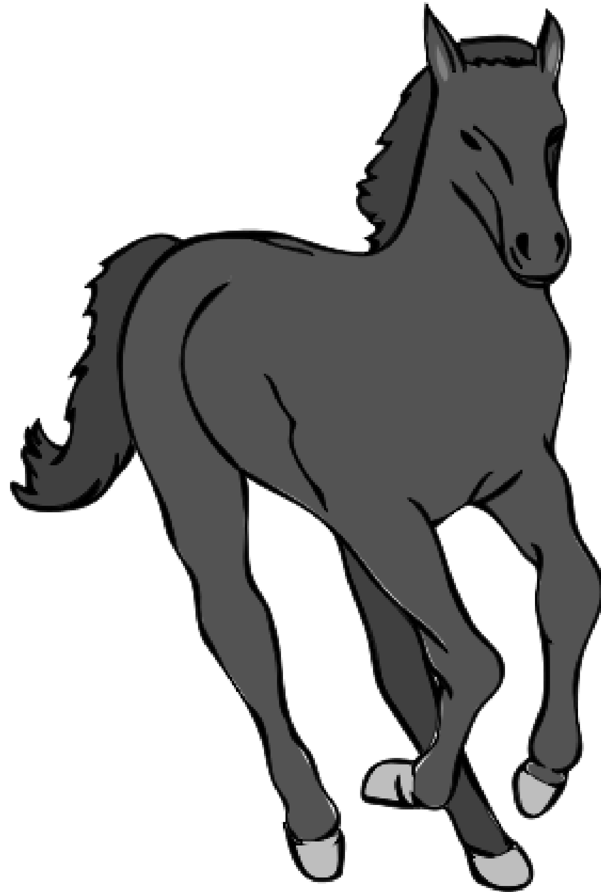
2D-Spieleprogrammierung in FreeBASIC

Ein anwendungsorientierter Leitfaden



von Stephan Markthaler

Stand: 24. September 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 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 Grafikbibliothek wie z. B. OpenGL zu nutzen.

6. Rechtliches

Das Dokument unterliegt der Lizenz Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported. Sie sind berechtigt, das Werk zu vervielfältigen, verbreiten und öffentlich zugänglich machen sowie Abwandlungen und Bearbeitungen des Werkes anfertigen, sofern dabei

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

Einen vollständigen Lizenztext erhalten Sie unter
`http://creativecommons.org/licenses/by-nc-sa/3.0/de/legalcode`
Unabhängig von oben genannten Linzenzbedingungen 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.

7. Weitere Informationen

Sämtliche längeren Quellcodes 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 zu erwähnen möchte ich

- Sebastian für die Bereitstellung des FreeBASIC-Portals
- MOD für die Hilfe bei der Restbildung und für das Korrekturlesen

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. Speicherung der Leveldaten . . . 4
  - 2.1.1. Interne Speicherung der Leveldaten . . . 5
  - 2.1.2. Leveldaten im Quelltext . . . 6
  - 2.1.3. Leveldaten in einer externen Datei . . . 8

3. Steuerung 12
- 3.1. Tastatursteuerung . . . 12
  - 3.1.1. Tastaturabfrage mit `INKEY` . . . 12
  - 3.1.2. Tastatendruck 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

II. Anhang **35**

A. ASCII-Zeichentabelle **36**

B. MULTIKEY-Scancodes **37**

C. Ereignisse von SCREENEVENT **38**

D. Modi für `SCREENRES` und `SCREEN` **39**

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 Projektes, 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 kleinen, 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 kleinen, für sich funktionierenden Teil

des Programms und erweitern Sie es dann nach und nach um weitere 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“ soll sich zunächst einmal auf einzelne ASCII-Zeichen in der Konsole beschränken. 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. 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 als externe Datei vorliegen und eingelesen werden. Doch zunächst wollen wir uns um die interne Speicherung der Leveldaten kümmern.

2.1.1. Interne Speicherung der Leveldaten

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.

2.1.2. 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.

Das folgende 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 starten 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

```
DIM AS STRING*1 felddata(20, 10) ' Daten des Spielfeldes
DIM AS STRING text               ' eingelesene DATA-Zeilen
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
    ' 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) = " "
      sx = spalte
      sy = zeile
    END IF
    IF felddata(spalte, zeile) = "A" THEN
      ' Ausgang entdeckt; Feld wird angepasst
      felddata(spalte, zeile) = " "
      ax = spalte
      ay = zeile
    END IF
  NEXT
NEXT

level1:
DATA "#####################"
DATA "#S  # #      #      #"
DATA "###   # ########### #"
DATA "# ### #    #      # #"
DATA "# #     #  # # ## # #"
DATA "# #     #    #  #   #"
DATA "# # #####  # # ######"
DATA "#      #   # #      #"
DATA "###### ##### ########"
DATA "#      #            A"
DATA "#####################"

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

Nachteil dieser Methode ist, dass Sie das Level nicht mehr verändern können, ohne das Programm neu compilieren zu müssen. 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.

2.1.3. Leveldaten in einer externen Datei

Wollen Sie die Level später verändern können, 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.

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 ausgelesen 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. Hier ist eine gute Kommentierung angeraten; Abschnitt „Sprechende Werte“ 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. Die Berechnungen im Datentyp `INTEGER` laufen jedoch wesentlich schneller; daher sollten sie andere Ganzzahl-Typen nur dann einsetzen, wenn die entstehende Speicherersparnis signifikant ist. Der 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

```
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

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
CLOSE #f
```

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
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 ...
  NEXT
NEXT
CLOSE #f
```

Sprechende Werte

Um die Zahlenwerte, die bei der ASCII- bzw. Binärspeichung auftreten, im Programmcode sofort verstehen zu können, bietet es sich an, sie mit sprechenden Namen auszustatten. 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 Compiliervorgang umgesetzt wird. Nach dem Befehl ersetzt der Compiler alle auftretenden *FeldLeer* durch die Zahl 32. Die Ersetzung findet, wie gesagt, bereits beim Compiliervorgang statt und hat daher keine negativen Geschwindigkeitsauswirkungen auf das spätere Programm.

Eine andere Möglichkeit ist die Verwendung von `ENUM`. Diese Anweisung bietet sich vor allem dann an, wenn einfach der Reihe nach durchnummeriert werden soll.

```
ENUM Felderliste
  FeldLeer
  FeldWand
  ' ...
END ENUM
' ...
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.

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
2. Vorteile der Binär-Speicherung:
   - kompaktes Speicherformat
   - leichter Zugriff auf Teildaten

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 folgenen 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
    CASE CHR(255, 72)
      ' Spielfigur nach oben bewegen
      IF felddata(sx, sy-1) = 32 THEN
        LOCATE sy, sx
        PRINT " ";      ' alte Position leeren
        sy -= 1         ' Spielerposition aendern
        LOCATE sy, sx
        PRINT "S";      ' neue Position schreiben
      END IF
    CASE CHR(255, 75)
      ' 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 das 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
  taste = INKEY
  SELECT CASE taste
    CASE CHR(255, 72)
      bewege(sx, sy, 0, -1)    ' nach oben
    CASE CHR(255, 75)
      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
  END SELECT
  SLEEP 1   ' Pause, um die Kontrolle an andere Prozesse zu uebergeben
LOOP UNTIL taste = CHR(27)

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
    ' das Zielfeld ist leer; die Bewegung kann stattfinden
    LOCATE sy, sx       ' alte Position leeren
    PRINT " ";
    sx += dx            ' Spielerposition aendern
    sy += dy
    LOCATE sy+1, sx+1   ' neue Position schreiben
    PRINT "S";
    RETURN -1           ' Bewegung erfolgreich
  END IF
  RETURN 0              ' keine Bewegung erfolgt
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. Tastatendruck 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, dann wird die Spielerfigur vermutlich nach dem ersten Schritt eine kleine Pause einlegen, bevor sie dann im schnelleren Tempo weitereilt. Der Grund 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 Spielfigur 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
    ' ...
  ELSEIF MULTIKEY(80) THEN
    ' Spielfigur nach unten bewegen
    ' ...
  ' ...
  END IF
  SLEEP 200, 1
LOOP UNTIL MULTIKEY(1)   ' ESC-Taste
```

Für die Pfeltasten 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 erfogreich. *buttons* enthält dann den Status 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.

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

Zumindest theoretisch funktioniert das sehr gut - jedoch nur, wenn der Joystick nach dem Loslassen der Steuerung alle Achsen wieder exakt auf den Wert 0 zurückstellt. Man sollte sich lieber nicht auf die exakten Werte 0, -1 bzw. 1 verlassen, sondern einen gewissen Spielraum einräumen, in dem der Achsenausschlag als Bewegung oder als Stillstand interpretiert wird.

Quelltext 3.4: Steuerung mit Joystick

```
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
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
    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
    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.

3.3. Maussteuerung

Für das hier vorgestellte Spiel ist eine Maussteuerung eher schwierig. Es gibt zwar die Möglichkeit, Mausklicks im Spielfeld auszuwerten und die Spielfigur an die angeklickte Stelle zu bewegen, aber dazu müsste man entweder eine Routine zur Wegsuche implementieren (womit die eigentliche Aufgabe des Spielers, einen Weg aus dem Labyrinth zu suchen, hinfällig wird), oder der Spieler kann immer nur ein Feld neben der augenblicklichen Position anklicken, womit die Mausbedienung recht unkonfortabel wird.

Konfortable 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 zu. Kapitel 4 beschäftigt sich eingehend mit den Grafikbefehlen. 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
```

Die Maus wurde zwar durch den vierten Parameter von `SETMOUSE` im Programmfenster „eingesperrt“, trotzdem funktioniert die Steuerung nicht mehr, wenn sich die Maus am Rand des Fensters befindet. Daher wird die Maus nun nach jeder Abfrage wieder in die Ausgangsposition zurückgesetzt.

Quelltext 3.5: Steuerung mit Maus

```
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
DO
  GETMOUSE mausX, mausY, , mausB              ' Maus abfragen ...
  SETMOUSE 100, 100                           ' ... und zuruecksetzen
  LINE (ballX-5, ballY-5)-STEP(10, 10), 0     ' 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
  CIRCLE (ballX, ballY), 5                    ' Ball zeichnen
  SLEEP 50
LOOP UNTIL mausB <> 0
```

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 abgebremst wird. 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 gegengesetzte Richtung zurückgeworfen. Der unten 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
SETMOUSE 100, 100, 0, 1  ' Maus im Fenster zentrieren und ausblenden
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
LOOP UNTIL mausB <> 0
```

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 compiliertes 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, dann 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 tiefer 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 initialisert, 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 zehnmal 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 9 bis 16

' Bild in den Puffer schreiben
bild = IMAGECREATE(40, 40)                ' Bildpuffer erstellen
CIRCLE bild, (20, 25), 15, farbe, PI, 0, .6 ' untere halbe Ellipse
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

' Puffer auf dem Bildschirm ausgeben
FOR i AS INTEGER = 1 TO 5
  PUT (i*50-20, 80), bild
NEXT

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 3.6 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 dieser Methode 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

' 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
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

' 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
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 ...
  IF mausX > 260 THEN mausX = 260           ' ... und an den Grenzen anpassen
  IF mausy > 160 THEN mausX = 160
  PUT (mausX, mausY), bild, XOR             ' neue Position zeichnen
  SLEEP 1
LOOP UNTIL mausB > 0 OR INKEY = CHR(27)     ' bei Mausklick oder ESC beenden
IMAGEDESTROY bild                           ' Bildpuffer freigeben
```

Es wurde hier 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 Alpha-Kanal einsetzen – FreeBASIC unterstützt auch Bitmaps mit Alpha-Kanal – 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 diese Berechnung mit ein.

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 UINTEGER hell = RGB(255, 64, 64)        ' heller Farbwert des Steins
DIM AS UINTEGER dunkel = RGB(192, 0, 0)        ' dunkler Farbwert des Steins

' Bild in den Puffer schreiben
bild = IMAGECREATE(40, 40)
hg = IMAGECREATE(40, 40)
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
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

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
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
  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
    SCREENUNLOCK
    altX = mausX                           ' neue Position merken
    altY = mausY
  END IF
  SLEEP 1
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

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

' Fenster und Bildpuffer erstellen
SCREENRES breit, hoch, 32
bild = IMAGECREATE(breit, hoch)

' Bild laden und ausgeben
BLOAD datei, bild
PUT (0,0), bild, PSET
IMAGEDESTROY bild
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 als 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*, 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.

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. Mit dem Befehl `DRAW STRING` lässt sich dagegen eine pixelgenaue Positionierung des Textes erreichen. 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 Aktionsworte verwenden. Die Aktionswörter funktionieren allerdings nur bei der Verwendung eines benutzerdefinierten Fonts. 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
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.

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 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, ein langjähriges Mitglied der FreeBASIC-Gemeinde, hat dazu eine Funktion zusammengestellt, die auf der nächsten Seite abgedruckt ist.

Quelltext 4.6: Verschiedene Schriftgrößen gleichzeitig

```
'fb_font_x.bas by Volta

TYPE fb_font_x
  AS INTEGER breit, hoch
  AS ANY PTR start
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

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
  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
    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
          IF (buffer = 0) THEN
            PSET (x,y),fgcol
          ELSE
            PSET buffer,(x,y),fgcol
          END IF
        END IF
          bits = bits SHR 1
      NEXT
      row +=1
    NEXT
    xpos +=f.breit
    IF (xpos-f.breit)>xend THEN EXIT FOR
  NEXT
  SCREENUNLOCK
END SUB

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
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 erhöhtem Speicheraufwand und 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 am einfachsten mit **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
  CLS
  PAINT (100, 100), &h00ff00        ' gruener Hintergrund
  LINE (50, 50)-(150, 150),, BF     ' blaues Rechteck
  SCREENSYNC
  SCREENCOPY                        ' jetzt das Gezeichnete anzeigen
  SLEEP 50, 1
  COLOR &hff0000, &h0000ff          ' rot auf blau
  CLS
  PAINT (100, 100), &h00ff00        ' gruener Hintergrund
  CIRCLE (100, 100), 70,,,,, f      ' roter Kreis
  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 vergessener 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 verschiedene Spielobjekte umgesetzt und dargestellt werden?
- Wie verwendet man unterschiedliche 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, dann 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 eine festdefinierte Größe haben, müssen Sie 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 durch Verwendung der Transparenzfarbe oder des Alphakanals – 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, dass 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 `UDT`s (*user defined type*) an.

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

hintergrund und *vordergrund* sind dabei Zeiger auf den Grafikpuffer, der das Hintergrund-

bzw. Vordergrundbild beinhaltet. *betretbar* speichert, ob das Feld betreten werden kann, und *bonus* 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 von der Art des Einsatzes 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

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

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

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 `UDT`s *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 ein weiteres Record *id*, das 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 bietet sich die Verwendung von `ENUM` an; der unten stehende Code ignoriert dies jedoch und überlässt die Umsetzung dem Leser als Übung.

Quelltext 5.2: Feld-Daten als UDT (2)

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

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

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

' ...

DIM AS feldtyp feld = felddata(sx, sy)    ' Information des Spielerfeldes
IF feld.untergrund.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
  ' ...
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. Um eine vom Rest des Programms völlig unabhängige Steuerung zu erreichen, bietet sich der Einsatz von Multithreading an. Alternativ dazu kann man auch versuchen, die Zeitsteuerung direkt in die Hauptschleife des Spieles zu integrieren. Selbstverständlich

kann der Spielablauf nicht einfach solange 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 Variable der Zeitpunkt festgehalten, an dem das Ereignis zuletzt ausgeführt wurde. Eine weitere Variable speichert, wie lange es bis zur nächsten Ausführung des Ereignis dauert. `TIMER` gibt die Anzahl der vergangenen Sekunden seit dem Systemstart zurück[1] – der Wert erhöht sich also ständig. Es muss nun also regelmäßig verglichen werden, ob bereits genug Zeit verstrichen ist, um das Ereignis auszulösen. Dieses Prinzip wird im Quelltext 5.3 demonstriert. Das Programm gibt alle fünf Sekunden 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

```
DIM AS DOUBLE letzteAusfuehrung = TIMER      ' Zeitpunkt der letzten Ausfuehrung
DIM AS DOUBLE naechsteAusfuehrung = 5        ' Abstand 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
```

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

Teil II.

Anhang

A. ASCII-Zeichentabelle

Zeichencodierung in der Konsole

0		26	→	52	4	78	N	104	h	130	é	156	£	182	Â	208	ð	234	Û
1	☺	27	←	53	5	79	O	105	i	131	â	157	Ø	183	À	209	Ð	235	Ù
2	☻	28	∟	54	6	80	P	106	j	132	ä	158	×	184	©	210	Ê	236	ý
3	♥	29	↔	55	7	81	Q	107	k	133	à	159	ƒ	185	╣	211	Ë	237	Ý
4	♦	30	▲	56	8	82	R	108	l	134	å	160	á	186	║	212	È	238	¯
5	♣	31	▼	57	9	83	S	109	m	135	ç	161	í	187	╗	213	ı	239	´
6	♠	32		58	:	84	T	110	n	136	ê	162	ó	188	╝	214	Í	240	-
7		33	!	59	;	85	U	111	o	137	ë	163	ú	189	¢	215	Î	241	±
8		34	"	60	<	86	V	112	p	138	è	164	ñ	190	¥	216	Ï	242	‗
9		35	#	61	=	87	W	113	q	139	ï	165	Ñ	191	┐	217	┘	243	¾
10		36	$	62	>	88	X	114	r	140	î	166	ª	192	└	218	┌	244	¶
11	♂	37	%	63	?	89	Y	115	s	141	ì	167	º	193	┴	219	█	245	§
12	♀	38	&	64	@	90	Z	116	t	142	Ä	168	¿	194	┬	220	▄	246	÷
13		39	'	65	A	91	[	117	u	143	Å	169	®	195	├	221	¦	247	¸
14	♫	40	(	66	B	92	\	118	v	144	É	170	¬	196	─	222	Ì	248	°
15	☼	41	)	67	C	93	]	119	w	145	æ	171	½	197	┼	223	▀	249	¨
16	►	42	*	68	D	94	^	120	x	146	Æ	172	¼	198	ã	224	Ó	250	·
17	◄	43	+	69	E	95	_	121	y	147	ô	173	¡	199	Ã	225	ß	251	¹
18	↕	44	,	70	F	96	`	122	z	148	ö	174	«	200	╚	226	Ô	252	³
19	‼	45	-	71	G	97	a	123	{	149	ò	175	»	201	╔	227	Ò	253	²
20	¶	46	.	72	H	98	b	124	¦	150	û	176	░	202	╩	228	õ	254	■
21	§	47	/	73	I	99	c	125	}	151	ù	177	▒	203	╦	229	Õ	255	
22	▬	48	0	74	J	100	d	126	~	152	ÿ	178	▓	204	╠	230	µ		
23	↨	49	1	75	K	101	e	127	⌂	153	Ö	179	│	205	═	231	þ		
24	↑	50	2	76	L	102	f	128	Ç	154	Ü	180	┤	206	╬	232	Þ		
25	↓	51	3	77	M	103	g	129	ü	155	ø	181	Á	207	¤	233	Ú		

Zeichencodierung in einem Grafikfenster

0		26	→	52	4	78	N	104	h	130	é	156	£	182	╢	208	╨	234	Ω
1	☺	27	←	53	5	79	O	105	i	131	â	157	¥	183	╖	209	╤	235	δ
2	☻	28	∟	54	6	80	P	106	j	132	ä	158	₧	184	╕	210	╥	236	∞
3	♥	29	↔	55	7	81	Q	107	k	133	à	159	ƒ	185	╣	211	╙	237	φ
4	♦	30	▲	56	8	82	R	108	l	134	å	160	á	186	║	212	╘	238	ε
5	♣	31	▼	57	9	83	S	109	m	135	ç	161	í	187	╗	213	╒	239	∩
6	♠	32		58	:	84	T	110	n	136	ê	162	ó	188	╝	214	╓	240	≡
7		33	!	59	;	85	U	111	o	137	ë	163	ú	189	╜	215	╫	241	±
8		34	"	60	<	86	V	112	p	138	è	164	ñ	190	╛	216	╪	242	≥
9		35	#	61	=	87	W	113	q	139	ï	165	Ñ	191	┐	217	┘	243	≤
10		36	$	62	>	88	X	114	r	140	î	166	ª	192	└	218	┌	244	⌠
11	♂	37	%	63	?	89	Y	115	s	141	ì	167	º	193	┴	219	█	245	⌡
12	♀	38	&	64	@	90	Z	116	t	142	Ä	168	¿	194	┬	220	▄	246	÷
13		39	'	65	A	91	[	117	u	143	Å	169	⌐	195	├	221	▌	247	≈
14	♫	40	(	66	B	92	\	118	v	144	É	170	¬	196	─	222	▐	248	°
15	☼	41	)	67	C	93	]	119	w	145	æ	171	½	197	┼	223	▀	249	∙
16	►	42	*	68	D	94	^	120	x	146	Æ	172	¼	198	╞	224	α	250	·
17	◄	43	+	69	E	95	_	121	y	147	ô	173	¡	199	╟	225	ß	251	√
18	↕	44	,	70	F	96	`	122	z	148	ö	174	«	200	╚	226	Γ	252	ⁿ
19	‼	45	-	71	G	97	a	123	{	149	ò	175	»	201	╔	227	π	253	²
20	¶	46	.	72	H	98	b	124	¦	150	û	176	░	202	╩	228	Σ	254	■
21	§	47	/	73	I	99	c	125	}	151	ù	177	▒	203	╦	229	σ	255	
22	▬	48	0	74	J	100	d	126	~	152	ÿ	178	▓	204	╠	230	µ		
23	↨	49	1	75	K	101	e	127	⌂	153	Ö	179	│	205	═	231	τ		
24	↑	50	2	76	L	102	f	128	Ç	154	Ü	180	┤	206	╬	232	Φ		
25	↓	51	3	77	M	103	g	129	ü	155	¢	181	╡	207	╧	233	Θ		

B. MULTIKEY-Scancodes

Die nachfolgende Liste enthält die Scancodes, die 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
SC_ESCAPE	01	1
SC_1	02	2
SC_2	03	3
SC_3	04	4
SC_4	05	5
SC_5	06	6
SC_6	07	7
SC_7	08	8
SC_8	09	9
SC_9	0A	10
SC_0	0B	11
SC_MINUS	0C	12
SC_EQUALS	0D	13
SC_BACKSPACE	0E	14
SC_TAB	0F	15
SC_Q	10	16
SC_W	11	17
SC_E	12	18
SC_R	13	19
SC_T	14	20
SC_Y	15	21
SC_U	16	22
SC_I	17	23
SC_O	18	24
SC_P	19	25
SC_LEFTBRACKET	1A	26
SC_RIGHTBRACKET	1B	27
SC_ENTER	1C	28
SC_CONTROL	1D	29

Konstante	hex	dez
SC_A	1E	30
SC_S	1F	31
SC_D	20	32
SC_F	21	33
SC_G	22	34
SC_H	23	35
SC_J	24	36
SC_K	25	37
SC_L	26	38
SC_SEMICOLON	27	39
SC_QUOTE	28	40
SC_TILDE	29	41
SC_LSHIFT	2A	42
SC_BACKSLASH	2B	43
SC_Z	2C	44
SC_X	2D	45
SC_C	2E	46
SC_V	2F	47
SC_B	30	48
SC_N	31	49
SC_M	32	50
SC_COMMA	33	51
SC_PERIOD	34	52
SC_SLASH	35	53
SC_RSHIFT	36	54
SC_MULTIPLY	37	55
SC_ALT	38	56
SC_SPACE	39	57
SC_CAPSLOCK	3A	58

Konstante	hex	dez
SC_F1	3B	59
SC_F2	3C	60
SC_F3	3D	61
SC_F4	3E	62
SC_F5	3F	63
SC_F6	40	64
SC_F7	41	65
SC_F8	42	66
SC_F9	43	67
SC_F10	44	68
SC_NUMLOCK	45	69
SC_SCROLLLOCK	46	70
SC_HOME	47	71
SC_UP	48	72
SC_PAGEUP	49	73
SC_LEFT	4B	75
SC_RIGHT	4D	77
SC_PLUS	4E	78
SC_END	4F	79
SC_DOWN	50	80
SC_PAGEDOWN	51	81
SC_INSERT	52	82
SC_DELETE	53	83
SC_F11	57	87
SC_F12	58	88
SC_LWIN	7D	125
SC_RWIN	7E	126
SC_MENU	7F	127

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 A). 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 Mausrad wurde benutzt. Die neue Position des Mausrads 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 Mausrad wurde benutzt. Die neue Position des Mausrads 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 auf Vollbild-Antialiasing durch die ARB_multisample-Erweiterung
-1	GFX_NULL	Grafikmodus ohne visuelles Feedback

Im OpenGL-Modus haben die *drawing primitives* keine Auswirkungen. Es steht 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

ASCII-Zeichentabelle, 36

Befehle
- #DEFINE, 10
- ALLOCATE, 20
- ALPHA, 23
- BINARY, 9
- BLOAD, 24
- BSAVE, 24
- BYVAL, 15
- CASE, 12
- CHR, 14
- CIRCLE, 20
- CLS, 28
- DATA, 6, 8
- DRAW, 20
- DRAW STRING, 26
- ENUM, 10
- FLIP, 28
- FUNCTION, 13
- GET, 20, 23
- GETJOYSTICK, 14
- GETKEY, 23
- IMAGECREATE, 20, 25
- IMAGEDESTROY, 20
- INKEY, 12, 14
- INTEGER, 8, 31
- LINE, 20, 21
- LINE INPUT, 8
- MID, 7
- MULTIKEY, 14
- PAINT, 20, 28
- PCOPY, 28
- PRESET, 20
- PRINT, 26
- PSET, 20, 23
- PUT, 21, 22
- RESTORE, 6
- RGB, 23
- SCREEN, 19, 28, 39
- SCREENCOPY, 28
- SCREENINFO, 19
- SCREENLOCK, 22, 23, 28
- SCREENRES, 19, 28, 39
- SCREENSET, 28, 29
- SCREENUNLOCK, 23
- SCREENWAIT, 28
- SETMOUSE, 16
- SUB, 14
- TIMER, 34
- TRANS, 23
- UBYTE, 8
- WIDTH, 26
- WSTRING, 8
- XOR, 21, 22

binäre Speicherung, 9

BMP-Bilder, 24

Copyright, iii

Danksagung, iv
double buffering, 28

Erweiterbarkeit, 2

Grafik
  native Befehle, 20
  Puffer, 20
Grafikbildschirm initialisieren, 19

Hintergrund, 21

Leveldaten, 6
Lizenz, iii

Projektseite, iv

Rechtliches, iii

Scancodes, 37
SCREEN-Modi, 39
SCREENEVENT, 38
Selbsteinschätzung, 2
Spielobjekte, 30
Steuerung
  Beschleunigung, 17
  Joystick, 14
  Maus, 16
  Tastatur, 12, 14
String-Indizierung, 8

Textausgabe, 26

Untergrund, 31
user defined types, 31

Wartbarkeit, 3

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