geladen und mit **BASS_MusicFree** freigegeben (Quelltext 9.2 nutzt stattdessen wieder **BASS_Free**). Des Weiteren gibt es zusätzliche Steuerungselemente wie **BASS_ChannelSetPosition** zur Ansteuerung der Wiedergabestelle und **BASS_ChannelSetAttribute** für die Einstellung bestimmter Eigenschaften (Lautstärke, Balance und viele mehr). Die Wiedergabe erfolgt wie gehabt über **BASS_ChannelPlay**.

Quelltext 9.2: Wiedergabe einer Trackermoduldatei

```

#INCLUDE "bass24.bi"

' BASS initialisieren
BASS_Init(-1, 44100, 0, 0, 0)

5 ' Channel einrichten; die Wiedergabe startet am Ende neu (BASS_SAMPLE_LOOP)
  DIM AS STRING musikname = "moduldatei.mod"
  DIM AS HMUSIC musik = BASS_MusicLoad(0, STRPTR(musikname), 0, 0, _
    BASS_MUSIC_POSRESET OR BASS_SAMPLE_LOOP, 0)
10 BASS_ChannelSetPosition musik, 0, BASS_POS_BYTE ' Startposition
    BASS_ChannelSetAttribute musik, BASS_ATTRIB_VOL, 1 ' volle Lautstaerke
    BASS_ChannelSetAttribute musik, BASS_ATTRIB_PAN, 0 ' Balance in der Mitte

' Musik abspielen
15 BASS_ChannelPlay musik, 0

GETKEY
BASS_Free ' wichtig, um den belegten Speicher wieder freizugeben

```

Die Soundbibliothek bietet noch viel mehr Möglichkeiten als das einfache Abspielen von Musikdateien. Dazu sollten Sie jedoch die Referenz der Bibliothek zu Rate ziehen, in der sämtliche zur Verfügung stehenden Befehle aufgeführt sind.

9.3. Rotation und Skalierung

Für die Rotation und die Skalierung von Grafiken existiert keine vorgefertigte FreeBASIC-Routine. Es gibt jedoch eine Reihe von Bibliotheken, die dafür Methoden zur Verfügung stellen. Eine direkte Umsetzung wurde von Joshy (D. J. Peters) mit der Prozedur **MultiPut**⁵ bereitgestellt.

```
MultiPut Zielpuffer, x, y, Quellpuffer, xSkalierung, ySkalierung, Transparenz
```

Alle Parameter bis auf *Quellpuffer* sind optional. *Zielpuffer* gibt den Puffer an, in den das neue Bild geschrieben wird; wird er ausgelassen oder der Wert 0 angegeben, dann erfolgt eine Ausgabe auf den Bildschirm. *x* und *y* bestimmen die Stelle, an welcher der Mittelpunkt (also nicht die linke obere Ecke!) des Bildes plziert wird. *xSkalierung* und *ySkalierung* erlauben eine Streckung oder Stauchung in x- bzw. y-Richtung, wobei Werte größer als 1 in die angegebene Richtung vergrößern und Werte zwischen 0 und 1 die Grafik verkleinern. *Transparenz* schließlich erlaubt es, die Transparenzfarbe zu nutzen – wird hier ein Wert ungleich 0 angegeben, dann erscheinen Flächen in dieser Farbe transparent.

⁵ <http://www.freebasic.net/forum/viewtopic.php?t=2441>

Quelltext 9.3 demonstriert den Umgang mit **MultiPut**. Eine Mausebewegung nach links oder rechts dreht das Bild, mit dem Mausewheel kann es vergrößert oder verkleinert werden. Selbstverständlich muss eine Datei, die den Befehl enthält, auf Ihrem Arbeitsplatz zur Verfügung stehen (siehe Fußnote 5).

Quelltext 9.3: Rotation und Skalierung

```

5  #INCLUDE "multiput.bi"
   SCREENRES 400, 300, 32
   DIM AS SINGLE rotation = 0, groesse = 1
   DIM AS INTEGER mausX, mausY, mausR, mausB, rad
10  ' Bild erstellen
   DIM AS ANY PTR bild = IMAGECREATE(50, 50, 0)
   LINE bild, (49, 49)-(0, 20), &h0000FF, BF
   LINE bild, -(24, 0), &h0000FF
   LINE bild, -(49, 20), &h0000FF
   PAINT bild, (24, 10), &h00FFFF, &h0000FF

   ' Hauptprogramm
   DO
15  SETMOUSE 200, 150, 0, -1
      SLEEP 10
      GETMOUSE mausX, mausY, mausR, mausB
      IF mausX < 0 THEN CONTINUE DO          ' ausserhalb des Fensters
      rotation += (mausX-200)/50
      groesse += (rad-mausR)/10
20  IF groesse < 0.1 THEN groesse = 0.1    ' minimale Groesse
      IF groesse > 4 THEN groesse = 4      ' maximale Groesse
      rad = mausR
      SCREENLOCK
25  CLS
      MultiPut , 200, 150, bild, groesse, groesse, rotation
      SCREENUNLOCK
      LOOP UNTIL mausB > 0 OR INKEY <> ""

30  ' aufräumen und beenden
      IMAGEDESTROY bild

```

9.4. Weitere Bibliotheken

Bei der Vielzahl der zur Verfügung stehenden Bibliotheken kann natürlich nicht annähernd auf alles Interessante eingegangen werden. Ein paar Aspekte sollen noch kurz genannt werden:

- Für aufwändigere Berechnungen bietet sich der Einbau einer mathematischen Bibliothek an. *FBMath*⁶ bietet eine Vielzahl mathematischer Operationen wie

⁶ <http://freebasic-portal.de/downloads/bibliotheken/fbmath-165.html>

Wahrscheinlichkeitsrechnung, Matrizen, lineare und nicht lineare Gleichungen, Integralrechnung, Zufallszahlen und vieles mehr. Auch wenn sich ein großer Teil der Funktionen in einem Spiel wohl nicht direkt verwenden lässt, handelt es sich bei vielen Fragestellungen um rein mathematische Probleme. Die Bibliothek kann beispielsweise bei der Berechnung komplizierterer Flugbahnen oder der Erfolgswahrscheinlichkeit bei Strategiespielen mit Glücksfaktor verwendet werden.

- Die *FreeBASIC Extended Library*⁷ erweitert den FreeBASIC-Befehlssatz um einige hilfreiche Elemente. Speziell für die Spieleprogrammierung interessant ist der Umgang mit verschiedenen Grafikformaten, animierten Sprites und Vektorklassen.
- Wer tiefer in die Spieleprogrammierung einsteigen möchte, dem sei die Bibliothek *Allegro*⁸ ans Herz gelegt. Vor allem, wenn Sie einmal eine andere Programmiersprache verwenden wollen, die keine eigene Grafikbibliothek einsetzt, bietet Allegro eine einfache und komfortable Lösung.
- Für Spiele über ein Netzwerk bietet sich die Funktionensammlung *TSNE* an. Diese wird in [Kapitel 13.2](#) genauer vorgestellt.

⁷ <http://freebasic-portal.de/downloads/bibliotheken/freebasic-extended-library-133.html>

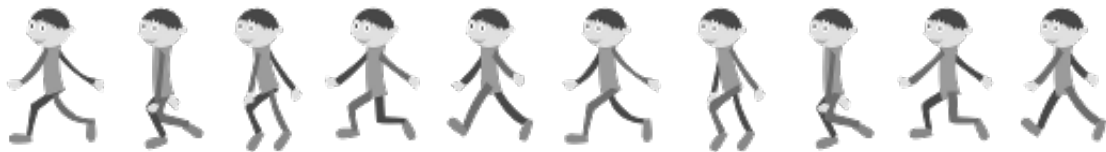
⁸ <http://alleg.sourceforge.net>

10. Erweiterte Grafikprogrammierung

Kapitel 4 hat bereits einige Grundlagen der Grafikprogrammierung behandelt. Die dort eingebundenen Grafiken waren jedoch rein statisch. Das folgende Kapitel geht auf bewegliche Grafiken ein. Zuerst wird die Animation einer Spielfigur erstellt. Anschließend erfahren Sie, wie Sie einen scrollbaren Hintergrund einbauen können und wie sich Parallax Scrolling, also die unterschiedlich schnelle Bewegung mehrerer Darstellungsebenen, umsetzen lässt.

10.1. Animation

Um Spielfiguren oder Gegenstände zu animieren, gibt es eine ganz einfache Möglichkeit: Erstellen Sie mehrere Bilder, die nacheinander abgespielt den gewünschten Bewegungsablauf ergeben. Um beispielsweise eine Figur über den Bildschirm laufen zu lassen, werden die Arme und Beine in unterschiedlichen Positionen gezeichnet.



„Walk cycle boy“ von shokunin (<http://openclipart.org>)

Sie können die einzelnen Bilder der Sequenz nun in verschiedenen Dateien speichern und diese der Reihe nach anzeigen. Alternativ dazu kann aber auch der ganze Bewegungsablauf in einer einzigen Datei gespeichert werden. Angezeigt wird dann immer nur der Ausschnitt mit der im Augenblick benötigten Stellung. Am einfachsten ist diese Methode umzusetzen, wenn jedes Teilbild dieselbe Größe besitzt.

Quelltext 10.1 geht von einer Sequenz von zehn Bildern aus, die von 0 bis 9 nummeriert werden. Nach dem neunten Bild muss wieder bei 0 angefangen werden, wofür der Operator **MOD** verwendet wird. Sie können den Quellcode – mit angepassten Werten – zum Testen Ihrer Animationen verwenden. *breite* und *hoehe* geben die Maße der einzelnen Bilder an, *anzahl* die Zahl der zur Sequenz gehörenden Bilder. Wenn Sie die Animation langsamer ablaufen lassen, können Sie kleine Unstimmigkeiten besser lokalisieren. Bei höherer

Geschwindigkeit erhalten Sie dagegen einen besseren Gesamteindruck und erkennen, ob die Animation insgesamt „rund“ läuft.

Quelltext 10.1: Animierte Spielfigur

```
5  DIM AS INTEGER breite = 80, hoehe = 150, anzahl = 10, nr = 0
    SCREENRES breite, hoehe, 32
    DIM AS ANY PTR animation = IMAGECREATE(breite*anzahl, hoehe)
    BLOAD "walking.bmp", animation
10 DO
    PUT (0, 0), animation, (nr*breite, 0)-STEP(breite-1, hoehe-1), PSET
    SLEEP 100
    nr = (nr+1) MOD anzahl      ' naechstes Bild
    LOOP UNTIL INKEY <> ""      ' Ende bei Tastendruck
    IMAGEDESTROY animation
```

Sinnvoll wäre es, noch ein Bild für ein stehendes Männchen einzufügen. Solange die Spielfigur steht, wird dann dieses Bild angezeigt. Wenn die Figur bewegt wird, folgen der Reihe nach die Bilder der Animation, solange bis die Figur wieder steht. Je nach Spiel können auch Animationen für weitere Tätigkeiten wie Springen, Klettern, Kämpfen oder ähnliches zum Einsatz kommen.

10.2. Scrolling

Viele Spiele, insbesondere aus dem Genre Jump 'n' Run oder Shoot 'em Up, besitzen sehr große Level, von denen immer nur ein kleiner Ausschnitt sichtbar ist. Um bei diesen Spielen durch die Ansicht zu scrollen, können Sie ähnlich vorgehen wie im vorigen Abschnitt bei der Animation: Das gesamte Spielfeld wird in einem Grafikpuffer zwischengespeichert und nur der Ausschnitt, der für den Spieler sichtbar sein soll, in das Fenster kopiert.

Quelltext 10.2 demonstriert ein mausgesteuertes Scrolling. Das Programm erstellt ein Fenster, in dem Sie einen kleinen Ausschnitt des geladenen Bildes sehen können. Bewegen Sie die Maus an den Rand des Fensters, um den angezeigten Bildausschnitt zu scrollen. *posX* und *posY* merken sich die momentane Position des Anzeigebereichs. Der Code ist allgemein gehalten, sodass Sie damit jedes 32bit-BMP laden können. Die für das korrekte Erstellen des Bildpuffers erforderlichen Maße des Bildes werden zu Beginn des Programms automatisch eingelesen.

Quelltext 10.2: Scrolling

```

SCREENRES 200, 200, 32
SETMOUSE 100, 100,, 1
DIM bild AS ANY PTR, datei AS STRING = "meinBild.bmp"
DIM AS INTEGER breit, hoch, dateinr
5 DIM AS INTEGER posX = 0, posY = 0, mausX, mausY, mausButton

' Bildgroesse (Breite und Hoehe) auslesen
dateinr = FREEFILE
OPEN datei FOR BINARY AS #dateinr
10 GET #dateinr, 19, breit
GET #dateinr, 23, hoch
CLOSE #dateinr

' Bildpuffer erstellen und Bild laden
15 bild = IMAGECREATE(breit, hoch)
BLOAD datei, bild
PUT (0, 0), bild, (posX, posY)-STEP(199, 199), PSET
DO
  GETMOUSE mausX, mausY,, mausButton
  ' Verschiebungen durchfuehren
  IF mausX < 10 AND posX > 0 THEN ' nach links schieben
    posX -= 1 : PUT (0, 0), bild, (posX, posY)-STEP(199, 199), PSET
  END IF
  IF mausY < 10 AND posY > 0 THEN ' nach oben schieben
25   posY -= 1 : PUT (0, 0), bild, (posX, posY)-STEP(199, 199), PSET
  END IF
  IF mausX > 190 AND posX < breit-200 THEN ' nach rechts schieben
    posX += 1 : PUT (0, 0), bild, (posX, posY)-STEP(199, 199), PSET
  END IF
  IF mausY > 190 AND posY < hoch-200 THEN ' nach unten schieben
30   posY += 1 : PUT (0, 0), bild, (posX, posY)-STEP(199, 199), PSET
  END IF
  SLEEP 1
LOOP UNTIL mausButton > 0
35 IMAGEDESTROY bild

```

Bei einem Shoot 'em Up wird für gewöhnlich automatisch gescrollt, wobei der Spieler keinen oder nur geringen Einfluss auf die Scrollgeschwindigkeit hat. In einem Spiel mit frei beweglicher Spielfigur dagegen hängt die Scrollbewegung von der Position der Figur ab: Wenn sie sich dem Fensterrand nähert, wird die Ansicht weitergescrollt.

Bei Änderungen auf dem Spielfeld müssen Sie sich entscheiden, ob diese direkt in das Fenster oder zuerst in den Grafikpuffer gezeichnet werden. Dauerhafte Änderungen speichern Sie besser erst im Grafikpuffer, während Sie Spielfiguren, die ihre Position ständig ändern, auch sofort in das Fenster zeichnen können. Bei einer hohen Anzahl an Spielfiguren – z. B. bei einem Jump 'n' Run mit vielen Gegnern – bietet es sich sowieso an, nur die aktuell sichtbaren Figuren zu zeichnen, um Rechenzeit zu sparen.

10.3. Parallax-Scrolling

Unter *Parallax-Scrolling* versteht man den Effekt, wenn sich mehrere Scroll-Ebenen unterschiedlich schnell bewegen. Gegenstände, die sich weiter entfernt befinden, werden dabei langsamer bewegt als weiter vorn befindliche Objekte. So wird ein räumlicher Effekt erzielt.

Die Ebenen werden zuerst nach Entfernung sortiert. Je weiter hinten sich eine Ebene befindet, desto langsamer bewegt sie sich auch. Zuerst wird die hinterste Ebene gezeichnet, also diejenige, die sich am langsamsten (oder gar nicht) bewegt, etwa der Himmel und die Wolken. Darüber kommt die nächste Ebene, z. B. die Bäume am Horizont. Alle Stellen, an denen der Himmel zu sehen sein soll, werden transparent gezeichnet, so dass die dahinter stehende Ebene sichtbar bleibt. Am einfachsten geht das über die Transparenzfarbe, im 32bit-Farbmodus die Farbe &hFF00FF.

Sie können prinzipiell beliebig viele Ebenen verwenden. Bedenken Sie jedoch, dass das Zeichnen jeder Ebene Rechenzeit benötigt und damit das Programm ausbremst. Leider müssen Sie auch die Ebenen zeichnen, die sich seit der letzten Bildschirmaktualisierung gar nicht geändert haben, da die alten Ausschnitte der höheren Ebenen wieder überschrieben werden müssen. Dafür ist es aber meist nicht erforderlich, dass die Ebenen das komplette Fenster ausfüllen. Zeichnen Sie dann für jede Ebene nur den Bereich, den Sie benötigen.

Die drei Scroll-Ebenen in [Quelltext 10.3](#) nehmen der Einfachheit halber den kompletten Bildschirm ein. Sie sind so geplant, dass der rechte Rand jeder Ebene nahtlos in deren linken Rand übergehen kann. Zu Beginn wird in jeder Ebene ein Stück des linken Randes an das rechte Ende der Ebene kopiert. Zu diesem Zweck muss der Bildspeicher größer als die Ebenenbreite gewählt werden.

Das Array *schnell()* regelt die Geschwindigkeit der Ebenen – Ebene 1 wird bei jedem Durchgang weiterbewegt, Ebene 2 nur jedes dritte Mal und Ebene 3 nur jedes achte Mal. *posX()* speichert, wie weit jede Ebene gescrollt wurde und welcher Ausschnitt dementsprechend kopiert werden muss.

Auf der [Projektseite](#) finden Sie ein leicht modifiziertes Beispiel, bei dem die Ebenen nur einen Teil des Bildschirms ausmachen, inklusive der Grafiken für drei Ebenen. Variieren Sie die Werte im Array *schnell()*, um einen Eindruck von der Funktionsweise des Programms zu bekommen.

Quelltext 10.3: Parallax Scrolling

```

DIM AS INTEGER bildbreite = 400, bildhoehe = 300
SCREENRES bildbreite, bildhoehe, 32
DIM AS ANY PTR ebene(1 TO 3) ' Bildspeicher der Ebenen
DIM AS INTEGER zaehler = 0, i
5 DIM AS INTEGER b(1 TO 3) = { 900, 900, 900 } ' Breite der Ebenen
DIM AS INTEGER posX(1 TO 3) = { 0, 0, 0 } ' Position innerhalb der Ebenen
DIM AS INTEGER schnell(1 TO 3) = { 8, 3, 1 } ' Geschwindigkeit der Ebenen

' Ebenen laden
10 FOR i = 1 TO 3
    ebene(i) = IMAGECREATE(b(i) + bildbreite, bildhoehe)
    BLOAD "ebene" & i & ".bmp", ebene(i)
    ' Stueck des linken Randes an den rechten Rand anhaengen
    PUT ebene(i), (b(i),0), ebene(i), (0,0)-STEP(bildbreite-1,bildhoehe-1), PSET
15 NEXT

DO
    ' Ebenen zeichnen
    SCREENLOCK
    20 FOR i = 1 TO 3
        PUT (0,0), ebene(i), (posX(i),0)-STEP(bildbreite-1,bildhoehe-1), TRANS
    NEXT
    SCREENUNLOCK
    ' neue Position der Ebenen bestimmen
    25 zaehler += 1
    FOR i = 1 TO 3
        IF (zaehler MOD schnell(i)) = 0 THEN posX(i) += 1
        ' Wenn das Ende der Ebene erreicht ist, an den Anfang zurueckkehren
        IF posX(i) >= b(i) THEN posX(i) -= b(i)
    30 NEXT
    SLEEP 1
    LOOP UNTIL INKEY <> ""

' Speicher bereinigen
35 FOR i = 1 TO 3
    IMAGEDESTROY ebene(i)
NEXT

```

Noch eine Anmerkung für alle Freunde der Modulo-Rechnung: Die Zeilen 27-29 im obigen Code hätten auch zu einer Zeile zusammengefasst werden können:

```
IF (zaehler MOD schnell(i)) = 0 THEN posX(i) = (posX(i) + 1) MOD b(i)
```

Kapitel 12.2 verwendet ebenfalls die Modulo-Rechnung, um die Programmierung kompakt zu gestalten. Dort wird auch etwas genauer auf die Funktionsweise dieser Rechenmethode eingegangen.

11. Kollisionskontrolle

Bisher wurde ein rasterförmig aufgeteiltes Spielfeld verwendet, was die Überprüfung erleichtert, welche Bereiche begehbar sind. Dafür ist die Größe und Position der Objekte stark eingeschränkt. Wenn Sie frei platzierbare Objekte verwenden wollen, müssen Sie sich etwas anderes ausdenken. Wir werden zuerst die Interaktion der Spielfigur mit dem Hintergrund und dann ihre Interaktion mit anderen, insbesondere beweglichen Objekten behandeln.

11.1. Kontrolle anhand von Farbwerten

Was Sie über den Hintergrund bereits ohne weiteren Aufwand wissen, ist der Farbwert der einzelnen Pixel. Es liegt also nahe, diese Farbwerte für die Kollisionskontrolle zu nutzen. Im einfachsten Fall sind alle Stellen, welche für die Figur ein Hindernis darstellen, in derselben Farbe gezeichnet. [Quelltext 11.1](#) skizziert das Verfahren anhand folgender Situation: *sx* und *sy* geben, diesmal pixelgenau, die linke obere Ecke der Spielfigur an, *sb* und *sh* liefern Breite und Höhe der Figur. Es wird getestet, ob sich die Figur auf festem Boden befindet, also ob eines der Pixel unter ihr den Farbwert eines Hindernisses besitzt.

Quelltext 11.1: Kollisionskontrolle mit einem Farbwert

```
5  DIM AS INTEGER hindernisGefunden = 0           ' 0 als false-Wert
    FOR i AS INTEGER = sx TO sx + sb - 1         ' ueber die ganze Breite der Figur
        IF POINT(i, sy+sh) = hindernisfarbe THEN ' hier ist ein Hindernis
            hindernisGefunden = -1               ' -1 als true-Wert
        EXIT FOR                                  ' keine weitere Kontrolle noetig
    END IF
NEXT
```

Wenn *hindernisGefunden* nach der Schleife noch den Wert 0 besitzt, befindet sich kein Boden unter der Figur; sie wird also (sofern sie nicht fliegen kann) nach unten fallen. Ähnlich kann natürlich auch überprüft werden, ob sich die Figur nach rechts oder links bewegen kann.

Wahrscheinlich wollen Sie die Hindernisse aber nicht alle in derselben Farbe zeichnen. Es würde zwar reichen, jedes Hindernis mit einem Rand in dieser Farbe zu umgeben, aber auch das schränkt den Spielraum in der grafischen Gestaltung ziemlich ein. Möglich

wäre noch die Verwendung eines bestimmten Farbspektrums oder umgekehrt der Einsatz einer einheitlichen Farbe für alle begehbaren Stellen – alle anderen Farben sind dann ein Hindernis.

11.2. Erstellung einer Maske

Eine Maske erledigt genau dieselbe Arbeit wie die Verwendung einer Hindernis-Farbe, nur mit dem Unterschied, dass dem Spieler die Informationen der Maske nicht direkt angezeigt werden. Sie kann also völlig unabhängig von der angezeigten Farbe verwendet werden. Damit stellt auch der Einbau verborgener Gänge (als Hindernis gezeichnet, in der Maske aber nicht als Hindernis markiert) und unsichtbarer Hindernisse (nur in der Maske als Hindernis gesetzt) kein Problem dar. Einziger Nachteil ist der erhöhte Speicherverbrauch, da Sie neben den Farbwerten für jedes Pixel noch weitere Informationen speichern müssen.

Wenn Sie beim Hintergrund keine raffinierten Effekte mit Teiltransparenz erzielen wollen, können Sie auch den Alphakanal für die Maske nutzen. Farben werden als RGB-Werte (rot-grün-blau) gespeichert; jeder der einzelnen Farbkanäle kann einen Wert von 0 bis 255 annehmen, belegt also 8 Bit. Bei 32bit-Variablen und einer Belegung von dreimal 8 Bit bleiben also noch 8 Bit frei, die für den Alphakanal genutzt werden können. Dabei handelt es sich um den Grad der Transparenz des Pixels. Der Wert 255 bedeutet, dass das Pixel nicht transparent ist, den Hintergrund also völlig überdeckt. Der Wert 0 bedeutet völlige Transparenz; der Hintergrund wird nicht verändert. Bei einem Wert von 127 mischen sich Hintergrund und Vordergrund in genau gleichem Maße.

Wenn Sie den Alphakanal für die Maske nutzen wollen, müssen Sie deswegen in Ihrem Spiel nicht auf Transparenz verzichten. Zum einen können Sie das Schlüsselwort **TRANS** und die dazu gehörige Transparenzfarbe verwenden. Auf der anderen Seite können Sie den Alphakanal für die Spielfigur ganz normal nutzen; nur der Hintergrund braucht ihn als Maskenfarbe.

In [Quelltext 11.2](#) werden drei Objekte mit unterschiedlichem Alphakanal gezeichnet. Dadurch erhalten sie unterschiedliche Eigenschaften. Das erste Objekt ist nur Verzierung und wird als Hintergrundgrafik behandelt. Das zweite ist zwar begehbar, die Spielfigur wird aber von ihm verdeckt. Das dritte Objekt ist ein Hindernis, durch das sich die Figur nicht bewegen kann. Die Figur wird mit den Pfeiltasten nach links und rechts gesteuert.

Die vorgeschlagene Zeichenroutine mag vielleicht auf den ersten Blick etwas umständlich wirken. Nach der Speicherung des Hintergrunds wird erst die Figur gezeichnet, dann noch einmal der Hintergrund darüber gesetzt. Die Stellen mit dem Alphawert 255 überdecken damit die Figur; Stellen mit dem Alphawert 0 verschwinden hinter der Figur. Als Sonderfall dient der Alphawert 127, der Hindernisse kennzeichnet und daher nicht betreten werden

kann. Das Hindernis wird der Einfachheit halber nur an einem Pixel getestet; natürlich muss der Test eigentlich über die ganze Höhe der Figur hinweg durchgeführt werden.

Quelltext 11.2: Kollisionskontrolle mit Alphakanal

```

SCREENRES 400, 50, 32
PAINT (0, 0), 0           ' Hintergrund durchsichtig setzen
DIM AS INTEGER sx = 50    ' Startposition der Figur
DIM AS STRING taste      ' zur spaeteren Tastaturabfrage

5  ' Figur in einen Puffer zeichnen; zweiter Puffer zur Hintergrundspeicherung
DIM AS ANY PTR figur, hintergrund
DIM AS UINTEGER farbe = &hFF907010 ' Farbe der Figur
figur = IMAGECREATE(30, 50, 0) ' durchsichtiger Hintergrund
10 hintergrund = IMAGECREATE(30, 50)
CIRCLE figur, (15, 5), 5, farbe,,, F ' Kopf
LINE figur, (15, 10)-(15, 40), farbe ' Rumpf
LINE figur, ( 0, 10)-(15, 20), farbe ' Arme
LINE figur,           -(29, 10), farbe
15 LINE figur, ( 0, 49)-(15, 40), farbe ' Beine
LINE figur,           -(29, 49), farbe

' Objekte zeichnen
LINE (100, 5)-(150, 49), &h006080F0, BF ' Alphawert 00: Hintergrund
20 LINE (200, 5)-(250, 49), &hFF6080F0, BF ' Alphawert FF: Vordergrund
LINE (300, 5)-(350, 49), &h7F6080F0, BF ' Alphawert 7F: Hindernis
LINE (0, 5)-(5, 49), &h7F6080F0, BF ' linke Wand zur Begrenzung

GET (sx, 0)-STEP(29, 49), hintergrund
25 PUT (sx, 0), figur, ALPHA
PUT (sx, 0), hintergrund, ALPHA
DO
taste = INKEY
IF taste = CHR(255, 75) AND _ ' Pfeiltaste links
30 ( POINT(sx-1, 20) AND &hFF000000) <> &h7F000000 THEN
PUT (sx, 0), hintergrund, PSET ' Hintergrund wiederherstellen
sx -= 1 ' Bewegung nach links
GET (sx, 0)-STEP(29, 49), hintergrund ' neuen Hintergrund speichern
PUT (sx, 0), figur, ALPHA ' Figur zeichnen
35 PUT (sx, 0), hintergrund, ALPHA ' Hintergrund darueber zeichnen

ELSEIF taste = CHR(255, 77) AND _ ' Pfeiltaste rechts
( POINT(sx+30, 20) AND &hFF000000) <> &h7F000000 THEN
PUT (sx, 0), hintergrund, PSET ' Hintergrund wiederherstellen
40 sx += 1 ' Bewegung nach rechts
GET (sx, 0)-STEP(29, 49), hintergrund ' neuen Hintergrund speichern
PUT (sx, 0), figur, ALPHA ' Figur zeichnen
PUT (sx, 0), hintergrund, ALPHA ' Hintergrund darueber zeichnen
END IF
45 SLEEP 10
LOOP UNTIL taste = CHR(27)

```

11.3. Kontrolle anhand der Objektposition

Des Weiteren besteht die Möglichkeit, von allen im Spiel befindlichen Objekten die Position zu speichern und diese für die Kollisionskontrolle zu nutzen. Dazu müssen Sie alle Objekte durchlaufen um zu prüfen, ob eines davon mit der Figur kollidiert.

Sinnvoll ist dieses Verfahren vor allem bei Objekten, deren Position sich verändern kann, also insbesondere bei gegnerischen Spielfiguren. Eine pixelgenaue Überprüfung der Objekte ist zwar möglich, aber recht aufwändig. Stattdessen werden wir die Objekte durch einfache geometrische Formen annähern. Hier bieten sich vor allem der Kreis und das Rechteck an.

Kreisförmige Objekte sind am einfachsten zu handhaben. Zwei Kreise überschneiden sich, wenn ihr Abstand kleiner ist als die Summe der beiden Radien. Für die Abstandsberechnung dient der Satz des Pythagoras:

```
IF (pos2x - pos1x)^2 + (pos2y - pos1y)^2 < (radius1 + radius2)^2 THEN
  ' beide Kreise ueberschneiden sich
END IF
```

pos1x und *pos1y* stellen dabei den Mittelpunkt des ersten Objekts dar, *pos2x* und *pos2y* den Mittelpunkt des zweiten Objekts. Da beim Zeichnen mit **PUT** nicht der Mittelpunkt, sondern die linke obere Ecke benötigt wird, müssen die Zahlenwerte vorher entsprechend umgewandelt werden.

Auch für die rechteckähnlichen Objekte wird hier der Einfachheit halber der Mittelpunkt des Rechtecks verwendet. Im Gegensatz zu der Annäherung durch Kreise benötigen wir nicht nur einen Radius, sondern zwei Seitenangaben, die mit *laenge1* und *breite1* bzw. mit *laenge2* und *breite2* angesprochen werden. Die Abstände in x- und y-Richtung werden einzeln durch die Funktion **ABS** überprüft.

```
IF ABS(pos2x - pos1x) < (breite1 + breite2)/2 AND _
  ABS(pos2y - pos1y) < (hoehe1 + hoehe2)/2 THEN
  ' beide Rechtecke ueberschneiden sich
END IF
```

Beide Versionen funktionieren allerdings nur zur Überprüfung, ob sich die Objekte berühren bzw. überschneiden. Wenn es entscheidend ist, auf welcher Seite die Objekte kollidieren, müssen die einzelnen Seiten getrennt untersucht werden.

Teil III.

Die Gegenspieler

12. Einfache Computersteuerung

Die Programmierung einer guten AI⁹ ist ein anspruchsvolles Thema, das nur schwer im Rahmen eines Buches erschöpfend behandelt werden kann. In diesem Kapitel wird lediglich auf einige Grundlagen für die Implementierung eines Computergegners eingegangen. Der Begriff AI bezieht sich hier also auf jede Art von Computersteuerung, ungeachtet derer „Intelligenz“. Neben der Umsetzung einfacher computergesteuerter Bewegungen für das bisher behandelte Labyrinth-Spiel wird auch der Minimax-Algorithmus für rundenbasierte Zwei-Personen-Spiele vorgestellt.

Das, was üblicherweise für Spiele entwickelt wird, ist natürlich keine „echte“ Intelligenz. Es geht vielmehr darum, ein Verhalten zu simulieren, das intelligent *wirkt*. Je nach Spiel kann sich das unterschiedlich auswirken: Ein Verfolger in einem Arcade-Spiel sucht den kürzesten Weg zu seinem Ziel, in einem Schachspiel bestimmt der Computer den besten nächsten Zug, und so weiter. Intelligenz kann aber auch bedeuten, dass die Computerreaktion nicht so leicht vorausberechnet werden kann, wodurch es für den menschlichen Spieler schwerer wird, im Vorfeld zu reagieren.

Wenn Sie sich an die Umsetzung einer AI wagen wollen, spielen Sie verschiedene Möglichkeiten durch und variieren Sie sie. Wenn das erzielte Ergebnis auf Sie ausreichend intelligent wirkt, dann sind Sie auf einem guten Weg.

12.1. Vordefinierte Computerzüge

In manchen Fällen ist das Verhalten der computergesteuerten Gegner fest vorgegeben. Denken Sie in diesem Zusammenhang beispielsweise an ein Jump'n'Run-Spiel. Das übliche Verhalten für die gegnerische Figur wird sein, solange geradeaus zu gehen, bis sie an eine Wand stößt, und dann umzukehren. Je nach Typ kehrt die Figur auch an einem Abgrund um, oder sie läuft weiter und fällt anschließend, wenn sie keinen Boden mehr unter den Füßen hat, nach unten. Möglicherweise bewegt sie sich auch nur auf einer bestimmten Art von Untergrund und kehrt um, wenn der Untergrund des folgenden Feldes nicht stimmt. Wenn die Figur dagegen fliegen kann, ist ihr der Untergrund egal und sie fällt auch keine Abgründe hinunter.

⁹ „*artificial intelligence*“, zu deutsch „*künstliche Intelligenz*“, auch KI

Die Berechnung des nächsten Schritts könnte also folgendermaßen ablaufen:

1. Wenn die Figur nicht fliegen kann, überprüfe den Untergrund.
 - a) Der Untergrund ist für die Figur tödlich – sie stirbt.
 - b) Unter der Figur befindet sich kein Boden – sie fällt nach unten. Der Zug ist beendet.
 - c) Der Untergrund ist begehbar.
2. Überprüfe, ob das nächste Feld begehbar ist. Sonst drehe um; der Zug ist beendet.
3. Überprüfe gegebenenfalls den Untergrund des nächsten Feldes. Wenn er für die Figur nicht begehbar ist, drehe um; der Zug ist beendet.
4. Sofern der Zug noch nicht beendet wurde, gehe ein Feld nach vorn.

Wie eine Computerfigur gesteuert wird, muss nicht immer offensichtlich sein. Sie können den Spielfeldern auch versteckte Attribute beifügen, die es der Figur z. B. verbieten, bestimmte Felder zu betreten. Besonders bei sonst sehr freizügigen Bewegungsmöglichkeiten ist das ein wirksames Mittel, um den Aktionskreis der Figur einzuschränken.

12.2. Zufällige Computerzüge

Um die Wanderung durch das in den vorigen Kapiteln aufgebaute Labyrinth etwas spannender zu machen, werden nun computergesteuerte Gegner eingeführt, deren Aufgabe es ist, die Spielfigur abzufangen. Wie intelligent sich ein Computergegner verhält, entscheidet maßgeblich über den Schwierigkeitsgrad des Levels. Die einfachste Art der Computersteuerung, die sich allerdings in keinsten Weise als „intelligent“ bezeichnen lässt, ist ein Zufallsgenerator. Immer wenn sich der Gegner bewegen soll, wird per Zufall die Richtung bestimmt. Dazu dient der Befehl **RND**, der eine Zufallszahl von (einschließlich) 0 bis (ausschließlich) 1 erzeugt. Damit bei jedem Programmdurchlauf andere Zahlen verwendet werden, muss der Zufallsgenerator zuerst mit **RANDOMIZE** initialisiert werden.

Da die rein zufällige Bewegung sehr unkoordiniert wirkt, sollte sie noch etwas modifiziert werden. In [Quelltext 12.1](#) versucht die computergesteuerte Figur, geradeaus zu gehen. Ist das nicht möglich, dann sucht der Computer mittels Zufallszahl aus, ob sie nach rechts oder links weitergehen soll (sofern der Weg in beide Richtungen frei ist). Nur wenn es keine andere Möglichkeit gibt, dreht die Figur um und geht zurück. Die Figur macht damit natürlich keine Jagd auf den Spieler; der Spieler muss lediglich aufpassen, nicht versehentlich mit der Computerfigur zu kollidieren.

Das Programm verwendet die Variablen *cx*, *cy* und *cr* für die x- bzw. y-Position sowie die momentane Laufrichtung der Computerfigur (für mehrere Computergegner bietet sich hier natürlich ein Array an). Zur Speicherung der Felddaten wird *felddata(x, y)* verwendet. Es handelt sich wieder nur um einen Codeausschnitt – das Einlesen der Felddaten ist im Code nicht enthalten.

Zur besseren Lesbarkeit werden die **ENUM**-Konstanten *Links=0*, *Oben=1*, *Rechts=2* und *Unten=3* verwendet. Wird zur aktuellen Richtung 1 addiert (z. B. *Links+1 = 1 = Oben*), dann resultiert daraus der Wert, der beim Rechtsabbiegen entsteht. Nur beim Abbiegen von *Unten* nach *Links* ist das Ergebnis noch falsch – bei einer Berechnung **MOD** 4 stimmt es aber wieder: $(Unten+1) \bmod 4 = 0 = Links$. Analog dazu kann zum Umdrehen 2 und zum Linksabbiegen 3 addiert werden (jeweils **MOD** 4). Dass zum Linksabbiegen 3 addiert wird und nicht 1 subtrahiert, liegt daran, dass $(Links-1) \bmod 4 = -1$ ist, also nicht das gewünschte Ergebnis 3 zurückgibt. Die Subtraktion eignet sich hier also nicht.

Quelltext 12.1: Zufallsgesteuerte Computerbewegung

```

DO
  ' Vorwaartszug testen
  SELECT CASE cr
    CASE Links
      IF felddata(cx-1, cy) = Leer THEN bewege cx, cy, Links : CONTINUE DO
    CASE Oben
      IF felddata(cx, cy-1) = Leer THEN bewege cx, cy, Oben : CONTINUE DO
    CASE Rechts
      IF felddata(cx+1, cy) = Leer THEN bewege cx, cy, Rechts : CONTINUE DO
    CASE Unten
      IF felddata(cx, cy+1) = Leer THEN bewege cx, cy, Unten : CONTINUE DO
  END SELECT

  ' verfuegbare Richtungen suchen
  DIM AS INTEGER linksabbiegen = 0, rechtsabbiegen = 0
  SELECT CASE cr
    CASE Links
      IF felddata(cx, cy+1) = Leer THEN linksabbiegen = -1
      IF felddata(cx, cy-1) = Leer THEN rechtsabbiegen = -1
    CASE Oben
      IF felddata(cx-1, cy) = Leer THEN linksabbiegen = -1
      IF felddata(cx+1, cy) = Leer THEN rechtsabbiegen = -1
    CASE Rechts
      IF felddata(cx, cy-1) = Leer THEN linksabbiegen = -1
      IF felddata(cx, cy+1) = Leer THEN rechtsabbiegen = -1
    CASE Unten
      IF felddata(cx+1, cy) = Leer THEN linksabbiegen = -1
      IF felddata(cx-1, cy) = Leer THEN rechtsabbiegen = -1
  END SELECT
  IF linksabbiegen AND (NOT rechtsabbiegen) THEN ' nur linksabbiegen
    cr = (cr+3) MOD 4
  ELSEIF (NOT linksabbiegen) AND rechtsabbiegen THEN ' nur rechtsabbiegen
    cr = (cr+1) MOD 4

```

```

35  ELSEIF linksabbiegen AND rechtsabbiegen THEN          ' beides moeglich
    ' zu 50% links , zu 50% rechts abbiegen
    IF RND < .5 THEN cr = (cr+3) MOD 4 ELSE cr = (cr+1) MOD 4
    ELSE                                                    ' keines moeglich
        cr = (cr+2) MOD 4
    END IF
40  LOOP UNTIL INKEY <> ""

```

Der Codeausschnitt ist für sich allein zwar nicht lauffähig, aber Sie können einmal versuchen, daraus ein komplettes Programm zu stricken.¹⁰

Ob eine Computerfigur, die sich nach dem Zufallsprinzip bewegt, dem Spieler gefährlich werden kann, hängt weitgehend von der Levelgröße und der Anzahl der Computergegner ab. Mit einer ausreichenden Gegnerzahl kann die Angelegenheit schon recht anspruchsvoll werden.

12.3. Verfolgungsjagd

Wesentlich schwieriger wird es für den Spieler, wenn sich die Computergegner gezielt zur Spielerfigur hin bewegen. Dies lässt sich relativ einfach bewerkstelligen. Eine mögliche Vorgehensweise: Der Computer prüft, wie groß die vertikale und wie groß die horizontale Distanz zur Spielerfigur ist. Dann versucht er, die größere der beiden Distanzen zu verringern. Geht das nicht, wird versucht, die kleinere Distanz zu reduzieren. Falls auch das nicht möglich ist, entscheidet er sich zufällig für eine der möglichen Laufrichtungen.

Die Distanzen lassen sich mit der Betragsfunktion **ABS** schnell vergleichen. Im Folgenden stehen die Variablen *sx* und *sy* wieder für die Koordinaten der Spielerfigur und die Variablen *cx* und *cy* für die Computerposition.

```

IF ABS(cx - sx) > ABS(cy - sy) THEN
    ' versuche, die horizontale Entfernung zu verringern
ELSE
    ' versuche, die vertikale Entfernung zu verringern
END IF

```

Bei einer solchen Verfolgungstaktik wird das Entkommen schon ziemlich schwer, besonders dann, wenn man es mit mehr als einem Gegner zu tun bekommt. Um die Taktik etwas abzuschwächen, können Sie eine der folgenden Methoden ausprobieren:

- Der Computer versucht nur, die größere der beiden Entfernungen zu verringern. Geht das nicht, findet eine zufällige Auswahl der Laufrichtung statt.
- Der Spieler wird nur innerhalb eines bestimmten Radius verfolgt, also wenn die Entfernung des Computerspielers nicht zu groß oder nicht zu klein ist (oder beides).

¹⁰ Auf der [Projektseite](#) finden Sie ein vollständiges Beispiel.

- Der Computer verfolgt den Spieler nur, wenn er ihn „sieht“, also z. B. wenn sich der Spieler in derselben horizontalen oder vertikalen Reihe befindet und keine „Wände“ dazwischen liegen.

Es gibt noch zahlreiche andere Möglichkeiten; die hier genannten sollen nur zur Anregung dienen.

Selbstverständlich kann das Spiel auch anders herum laufen – warum lassen Sie den Spieler nicht zur Abwechslung einmal eine flüchtende Computerfigur jagen?

12.4. Computereinsatz in Strategiespielen

Ziel der Computerberechnung ist es, einen „Zug“ zu machen, der die für den Computer bestmögliche Situation herbeiführt. Im vorigen Kapitel war diese beste Situation schnell gefunden: gut ist, wenn sich die Computerfigur möglichst nahe beim Spieler befindet. Ein guter Zug ist also einer, bei dem sich die Computerfigur an den Spieler annähert.

In Strategiespielen, etwa bei Schach oder Dame, ist die Berechnung der „bestmöglichen Situation“ schon schwieriger. Für diese doch sehr bekannten Spiele haben sich aber bereits Berechnungsroutinen eingebürgert, die in Abhängigkeit von der Anzahl der sich auf dem Feld befindenden Steine und deren Position einen Spielwert errechnen. Der Computer müsste nun eigentlich nur alle möglichen Züge durchprobieren und denjenigen ausführen, bei dem der Spielwert für ihn am besten ausfällt.

Ganz so leicht ist es natürlich nicht. Manchmal würde eine Position für sich genommen einen recht hohen Spielwert ergeben, im nächsten Zug kann der Gegner jedoch vielleicht zu einem entscheidenden Schlag ausholen und das Ergebnis der Berechnung komplett umwerfen. Es ist also nötig, einige Schritte im Voraus zu planen.

Der Einfachheit halber wird im Folgenden von einem hohen Spielwert gesprochen, wenn die Situation für den Computer besonders günstig ist. Ein niedriger Spielwert ist dementsprechend für den Computer ungünstig und für den Gegner (z. B. den menschlichen Spieler) günstig. Das Vorausberechnen mehrerer Züge läuft dann in der Praxis folgendermaßen ab: Der Computer führt der Reihe nach alle möglichen Züge durch und berechnet für jeden dieser Züge, welche gegnerische Antwort darauf die beste ist. Mit anderen Worten: Für jeden Computerzug werden nun sämtliche möglichen Gegenzüge durchgeführt und der beste davon ausgewählt – also derjenige, bei dem der Spielwert möglichst niedrig ausfällt. Derjenige Computerzug, bei dem auch der beste Gegenzug zum höchsten Spielwert führt, wird letztendlich durchgeführt.

Selbstverständlich kann zur Berechnung der Gegenzüge auch wieder die Antwort des Computers mit einberechnet werden und so weiter. Je tiefer die Berechnung der abwechselnden Züge verschachtelt ist, desto mehr Züge werden vorausberechnet und

desto besser sollte dementsprechend das Ergebnis sein. Das Problem ist natürlich, dass die Anzahl der zu berechnenden Züge mit der Berechnungstiefe sehr schnell wächst. Wenn es z. B. bei einem Spiel in jeder Situation je zehn mögliche Züge gibt, dann müssen dazu insgesamt 100 gegnerische Züge berechnet werden. Auf diese gibt es wiederum 1000 Antworten usw. Der Berechnungsaufwand wächst damit exponentiell zur Berechnungstiefe.

Die vorgestellte Methode wird als *Minimax-Suchverfahren* bezeichnet. Bei [Quelltext 12.2](#) handelt es sich genau genommen um den *NegaMax-Algorithmus*, einer Variante der Minimax-Suche. Der Computer nimmt abwechselnd die Rolle der beiden Spieler an und berechnet dafür jeweils den maximalen Spielwert. Bei der Rückgabe wird das Vorzeichen dieses Wertes umgedreht, da ja ein hoher Wert für den einen Spieler ein entsprechend niedriger Wert für den anderen Spieler darstellt und umgekehrt.

Quelltext 12.2: Minimax-Suche

```

FUNCTION minimax (resttiefe AS INTEGER) AS INTEGER
  DIM AS INTEGER ermittelt, zugWert
  ermittelt = -2^31 ' auf den kleinstmöglichen Wert stellen
  FOR i AS INTEGER = 1 TO anzahlZuege ' alle möglichen Zuege durchspielen
5    fuehreZugAus(i)
    IF restTiefe <= 1 THEN ' wurde die maximale Berechnungstiefe
      zugWert = berechneSpielwert ' erreicht, dann bewerte die Situation
    ELSE
      zugWert = -minimax(restTiefe-1) ' rekursiv den Gegenzug berechnen
10    END IF
    nimmZugZurueck(i) ' Ausgangssituation wiederherstellen
    IF zugWert > ermittelt THEN
      ermittelt = zugWert ' besten Zug merken
    END IF
15  NEXT
  RETURN ermittelt
END FUNCTION

```

Ein verbessertes Verfahren liefert die *Alpha-Beta-Suche*. Hierbei wird frühzeitig entschieden, an welchen Stellen keine weitere Suche nötig ist, weil das bestmögliche Ergebnis auf jeden Fall schlechter sein wird als das Ergebnis eines anderen, schon berechneten Zuges. Durch geschickte Optimierung der Alpha-Beta-Suche kann die Berechnungsdauer wesentlich verkürzt werden. Auf diesen Algorithmus wird hier jedoch nicht weiter eingegangen.

Der Minimax-Algorithmus eignet sich im Prinzip für alle Zwei-Personen-Spiele, bei denen abwechselnd gezogen wird. Die Berechnung des Spielwertes sollte nicht zu aufwändig ausfallen, da sie ganz am Ende des Algorithmus steht und daher sehr häufig aufgerufen wird. Wenn sie zu viel Rechenzeit verbraucht, bremst das die Zugberechnung sehr stark aus. An dieser Stelle soll auch noch einmal darauf hingewiesen werden, dass die Wahl des richtigen Datentyps entscheidende Vorteile bringt. Berechnungen mit **INTEGER**-Variablen

werden wesentlich schneller durchgeführt als z. B. mit **SHORT**-Variablen, während Sie bei zeitkritischen Berechnungen auf den Datentyp **DOUBLE** nach Möglichkeit komplett verzichten sollten.

Am einfachsten können Sie die Spieltaktik des Computers überprüfen, indem Sie zwei Computerspieler gegeneinander antreten lassen. Achten Sie darauf, ob alle durchgeführten Züge erlaubt sind (sonst liegt ein Programmierfehler vor), ob sie sinnvoll sind und ob eine gewisse Variation der Züge stattfindet – das Abspulen der immer gleichen Züge wirkt nicht intelligent, sondern einprogrammiert.

Auf der nächsten Seite wird das Minimax-Verfahren anhand eines einfachen Nim-Spiels umgesetzt. Das Spielfeld besteht aus mehreren Reihen mit Streichhölzern, wobei in der ersten Reihe ein Streichholz, in der zweiten Reihe zwei Streichhölzer usw. liegen. Nacheinander darf sich jeder der beiden Spieler eine Reihe aussuchen und aus dieser beliebig viele Streichhölzer entfernen; mindestens jedoch muss eines entfernt werden. Derjenige Spieler, der das letzte Streichholz nimmt, verliert.

Vorteil des Spiels ist die relativ geringe Zahl an möglichen Zügen. Bei vier Startreihen gibt es für den ersten Zug 10 Möglichkeiten, diese Zahl schrumpft jedoch bei voranschreitendem Spiel immer weiter. Als Spielwert kommen nur die Zahlen 1 (Spielgewinn), -1 (Spielverlust) und 0 (keine Aussage möglich) zum Tragen. [Quelltext 12.3](#) berechnet das Spiel bereits bis zum Ende durch, womit sogar nur noch zwei Spielwerte auftreten können. Damit nicht bei jedem Programmdurchlauf immer dieselben Züge abgespult werden, entscheidet der Computer bei gleich guten Alternativen nach dem Zufallsprinzip. Das eingeführte **UDT** *zugtyp* dient dazu, nicht nur den besten Spielwert, sondern auch den dazugehörigen Zug (also die ausgewählte Reihe sowie die daraus entfernte Anzahl an Streichhölzern) schnell und einfach speichern und zurückgeben zu können.

Wenn Sie mit einer höheren Anzahl an Startreihen spielen wollen, sollten Sie die Rechentiefe verringern, da die ersten Züge ansonsten sehr viel Zeit in Anspruch nehmen. Der Computer spielt dann allerdings nicht mehr von Anfang an perfekt.¹¹

¹¹ Es sei darauf hingewiesen, dass die verwendete Bewertungsroutine nicht optimal ist – eine Siegstellung ließe sich bereits viel früher erkennen. Das würde dem Leser jedoch die Lösungsstrategie verraten und ihm das Vergnügen nehmen, sie selbst zu entdecken.

Quelltext 12.3: Nim-Spiel

```

TYPE zugtyp
  AS INTEGER reihe, anz, wert
END TYPE
DIM SHARED AS INTEGER reihenzahl = 4, rechentiefe = 10, reihe(1 TO reihenzahl)
5 FOR r AS INTEGER = 1 TO reihenzahl : reihe(r) = r : NEXT
DIM spieler AS INTEGER = 1, zug AS zugtyp
RANDOMIZE TIMER

FUNCTION berechneSpielwert AS INTEGER
10 FOR i AS INTEGER = 1 TO reihenzahl
  IF reihe(i) <> 0 THEN RETURN 0 ' nicht das letzte Holz genommen
NEXT
RETURN -1 ' Verlust des Spiels
END FUNCTION
15 FUNCTION minimax(resttiefe AS INTEGER) AS zugtyp
  DIM AS INTEGER besteReihe, besteAnz, besterWert, zugWert, gefunden
  besterWert = -2^31 ' auf den kleinstmoeglichen Wert stellen
  FOR r AS INTEGER = 1 TO reihenzahl ' moegliche Zuege: aus jeder Reihe jede
    FOR anz AS INTEGER = 1 TO reihe(r) ' moegliche Streichholzzahl entfernen
20 reihe(r) -= anz ' Zug durchfuehren
    IF restTiefe <= 1 THEN ' wurde die maximale Berechnungstiefe
      zugWert = berechneSpielwert ' erreicht, dann bewerte die Situation
    ELSE
      zugWert = -minimax(restTiefe - 1).wert ' sonst rekursiv weiterrechnen
25 END IF
    reihe(r) += anz ' Ausgangssituation wiederherstellen
    IF zugWert > besterWert THEN
      besterWert = zugWert : besteReihe = r : besteAnz = anz : gefunden = 1
    ELSEIF zugWert = besterWert THEN
30 gefunden += 1
    IF INT(RND*gefunden) = 0 THEN
      besterWert = zugWert : besteReihe = r : besteAnz = anz
    END IF
  END IF
35 NEXT anz, r
  IF besterWert = -2^31 THEN RETURN TYPE(0, 0, 1) ' Spieler hat gewonnen
  RETURN TYPE(besteReihe, besteAnz, besterWert)
END FUNCTION

40 DO ' Hauptprogramm
  CLS
  FOR r AS INTEGER = 1 TO reihenzahl
    FOR anz AS INTEGER = 1 TO reihe(r) : PRINT "| "; : NEXT
    PRINT : PRINT
45 NEXT
  IF GETKEY = 27 THEN EXIT DO
  spieler = -spieler
  zug = minimax(rechentiefe)
  IF zug.reihe <> 0 THEN reihe(zug.reihe) -= zug.anz
50 LOOP UNTIL zug.reihe = 0

```

13. Menschliche Gegner

13.1. Mehrere Spieler an einem Computer

Um mehrere Spieler am selben Computer spielen zu lassen, gibt es verschiedene Ansätze. Bei Spielen wie Dame, Schach o. ä. ist das kein Problem: beide Spieler können gemeinsam vor dem Computer sitzen und abwechselnd ihren Zug eingeben. Das funktioniert grundsätzlich bei allen rundenbasierten Spielen, bei denen die Züge offen einsehbar sind. Sollen dem Gegner die Züge verborgen bleiben, dann müssen die Spieler abwechselnd den Platz vor dem Computer räumen und dürfen während des gegnerischen Zuges nicht auf den Bildschirm sehen. Diese Technik wird *Hot Seat* (zu deutsch „Schleudersitz“ oder „heißer Stuhl“) genannt. Damit die Wartezeiten nicht zu lang werden, sollten Sie eine Begrenzung der Rundenzeit in Erwägung ziehen. Außerdem darf auch beim Platzwechsel keiner der beiden Spieler den Bildschirm des jeweils anderen zu Gesicht bekommen.

Wenn alle Spieler zeitgleich agieren sollen, benötigt jeder eine eigene Steuerungsmöglichkeit. Das kann über mehrere Joysticks oder verschiedene Tasten geschehen. Auch hier ist es einfacher, wenn alle Beteiligten dasselbe Bild zu sehen bekommen, jedoch kann auch jeder Spieler mithilfe eines *Split Screens* seine eigene Spielansicht erhalten.

Das soll soweit als Denkanstoß genügen. Um einiges anspruchsvoller ist das Spiel über Netzwerk, auf das im nächsten Kapitel eingegangen wird.

13.2. Netzwerkspiel

Um ein Netzwerk aufzubauen, werden zwei „Arten“ von Programm benötigt. Zunächst einmal ist ein Server erforderlich, der die Spieldaten verwaltet. Die einzelnen Spieler nehmen über einen Client Kontakt mit dem Server auf, erhalten von diesem die Informationen über die aktuelle Spielsituation und teilen ihm die durchgeführten Züge mit.

Da Netzwerkprogrammierung keine ganz so einfache Sache ist, greifen wir auf die Bibliothek TSNE¹² von ThePuppetMaster (Martin Wiemann) zurück.

```
#INCLUDE "TSNE_V3.bi"
```

¹² TCP Socket Networking Eventing; <http://www.freebasic-portal.de/projekte/11>

Dazu muss sich die Datei *TSNE_V3.bi* im selben Verzeichnis befinden wie Ihre Quellcode-Datei. Wenn Sie mehrere externe Bibliotheken verwenden, bietet es sich an, diese alle in einen gemeinsamen Unterordner zu packen. Dann muss für das **#INCLUDE** der Pfad entsprechend angepasst werden.

Beim Einrichten des Servers werden einige Callback-Funktionen übergeben, die das Verhalten bei bestimmten Ereignissen – Verbindung oder Trennung eines Client, Datenempfang – regelt. Die Ereignisüberwachung wird immer dann, wenn ein solches Ereignis eintritt, automatisch aufgerufen.

Ein sehr einfach gehaltener Server, der nicht viel macht außer auf Anfragen zu reagieren, könnte folgendermaßen aussehen:

Quelltext 13.1: Einfacher Netzwerk-Server

```

5  #INCLUDE "TSNE_V3.bi"

SUB Verbunden(BYVAL id AS INTEGER)
    PRINT "Client " & id & " wurde verbunden."
END SUB

10 SUB Getrennt(BYVAL id AS INTEGER)
    PRINT "Client " & id & " wurde getrennt."
END SUB

SUB NeueDaten(BYVAL id AS INTEGER, BYREF daten AS STRING)
15    PRINT "Neue Daten von Client " & id & ": " & daten
END SUB

SUB Verbindung(BYVAL id AS INTEGER, BYVAL req AS Socket, BYVAL ipa AS STRING)
    ' wenn versucht wird, eine neue Verbindung herzustellen
15    PRINT "Ein Client versucht zu verbinden"
    DIM TSNEID AS INTEGER
    DIM returnIPA AS STRING
    ' Akzeptieren der Verbindung
    TSNE_Create_Accept req, TSNEID, returnIPA, @Getrennt, @Verbunden, @NeueDaten
20    END SUB
SUB Verbindungsabbruch(BYVAL id AS INTEGER, BYVAL ipa AS STRING)
    ' Verhalten, wenn die Verbindung abgelehnt wird
    PRINT "Verbindung wurde abgelehnt"
25    END SUB

' Server erstellen
DIM bv AS INTEGER, Server AS INTEGER
bv = TSNE_Create_Server(Server, 1100, 10, @Verbindung, @Verbindungsabbruch)
IF bv <> TSNE_Const_NoError THEN PRINT "Server-Fehler: " & BV : END
30 DO : SLEEP 1 : LOOP UNTIL INKEY <> "" ' jetzt wird gewartet ...
    TSNE_Disconnect(Server) ' Server beenden
    TSNE_WaitClose(Server) ' warten bis Server beendet ist

```

Die eigentliche Erstellung des Servers geschieht mit *TSNE_Create_Server*. In der Variable *Server* wird eine interne Identifikationsnummer gespeichert, um den Server später ansprechen zu können. Danach folgt die gewünschte Port-Nummer, hier 1100. Verwenden

Sie möglichst keine reservierte Portnummer – wichtig ist aber vor allem, dass der Port frei ist. Im Internet finden Sie ausführliche Listen über die reservierten Portnummern.

Achtung:

Die Portnummern bis 1023 sind generell reserviert und sollten daher nicht für eigene Programme verwendet werden. Unter Linux kann ein Port in diesem Bereich nur mit root-Rechten gebunden werden. Sie sollten aber generell nur Portnummern ab 1024 wählen.

Die nächste Zahl 10 gibt die Anzahl der Verbindungsanfragen an, die der Server gleichzeitig bearbeiten kann. Diese Zahl ist nur relevant, wenn sehr viele Verbindungsanfragen zur gleichen Zeit eingehen. Es handelt sich hier nicht um die Anzahl der Verbindungen, die gleichzeitig *bestehen* können – diese ist nur durch die Größe eines **UINTEGERs** (maximal 65535 gleichzeitig bestehende Verbindungen) beschränkt.

Die Prozeduren *Verbinden* und *Verbindungsabbruch* können im Prinzip so übernommen werden. Die darin enthaltenen **PRINT**-Statusrückmeldungen sind nicht unbedingt erforderlich, helfen aber herauszufinden, ob der Verbindungsaufbau wie geplant funktioniert. Auch die Prozeduren *Verbunden* und *Getrennt* dienen zunächst nur der Verbindungskontrolle. Interessant ist zunächst vor allem die Prozedur *NeueDaten*. Sobald ein Client Daten an den Server schickt, landen sie in dieser Prozedur. In der Regel wird der Server die Daten dann auswerten und darauf reagieren.

Dazu ein passender Client, der eine kurze Nachricht an den Server schickt:

Quelltext 13.2: Einfacher Netzwerk-Client

```
#INCLUDE "TSNE_V3.bi"

SUB Verbunden(BYVAL id AS UINTEGER)
    PRINT "Die Verbindung wurde hergestellt."
5  END SUB
SUB Getrennt(BYVAL id AS UINTEGER)
    PRINT "Die Verbindung wurde getrennt."
    END SUB
10 SUB NeueDaten(BYVAL id AS UINTEGER, BYREF daten AS STRING)
    PRINT "Neue Daten von Client " & id & ": " & daten
    END SUB

' Client erstellen
DIM bv AS INTEGER, Client AS UINTEGER, host AS STRING = "127.0.0.1"
15 bv = TSNE_Create_Client(Client, host, 1100, @Getrennt, @Verbunden, @NeueDaten)
IF bv <> TSNE_Const_NoError THEN PRINT "Server-Fehler: " & BV : END
bv = TSNE_WaitConnected(Client) ' auf Verbindung warten
IF bv <> TSNE_Const_NoError THEN PRINT "Fehler: " & TSNE_GetGURUCode(BV) : END
TSNE_Data_Send 1, "Hallo Server!" ' Daten senden
20 TSNE_Disconnect(Client) ' Client beenden
    TSNE_WaitClose(Client) ' warten bis Client beendet ist
```

Das beim Server verwendete Prinzip mit den Callback-Funktionen bleibt gleich, jedoch muss der Client die Adresse und den Port kennen, unter denen der Server angesprochen werden kann. Wenn Server und Client auf demselben Computer laufen, lautet die Adresse 127.0.0.1; ansonsten müssen Sie die Adresse zuerst ermitteln (und darauf achten, dass keine Firewall den Zugriff verhindert).

Die Daten werden mithilfe des Befehls `TSNE_Data_Send` verschickt:

<code>TSNE_Data_Send ID_des_Empfaengers , zu_sendende_Nachricht</code>
--

Die Funktion sendet sowohl vom Server an den Client als auch vom Client an den Server oder vom Client an einen anderen Client. Sie kann zwar nur Strings verschicken, doch das lässt sich sehr vielseitig einsetzen. Wenn Sie verschiedene Arten von Daten verschicken wollen (z. B. Spielzug, Statusabfrage, Chatmeldung ...), können Sie die Nachricht mit einem „Identifikationsbyte“ beginnen, das genauere Angaben über die Art der Daten macht. Je nach dessen Inhalt wird dann der Rest der Daten ausgewertet. Zum Versenden von Zahlenwerten bietet sich unter Umständen auch **MKI** oder ein ähnlicher Befehl an. Damit wird eine Integerzahl in einen String umgewandelt, der ein Speicheräquivalent der Zahl darstellt. Zur Rückumwandlung dient der Befehl **CVI**.

Achtung:

Wenn mehrere Datensendungen innerhalb kurzer Zeit stattfinden, kann es vorkommen, dass die Daten nicht einzeln geschickt werden, sondern zusammen in einem Paket. Das bedeutet, dass die gesendeten Strings einfach aneinander gehängt werden. In diesem Fall können Sie die Strings mit einem speziellen Trennzeichen beenden und die empfangenen Daten bei diesem Trennzeichen wieder per Hand auseinandernehmen.

Spielzüge sollten immer über den Server laufen, der sie auf Gültigkeit überprüft. Gehen Sie nicht davon aus, dass das, was ein Client sendet, schon korrekt sein wird! Es ist leicht, gefälschte Daten zu verschicken; solange aber der Server immer weiß, welcher Spieler welche Züge machen darf, kann er ungültige Angaben leicht herausfiltern.

Wenn Sie wollen, können Sie sich auch die Erweiterungsbibliothek `TSNE_Play`¹³ ansehen. Sie stellt unter anderem eine einfache Möglichkeit zur Verfügung, einen passwortgeschützten Server zu erstellen. Außerdem wurde darin eine allgemein gehaltene Übertragung von Spielzügen eingebaut, die es ermöglicht, drei **DOUBLE**-Zahlen (etwa die drei Koordinatenrichtungen eines Zuges) und ein **UINTINTEGER** zu übermitteln. `TSNE_Play` greift dabei komplett auf `TSNE` zurück.

¹³ <http://www.freebasic-portal.de/projekte/36>

Teil IV.

Anhang

A. ASCII-Zeichentabelle

Zeichencodierung in der Konsole

0	26	52	78	104	130	156	182	208	234
1	27	53	79	105	131	157	183	209	235
2	28	54	80	106	132	158	184	210	236
3	29	55	81	107	133	159	185	211	237
4	30	56	82	108	134	160	186	212	238
5	31	57	83	109	135	161	187	213	239
6	32	58	84	110	136	162	188	214	240
7	33	59	85	111	137	163	189	215	241
8	34	60	86	112	138	164	190	216	242
9	35	61	87	113	139	165	191	217	243
10	36	62	88	114	140	166	192	218	244
11	37	63	89	115	141	167	193	219	245
12	38	64	90	116	142	168	194	220	246
13	39	65	91	117	143	169	195	221	247
14	40	66	92	118	144	170	196	222	248
15	41	67	93	119	145	171	197	223	249
16	42	68	94	120	146	172	198	224	250
17	43	69	95	121	147	173	199	225	251
18	44	70	96	122	148	174	200	226	252
19	45	71	97	123	149	175	201	227	253
20	46	72	98	124	150	176	202	228	254
21	47	73	99	125	151	177	203	229	255
22	48	74	100	126	152	178	204	230	
23	49	75	101	127	153	179	205	231	
24	50	76	102	128	154	180	206	232	
25	51	77	103	129	155	181	207	233	

Zeichencodierung in einem Grafikfenster

0	26	52	78	104	130	156	182	208	234
1	27	53	79	105	131	157	183	209	235
2	28	54	80	106	132	158	184	210	236
3	29	55	81	107	133	159	185	211	237
4	30	56	82	108	134	160	186	212	238
5	31	57	83	109	135	161	187	213	239
6	32	58	84	110	136	162	188	214	240
7	33	59	85	111	137	163	189	215	241
8	34	60	86	112	138	164	190	216	242
9	35	61	87	113	139	165	191	217	243
10	36	62	88	114	140	166	192	218	244
11	37	63	89	115	141	167	193	219	245
12	38	64	90	116	142	168	194	220	246
13	39	65	91	117	143	169	195	221	247
14	40	66	92	118	144	170	196	222	248
15	41	67	93	119	145	171	197	223	249
16	42	68	94	120	146	172	198	224	250
17	43	69	95	121	147	173	199	225	251
18	44	70	96	122	148	174	200	226	252
19	45	71	97	123	149	175	201	227	253
20	46	72	98	124	150	176	202	228	254
21	47	73	99	125	151	177	203	229	255
22	48	74	100	126	152	178	204	230	
23	49	75	101	127	153	179	205	231	
24	50	76	102	128	154	180	206	232	
25	51	77	103	129	155	181	207	233	

B. MULTIKEY-Scancodes

Die nachfolgende Liste enthält die Scancodes, die z. B. bei MULTIKEY verwendet werden. Sie entsprechen den DOS-Scancodes und funktionieren auch plattformübergreifend. Sie finden diese Liste ebenfalls in der Datei *fbgfx.bi*, die sich in Ihrem inc-Verzeichnis befinden sollte.

Die Liste führt die definierte Konstante sowie den dazu gehörigen Hexadezimal- und den Dezimalwert auf.

Konstante	hex	dez	Konstante	hex	dez	Konstante	hex	dez
SC_ESCAPE	01	1	SC_A	1E	30	SC_F1	3B	59
SC_1	02	2	SC_S	1F	31	SC_F2	3C	60
SC_2	03	3	SC_D	20	32	SC_F3	3D	61
SC_3	04	4	SC_F	21	33	SC_F4	3E	62
SC_4	05	5	SC_G	22	34	SC_F5	3F	63
SC_5	06	6	SC_H	23	35	SC_F6	40	64
SC_6	07	7	SC_J	24	36	SC_F7	41	65
SC_7	08	8	SC_K	25	37	SC_F8	42	66
SC_8	09	9	SC_L	26	38	SC_F9	43	67
SC_9	0A	10	SC_SEMICOLON	27	39	SC_F10	44	68
SC_0	0B	11	SC_QUOTE	28	40	SC_NUMLOCK	45	69
SC_MINUS	0C	12	SC_TILDE	29	41	SC_SCROLLLOCK	46	70
SC_EQUALS	0D	13	SC_LSHIFT	2A	42	SC_HOME	47	71
SC_BACKSPACE	0E	14	SC_BACKSLASH	2B	43	SC_UP	48	72
SC_TAB	0F	15	SC_Z	2C	44	SC_PAGEUP	49	73
SC_Q	10	16	SC_X	2D	45	SC_LEFT	4B	75
SC_W	11	17	SC_C	2E	46	SC_RIGHT	4D	77
SC_E	12	18	SC_V	2F	47	SC_PLUS	4E	78
SC_R	13	19	SC_B	30	48	SC_END	4F	79
SC_T	14	20	SC_N	31	49	SC_DOWN	50	80
SC_Y	15	21	SC_M	32	50	SC_PAGEDOWN	51	81
SC_U	16	22	SC_COMMA	33	51	SC_INSERT	52	82
SC_I	17	23	SC_PERIOD	34	52	SC_DELETE	53	83
SC_O	18	24	SC_SLASH	35	53	SC_F11	57	87
SC_P	19	25	SC_RSHIFT	36	54	SC_F12	58	88
SC_LEFTBRACKET	1A	26	SC_MULTIPLY	37	55	SC_LWIN	7D	125
SC_RIGHTBRACKET	1B	27	SC_ALT	38	56	SC_RWIN	7E	126
SC_ENTER	1C	28	SC_SPACE	39	57	SC_MENU	7F	127
SC_CONTROL	1D	29	SC_CAPSLOCK	3A	58			

C. Ereignisse von SCREENEVENT

Nr.	Konstante	Beschreibung
1	EVENT_KEY_PRESS	Eine Taste wurde gedrückt. Der Record 'scancode' enthält den plattformunabhängigen Scancode der Taste (siehe Anhang B). Ist dieser Taste ein ASCII-Code zugeordnet, so kann dieser aus dem Record 'ascii' gelesen werden.
2	EVENT_KEY_RELEASE	Eine gedrückte Taste wurde wieder losgelassen. Die Records 'scancode' und 'ascii' werden in gleicher Weise ausgefüllt wie bei EVENT_KEY_PRESS.
3	EVENT_KEY_REPEAT	Eine Taste wird so lange gedrückt gehalten, bis sie als wiederholter Tastenanschlag behandelt wird. Die Records 'scancode' und 'ascii' werden in gleicher Weise ausgefüllt wie bei EVENT_KEY_PRESS.
4	EVENT_MOUSE_MOVE	Die Maus wurde im Programmfenster bewegt. Die Records 'x' und 'y' enthalten die neuen Koordinaten des Mauszeigers. Die Records 'dx' und 'dy' enthalten die Differenz der alten Koordinaten zu den neuen.
5	EVENT_MOUSE_BUTTON_PRESS	Ein Mausbutton wurde gedrückt. Der Record 'button' gibt die Taste an: 1 = linke, 2 = rechte, 3 = mittlere Maustaste
6	EVENT_MOUSE_BUTTON_RELEASE	Ein Mausbutton wurde wieder losgelassen. Der Record 'button' wird in gleicher Weise ausgefüllt wie bei EVENT_MOUSE_BUTTON_PRESS.
7	EVENT_MOUSE_DOUBLE_CLICK	Ein Mausbutton wurde doppelt angeklickt. Der Record 'button' wird in gleicher Weise ausgefüllt wie bei EVENT_MOUSE_BUTTON_PRESS.
8	EVENT_MOUSE_WHEEL	Das Mousrad wurde benutzt. Die neue Position des Mousrads wird im Record 'z' eingetragen.
9	EVENT_MOUSE_ENTER	Die Maus wurde in das Programmfenster bewegt.
10	EVENT_MOUSE_EXIT	Die Maus wurde aus dem Programmfenster bewegt.
11	EVENT_WINDOW_GOT_FOCUS	Das Programmfenster hat den Fokus bekommen (es wurde also zum aktiven Fenster).
12	EVENT_WINDOW_LOST_FOCUS	Das Programmfenster hat den Fokus verloren (es ist also in den Hintergrund getreten).
13	EVENT_WINDOW_CLOSE	Der Benutzer hat versucht das Fenster zu schließen, z.B. über den Schließen-Button in der Titelleiste.
14	EVENT_MOUSE_HWHEEL	Das horizontale Mousrad wurde benutzt. Die neue Position des Mousrads wird im Record 'w' eingetragen. Zur Zeit der Drucklegung war der Befehl nicht vollständig implementiert.

D. Modi für SCREENRES und SCREEN

Wert	Symbol	Wirkung
&H00	GFX_WINDOWED	Normaler Fenstermodus (Standard-Option)
&H01	GFX_FULLSCREEN	Vollbildmodus
&H02	GFX_OPENGL	OpenGL-Modus
&H04	GFX_NO_SWITCH	kein Moduswechsel
&H08	GFX_NO_FRAME	kein Rahmen (ab v0.17)
&H10	GFX_SHAPED_WINDOW	Splashscreen-Modus (ab v0.17)
&H20	GFX_ALWAYS_ON_TOP	Fenster, das immer auf oberster Ebene bleibt (ab v0.17)
&H40	GFX_ALPHA_PRIMITIVES	bearbeitet bei Drawing Primitives wie PSET, LINE etc. auch ALPHA-Werte (ab v0.17)
&H80	GFX_HIGH_PRIORITY	höhere Priorität für Grafikprozesse, nur unter Win32 (ab v0.18)
&H10000	GFX_STENCIL_BUFFER	Stencil Buffer (Schablonenpuffer) verwenden (nur im OpenGL-Modus)
&H20000	GFX_ACCUMULATION_BUFFER	Accumulation Buffer (nur im OpenGL-Modus)
&H40000	GFX_MULTISAMPLE	bewirkt im Vollbildmodus Antialiasing durch die ARB_multisample-Erweiterung
-1	GFX_NULL	Grafikmodus ohne visuelles Feedback

Im OpenGL-Modus haben die *drawing primitives* keine Auswirkungen. Es stehen nur ein funktionierendes OpenGL-Fenster und die Befehle zum direkten Speicherzugriff auf den VideoRAM zur Verfügung.

Der Stencil Buffer steht nur im OpenGL-Modus zur Verfügung.

Das Rahmen-Flag bewirkt ein Fenster ohne Titelleiste und Fensterrahmen. Das Splashscreen-Flag bewirkt dasselbe wie das Rahmen-Flag. Zusätzlich ist die transparente Farbe tatsächlich transparent, d.h. der Desktop bzw. die Fenster, die sich hinter dem FreeBASIC-gfx-Fenster befinden, sind 'durch das gfx-Fenster hindurch' sichtbar.

Index

A

Abstandsberechnung	64, 69
AI	66
Allegro (Bibliothek)	55
Alpha-Beta-Suche	71
Alphakanal	23, 62
Animation	56
Arrays	5
ASCII-Zeichentabelle	79

B

Befehle

#DEFINE	10
#INCLUDE	50
ABS	64, 69
ALLOCATE	20
ALPHA	23
BINARY	9
BLOAD	24
BSAVE	24
BYVAL	16
CASE	13
CHR	14
CIRCLE	20
CLS	28
CVI	77
DATA	6
DRAW	20

DRAW STRING	26
ENUM	11, 68
FLIP	28
FUNCTION	13
GET	20, 23
GETJOYSTICK	14
GETKEY	23
GETMOUSE	16
IMAGECREATE	20, 25
IMAGEDESTROY	20
INKEY	12
INSTR	41
INTEGER	8, 31, 71
LBOUND	42
LINE	20 f.
LINE INPUT	8
MID	7, 42
MKI	77
MOD	56, 68
MULTIKEY	14
MUTEXCREATE	44
MUTEXDESTROY	44
MUTEXLOCK	44
MUTEXUNLOCK	44
PAINT	20, 28
PCOPY	28
PRESET	20
PRINT	26
PSET	20, 23

PUT	21 f.	Dateizugriff	8
RANDOMIZE	67	double buffering	28
RESTORE	6	E	
RGB	23	Erweiterbarkeit	2
RND	67	F	
SCREEN	19, 28, 82	FBMath (Bibliothek)	54
SCREENCOPY	28	Flags	35, 37
SCREENEVENT	17	FreeBASIC Extended Library (Bibliothek)	55
SCREENINFO	19	G	
SCREENLOCK	22 f., 28	Grafik	
SCREENRES	19, 28, 82	native Befehle	20
SCREENSET	28 f.	Puffer	20
SCREENUNLOCK	23	Grafikbildschirm initialisieren	19
SCREENWAIT	28	H	
SETMOUSE	16	Header-Dateien	50
SGN	16	Highscore	41
SUB	14	Hintergrund	21
THREADCREATE	44	Hot Seat	74
THREADWAIT	44	K	
TIMER	34	künstliche Intelligenz	66
TRANS	23	Kachelgrafik	31
UBYTE	8	Kollision	61
VALINT	41	L	
WIDTH	26	Leveldaten	6
WSTRING	8	Lizenz	iii
XOR	21 f., 38	M	
Bewegung	56	Maske	23, 62
Bibliotheken	50		
binäre Speicherung	9		
BMP-Bilder	24		
C			
Client	76		
Copyright	iii		
D			
Danksagung	iv		

Mehrspieler 74
 Minimax-Suchverfahren 70
 MultiPut 53

N

nebenläufige Programme 44
 NegaMax-Algorithmus 71
 Netzwerk 74
 Nim-Spiel 73

P

Parallax Scrolling 59
 Pfeiltasten 12
 Projektseite iv
 Prozeduren 13

R

Rechtliches iii
 Rekursion 36
 RGB-Farbwerte 62

S

Scancodes 14, 80
 SCREEN-Modi 82
 SCRENEVENT 81
 Scrolling 57
 Selbsteinschätzung 2
 Server 75
 Spielobjekte 30
 Split Screen 74
 Steuerung
 Beschleunigung 17
 Joystick 14
 Maus 16
 Tastatur 12, 14
 String-Indizierung 8

T

Textausgabe 26
 Threading 44
 Tiles 31
 Transparenzfarbe 59
 TSNE 74
 TSNE_Play 77

U

Untergrund 31
 User Defined Type (UDT) 31

W

Wartbarkeit 3

Z

Zeitsteuerung
 Hauptschleife 34
 Thread 45
 zufällige Züge 67

Liste der Quelltexte

2.1. Leveldaten über DATA-Zeilen einlesen	7
2.2. Leveldaten über eine ASCII-Datei einlesen	9
2.3. Leveldaten über eine Binär-Datei einlesen	10
3.1. Steuerung über Tastatur (INKEY)	12
3.2. Verbesserte Steuerung über Tastatur	13
3.3. Steuerung über Tastatur (MULTIKEY)	14
3.4. Steuerung mit Joystick	15
3.5. Steuerung mit Maus	17
3.6. Bewegung mit Beschleunigung	18
4.1. Arbeiten mit dem Grafikpuffer	21
4.2. PUT mit Aktionswort XOR	22
4.3. PUT mit Hintergrund-Speicherung	24
4.4. Bildgröße ermitteln	25
4.5. DRAW STRING	26
4.6. Verschiedene Schriftgrößen gleichzeitig	27
4.7. Double Buffering	29
5.1. Feld-Daten als UDT	32
5.2. Feld-Daten als UDT (2)	33
5.3. Zeitsteuerung in der Hauptschleife	34
6.1. Minensuchspiel – Spielelemente	35
6.2. Minensuchspiel – Bomben verteilen	36
6.3. Minensuchspiel – Feld aufdecken	37
6.4. Minensuchspiel – Hauptprogramm	38
7.1. Highscore-Type	41
7.2. Highscore einlesen	42
7.3. Highscore schreiben	42

7.4. Highscore bearbeiten	43
8.1. Zeitsteuerung in einem Thread	45
8.2. Mutex-Fehler im rekursiven Aufruf	48
8.3. Mutex-Fehler bei der Rückgabe	48
8.4. Mutex-Fehler bei zu früher Freigabe	49
8.5. Korrekte Mutex-Verwendung	49
8.6. Mutex-Fehler bei Zwischenspeicherung	49
9.1. Sound- und Musikausgabe	52
9.2. Wiedergabe einer Trackermoduldatei	53
9.3. Rotation und Skalierung	54
10.1. Animierte Spielfigur	57
10.2. Scrolling	58
10.3. Parallax Scrolling	60
11.1. Kollisionskontrolle mit einem Farbwert	61
11.2. Kollisionskontrolle mit Alphakanal	63
12.1. Zufallsgesteuerte Computerbewegung	68
12.2. Minimax-Suche	71
12.3. Nim-Spiel	73
13.1. Einfacher Netzwerk-Server	75
13.2. Einfacher Netzwerk-Client	76