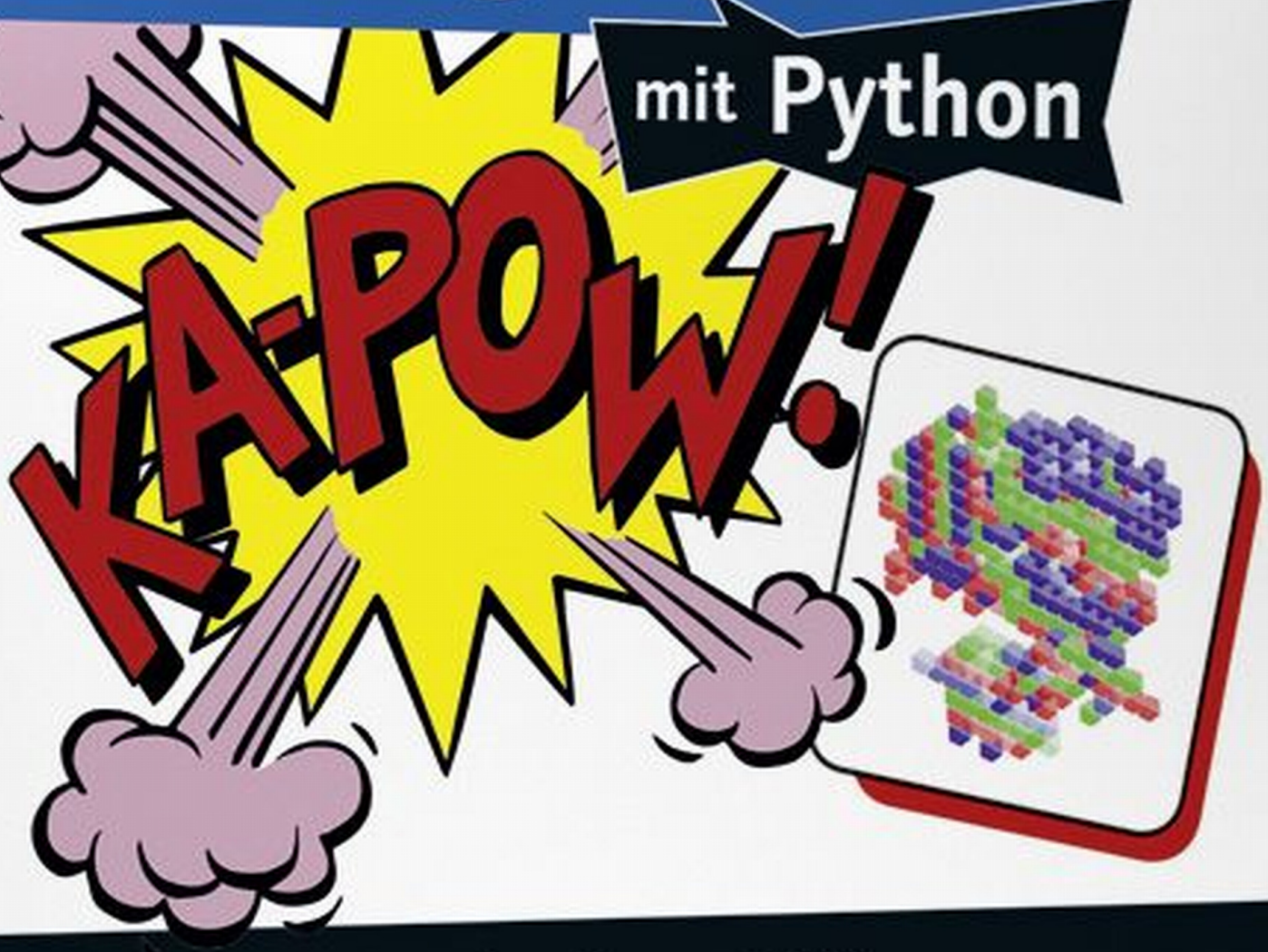
Lars Heppert

Coding for Fun

mit Python



- Das nächste Level für Python-Programmierer!
- Spannende Programme verstehen und selbst entwickeln
- Webcrawler, Snake, Kryptologie, genetische Algorithmen ...

CD ROM Alle Beispiele sowie die komplette Software (Eclipse JDT, Python etc.) zum sofortigen Start!

Galileo Computing

Lars Heppert

Coding for Fun mit Python

Galileo Press

Der Name Galileo Press geht auf den italienischen Mathematiker und Philosophen Galileo Galilei (1564–1642) zurück. Er gilt als Gründungsfigur der neuzeitlichen Wissenschaft und wurde berühmt als Verfechter des modernen, heliozentrischen Weltbilds. Legendär ist sein Ausspruch *Eppur se muove* (Und sie bewegt sich doch). Das Emblem von Galileo Press ist der Jupiter, umkreist von den vier Galileischen Monden. Galilei entdeckte die nach ihm benannten Monde 1610.

Lektorat Sebastian Kestel, Christine Siedle
Korrektorat Petra Biedermann, Reken
Typografie und Layout Vera Brauner
Herstellung Steffi Ehrentraut
Satz Lars Heppert
Einbandgestaltung Barbara Thoben, Köln
Titelbild Getty images/Paul Gilligen
Druck und Bindung Bercker Graphischer Betrieb, Kevelaer

Dieses Buch wurde gesetzt aus der Linotype Syntax Serif (9,25/13,25 pt) in LaTeX.

Gerne stehen wir Ihnen mit Rat und Tat zur Seite:
sebastian.kestel@galileo-press.de bei Fragen und Anmerkungen zum Inhalt des Buches
service@galileo-press.de für versandkostenfreie Bestellungen und Reklamationen
britta.behrens@galileo-press.de für Rezensions- und Schulungsexemplare

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über *http://dnb.d-nb.de* abrufbar.

ISBN 978-3-8362-1513-8

1. Auflage 2010

Liebe Leserin, lieber Leser,

Sie spielen gerne echte Klassiker, stellen sich auch in Ihrer Freizeit immer wieder neuen Herausforderungen, finden Kniffliges spannend und haben schon Erfahrungen mit Python sammeln können? Sehr gut! Dann liegt dieses Coding for Fun-Buch in Ihren Händen absolut richtig!

Denn dieses Buch will Sie unterhalten – zum einen mit genialen Spielen wie beispielsweise »Snake«, die Sie einfach nachprogrammieren und spielen können. Zum anderen mit Aufgaben zum Tüfteln und Knobeln. Ob »Game of Life«, Verschlüsselung oder der eigene Webcrawler: Unser Autor Lars Heppert hat für Sie einfache bis anspruchsvolle Themen zusammengestellt, mit denen Sie richtig spannende Stunden erleben werden!

Eines lässt sich an dieser Stelle bereits sagen: Sie werden über die Beispiele und das erfrischend andere Wissen in diesem Buch immer wieder staunen. Spätestens dann, wenn Ihnen der Autor die Verbindung von Mondlandung und Suprematismus vor Augen führt, dürften Sie dem wohl zustimmen …

Damit Sie sofort beginnen können, finden Sie die erforderliche Software und alle Codebeispiele auf der beiliegenden CD-Rom. Jetzt müssen Sie zu Beginn nur noch Ihr persönliches Lieblingskapitel wählen – und schon geht's los!

Noch eine Anmerkung in eigener Sache: Dieses Buch wurde mit großer Sorgfalt geschrieben, lektoriert und produziert. Doch kein Buch ist perfekt. Sollte also etwas nicht so funktionieren, wie Sie es erwarten, dann scheuen Sie sich nicht, sich mit mir in Verbindung zu setzen. Ihre Fragen und Anmerkungen sind jederzeit willkommen.

Viel Freude beim Lesen und Programmieren wünscht Ihnen

Ihr Sebastian Kestel
Lektorat Galileo Computing

sebastian.kestel@galileo-press.de
www.galileocomputing.de
Galileo Press · Rheinwerkallee 4 · 53227 Bonn

Auf einen Blick

Inhalt

Vorwort

Wie Sie vermutlich bereits wissen, ist Python eine sehr interessante Sprache mit viel Potenzial für die verschiedensten Einsatzbereiche. Genau aus diesem Grund und der Tatsache, dass sich Python-Code sehr leicht lesen lässt, ist diese Sprache für ein Buch, in dem es vor allem um den Spaß am Programmieren geht, eine ausgezeichnete Wahl. Genaugenommen die ideale Sprache, um schnell Probleme zu lösen und Dinge ganz einfach einmal auszuprobieren, ohne viel Arbeit investieren zu müssen. Denn gerade der sonst doch recht hohe Arbeitsaufwand der Softwareerstellung ist es, der viele abschreckt und oft zu Standardsoftware greifen lässt. Der Haken hierbei ist jedoch die Bevormundung der Nutzer in der Form, dass dann eben auch nur der vorgesehene Standard möglich ist. Der Erfinder von Python hat diesen Sachverhalt in einem sehr schönen Zitat treffend zum Ausdruck gebracht.

> *»However, while many people nowadays use a computer, few of them are computer programmers. Non-programmers aren't really empowered in how they can use their computer: they are confined to using applications in ways that programmers have determined for them. One doesn't need to be a visionary to see the limitations here.«*
>
> *Guido van Rossum*

Guido van Rossum hat hier Abhilfe geschaffen, indem er eine leichte und hoch produktive Sprache zur Verfügung stellte, welche es seit kurzem in der Version 3.1 gibt. Auch mit der neuen Version wurde weiterhin an der Klarheit der Sprache gefeilt, um in Python formulierte Programme noch klarer zu gestalten und die Sprachlogik insgesamt so einheitlich wie möglich zu halten. Dieses Ziel für Klarheit und Gradlinigkeit ist sehr schön im „Zen of Python" festgehalten:

> *»Beautiful is better than ugly.*
> *Explicit is better than implicit.*
> *Simple is better than complex.*
> *Complex is better than complicated.*
> *Flat is better than nested.*
> *Sparse is better than dense.*
> *Readability counts.*
> *Special cases aren't special enough to break the rules.*
> *Although practicality beats purity.*
> *Errors should never pass silently.*
> *Unless explicitly silenced.*
> *In the face of ambiguity, refuse the temptation to guess.*
> *There should be one – and preferably only one – obvious way to do it.*

Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
*Although never is often better than *right* now.*
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea – let's do more of those!«

The Zen of Python, by Tim Peters

Ein Beispiel aus der Praxis, das die Einfachheit der Sprache nicht besser ausdrücken könnte, ist die folgende Implementierung des QuickSort-Algorithmus:

```
def quicksort(liste):
    if len(liste) <= 1:
        return liste
    pivotelement = liste.pop()
    links  = [element for element in liste if element <  pivotelement]
    rechts = [element for element in liste if element >= pivotelement]
    return quicksort(links) + [pivotelement] + quicksort(rechts)
```

Listing 0.1 Die einfachste mir bekannte Implementierung von »QuickSort«

Der Algorithmus basiert auf einem einfachen und altbekannten Prinzip: »Divide et Impera« – Zu deutsch: »Teile und herrsche«. Die eigentliche Aufgabe, eine Liste von vergleichbaren Elementen zu sortieren, wird in mehrere Teilschritte zerlegt. Ein Teilschritt ist dabei durch die Wahl eines beliebigen Elements gekennzeichnet, welches daraufhin als »Mitte« definiert wird. Ausgehend von dieser neu definierten Mitte werden daraufhin die kleineren und größeren Elemente in die Listen `links` und `rechts` sortiert. Zu guter Letzt wird durch Rekursion für die beiden genannten Listen dieser Schritt erneut ausgeführt. Die Rekursion endet erst, wenn die Teillisten nur noch ein Element enthalten, da eine Liste mit genau einem Element – wie der Informatiker sagen würde – immer sortiert ist.

Wie Sie sehen, geht es in diesem Buch immer direkt ans Eingemachte, so dass Sie sich nicht mit trockener Theorie abfinden müssen. Insofern kann ich Ihnen ein sehr praxisorientiertes Buch mit viel Freiraum für Ihre eigene Kreativität versprechen. Hier wird alles direkt ausprobiert und kodiert. Hin und wieder mag das auf den ersten Blick dann auch zu Code führen, der noch ein wenig Aufräumarbeiten verkraften kann, aber dafür kommen wir schneller ans Ziel und sparen uns einige trockene Passagen. Zwischendrin gibt es immer wieder Hinweise auf mögliche Fallgruben und Optimierungsmöglichkeiten, die Sie jederzeit selbst noch

umsetzen können und die auch zum Teil schon innerhalb des Buches umgesetzt werden.

Aber nun wünsche ich Ihnen viel Spass mit diesem Buch – Ihre Meinung und Ideen, aber natürlich auch Fragen nehme ich gerne per E-Mail an *Lars@Heppert.com* entgegen. Zudem habe ich auch eine Website zum Buch eingerichtet, welche Sie unter *http://coding.heppert.com* finden.

Danksagung

An dieser Stelle möchte ich mich bei Tino Wagner für die sehr schöne Implementierung eines Doppelpendels bedanken, die mich dazu verleitet hat, auch etwas zur Chaostheorie zu bringen und zudem ein Einfachpendel zu implementieren. Darüber hinaus hat er die Mondlandung »debugged« – was Ihn um ein paar Stunden Schlaf gebracht hat. Danken möchte ich auch Andreas Oswald für seine Tipps bezüglich OpenGL und Spieleprogrammierung im allgemeinen, sowie Torsten Kleinwächter als ausdauerndem Probeleser. Darüber hinaus danke ich für die Unterstützung seitens des Verlags durch Christine Siedle, Sebastian Kestel und Petra Biederman, die alle Kapitel sehr sorgfältig lektoriert und dabei auch gut mitgedacht haben. Zu guter Letzt noch ein Dank an Steffi Ehrentraut für die Überarbeitung der im Buch verwendeten Grafiken.

Lars Heppert
Lars@Heppert.com

»Zeit ist das, was man an der Uhr abliest.«

Albert Einstein

1 Es schlägt 12

In diesem Kapitel werden wir eine analoge Uhr entwickeln, die, wie sollte es auch anders sein, die momentane Zeit anzeigt. Wir stellen die Uhr zunächst zweidimensional dar; wer dann noch nicht genug hat, bastelt sich im Anschluss noch eine schickere 3D-Variante. Bei der 3D-Darstellung werde ich mich allerdings auf Rotationen der Uhr beschränken. Die Uhr an sich wird dann immer noch keine Tiefe besitzen, aber auch die 2D-Darstellung können Sie mit OpenGL leicht rotieren lassen. Meine Empfehlung: Wenn möglich, starten Sie gleich mit OpenGL, denn die dadurch gewonnene Flexibilität in der Darstellung und insgesamt höhere Güte der Grafikausgabe ist durch nichts zu ersetzen. Außerdem ist OpenGL richtig cool – aber das sehen Sie spätestens in Abschnitt 1.8, »Zeichnen mit OpenGL«.

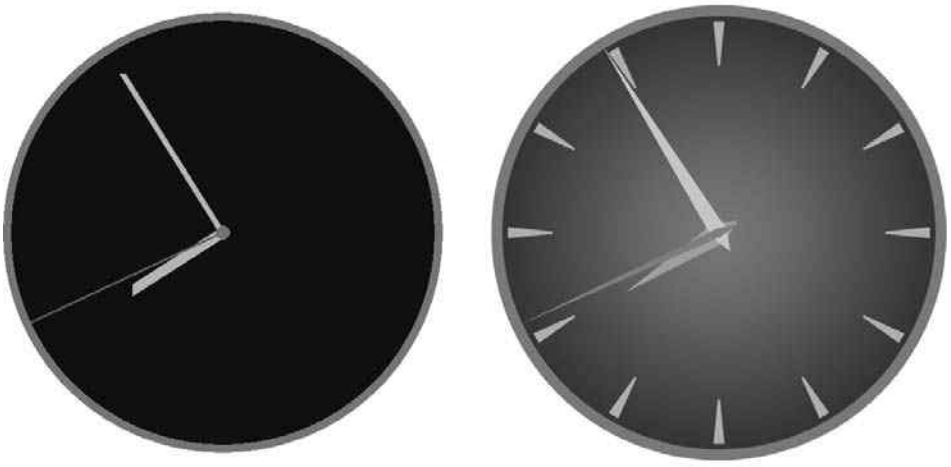


Abbildung 1.1 Links die OpenGL-Variante, rechts die 2D-API-Version

1.1 Eine Uhr?

Zunächst stellen wir, wie bereits in der Kapiteleinleitung geschildert, lediglich eine Uhr dar. Das hört sich nicht gerade spektakulär an und ist an sich auch nicht

wirklich schwer, aber was genau benötigen wir dafür? Für die zweidimensionale Darstellung reicht es, einen Kreis zu zeichnen und die Zeiger entsprechend der aktuellen Uhrzeit innerhalb des Kreises als Linien darzustellen. Für die dreidimensionale Darstellung ist natürlich analog vorzugehen, allerdings kommt hier die Besonderheit der Verwendung von OpenGL hinzu. Nebenbei bemerkt handelt es sich bei der darzustellenden Uhr natürlich um eine »schöne« analoge Uhr, wenn »schön« auch, wie immer, sicherlich eine Geschmacksfrage ist. Eine Frage stellt sich jedoch noch: Wie genau ermitteln wir den für die aktuelle Stunde und Minute korrekten Winkel der darzustellenden Zeiger? Vermutlich ist Ihnen die Antwort schon klar – sie folgt in Kürze.

[◉] Den kompletten Quelltext zur 2D-Clock finden Sie unter *Kapitel01/clock.py* auf der beiliegenden CD.

1.2 Das Importgeschäft – Bibliotheken

Für die 2D-Darstellung benötigen Sie zunächst einmal »pygame«. »pygame« ist eine Bibliothek, die die Entwicklung von Spielen unterstützt und vereinfacht. Bitte installieren Sie »pygame« bei Bedarf von der beiliegenden Buch-CD, oder laden Sie die neueste Version unter *http://www.pygame.org* herunter. Außer »pygame« brauchen Sie die Systembibliothek »sys« (für den Aufruf zum Beenden des Programms) sowie »math« (zur Berechnung der Zeigerstellung) und »datetime« (zur Ermittlung der aktuellen Uhrzeit). Beginnen Sie Ihr Programm also mit dem folgenden Import:

```
import pygame, sys, math, datetime
```

»pygame« existiert auch für Python 3.1, so dass Sie bei der Wahl der Python-Version nur insofern eingeschränkt sind, dass Sie sich auf die 32-Bit-Version beschränken müssen, da »pygame« – Stand zur Drucklegung dieses Buches – nur für diese Versionen zur Verfügung steht. Dennoch empfehle ich Ihnen, noch eine Weile auf die »alten« Python-Versionen zu setzen, da auch viele andere Bibliotheken noch nicht den Sprung zu Python 3.x vollzogen haben, darunter die beiden sehr nützlichen Bibliotheken »NumPy« und »SciPy«.

1.3 Farben und Maße der Uhr

Als Nächstes legen wir die Dimensionen, Farben und sonstigen Eingangsgrößen für unsere Uhr fest.

```
windowMargin            = 30
windowWidth             = 600
windowHeight            = windowWidth
windowCenter            = windowWidth/2, windowHeight/2
clockMarginWidth        = 20
secondColor             = (255, 0, 0)
minuteColor             = (100, 200, 0)
hourColor               = (100, 200, 0)
clockMarginColor        = (130, 130, 0)
clockBackgroundColor    = (20, 40, 30)
backgroundColor         = (255, 255, 255)
hourCursorLength        = windowWidth/2.0-windowMargin-140
minuteCursorLength      = windowWidth/2.0-windowMargin-40
secondCursorLength      = windowWidth/2.0-windowMargin-10

virtualSpeed    = 1
useVirtualTimer = False
```

Listing 1.1 Farben und Maße der Uhr

Farbe	Rotanteil	Grünanteil	Blauanteil
Schwarz	0	0	0
Rot	255	0	0
Grün	0	255	0
Gelb	255	255	0
Blau	0	0	255
Magenta	255	0	255
Zyan	0	255	255
Dunkelgrau	64	64	64
Hellgrau	192	192	192
Braun	153	102	31
Orange	250	160	31
Pink	250	10	178
Lila	153	102	178
Weiß	255	255	255

Tabelle 1.1 RGB-Werte häufig verwendeter Farben

Die meisten der in Listing 1.1 aufgezählten Variablen erklären sich von selbst. Beispielsweise steht `windowMargin` für den Abstand der Uhr vom Fensterrand, `windowHeight` sowie `windowWidth` repräsentieren die Fensterhöhe bzw. Fenster-

breite, und die x- und y-Koordinaten des Tupels `windowCenter` beschreiben die Fenstermitte. Die Variable `clockMargin` legt die Breite des Uhrrandes in Pixel fest. Die darauffolgenden Dreier-Tupel stehen für die verschiedenen Farbwerte, die jeweils durch den Rot-, Grün-, und Blauanteil im Bereich von 0 bis 255 definiert sind. Sie können hier natürlich auch andere Farben verwenden als ich. In Tabelle 1.1 finden Sie dazu eine Übersicht einiger Farben mit den zugehörigen RGB-Werten. Die Variablen `hourCursorLength`, `minuteCursorLength` und `secondCursorLength` geben die jeweilige Zeigerlänge an. Ganz zum Schluss werden noch zwei Variablen für den »virtuellen Timer« gesetzt: `virtualSpeed` bestimmt die Ablaufgeschwindigkeit unserer virtuellen Zeit, und `useVirtualTimer` wählt zwischen echter und virtueller Zeit aus. Den »virtuellen Timer« verwenden wir später, um die Zeit schneller ablaufen zu lassen, so dass sofort sichtbar wird, ob die Uhr prinzipiell richtig funktioniert; ohne eine Beschleunigung der Zeit wäre ansonsten eine Überprüfung sehr langwierig. Abbildung 1.2 veranschaulicht noch einmal die Maße der Uhr.

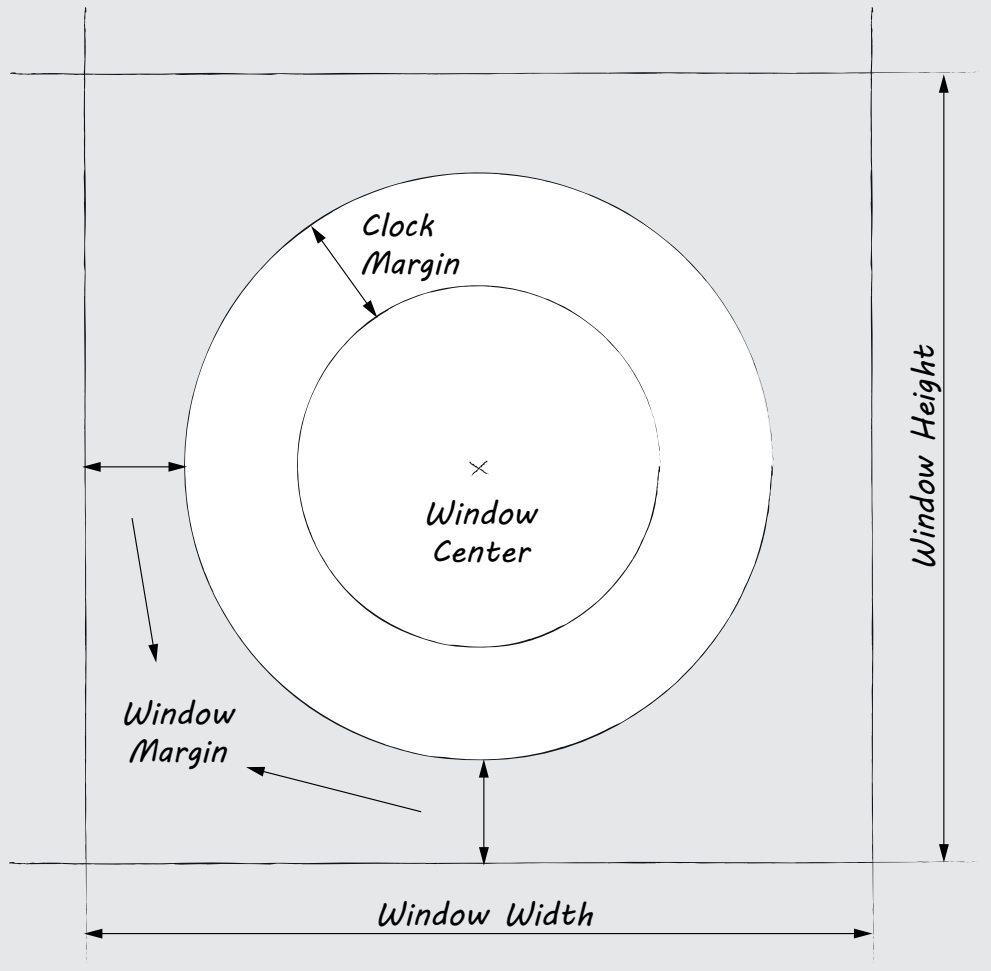


Abbildung 1.2 Die Maße der Uhr

1.4 Steuerung der Uhrzeiger

Um die Uhrzeiger steuern zu können, muss ihre jeweilige Endposition berechnet werden. Das geht relativ einfach über die Winkelfunktionen. Für diejenigen, die sich nicht mehr daran erinnern können, wie das genau funktioniert, illustriert Abbildung 1.3 noch einmal das Prinzip.

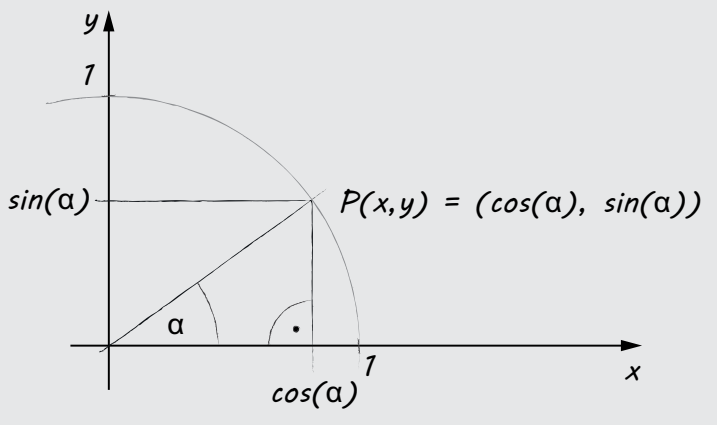


Abbildung 1.3 Berechnung der Zeigerendposition

Die Startposition der Zeiger ist jeweils der Mittelpunkt unserer Uhr, den wir durch einen Kreis (genauer einen Ring) darstellen. Der Mittelpunkt der Uhr wird im Quelltext durch das Tupel `windowCenter` repräsentiert. Für die Berechnung der einzelnen Endpositionen benötigen Sie noch den Winkel, der zu der entsprechenden Zeitangabe passt. Da die Gradzahl für den Winkel von der x-Achse aus bestimmt wird, entspräche zum Beispiel die Zeigerposition von 3 Uhr einem Winkel von 0 Grad.

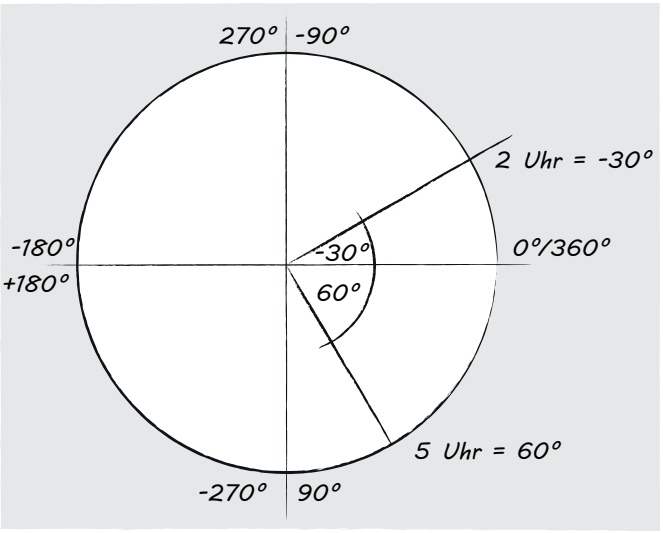


Abbildung 1.4 Zeiten in Gradzahlen ohne Anpassung

Um es etwas intuitiver zu machen, ziehen wir von allen Winkeln 90 Grad ab, so dass 0 Grad 12 Uhr entspricht.

```
def getCursorPositionDegrees(position, scale):
    # 12 Uhr entspricht -90 Grad
    cursorOffset = -90
    degrees = 360 / scale * position + cursorOffset
    return degrees
```

Listing 1.2 Den Zeigerwinkel für die Uhrzeit bestimmen

Durch die Reduzierung des Winkels um 90 Grad können wir also 0 Grad als 12 Uhr deklarieren. Danach rechnen wir die Skala auf 360 Grad um, um die Gradzahl für den Zeiger zu ermitteln. `scale` gibt also an, ob wir gerade den Winkel des Minutenzeigers ermitteln, oder den des Stundenzeigers. Je nachdem, welcher Winkel berechnet wird, wird durch `scale` entweder 60 oder 12 als Skala festgelegt.

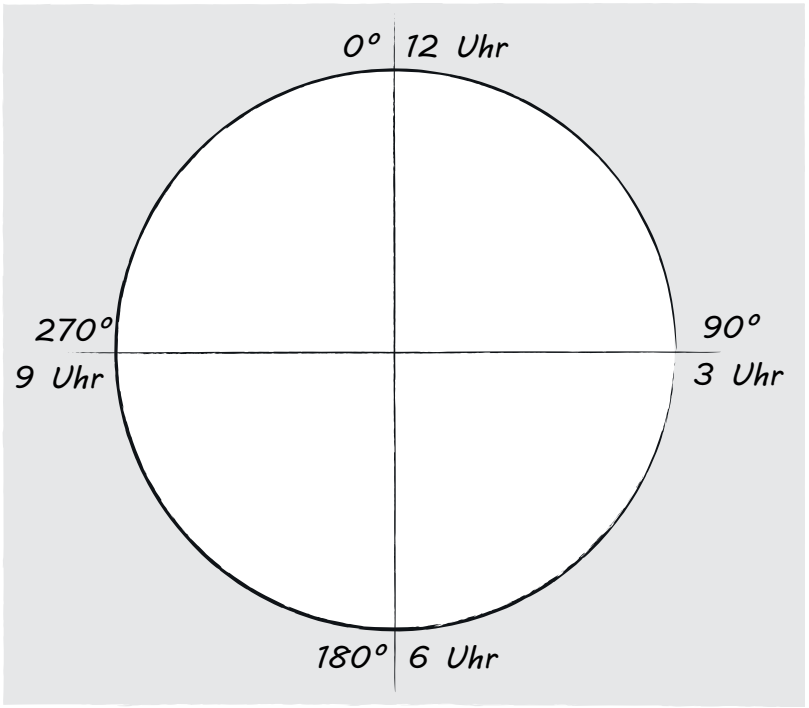


Abbildung 1.5 Zeiten in Gradzahlen mit Anpassung

Eine weitere kleine Hürde ist die Umrechnung von Grad in Bogenmaß. Dazu teilen wir die Gradzahl durch 180 und multiplizieren sie mit Pi, denn 180 Grad entsprechen Pi im Bogenmaß. Die Umrechnung benötigen wir, da alle Berechnungen mit den Winkelfunktionen in der Bibliothek `math` in Bogenmaß durchgeführt werden. Die entsprechende Funktion sieht wie folgt aus:

```
def gradToBogenmass(degrees):
    # python bietet auch die Funktion math.radians(degrees),
    # welche die Umrechnung genauso ausfuehrt, aber so wird
    # der Sachverhalt deutlicher
    return degrees/180.0*math.pi
```

Listing 1.3 Umrechnung von Grad in Bogenmaß

Da die Bibliothek `math` alles in Bogenmaß berechnet, bietet sie natürlich auch entsprechende Funktionen, die wir für die Umrechnung verwenden können. Der Sachverhalt der Umrechnung ist jedoch so einfach und gleichzeitig nützlich, dass ich ihn hier kurz durch die Definition der eigenen Funktion klarer darstellen wollte. Alternativ bedienen Sie sich gerne auch der im Kommentar genannten Funktion `math.radians()`. Der entsprechende Gegenpart zur Umrechnung von Bogenmaß in Grad ist die Funktion `math.degrees()`.

Da Sie alles im Einheitskreis rechnen, also in einem Kreis mit einem Radius von 1, müssen Sie die Zeiger jetzt noch »verlängern«. Dafür multiplizieren Sie einfach die errechneten Werte für x und y mit einem Skalierungsfaktor, den Sie im Quelltext als `cursorLength` wiederfinden. Der Parameter `scale` gibt dabei die Skala für den zu zeichnenden Zeiger an, so dass `scale` für den Minutenzeiger zum Beispiel 60 annimmt, entsprechend der Anzahl der Minuten pro Stunde, für den Stundenzeiger hingegen 12; die Skala gibt also die Anzahl der Einheiten vor, die einer kompletten Umdrehung des Zeigers in der Uhr entsprechen.

Den Endpunkt eines Zeigers errechnen Sie also wie folgt:

```
def getCirclePoint(position, scale, cursorLength):
    degrees = getCursorPositionDegrees(position, scale)
    bogenmass = gradToBogenmass(degrees)
    xPos = round(math.cos(bogenmass)*cursorLength+windowCenter[0])
    yPos = round(math.sin(bogenmass)*cursorLength+windowCenter[1])
    return (xPos, yPos)
```

Listing 1.4 Ermittlung der Zeigerendposition

Im Code steht `position` für die aktuelle Position in der angegebenen Skala. Das heißt, für 3 Uhr würde die Position 3 mit einer Skala von 12 übermittelt, für 15 Minuten die Position 15 mit einer Skala von 60. Die jeweiligen Zeigerpositionen für 3 Uhr bei einer Skala von 12 und für 15 Minuten bei einer Skala von 60 sind also identisch. Die Funktion `getCursorPositionDegrees` ermittelt entsprechend der verwendeten Skala die relative Position des Zeigers.

1.5 Hauptroutinen der Uhr

Um unsere Uhr nun auch zu zeichnen, verwenden wir, wie bereits erwähnt, `pygame`.

Die folgenden Listings zeigen die Hauptroutinen, die Eventhandling und Bildschirmaktualisierung realisieren.

```
def handleEvents():
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit(0)
        elif event.type == pygame.KEYDOWN:
            sys.exit(0)
        elif event.type == pygame.MOUSEBUTTONDOWN:
            sys.exit(0)
```

Listing 1.5 Eventhandling der Uhr

Die Funktion `handleEvents()` reagiert auf jegliche Maus- und Tastatureingaben mit dem Beenden des Programms. Das hat den Vorteil, dass man die Uhr auch im `pygame.FULLSCREEN`-Modus laufen lassen kann, ohne das Programm über den Task-Manager beenden zu müssen. Das Event `pygame.QUIT` signalisiert einfach das Schließen des Fensters über einen der üblichen Wege.

Im Anschluss folgt die Haupteinstiegsroutine unserer Uhr-Applikation. In ihr wird `pygame` initialisiert und das Fenster für die Darstellung geöffnet.

```
def main():
    # Initialisierung des Screens
    pygame.init()
    screen = pygame.display.set_mode(\
     (windowWidth, windowHeight)\
     ,pygame.HWSURFACE | pygame.DOUBLEBUF);
    pygame.display.set_caption('Analog Clock');

    # Endlosschleife für den Hauptablauf der Uhr
    while True:
        handleEvents()
        screen.fill(backgroundColor)

        drawBackground()
        drawCurrentTime()
        drawForeground()

        pygame.display.flip()
```

```
        pygame.time.delay(10)

if __name__ == '__main__':
    main()
```

Listing 1.6 Der Hauptablauf für die Uhr

Auch wenn Sie nur eine Uhr darstellen, so lohnt es sich dennoch, die Optionen `pygame.HWSURFACE` und `pygame.DOUBLEBUF` zu verwenden, denn die Zeiger sollen kontinuierlich gezeichnet werden und nicht flackern. Die Optionen sorgen für ein hardwarebeschleunigtes Display, das doppelt gepuffert ist. Das heißt, dass zunächst auf einem zweiten Surface gezeichnet wird und dann der Speicherinhalt »in einem Rutsch« mittels `pygame.display.flip()` umgeschaltet wird. Mit »kontinuierlich« meine ich eine flüssige Bewegung der Zeiger, im Gegensatz zu springenden Zeigern. Wer möchte, kann natürlich gerne auch springende Zeiger darstellen, das ist reine Geschmackssache. Springende Zeiger bieten auch noch Raum für ein paar Specials, wie zum Beispiel leichte »Überschwinger«, das heißt leichtes Weiterschwingen des Sekundenzeigers beim Sprung zur nächsten Sekunde. Am Ende des nächsten Abschnitts gebe ich Ihnen einen Hinweis, wie Sie das realisieren könnten.

1.6 Zeichnen der Uhr in 2D

Die ersten beiden Funktionen für das Zeichnen der Uhr sind `drawBackground` und `drawForeground`, die wie folgt definiert sind:

```
def drawBackground():
    screen.fill(backgroundColor)
    pygame.draw.ellipse(screen, clockMarginColor, (windowMargin,\
     windowMargin, windowWidth-2*windowMargin,\
     windowWidth-2*windowMargin))
    pygame.draw.ellipse(screen, clockBackgroundColor,\
     (windowMargin+clockMarginWidth/2,\
     windowMargin+clockMarginWidth/2,\
     windowWidth-(windowMargin+clockMarginWidth/2)*2,\
     windowWidth-(windowMargin+clockMarginWidth/2)*2))

def drawForeground():
    pygame.draw.ellipse(screen, clockMarginColor,\
     (windowWidth/2.0-9, windowHeight/2.0-9, 18, 18))
```

Listing 1.7 Das Zeichnen der Uhr

Die Variablen im Listing 1.7 sind in Abschnitt 1.3 erläutert. Der Aufruf der Funktion `pygame.draw.ellipse` zeichnet eine Ellipse innerhalb des durch die vier Punkte definierten Vierecks. Da wir die Punkte so wählen, dass dieses Viereck ein Quadrat ist, erhalten wir entsprechend einen Kreis. Der Einfachheit halber zeichnen wir zunächst eine ausgefüllte Ellipse und darauf eine kleinere Ellipse, die ebenfalls ausgefüllt ist, und zwar mit der Hintergrundfarbe. Sie könnten auch die Funktion `pygame.draw.circle()` zum Zeichnen eines Kreises verwenden, aber mir kam die Art der Definition für Ellipsen hier sehr gelegen, da nur das umgebende Viereck zu bestimmen ist.

Die Koordinaten für das die Ellipse umgebene Viereck ermitteln sich dabei entsprechend aus den jeweiligen Abständen und Breiten. Der Rand zwischen der Uhr und dem Fensterrahmen wird durch die Variable `windowMargin` festgelegt. Die Koordinaten für das Zeichnen der Ellipse müssen den oberen linken Eckpunkt des die Ellipse umschließenden Vierecks sowie die Breite und Höhe beschreiben. Der Rand zwischen Uhr und Fensterrahmen soll horizontal und vertikal identisch sein, weshalb der obere linke Eckpunkt des Vierecks sich als Punkt P_1 = (windowMargin, windowMargin) darstellt. Damit die Uhr mittig im Fenster platziert wird, muss auch der Abstand nach unten und nach rechts identisch sein. Hierfür ermitteln wir die Breite entsprechend, indem wir `windowMargin` doppelt von `windowWidth` abziehen.

Die verschiedenen Zeiger zeichnen wir mit der gleichen Funktion. Darum wählen wir geeignete Parameter, die für unterschiedliche Zeigerbreite und -länge sorgen.

```
def drawCursor(color, width, length, position, scale):
    end = getCirclePoint(position, scale, length);
    pygame.draw.line(screen, color, windowCenter, end, width)
```

Listing 1.8 Die Zeiger zeichnen

Die Parameter sind weitgehend selbsterklärend und wurden zum Teil schon früher im Zusammenhang mit der Funktion `getCirclePoint()` in Abschnitt 1.4 beschrieben. Die Parameter `width` und `length` definieren, wie ihr Name schon verrät, die Zeigerbreite und Zeigerlänge. Die Funktion `pygame.draw.line` zeichnet eine Linie mit der durch `color` definierten Farbe vom Ausgangspunkt `windowCenter` zum Endpunkt `end`. Die Punkte werden durch Tupel aus zwei Werten für x und y repräsentiert. Der letzte Parameter gibt die Linienstärke an.

Nun fehlt nur noch die Darstellung der aktuellen bzw. virtuellen Uhrzeit. Dafür ist die Methode `drawCurrentTime()` zuständig, die wie folgt aufgebaut ist:

```
def drawCurrentTime():
    if useVirtualTimer:
        global hour, minute, second, micro
        timeGoesOn()
    else:
        now     = datetime.datetime.now()
        micro   = now.microsecond
        hour    = now.hour
        minute  = now.minute
        second  = now.second

    drawCursor(hourColor, 15,
     hourCursorLength,
     hour+minute/60.0, 12)
    drawCursor(minuteColor, 8,
     minuteCursorLength,
     minute+second/60.0, 60)
    drawCursor(secondColor, 3,
     secondCursorLength,
     second+micro/1000000.0, 60)
```

Listing 1.9 Die aktuelle Uhrzeit visualisieren

Zunächst wird überprüft, ob wir die virtuelle Zeit verwenden. Falls dem so ist, so wird jeweils vor dem Zeichnen die Zeit durch `timeGoesOn()` fortgeschrieben. Ansonsten wird die Systemzeit abgefragt. Im Anschluss wird die aktuell zu verwendende Zeit grafisch dargestellt, wofür die einzelnen Zeiger gezeichnet werden.

Um die Uhr im Anfangsstadium besser testen zu können, nutzen Sie den bereits erwähnten virtuellen Timer, so dass aus Stunden wenige Sekunden werden können und unsere Uhr sich fast überschlägt.

```
hour    = 0
minute  = 0
second  = 0
micro   = 0
def timeGoesOn():
    global hour, minute, second, micro
    micro += virtualSpeed
    if micro >= 2: # halve seconds - not micro seconds
        second += 1
        micro %= 2
    if second > 60:
        minute += 1
        second %= 60
```

```
if minute > 60:
    hour += 1
    minute %= 60
if hour > 12:
    hour %= 12
```

Listing 1.10 Unser virtueller Timer

In der Funktion `drawCurrentTime()` aus Listing 1.9 wird für jeden Zeiger die jeweils kleinere Zeiteinheit mit eingerechnet. Im folgenden Listing sehen Sie die entsprechenden Stellen noch einmal im Quelltext:

```
drawCursor(hourColor, 15,
 hourCursorLength,
 hour+minute/60.0, 12)
drawCursor(minuteColor, 8,
 minuteCursorLength,
 minute+second/60.0, 60)
drawCursor(secondColor, 3,
 secondCursorLength,
 second+micro/1000000.0, 60)
```

Listing 1.11 Aufrufe zum Zeichnen der Zeiger

[+] Teilung durch Float-Werte

Um bei der Division die Umwandlung in einen Integer zu vermeiden, wird hier durch Float geteilt. Python sorgt somit dafür, dass die genauere Zahlendarstellung für das Ergebnis verwendet wird. Ohne das Teilen durch einen Float-Wert würde eine Integer-Division stattfinden, deren Ergebnis ebenfalls ein Integer ist, dessen Wert dem ganzzahligen Anteil des Ergebnisses entspricht.

Die jeweils kleinere Einheit – zum Beispiel beim Stundenzeiger die Minuten – wird bei der Darstellung einbezogen. Dazu teilen wir einfach die aktuelle Anzahl der Minuten durch 60 und addieren das Ergebnis zu den Stunden. Analog dazu rechnen wir die Sekunden mit in die Minuten ein.

Durch das Einbeziehen der nächstkleineren Einheit laufen die Zeiger kontinuierlich, anstatt zu springen. Wem springende Zeiger besser gefallen, der erreicht das Gewünschte leicht, indem er die nächstkleinere Einheit nicht anteilig in die Zeigerposition aufnimmt. Akzeptabel ist das allerdings nur für den Sekunden- und Minutenzeiger; den Stundenzeiger lässt man in der Regel nicht springen. Um den Sprung schöner darzustellen, können Sie einen »Überschwinger« animieren. Dazu zeichnen Sie den Zeiger kurz vor dem Sprung kontinuierlich, wobei Sie einen Punkt etwas über dem eigentlichen Ziel anpeilen müssen. Es gilt also, eine

gute Animationszeitspanne zu finden und dann etwa 50 Prozent über das Ziel hinauszuschießen.

1.7 Eine schickere Uhr im 3D-Raum

Alles in allem kann man von unserer Uhr jetzt zwar die Zeit ablesen, aber wirklich schön ist sie nicht gerade. Deshalb zeige ich Ihnen jetzt, wie Sie mittels direkter Verwendung der OpenGL-API eine Uhr entwickeln, die schickere Zeiger und geglättete Kanten hat. Da wir dabei auch im 3D-Raum zeichnen, können wir die Uhr auch rotieren lassen.

Den kompletten Quelltext zu diesem Beispiel finden Sie auf der CD unter *Kapitel01/openGLClock.py*. [●]

OpenGL steht für »Open Graphics Library« und ist eine plattform- und programmiersprachenunabhängige Programmierschnittstelle zur Darstellung von 2D- und 3D-Grafiken. OpenGL bietet etwa 250 Befehle zur Darstellung komplexer 3D-Szenen in Echtzeit. Ein großer Vorteil und Hauptgrund für die weite Verbreitung von OpenGL ist die absolute Unabhängigkeit der Spezifikation von einzelnen Herstellern. Erweiterungen werden vom OpenGL ARB (Architecture Review Board) überprüft und gegebenenfalls dem Standard hinzugefügt. Nähere Informationen zu OpenGL finden Sie unter *http://www.opengl.org*.

Das Grundprinzip der Uhr verändert sich natürlich nicht wesentlich, nur weil sie jetzt mit OpenGL gezeichnet wird. Und auch das Zeichnen der Uhr mit OpenGL ist an sich nicht schwer. Wenn Sie sich etwas detaillierter mit OpenGL auseinandersetzen möchten, dann finden Sie im Anhang eine kurze Einleitung, in der sowohl *Winding* als auch *Culling* und andere noch auftauchende Begriffe näher erläutert werden.

Bevor wir jetzt aber mit OpenGL so richtig loslegen, müssen wir zunächst ein bisschen Initialisierungsarbeit leisten. Die erste Frage lautet wieder einmal: Was müssen wir importieren?

```
from OpenGL.GL import *
from OpenGL.GLU import *
import pygame, math, datetime
from pygame.locals import *
```

Listing 1.12 All you need is ...

Wie auch schon bei der Darstellung im 2D-Raum benötigen wir noch die Bibliotheken `datetime` und `math`, um die Systemzeit auszulesen und den Sinus und Kosinus für den Winkel zum aktuell darzustellenden Zeiger in Koordina-

ten umzurechnen. Darüber hinaus importieren wir die für OpenGL notwendigen Bibliotheken.

Die eigentliche Initialisierung unterscheidet sich nicht grundlegend von der bereits bekannten Initialisierung für die 2D-API. Die Hauptroutine unserer 3D-Uhr sieht wie folgt aus:

```
def main():
    video_flags = OPENGL|HWSURFACE|DOUBLEBUF
    screenSize  = (600, 600)

    pygame.init()
    pygame.display.gl_set_attribute(pygame.GL_MULTISAMPLEBUFFERS, 1)
    pygame.display.gl_set_attribute(pygame.GL_MULTISAMPLESAMPLES, 4)
    pygame.display.set_mode(screenSize, video_flags)

    resize(screenSize)
    init()

    frames = 0
    ticks = pygame.time.get_ticks()
    while True:
        event = pygame.event.poll()
        if event.type == QUIT or\
         (event.type == KEYDOWN and\
          event.key  == K_ESCAPE):
            break

        draw()
        pygame.display.flip()
        frames = frames+1

    print "fps:  %d" % ((frames*1000)/(pygame.time.get_ticks()-ticks))

if __name__ == '__main__': main()
```

Listing 1.13 Die Hauptroutine

Neu hinzugekommen ist das Flag `pygame.OPENGL`. Es wird hier im Quelltext verkürzt dereferenziert, was aufgrund des Imports mit Hilfe von `from pygame.locals import *` möglich ist. Von besonderer Bedeutung sind die zwei Zeilen, die die Optionen `pygame.GL_MULTISAMPLEBUFFERS` und zusätzlich `pygame.GL_MULTISAMPLESAMPLES` setzen. Diese Zeilen bewirken eine sehr glatte und damit auch viel schönere Darstellung der Uhr. Dass im Quelltext vier Samples angegeben sind, ergab sich einfach durch Testen. Probieren Sie ruhig auch einmal zwei Samples aus – Sie werden feststellen, dass die Glättung dann nicht so schön

aussieht. Ein größerer Wert als `4` ist jedoch unnötig – glatter als glatt ist einfach nicht möglich. Die Zeile danach setzt den gewählten Modus.

Die weiter unten stehenden Endlosschleife `while True` in Listing 1.13 kümmert sich um das Eventhandling und Auffrischen des Bildschirms. Nebenbei werden noch die *Frames per Second* mitgezählt und beim Beenden des Programms ausgegeben. Für dieses einfache Beispiel wäre die Ermittlung und Ausgabe der Bilder pro Sekunde natürlich nicht notwendig. Allerdings können Sie so bei eigenen Projekten, welche aufwendigere Szenarien rendern müssen, leicht feststellen, ob die Performance der Hardware für die Darstellung noch ausreicht.

Die Methode `resize()` erstellt passend zur Fenstergröße einen entsprechenden OpenGL-Kontext und sieht wie folgt aus:

```
def resize((width, height)):
    glViewport(0, 0, width, height)
    glMatrixMode(GL_PROJECTION)
    glLoadIdentity()
    glOrtho(-0.2, 4.2, 4.2, -0.2, -6.0, 0.0)
    glMatrixMode(GL_MODELVIEW)
    glLoadIdentity()
```

Listing 1.14 Initialisierung der Perspektive

`glViewport()` setzt das aktuelle Betrachtungsfenster auf den 3D-Raum und legt damit den Teilabschnitt der Matrix fest, der gerendert wird. Die Zeilen danach wählen den Modus und laden die darzustellenden Inhalte des Teilbereichs. Falls Sie OpenGL noch nicht kennen, möchte ich Sie für detailliertere Informationen erneut auf den Anhang verweisen. An dieser Stelle empfiehlt es sich, dort auch gleich nachzulesen, falls etwas zu verwirrend erscheint. Zugunsten einer kompakten Darstellung verzichte ich hier auf die Erklärung sämtlicher Zusammenhänge. Um den Quelltext zusammenzufassen: Zunächst wird die Projektionsmatrix geladen, durch eine Parallelprojektionsmatrix ersetzt und danach die Modelmatrix aktiviert, um in ihr zeichnen zu können.

Doch kommen wir nun noch zur `init()`-Methode:

```
def init():
    glShadeModel(GL_SMOOTH)
    glClearColor(0.0, 0.0, 0.0, 0.0)
    glClearDepth(1.0)
    glEnable(GL_DEPTH_TEST)
    glDepthFunc(GL_LEQUAL)
    glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST)
```

Listing 1.15 Initialisierung von OpenGL

`glShadeModel(shadeModel)` bestimmt die Art der Farbfüllung der gezeichneten Inhalte. Dabei sorgt `GL_SMOOTH` für einen Farbverlauf entsprechend der einzelnen Farben je Punkt, und `GL_FLAT` erstellt eine einfarbige Füllung mit der Farbe des letzten gesetzten Punkts je Grafikprimitiv. `glClearColor(rot, grün, blau, Transparenz)` setzt die Farbe, die zum Löschen der Ansicht dient, im Prinzip also die Hintergrundfarbe.

`glClearDepth(1.0)` bestimmt den Wert für die Löschung des Tiefenpuffers, das heißt, dieser Wert wird gesetzt, wenn der Tiefenpuffer gelöscht wird. `glEnable(GL_DEPTH_TEST` aktiviert den Tiefentest, der dafür sorgt, dass weiter hinten liegende, verdeckte Objekte unabhängig von der Zeichenreihenfolge nicht sichtbar sind. Auf den Tiefenpuffer könnte man verzichten, wenn man die Zeichenreihenfolge so wählt, dass die weiter vorn liegenden Inhalte zum Schluss gezeichnet werden. Über `glEnable(Option)` lassen sich außerdem viele andere Optionen aktivieren. Die Deaktivierung erfolgt bei Bedarf einfach durch den Aufruf von `glDisable(Option)`. Durch `glDepthFunc(Option)` wird die Art der Tiefenprüfung bestimmt. `GL_LEQUAL` lässt alle kleineren und gleich großen Werte die Prüfung bestehen. Was die Tiefenprüfung nicht besteht, ist auch nicht sichtbar und muss entsprechend nicht gezeichnet werden. Der Aufruf von `glHint()` mit den Parametern `GL_PERSPECTIVE_CORRECTION_HINT` und `GL_NICEST` sorgt für die realistischste perspektivische Interpolation der Farben.

1.8 Zeichnen mit OpenGL

Wie auch bei der 2D-Uhr zeichnen wir zunächst einen Ring. Um diesen Ring, wie auch jede andere Art von Linie, in OpenGL zu zeichnen, verwenden wir einen *Triangle-Strip*. Das ist ein aus Dreiecken bestehender Streifen.

In Abbildung 1.6 sehen Sie einen Triangle-Strip. Er sieht zwar noch nicht ganz wie ein Ring aus, aber auf genau die Art und Weise zeichnen wir gleich einen Ring.

Die Reihenfolge, in der die Punkte für die einzelnen Dreiecke oder auch Polygone angegeben sind, ist nicht, wie vielleicht vermutet, vollkommen egal. OpenGL wertet die Punktreihenfolge insofern aus, als es Polygone, die im Uhrzeigersinn gezeichnet wurden, als von hinten dargestellt annimmt. Diese Auswertung der Punktreihenfolge wird als *Winding* bezeichnet. Für Polygone, die gegen den Uhrzeigersinn definiert wurden, gilt das Gegenteil. Auch hierzu erfahren Sie Näheres im Anhang A, »OpenGL-Kurzreferenz«.

In OpenGL ist die kleinste Einheit ein Punkt im 3D-Raum. Mehrere Punkte lassen sich als Flächen darstellen, die mit Farbe gefüllt werden können. Die Füllung

kann, wie bereits bei der Erläuterung von `glShadeModel()` erwähnt, einfarbig oder ein Farbverlauf sein. Zeichenoperationen finden in Blöcken statt, die jeweils auf bestimmte Formen spezialisiert sind.

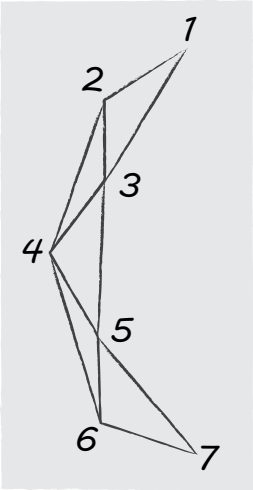


Abbildung 1.6 Ein Triangle-Strip – die Zahlen stellen die Reihenfolge der Punkte dar

Je mehr Dreiecke wir für den Triangle-Strip verwenden, desto glatter ist der Ring. Die Zeiger können wir direkt als Dreiecke darstellen, denn spitze Zeiger sehen sowieso besser aus – okay, das ist wieder Geschmackssache.

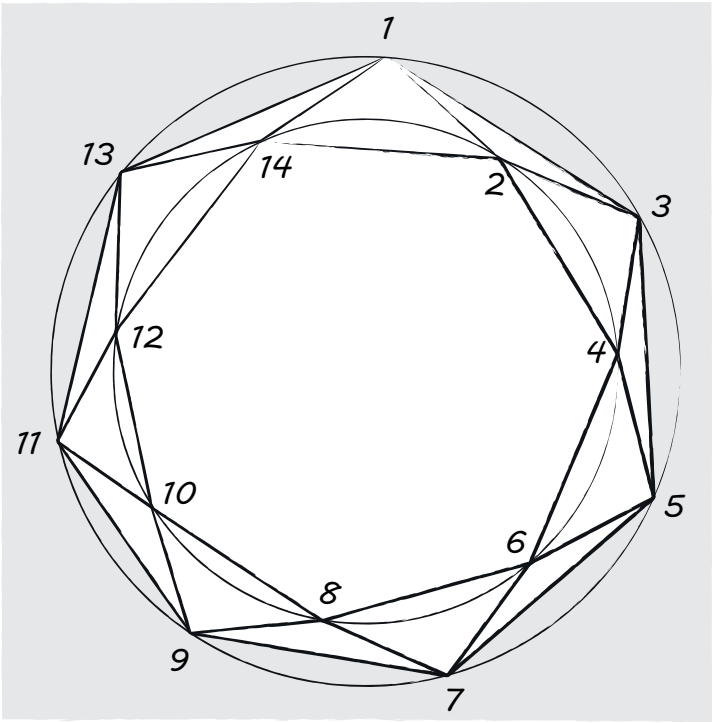


Abbildung 1.7 Ein Ring mittels Triangle-Strip

Wie definiert man nun einen solchen Streifen aus Dreiecken? Ganz einfach: Sie geben mehrere Punkte in Folge an, wobei ab dem zweiten Punkt jeder weitere Punkt mit den beiden vorherigen ein neues Dreieck bildet. Also ist ein Triangle-Strip nur eine Auflistung von Punkten, die entsprechend der gerade genannten Bedingung in Dreiecke umgewandelt werden. Im folgenden Listing 1.16 sehen Sie die Verwendung von `GL_TRIANGLE_STRIP`, wobei der Block durch Aufrufe von `glBegin(ZeichenModus)` und `glEnd()` eingeschlossen wird.

```
def drawRing(radius, width, depth):
    outerRadius = radius+width/2
    innerRadius = radius-width/2

    numCircleVertices = 1000
    inner = True
    glColor4f(0.9, 0.4, 0.2, 1.0)
    glBegin(GL_TRIANGLE_STRIP)
    for i in range(numCircleVertices):
        angle = (i / float(numCircleVertices-2)) * 2 * math.pi;
        if inner:
            currentRadius = innerRadius
        else:
            currentRadius = outerRadius
        x = math.cos(angle) * currentRadius
        y = math.sin(angle) * currentRadius
        inner = not inner
        glVertex3f(x, y, depth)
    glEnd()
```

Listing 1.16 Einen Ring mit Hilfe von OpenGL zeichnen

Für unseren Triangle-Strip definieren wir zu Beginn der Methode zwei Variablen, die für den inneren und äußeren Radius unseres Rings stehen. Diese Radien ergeben sich aus dem gewünschten Radius des Rings sowie der Breite des Rings.

Die Variable `numCircleVertices` legt die Anzahl der für den Ring zu zeichnenden Dreiecke fest. Je mehr Dreiecke wir verwenden, desto glatter wird der Ring. Die definierte Anzahl von 1.000 Dreiecken ist mehr als genug.

In der Schleife wird jeweils der Winkel für den nächsten Punkt des aktuell zu zeichnenden Dreiecks ermittelt und immer abwechselnd ein Punkt außen und innen vom Ring gesetzt. Die Funktion `glVertex3f` setzt jeweils den aktuellen Punkt, wobei `3f` für einen Punkt bestehend aus drei Koordinaten im `float`-Format steht. Die drei Koordinaten stehen für die x-, y- und z-Achse des Koordinatensystems.

Die Punkte werden in einem `glBegin(GL_TRIANGLE_STRIP) ... glEnd()`-Block gesetzt. Durch den umgebenden Block ist für OpenGL klar, wie die Darstellung

zu erfolgen hat. Zuvor wird noch die Farbe gesetzt, wobei `glColor4f()` vier `float`-Werte erwartet, die für die Farbanteile von Rot, Grün und Blau sowie die Deckkraft stehen.

Farbe	Rotanteil	Grünanteil	Blauanteil
Schwarz	0.0	0.0	0.0
Rot	1.0	0.0	0.0
Grün	0.0	1.0	0.0
Gelb	1.0	1.0	0.0
Blau	0.0	0.0	1.0
Magenta	1.0	0.0	1.0
Zyan	0.0	1.0	1.0
Dunkelgrau	0.25	0.25	0.25
Hellgrau	0.75	0.75	0.75
Braun	0.60	0.40	0.12
Orange	0.98	0.625	0.12
Pink	0.98	0.04	0.7
Lila	0.6	0.4	0.7
Weiß	1.0	1.0	1.0

Tabelle 1.2 RGB-Anteile häufig genutzter Farben als Float-Werte

Da in diesem Beispiel alle Polygone undurchsichtig gezeichnet werden, könnten wir auch die Funktion `glColor3f(rot, grün, blau)` aufrufen, die standardmäßig `1.0` für die Deckkraft annimmt. Da der Datentyp für den Funktionsaufruf grundsätzlich wählbar ist, sind Sie hier nicht auf `float` angewiesen. Dennoch empfehle ich Ihnen, es bei `float` zu belassen, da OpenGL intern ebenfalls mit Floats rechnet und einen anderen Datentyp sehr häufig konvertieren müsste. Für Sie heißt das im Prinzip: Alles, was geht, mit `float` realisieren, da so keine Performanceeinbußen zu erwarten sind.

Für die einzelnen Parameter gilt ein Wertebereich von `0.0` bis `1.0`, wobei `1.0` für 100 Prozent des jeweiligen Anteils bzw. für die volle Deckkraft steht.

Nun müssen noch die Zeiger her:

```
def drawPointer(width, length, position, scale):
    angle = float(position)/scale*360.0

    glRotatef(+angle, 0.0, 0.0, 1.0)
    glBegin(GL_TRIANGLES)
```

```
    glVertex3f(-width/2, 0.0, 0.0)
    glVertex3f(+width/2, 0.0, 0.0)
    glVertex3f(0.0, -length, 0.0)

    glVertex3f(-width/2, 0.0, 0.0)
    glVertex3f(+width/2, 0.0, 0.0)

    if length > 0:
        glVertex3f(0.0, +0.2, 0.0)
    else:
        glVertex3f(0.0, -0.2, 0.0)

    glEnd()
    glRotatef(-angle, 0.0, 0.0, 1.0)
```

Listing 1.17 Zeiger zeichnen

Die Zeiger werden einfach als Dreieck dargestellt und immer auf 12 Uhr gemalt, denn diese Variante war sehr leicht umzusetzen, wie Sie gleich sehen werden. Um immer auf 12 Uhr zeichnen zu können, muss vorher einfach alles entgegengesetzt zum Zeigerwinkel rotiert werden. Nach dem Zeichnen machen wir dann die Rotation rückgängig, und schon steht der Zeiger an der korrekten Stelle. So ersparen wir es uns, nicht nur den Endpunkt des Zeigers zu ermitteln, sondern auch die Endpunkte des senkrecht zum Zeiger stehenden Dreiecks. Durch die Rotation müssen wir überhaupt keinen Endpunkt mit Hilfe der Winkelfunktionen errechnen.

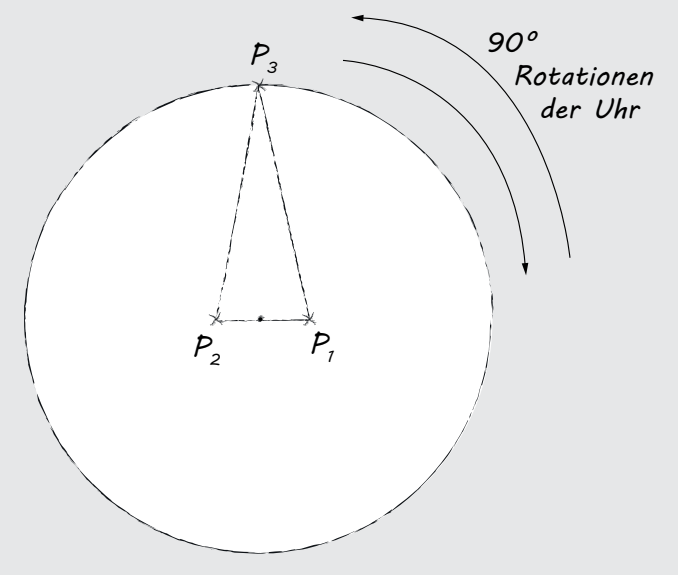


Abbildung 1.8 Rotation für die vereinfachte Zeigerdarstellung

Stellen Sie sich vor, dass Sie auf einem Blatt Papier eine Uhr zeichnen wollen, zum Einzeichnen des Stundenzeigers für 3 Uhr einfach das Blatt um 90 Grad gegen den Uhrzeigersinn rotieren und den Stundenzeiger für 12 Uhr eintragen. Im Anschluss drehen Sie das Blatt einfach wieder um 90 Grad, allerdings diesmal im Uhrzeigersinn, und schon haben Sie wie geplant den Stundenzeiger für 3 Uhr eingezeichnet. Für das Zeichnen auf Papier ist das sicher nicht die geeignetste Vorgehensweise, aber für die Darstellung der Uhr mittels OpenGL bringt dieser Ansatz eine klare Vereinfachung. Uns bleibt also eine Menge Arbeit erspart, wenn wir immer die gleichen Punkte zum Eintragen des Zeigers verwenden können und den Rest den zwei Rotationen überlassen. Abbildung 1.8 stellt den Sachverhalt noch einmal grafisch dar.

In dem `GL_TRIANGLES`-Block werden jeweils drei aufeinanderfolgende Punkte zu einem Dreieck zusammengefasst. Dabei hat allerdings jedes Dreieck seine eigenen drei Punkte, anders als bei `GL_TRIANGLE_STRIP`, wo jeweils die zwei letzten Punkte mit dem neu dazugekommenen ein Dreieck bilden. Um die Zeiger schöner darzustellen, zeichnen wir noch ein kleines Dreieck in entgegengesetzter Richtung. Das Rotieren erspart uns somit die Berechnung von Sinus und Kosinus für die jeweiligen Uhrzeiten.

Um die Zeit darzustellen, wird die Methode `drawCurrentTime()` aufgerufen:

```
def drawCurrentTime(hour, minute, second, micro):
    glColor4f(0.3, 0.7, 0.2, 1.0)
    drawPointer(0.20, 1.0,   hour+(minute/60.0), 12)
    glColor4f(0.3, 0.9, 0.5, 1.0)
    drawPointer(0.15, 2.0, minute+(second/60.0), 60)
    glColor4f(0.8, 0.4, 0.8, 1.0)
    drawPointer(0.08, 2.0, second+(micro/1000000.0), 60)
```

Listing 1.18 Die aktuelle Zeit darstellen

Die Farben für die einzelnen Zeiger sind diesmal nicht global definiert, sondern werden direkt im Aufruf von `glColor4f()` definiert.

1.9 Ein Ziffernblatt für die Uhr

Was jetzt noch fehlt, ist eine klare Markierung der Stunden und Minuten – kurz: ein Ziffernblatt. Zunächst reicht uns aber eine Markierung, so dass wir auf Ziffern verzichten. Das Einzeichnen der Markierungen verhält sich analog zum Darstellen der Zeiger, nur mit dem Unterschied, dass die Markierungsstriche nicht im Mittelpunkt der Uhr beginnen.

```
def drawClockFace():
    for hour in range(12):
        angle = float(hour)/12*360.0
        glColor4f(0.4, 0.7, 0.9, 1.0)
        glRotatef(+angle, 0.0, 0.0, 1.0)
        glBegin(GL_QUADS)
        glVertex3f(-0.01, 1.5, 0.0)
        glVertex3f(+0.01, 1.5, 0.0)
        glVertex3f(+0.05, 1.9, 0.0)
        glVertex3f(-0.05, 1.9, 0.0)
        glEnd()
        glRotate(-angle, 0.0, 0.0, 1.0)
```

Listing 1.19 Die Stundenmarkierungen zeichnen

Beim Zeichnen bedienen wir uns wieder der Rotation, wie Sie es schon vom Zeichnen der Zeiger her kennen. Wir rotieren also auch hier zunächst gegen den Uhrzeigersinn, tragen die Markierung auf 12 Uhr ein und machen die Rotation wieder rückgängig. Die Ziffernblattmarkierung für 3 Uhr wird entsprechend mit einer Rotation um -90 Grad gestartet. Dass die Uhr nur für das Einzeichnen der Markierungen zu den entsprechenden Stunden verwendet wird, sehen Sie gut an den beiden sich aufhebenden Aufrufen von `glRotate()`.

```
def drawClockFace():
    for hour in range(12):
        angle = float(hour)/12*360.0
        glColor4f(0.4, 0.7, 0.9, 1.0)
        glRotatef(+angle, 0.0, 0.0, 1.0)
        glBegin(GL_QUADS)
        glVertex3f(-0.01, 1.5, 0.0)
        glVertex3f(+0.01, 1.5, 0.0)
        glVertex3f(+0.05, 1.9, 0.0)
        glVertex3f(-0.05, 1.9, 0.0)
        glEnd()
        glRotate(-angle, 0.0, 0.0, 1.0)
```

Listing 1.20 Der Trick mit der Rotation

Anders als beim Zeichnen der Zeiger verwenden wir hier nicht `GL_TRIANGLES`, sondern `GL_QUADS`, da die Markierungen viereckig sein sollen, nicht dreieckig.

Nun fehlt noch ein ansprechender Hintergrund für das Ziffernblatt, den wir als einen `GL_TRIANGLE_FAN` zeichnen werden. Den Unterschied dieser beiden Grafikprimitive können Sie im Anhang zu OpenGL nachlesen. Dabei werden Sie die Wirkung von `glShadeModel(GL_SMOOTH)` erleben, denn im Quelltext wurde für den Mittelpunkt des Triangle-Fans eine andere Farbe gesetzt als für die

Randpunkte, denn deren Farben richten sich nach der aktuellen Uhrzeit; so ist gewährleistet, dass Ihre Uhr zu jeder Zeit ein andersfarbiges Ziffernblatt erhält. Der Rotanteil wird anhand der Sekunden, der Grünanteil anhand der Minuten und der Blauanteil anhand der Stunden festgegelegt.

```
def drawClockBackground(hour, minute, second):
    glBegin(GL_TRIANGLE_FAN)

    # center of the fan with last used color
    glVertex3f(0.0, 0.0, 0.0)

    numCircleVertices = 30
    clockRadius = 2.0
    for i in range(numCircleVertices):
        angle = (i / float(numCircleVertices-1)) * 2 * math.pi;
        x = math.cos(angle) * clockRadius
        y = math.sin(angle) * clockRadius

        # red, green, blue in relation to the current time
        glColor4f(second/60.0, minute/60.0, hour/12.0, 1.0)

        glVertex3f(x, y, 0.0)
    glEnd()
```

Listing 1.21 Das Ziffernblatt zeichnen

Abbildung 1.9 Das Ziffernblatt der Uhr

Dank der fixen Farbe für den Mittelpunkt wird zudem ein Farbverlauf aus dem Mittelpunkt zum Rand hinaus dargestellt. Möchten Sie lieber ein einfarbiges Ziffernblatt, so ändern Sie testhalber einfach das Shademodel durch den Aufruf von `glShadeModel(GL_FLAT)`. Danach wird statt eines Farbverlaufs die Farbe des zuletzt gesetzten Punktes einer Fläche als Füllfarbe für die gesamte Fläche verwendet.

1.10 Die Ansicht rotieren lassen

Bisher unterscheidet sich unsere neue Uhr kaum von der bereits in 2D gezeichneten Uhr. Allenfalls ist sie vielleicht aufgrund der Glättung und der spitzen Zeiger ein wenig schöner. Um hier ein bisschen mehr zu leisten, wollen wir die Uhr um die x- sowie um die y-Achse rotieren lassen. Als Auslöser für die Rotation nehmen wir die aktuelle Sekundenanzeige. Jeweils zur halben Minute wird sich die Uhr um die y-Achse drehen und zur vollen Minute um die x-Achse.

```
def clockRotate(second, micro, start, stop, rotation):
    x, y, z = rotation
    clockRotation = (second - start +
      micro/1000000.0)/(stop-start)*360.0
    if clockRotation <= 360:
        glRotate(clockRotation, x, y, z)
```

Listing 1.22 Die Uhr rotieren lassen

Die Methode zum Rotieren der Uhr ist sehr einfach: Es wird lediglich aufgrund des aktuellen Sekunden- und Mikrosekundenstandes der Grad der Drehung ermittelt und die Uhr dann rotiert. Dabei sind 360 Grad wie üblich eine volle Drehung. Die Rotation der Uhr beginnt jeweils bei 30 und 60 Sekunden und dauert 2 Sekunden an. Der Aufruf von `glRotate()` rotiert die komplette Ansicht um die Achsen, die nicht mit »0.0« angegeben sind. Die in Listing 1.22 definierte Methode `clockRotate()` wird nur aufgerufen, wenn sich die aktuelle Zeit innerhalb der Rotationszeit befindet, das heißt, zwischen 30 und 32 Sekunden sowie zwischen 0 und 2 Sekunden. Der aktuelle Rotationswinkel entspricht dem prozentualen Fortschritt im Rotationszeitraum. Der Winkel startet also bei 0 Grad, erreicht nach einer Sekunde 180 Grad und hat nach zwei Sekunden die volle Rotation abgeschlossen. Um die Rotation ohne Sprünge zu vollziehen, werden die Mikrosekunden in den entsprechenden Sekundenbruchteil umgerechnet und zu den Sekunden addiert; die Vorgehensweise kennen Sie ja bereits von den kontinuierlich dargestellten Zeigern.

Damit die Rotation tatsächlich auch sichtbar wird, müssen Sie darauf achten, dass nach der Rotation die Matrix nicht wieder in ihren Ausgangszustand zurückversetzt wird. Das heißt, alle folgenden Matrix-Transformationen sollten den Zustand der Matrix sichern und nach Vollendung wiederherstellen. Dies wird mittels `glPushMatrix()` und `glPopMatrix()` erreicht. Diese beiden Aufrufe werden zum Beispiel beim Zeichnen der Zeiger und des Ziffernblattes verwendet, um die Matrix nach der Rotation wiederherzustellen. Nähere Erläuterungen zu diesen Aufrufen finden Sie im Anhang A.4, »Die Matrix manipulieren«.

Alle notwendigen Zeichenoperationen werden gesammelt auf der Methode `draw()` aufgerufen.

```
def draw():
# Tiefenpuffer und Farbpuffer löschen
    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT)

    # Systemzeit auslesen
    now         = datetime.datetime.now()
    micro       = now.microsecond
    hour        = now.hour
    minute      = now.minute
    second      = now.second

    # Identitätsmatrix laden und Ansicht verschieben
    glLoadIdentity()
    glTranslatef(2.0, 2.0, 3.0)

# Drehung der Uhr
    if second >= 58:
        clockRotate(second, micro, 60.0, (1.0, 0.0, 0.0))
    elif 28 <= second <= 30:
        clockRotate(second, micro, 30.0, (0.0, 1.0, 0.0))

# Zeichnen der Uhr
drawClockFace()
    drawCurrentTime(hour, minute, second, micro)
    drawRing(2.0, 0.1, -0.1)
    drawRing(2.0, 0.1, +0.1)
```

Listing 1.23 Die Hauptroutine zum Zeichnen

Zu Beginn des Listings wird die Ansicht komplett gelöscht; wir beginnen also mit einem leeren Bildschirm, und alle Grafiken werden bei jedem Aufruf von `draw()` neu gezeichnet. Außerdem wird der Tiefenpuffer zurückgesetzt. Die `if`-Bedingung startet abhängig von der Zeit die entsprechende Rotation der Uhr.

Der Aufruf von `glLoadIdentity()` sorgt dafür, dass die ursprüngliche Ansicht auf unsere Uhr wiederhergestellt wird. Dementsprechend macht dieser Aufruf alle bis dahin erfolgten Translationen und Rotationen rückgängig. Der darauffolgende Aufruf führt sofort eine Translation aus, um die Uhr in der Ansicht zu zentrieren.

Ganz zum Schluss werden die Zeit sowie die Umrandung der Uhr dargestellt. Für den 3D-Raum habe ich hier zwei Ringe zeichnen lassen, die sich jeweils nach vorn bzw. nach hinten von der Uhr absetzen. Zu sehen sind diese beiden Ringe allerdings nur bei den Rotationen, da der Betrachter ansonsten aufgrund der Parallelprojektion, die mittels `glOrtho()` in Listing 1.14 erzeugt wurde, frontal auf die Uhr schaut.

Bei der Rotation werden Sie allerdings noch eine etwas unrealistisch wirkende Lücke zwischen den Ringen der Uhr bemerken. Als wir die Uhr noch nicht rotiert haben, war das natürlich kein Problem, aber jetzt fällt die Lücke unangenehm auf. Solche »Fehler« beim Perspektivenwechsel sind in der 3D-Programmierung nicht unüblich und müssen immer bedacht werden.

Abbildung 1.10 Die Rotation der Uhr mit frei schwebenden Ringen

Wir korrigieren diesen Fehler, indem wir einfach ein »Band« am Rand der Uhr zeichnen. Ein »Band« heißt in dem Fall wieder ein Triangle-Strip. Die Funktion `connectRings()` übernimmt diese Aufgabe und gestaltet sich wie folgt:

```
def connectRings(radius, depth1, depth2):
    numCircleVertices = 1000
    inner = True
    glColor4f(0.9, 0.4, 0.2, 1.0)
    glBegin(GL_TRIANGLE_STRIP)
    for i in range(numCircleVertices):
        angle = (i / float(numCircleVertices-2)) * 2 * math.pi;
        if inner:
            depth = depth1
        else:
            depth = depth2
        x = math.cos(angle) * radius
        y = math.sin(angle) * radius
        inner = not inner
        glVertex3f(x, y, depth)
    glEnd()
```

Listing 1.24 Das »Band« für die Uhr

Abbildung 1.11 Die Rotation der Uhr ohne Lücken

Die Parameter `depth1` und `depth2` geben die Werte für die Tiefe an, in der sich die beiden zu verbindenden Ringe befinden. Der Parameter `radius` repräsentiert den äußeren Radius der zu verbindenden Ringe. Anstelle des Wechsels zwischen innerem und äußerem Radius, wie Sie es schon vom Zeichnen der Ringe her kennen, wechseln wir nun zwischen vorderem und hinterem Ring. Der Radius

bleibt hierbei diesmal identisch. In Abbildung 1.11 sehen Sie die Auswirkung der neuen Funktion.

Falls es bezüglich der beim Aufruf verwendeten Parameter noch Verwirrung gibt, schauen Sie sich die Zusammenhänge noch einmal genau an.

```
drawRing(2.0, 0.1, -0.1)
drawRing(2.0, 0.1, +0.1)
connectRings(2.05, -0.1, +0.1)
```

Die Funktion `drawRing()` erwartet als zweiten Parameter die Breite des Rings. Der Ring an sich wird mittig zur Breite gezeichnet, weshalb dann entsprechend die Funktion `connectRings()` als Radius den um die Hälfte der Breite der Ringe erhöhten Radius erhält. Die Parameter für die Tiefe sind auf Anhieb wiederzufinden und bedürfen deshalb keiner weiteren Erklärung. Dass die Funktion `drawRing()` die Ringe plus bzw. minus der Hälfte der angegebenen Breite zeichnet, können Sie direkt aus den beiden ersten Zeilen der Funktion `drawRing()` ablesen:

```
outerRadius = radius+width/2
innerRadius = radius-width/2
```

1.11 Individuelle Beleuchtung

Wie so oft im Leben kommt der interessanteste Teil ganz zum Schluss – in diesem Fall die Beleuchtung mittels OpenGL. Bisher haben Sie alle Farben immer in der gleichen Intensität wahrgenommen, auch wenn die Uhr sich gedreht hat. Dies entspricht natürlich nicht der Realität, da die Intensität der Farben vom Reflexionswinkel des für unsere Perspektive sichtbaren Lichtes abhängt. Je direkter die Lichtquelle von dem von Ihnen betrachteten Objekt reflektiert wird, desto stärker sollten die Farben wiedergegeben werden. Da gerade bei der Rotation der Uhr dieser Effekt ganz deutlich zutage tritt, werden wir nun auch für die korrekte Beleuchtung sorgen.

In OpenGL ist Licht in drei Arten unterteilt: *ambient*, *diffuse* und *specular*. Dabei dient das »ambiente« Licht sozusagen der Grundausleuchtung, die alle Objekte unabhängig vom Einfallswinkel aufhellt. Es ist somit wie eine Anpassung des Gamma-Kanals zu verstehen, ganz nach dem Motto: Alles ein bisschen heller beziehungsweise dunkler bitte. Es eignet sich zum Beispiel für die Dämmerung in Spielen. Das diffuse Licht dagegen ändert die Intensität der wiedergegebenen Farben abhängig vom Reflexionswinkel des Lichtes, während das direkte Licht (specular) sehr punktuell aufhellend wirkt – sozusagen ein »Spotlight«. Die verschiedenen Lichtarten werden in der Regel nicht einzeln verwendet, sondern

treten immer in Kombination auf, wobei es letztendlich auf die richtige Mischung für den jeweils angestrebten Einsatzzweck ankommt.

Bis jetzt haben wir lediglich Farben festgelegt, was für die Beleuchtung von Flächen unzureichend ist, da die Reflexion des Lichtes nicht nur von der Farbe, sondern auch von den Materialeigenschaften abhängt. Dementsprechend werden Sie für die bis jetzt verwendeten Flächen auch Materialeigenschaften zuweisen, die für die Beleuchtung verwendet werden. Wenn Sie die Beleuchtung der Szene mittels `glEnable(GL_LIGHTING)` aktivieren, so wird die Uhr zunächst schwarz sein, da es weder Materialeigenschaften noch Lichtquellen gibt. Zudem ist noch unklar, wie das Licht reflektiert werden soll, da hierfür zusätzlich zu den Punkten *Normalen* anzugeben sind. Normalen sind senkrecht zu einer Fläche stehende Vektoren. Sie bilden das Lot für die Berechnung jeglicher Reflexionswinkel. Immer wenn ein Punkt angelegt wird, kann gleichzeitig eine Normale für diesen Punkt definiert werden.

Um Normalen zu erstellen, rufen Sie `glNormal3f(x, y, z)` mit dem Richtungsvektor als Parameter auf. OpenGL erwartet normalisierte Normalen, das heißt Normalen der Länge 1. Mittels `glEnable(GL_NORMALIZE)` können Sie die Normalisierung der Normalen von OpenGL automatisch durchführen lassen. Wir bräuchten im Prinzip diesen Aufruf nicht, da wir nur normalisierte Vektoren verwenden.

In folgendem Code wird nun eine Grundausleuchtung von 10 Prozent über alle Farben mittels `glLightfv(GL_LIGHT0, GL_AMBIENT, [0.1, 0.1, 0.1, 1.0])` definiert. Darauf folgen noch 90 Prozent winkelabhängige Beleuchtung durch `glLightfv(GL_LIGHT0, GL_DIFFUSE, [0.9, 0.9, 0.9, 1.0])` und die Positionierung der so definierten Lichtquelle mit `glLightfv(GL_LIGHT0, GL_POSITION, [0.0, 0.0, -6.0, 1.0])`. Der darauffolgende Aufruf aktiviert die Lichtquelle und macht im Zusammenhang mit den an anderen Stellen definierten Normalen die Beleuchtung perfekt.

```
glEnable(GL_LIGHTING)
glLightfv(GL_LIGHT0, GL_AMBIENT, [0.1, 0.1, 0.1, 1.0])
glLightfv(GL_LIGHT0, GL_DIFFUSE, [0.9, 0.9, 0.9, 1.0])
glLightfv(GL_LIGHT0, GL_POSITION, [0.0, 0.0, -6.0, 1.0])
glEnable(GL_LIGHT0)

glEnable(GL_COLOR_MATERIAL)
glColorMaterial(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE)
glEnable(GL_NORMALIZE)
```

Listing 1.25 Die Initialisierung für die Beleuchtung

Dass wir die Materialeigenschaften nicht explizit angeben müssen, verdanken wir dem sogenannten *Color Tracking*, das Aufrufe von `glColor()` zum Setzen von Materialeigenschaften auswertet. Um den Reflexionswinkel für das Licht zu bestimmen, müssen wir die Ausrichtung der Polygone kennen. Hierfür definieren wir einmalig eine Normale, die senkrecht zur Uhr platziert ist. Für fortgeschrittenere Anwendungen können Sie, wie bereits erwähnt, je Punkt eine Normale festlegen und somit eine sehr viel realistischere Beleuchtung bei nicht planaren Objekten erreichen.

```
    glNormal3f(0.0, 0.0, -1.0)

    if   0 <= second <= 10:
        clockRotate(second, micro,  0, 10, (1.0, 0.0, 0.0))
    elif 30 <= second <= 40:
        clockRotate(second, micro, 30, 40, (0.0, 1.0, 0.0))
```

Listing 1.26 Die Rotation in Abhängigkeit von der Zeit

In den Listings wurde jeweils ein Aufruf von `glNormal3f(0.0, 0.0, -1.0)` oder `glNormal3f(0.0, 0.0, +1.0)` mit aufgenommen, der die Normale auf der z-Achse definiert, und je nachdem, ob wir die Vorder- oder die Rückseite sehen, wird die Richtung des Vektors umgekehrt. Die Umkehrung erfolgt anhand der aktuellen Gradzahl der Rotation der Uhr und stellt so sicher, dass sowohl die Vorderseite als auch die Rückseite der Uhr korrekt ausgeleuchtet werden. Sie sehen die zwei verschiedenen Definitionen der Normalen beim Vergleich des vorherigen und des nächsten Listings jeweils **fett** hervorgehoben. Im Klartext heißt das: Jede Seite, die beleuchtet wird, benötigt auch eine Normale für die Fläche, damit sie sichtbar ist. Im Quelltext ist der Sachverhalt insofern vereinfacht, als die Normale immer senkrecht auf der gerade sichtbaren Fläche steht.

```
def clockRotate(second, micro, start, stop, rotation):
    x, y, z = rotation
    clockRotation =
     (second - start + micro/1000000.0)/(stop-start)*360.0

    # new handling for Lighting
    if 90 < clockRotation < 270:
        # invert the normal vector
        glNormal3f(0.0, 0.0, +1.0)

    if clockRotation <= 360:
        glRotate(clockRotation, x, y, z)
```

Listing 1.27 Die Rotation der Uhr

So weit zum Thema Beleuchtung mit OpenGL. Die Möglichkeiten sind hier natürlich bei weitem nicht erschöpft und lassen sich bei einem so einfachen Beispiel auch nicht demonstrieren. Probieren Sie einfach ein bisschen herum. Ergänzen Sie zum Beispiel die Szene um weitere Lichtquellen und eventuell auch um nicht planare Objekte. Dabei sollten Sie allerdings beachten, dass Sie dann mehr als eine Normale definieren müssen, um eine realistische und vor allem korrekt berechnete Beleuchtung zu erhalten. Zur Erinnerung: Die Normale soll den zum Polygon senkrecht stehenden Vektor kennzeichnen, um die Berechnung des Reflexionswinkels zu ermöglichen.

1.12 Das war schon alles?

Zugegebenermaßen ist die Uhr immer noch nicht der letzte Schrei. Was hier noch fehlt, wäre ein schöneres Ziffernblatt, vielleicht auch noch ein paar Komplikationen. Für das Ziffernblatt könnten Sie zum Beispiel ein Bild auf die Fläche mappen. Die Komplikationen stellen schon größere Hindernisse dar. Denkbar wäre zum Beispiel die Visualisierung der Mondphase und des Tages. Wenn Sie auch ein Automatikfan sind, dann könnten Sie etwa eine Gangreserve einbauen, die zum Beispiel die Laufzeit der Uhrenapplikation darstellen könnte. Wenn die Uhr dann tatsächlich zu einem Schmuckstück geworden ist, ließe sich die ganze Anwendung auch in einen Bildschirmschoner verwandeln. Eine einfache Alternative bliebe aber auch noch: Entwickeln Sie doch eine digitale Variante der Uhr. Vielleicht gefallen Ihnen ja digitale Uhren besser als analoge.

Ich bin sicher, Ihnen fallen noch viele andere Möglichkeiten ein – eine Uhr bietet doch mehr Spielraum für Kreativität, als man denkt. Wer besonders viel Begeisterung mitbringt, der könnte sich an einer Kuckucksuhr versuchen; das dürfte so ziemlich die anspruchsvollste Ausbaustufe darstellen. Für die diejenigen, die etwas kleinere Schritte bevorzugen, bietet sich auch die Bezifferung des Ziffernblatts an. Damit wäre dann auch der Name »Ziffernblatt« gerechtfertigt.

»Wer mit dem Leben spielt,
kommt nie zurecht;
wer sich nicht selbst befiehlt,
bleibt immer ein Knecht.«

Johann Wolfgang von Goethe

2 Game of Life

In diesem Kapitel werden Sie einen Klassiker nachprogrammieren – John Conways »Game of Life«. Das »Game of Life« ist ein von John Conway definiertes Modell einer in sich abgeschlossenen Welt. In dieser Welt leben Individuen, deren Überleben allein von der Gesellschaft abhängt. Lebewesen in guter Gesellschaft leben entsprechend lange, während vereinsamte Lebewesen sterben. Ein Anhäufung von zu vielen Individuen auf kleinem Raum ist jedoch ebenfalls nicht gesund und führt zum Tod aufgrund von Überbevölkerung. Zusammengefasst geht stellt das Modell genau zwei Eigenschaften in den Vordergrund. Zum einen das Streben nach sozialer Integration und gegenseitiger Unterstützung, und zum anderen den Mangel an Ressourcen, wenn zu viele Lebewesen auf zu kleinem Lebensraum aufeinandertreffen. Durch die beiden genannten Einschränkungen pegelt sich das Modell in der Regel relativ gut ein. Voraussetzung dafür ist jedoch eine ausgewogene Bevölkerungsdichte zu Beginn. Man sollte also nicht mit zu vielen, aber auch nicht mit zu wenigen Lebewesen starten.

Aber kommen wir nun zu einer etwas technischeren Betrachtung des Modells.

Es handelt sich dabei um einen einfachen zellulären Automaten im 2D-Raum, den Sie leicht abgewandelt später auch in 3D darstellen werden. Das Beeindruckende am »Game of Life« ist die Vielfalt an möglichen Abläufen, die unvorhersehbar auf dem Bildschirm sichtbar werden. Während der Entwicklung werden Sie weitere interessante zelluläre Automaten kennenlernen.

Alle grafischen Ausgaben werden Sie ab jetzt nur noch über OpenGL realisieren. Nähere Informationen zu OpenGL finden Sie im Anhang; falls Sie bereits das erste Kapitel, »Es schlägt 12«, gelesen haben, so werden Sie bezüglich OpenGL keine Überraschungen erleben.

2.1 Spiel des Lebens?

Das »Game of Life« wurde von dem englischen Mathematiker John Horton Conway entwickelt und ist als mathematisches Spiel zu betrachten. Conway ist außerdem für seine Arbeiten zur kombinatorischen Spieltheorie bekannt, die er in den Büchern »Zahlenzauber«, »Gewinnen: Strategien für mathematische Spiele« und »Über Zahlen und Spiele« zusammen mit Richard Kenneth Guy und Elwyn R. Berlekamp veröffentlicht hat.

Das »Game of Life« in der von Conway ursprünglich beschriebenen Darstellung ist ein zweidimensionaler zellulärer Automat. Ein zellulärer Automat modelliert ein räumlich diskretes dynamisches System, wobei die Zellzustände zum Zeitpunkt x von dem eigenen Zustand sowie dem Zustand der direkten Nachbarn zum Zeitpunkt x – 1 abhängen. So lautet die für Informatiker eingängige und absolut sinnvolle Erklärung. Ohne das Fachchinesisch heißt das einfach nur: Es gibt ein Spielfeld, das in Felder aufgeteilt ist. Diese Felder haben Zustände, die sich aus dem zeitlich direkt vorhergehenden Zustand der näheren Umgebung ermitteln lassen.

Im Verlaufe dieses Kapitels werden wir aber noch eine andere Form des »Game of Life« entwickeln, die nicht nur zweidimensional ist, sondern dreidimensional.

Am »Game of Life« lässt sich sehr schön erkennen, wie einfache Regeln über viele Zeitabschnitte zu komplexen unvorhersagbaren Vorgängen führen können. Diese ergeben sich durch den rekursiven Charakter des Automaten. Jeder Spielschritt lässt sich zwar mit einfachen Regeln berechnen, aber alle Spielschritte sind miteinander gekoppelt. Darum ist es auch nicht einfach möglich, vorherzusagen, ob eine gegebene Ausgangskonfiguration besonders lange zu einer hohen Spieldynamik verhilft. Mit »Spieldynamik« meine ich hier, dass sich die Zellzustände des Automaten möglichst stark und möglichst nicht periodisch ändern sollen – was soviel heißen soll, wie: der Spielfeldzustand soll sich nicht wiederholen. Eine weitere Beobachtung dieses Automaten lässt gewisse Strukturen erkennen, die scheinbar »intelligentes« Verhalten zeigen, da es zu Strukturen kommt, die einer gewissen Handlungsvorschrift folgen und dabei länger ihre Form behalten – diese Strukturen nennt man Oszillatoren. Das klingt sicher noch etwas abstrakt, darum schauen wir uns doch einfach an, was damit genau gemeint ist.

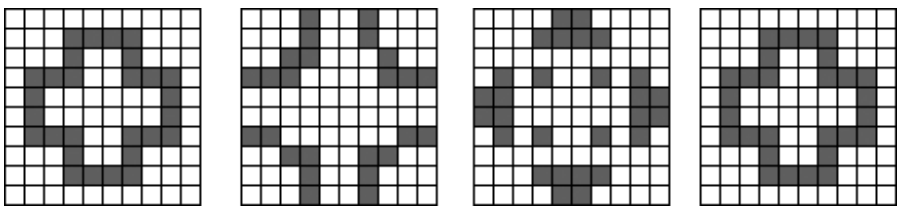


Abbildung 2.1 Periode-3-Oszillator

In Abbildung 2.1 sehen Sie einen Oszillator, der sich nach drei Iterationen wiederholt. Beim Durchlaufen der drei Phasen breitet er sich dabei zunächst aus, um beim Übergang in den Anfangszustand wieder in sich zusammenzufallen.

Eine andere interessante Erscheinung beim »Game of Life« sind die »Slider«, die – wie der Name schon andeutet – über den Bildschirm beziehungsweise durch den Lebensraum gleiten.

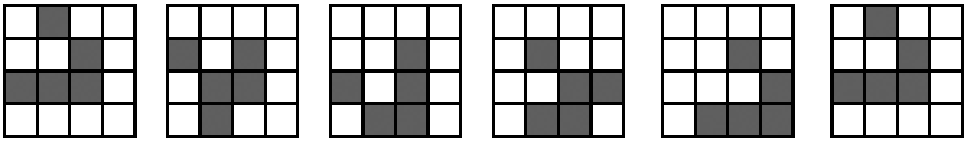


Abbildung 2.2 Gleiter

Die verschiedenen Formen des Gleiters treten in Kombination mit dem Gleiten über den Bildschirm auf. Da das »Game of Life« von uns in einer »Donut-Welt« – was genau man unter einer Donut-Welt versteht wird später noch erläutert – durchgeführt wird, treten Gleiter beim Verlassen des Bildschirms an einem Ende der Welt auf der gegenüberliegenden Seite der Welt wieder ein.

Ob ein Feld belebt ist oder nicht, ergibt sich aus dem vorherigen Zustand des Feldes und der Anzahl direkter Nachbarn. Grob gesagt sterben belebte Felder mit einer zu geringen Anzahl an Nachbarn aufgrund von Vereinsamung; belebte Felder mit einer zu hohen Anzahl an Nachbarn sterben hingegen aufgrund von Überbevölkerung.

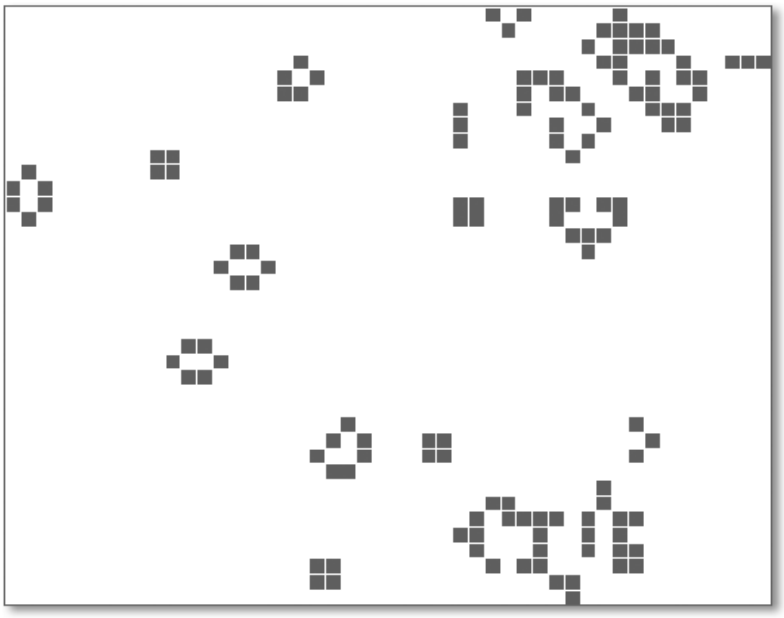


Abbildung 2.3 »Game of Life« in 2D

Bei einer adäquaten Anzahl von Nachbarn bleiben die belebten Felder am Leben. Nicht belebte Felder werden lebendig, wenn sie von einer fest definierten Anzahl von Nachbarn umgeben sind. Die genauen Bedingungen sehen wir uns in Abschnitt 2.3, »Die Regeln des Game of Life«, an. Im Folgenden beschäftigen wir uns zunächst verschiedenen Darstellungsvarianten des »Game of Life«.

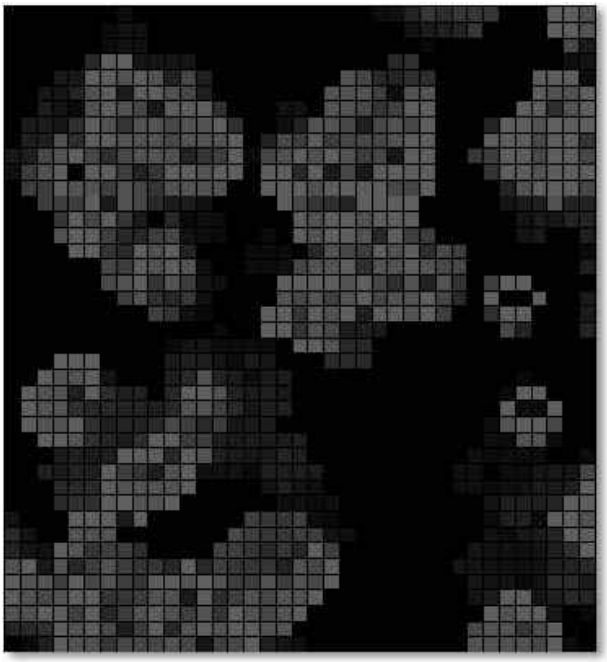


Abbildung 2.4 »Game of Life« in 2D mit »langsam sterbenden« Lebewesen

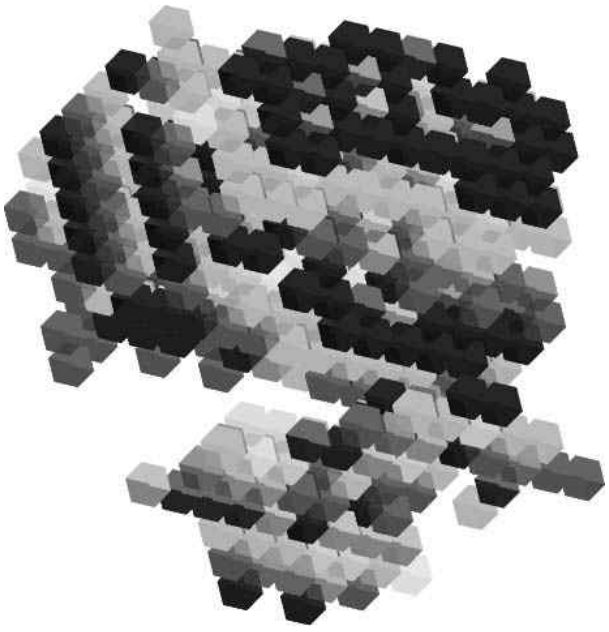


Abbildung 2.5 »Game of Life« in 3D

Wir werden mit der 2D-Variante starten, wofür wir zunächst nur eine zufällige Ausgangskonfiguration für die am Anfang als belebt geltenden Zellen erstellen müssen. Aufbauend auf diese Ausgangskonfiguration wird dann für jede Zelle des Spielfeldes anhand der Nachbarn der Folgezustand ermittelt. Im nächsten Schritt werden alle Zellen gleichzeitig auf den Folgezustand umgeschaltet und entsprechend dargestellt. Dann erweitern wir die 2D-Variante noch um ein kleines Gimmick: Wir lassen die Zellen langsam sterben, indem wir die toten Zellen schrittweise ausblenden. Als letzten Schritt werden wir eine 3D-Variante erstellen, die natürlich auch das Feature des langsamen Sterbens der einzelnen Zellen unterstützen wird. Aber kommen wir nun zunächst zur Repräsentation des Lebensraumes.

2.2 Der Lebensraum

Als Erstes benötigen die Lebewesen natürlich einen Lebensraum. Da wir uns zunächst auf zwei Dimensionen beschränken, reichen für den Lebensraum Variablen für Höhe und Breite. Dafür definieren wir zu Beginn unseres Programms die Variablen `livingSpaceWidth` und `livingSpaceHeight`.

Den Quelltext zu diesem Beispiel finden Sie auf der CD im Verzeichnis *Kapitel02/gameOfLife_v1.py*. [●]

```
livingSpaceWidth = 100
livingSpaceHeight = 60
```

Listing 2.1 Die Größe des Lebensraumes

Die Lebewesen werden im Lebensraum als Punkte dargestellt. Hierfür brauchen wir also ein Tupel aus x- und y-Koordinaten. Ob an einer bestimmten Stelle ein Lebewesen lebt oder nicht, halten wir in einer verschachtelten Liste fest. Bewohnte Felder werden mit einer `1` markiert, unbewohnte mit einer `0`. Ob ein Feld anfangs bewohnt oder unbewohnt ist, entscheidet der Zufall per Aufruf von `random.randint(0,1)`.

```
livingSpace = []
def initLivingSpace():
    for x in range(livingSpaceWidth):
        livingSpace.append([])
        for y in range(livingSpaceHeight):
            livingSpace[x].append(random.randint(0, 1))
```

Listing 2.2 Zufällige Initialisierung des Lebensraumes

2.3 Die Regeln des »Game of Life«

In dem im letzen Abschnitt definierten Lebensraum gelten natürlich, wie in jedem anderen Lebensraum auch, bestimmte Gesetzmäßigkeiten. Über Leben und Tod entscheiden beim »Game of Life« die folgenden einfachen Regeln:

- Ein nicht belebtes Feld mit genau drei belebten Nachbarfeldern wird in der Folgegeneration neu geboren.
- Belebte Felder mit weniger als zwei belebten Nachbarfeldern sterben in der Folgegeneration an Einsamkeit.
- Ein belebtes Feld mit zwei oder drei belebten Nachbarfeldern bleibt in der Folgegeneration belebt.
- Belebte Felder mit mehr als drei belebten Nachbarfeldern sterben in der Folgegeneration an Überbevölkerung.

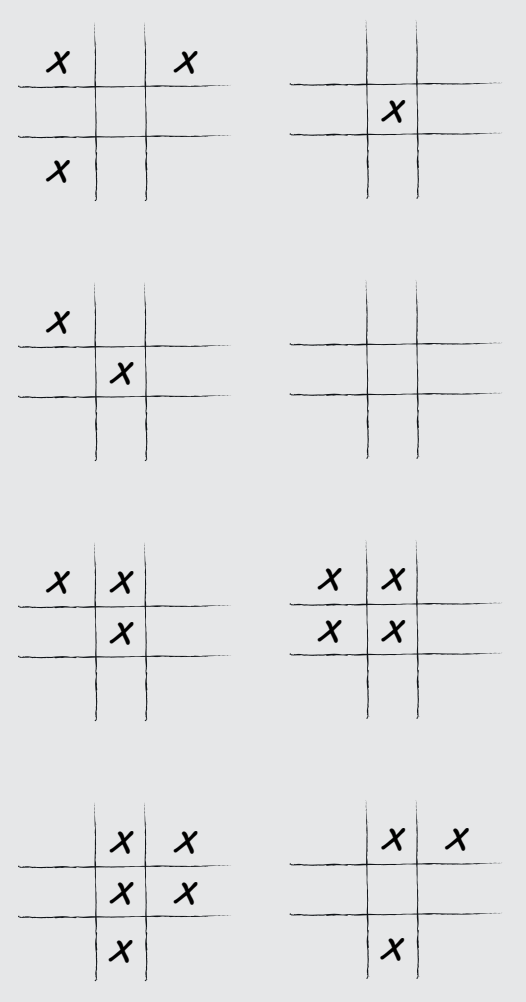


Abbildung 2.6 Die Regeln des »Game of Life«

Diese einfachen Regeln erlauben mit den richtigen Konstellationen sogar UND- und ODER-Operationen, so dass es möglich ist, mit diesem Automaten komplexe elektronische Schaltungen darzustellen. Per Zufall wird das zwar höchstwahrscheinlich nicht funktionieren, aber wenn Sie die Ausgangskonfiguration nicht zufällig erzeugen, sondern mit einem gesetzten Ziel, so ist sehr viel möglich. Die Grafiken in Abbildung 2.6 stellen jede Regel noch einmal anhand eines Beispiels dar. Die Ausgangsgeneration ist jeweils links dargestellt, gefolgt von der Folgegeneration rechts daneben. Die Reihenfolge der grafischen Beispiele entspricht der Reihenfolge der Erläuterung im Text. Die belebten Felder des Lebensraumes sind durch die Kreuze markiert; Felder ohne Kreuze sind unbelebt.

2.4 Darstellung des Lebensraumes

Für die Darstellung reichen in diesem Fall einfachste OpenGL-Mittel. Sie können hier auf Licht, Antialiasing, Rotation und Translation vollkommen verzichten. Sie müssen lediglich kleine Quadrate zeichnen, die Sie für die Darstellung der Lebewesen verwenden. Der Lebensraum wird sozusagen wie ein kariertes Blatt Papier dargestellt, wobei die belebten Felder farbig hervorgehoben werden. Der Lebensraum – die verschachtelte Liste – wird entsprechend einfach in Spalten und Zeilen auf dem Bildschirm dargestellt, wobei jedes Lebewesen als kleines Quadrat gezeichnet wird. In Listing 2.3 sehen Sie die Hauptablaufroutine.

```
def main():
    video_flags = OPENGL | HWSURFACE | DOUBLEBUF
    screenSize = (livingSpaceWidth * creatureSize,
     livingSpaceHeight * creatureSize)

    pygame.init()
    pygame.display.set_mode(screenSize, video_flags)

    initLivingSpace()
    resize(screenSize)
    init()

    frames = 0
    ticks = pygame.time.get_ticks()
    while True:
        event = pygame.event.poll()
        if event.type == QUIT or
         (event.type == KEYDOWN and
          event.key  == K_ESCAPE):
            break
```

```
        draw()
        calculateNextGeneration()
        pygame.display.flip()

if __name__ == '__main__': main()
```

Listing 2.3 Hauptablauf des »Game of Life«

Die meisten Aufrufe innerhalb von `main()` kennen Sie bereits, wenn Sie das erste Kapitel, »Es schlägt 12«, gelesen haben. Dann können Sie den folgenden Absatz überspringen.

Über die Variable `video_flags` aktivieren wir die Hardwarebeschleunigung der Grafikkarte. Es wird ein doppelt gepufferter hardwarebeschleunigter Screen erstellt, so dass alle Zeichenoperationen im nicht sichtbaren Puffer ausgeführt werden können. Dieser zweite Puffer wird dann schlagartig durch den Aufruf `pygame.display.flip()` auf dem Screen dargestellt. Der Aufruf `pygame.init()` initialisiert die Spielebibliothek und ist immer vor Verwendung von `pygame` durchzuführen. Der folgende Aufruf setzt die Parameter für die Hardwarebeschleunigung. Hier sind noch weitergehende Parameter möglich, die diverse Einstellungen erlauben. Diese zusätzlichen Parameter finden Sie in der Dokumentation von `pygame` auf der Internetseite *http://www.pygame.org*.

`initLivingSpace()` erzeugt einen zufällig bevölkerten Lebensraum. Der Aufruf danach erstellt das Fenster für die Darstellung in der von uns benötigten Größe. `init()` initialisiert OpenGL, was in diesem Fall sehr spartanisch ausfällt, da lediglich die Farbe zum Löschen des Bildschirminhalts gesetzt wird.

In Kapitel 1 noch nicht verwendet wurden lediglich `initLivingSpace()` und `calculateNextGeneration()`.

Die Bildschirmgröße ermittelt sich ganz einfach aus der Größe eines Lebewesens multipliziert mit der jeweiligen Dimension der x- oder y-Achse, wobei die Größe in `creatureSize` mit `10` definiert ist:

```
creatureSize = 10
```

Natürlich hat ein Punkt keine Maße, aber um ihn sichtbar zu machen, ist ein bestimmtes Maß dennoch notwendig; wie groß Sie die Lebewesen darstellen wollen, können Sie selbst entscheiden. Um die Größe der Lebewesen anzupassen, ändern Sie einfach die Variable `creatureSize` entsprechend. Ganz korrekt ist die Definition allerdings nicht, denn tatsächlich ist `10` der Platz, der inklusive des Abstandes zwischen den Feldern für ein Feld reserviert wird, wovon 90 Prozent für die Darstellung der Kreatur und 10 Prozent für die Darstellung des Gitternetzes dienen. Das Gitternetz wird nicht explizit dargestellt, sondern ergibt sich

automatisch, da die Lebewesen in einem Abstand von 10 Einheiten gezeichnet werden, aber nur 9 Einheiten breit sind. Entsprechend bleibt eine Einheit zwischen den Lebewesen in der Hintergrundfarbe erhalten. Das Ergebnis sieht dann wie ein Gitternetz aus.

Die Funktion `resize(screenSize)` erstellt für uns die View auf den Lebensraum innerhalb der 3D-Matrix von OpenGL.

```
def resize((width, height)):
    if height == 0:
        height = 1
    glViewport(0, 0, width, height)
    glMatrixMode(GL_PROJECTION)
    glLoadIdentity()
    glOrtho(-10.0, livingSpaceWidth * 10.0 + 10.0,
     livingSpaceHeight * 10.0 + 10.0, -10.0, -6.0, 0.0)
    glMatrixMode(GL_MODELVIEW)
    glLoadIdentity()
```

Listing 2.4 Initialisierung der Darstellungsebene

Vor der Manipulation einer Matrix muss diese geladen werden. Dies geschieht hier jeweils durch den Aufruf von `glMatrixMode()`. `glLoadIdentity()` lädt zur ausgewählten Matrix die Identitätsmatrix; das ist jene Matrix, die ohne jegliche Transformationen den Inhalt darstellt. Nähere Informationen entnehmen Sie bitte Anhang A, »OpenGL«.

`glOrtho()` wird hier mit `-10.0` und zweimal `+10.0` aufgerufen, um zum Fensterrand einen entsprechenden Abstand zu halten. Da es sich um eine Parallelprojektion handelt, ist der Wert von `-6.0` für »nah« absolut in Ordnung. Jeder andere negative Wert würde ebenfalls funktionieren. Wie im Anhang zu OpenGL beschrieben, ist die Größe der dargestellten Objekte bei einer Parallelprojektion mittels `glOrtho()` unabhängig von der Entfernung zum Objekt.

Der Aufruf von `init()` setzt in diesem Beispiel nur die Farbe für das Löschen des Bildschirms:

```
def init():
    glClearColor(0.0, 0.0, 0.0, 0.0)
```

Listing 2.5 Die Farbe zum Löschen des Bildschirmes setzen

Beim Zeichnen des Lebensraumes überprüfen Sie je Feld, ob sich dort ein Lebewesen aufhält oder nicht. Dafür verwenden Sie die sehr einfach gestaltete Funktion `isAlive(x, y)`, die wie folgt aufgebaut ist:

```
def isAlive(x, y):
    return livingSpace[x][y] == 1
```

Listing 2.6 Den Zustand eines Feldes prüfen

Der Vergleich liefert einen booleschen Wert zurück, entweder `True` oder `False`. Da nicht belebte Felder mit `0` belegt werden, wäre es auch möglich, auf den Vergleich zu verzichten und nur den Inhalt zurückzugeben, der, wenn er `0` ist, von Python automatisch als `False` interpretiert würde. Diese interne Verarbeitung von Python für eigene Zwecke zu nutzen, ist zwar möglich, führt aber in der Regel schnell zu Verwirrung und unerklärlichen Fehlern. Deshalb sollten Sie auf automatische Auswertungen solcher Art verzichten. In unserem Fall würde zum Beispiel jeder Wert ungleich `0` entsprechend als `True` ausgewertet, was aber später nicht mehr korrekt sein wird, sobald wir die Lebewesen »langsam sterben« lassen.

Das eigentliche Zeichnen erfolgt nun durch Aufruf von `draw()`:

```
def draw():
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
    glLoadIdentity()
    glTranslatef(0.0, 0.0, 3.0)
    glColor4f(1.0, 0.0, 0.0, 1.0)
    glBegin(GL_QUADS)
    for column in range(livingSpaceWidth):
        for row in range(livingSpaceHeight):
            if isAlive(column, row):
            x = column * 10.0
            y = row * 10.0
            glVertex3f(x, y, 0.0)
            glVertex3f(9.0 + x, y, 0.0)
            glVertex3f(9.0 + x, 9.0 + y, 0.0)
            glVertex3f(x, 9.0 + y, 0.0)
    glEnd()
```

Listing 2.7 Zeichnen des Lebensraumes

Innerhalb von `draw()` wird `isAlive()` aufgerufen, um den Zustand der einzelnen Felder abzufragen. Belebte Felder werden rot eingefärbt, alle anderen werden nicht eingefärbt. Wie bereits erwähnt, erfolgt die Darstellung immer in 10er-Schritten, während beim Zeichnen nur eine Breite von 9 verwendet wird; dadurch entsteht das Gitternetz zwischen den Lebewesen, das also nicht explizit gezeichnet wird, sondern sich aus den freigelassenen Zwischenräumen beim Zeichnen ergibt. Der Lebensraum wird in Spalten und Zeilen auf dem Bildschirm dargestellt, wobei jedes Lebewesen als kleines Quadrat gezeichnet wird.

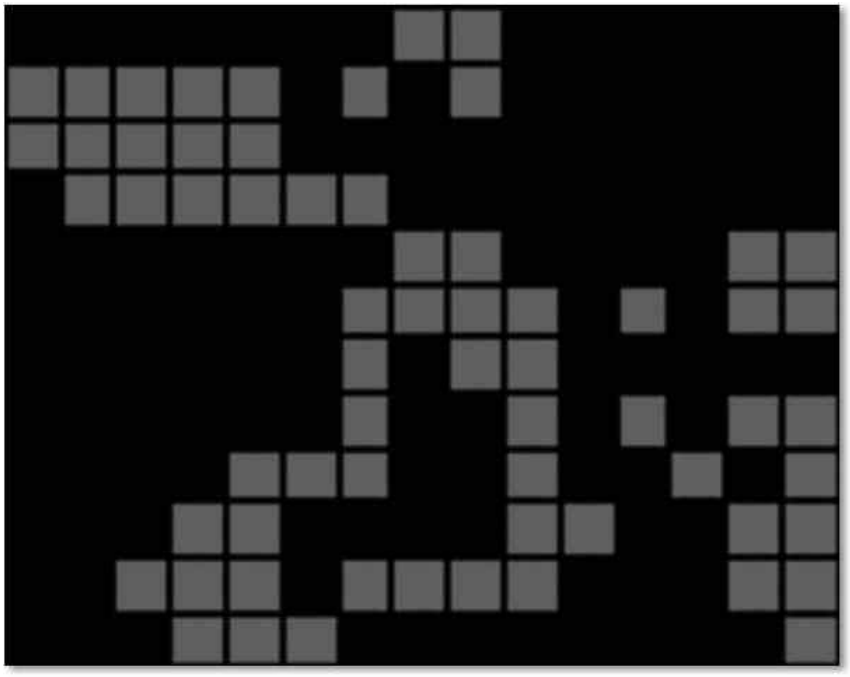


Abbildung 2.7 Das Gitternetz definiert sich wie ein Loch durch seine Umgebung – der Hintergrund scheint einfach durch

2.5 Die nächste Generation

Zum Berechnen der Folgegeneration unserer Simulation befolgen wir lediglich die oben definierten Regeln konsequent. Dafür müssen wir jedoch zunächst die Anzahl der belebten Nachbarzellen ermitteln. Dies leistet die folgende Funktion auf einfache Weise:

```
def getNeighborCount(x, y):
    count = 0

    xpn = (x + 1) % livingSpaceWidth
    ypn = (y + 1) % livingSpaceHeight

    count += isAlive(x  , ypn)
    count += isAlive(xpn, ypn)
    count += isAlive(xpn, y)
    count += isAlive(xpn, y - 1)
    count += isAlive(x  , y - 1)
    count += isAlive(x - 1, y - 1)
    count += isAlive(x - 1, y)
    count += isAlive(x - 1, ypn)
    return count
```

Listing 2.8 Die Anzahl der Nachbarn ermitteln

Die Überprüfung des Zustandes der Nachbarzellen führen wir erneut über die Funktion `isAlive()` durch:

```
def isAlive(x, y):
    return livingSpace[x][y] == 1
```

Listing 2.9 Zustandsüberprüfung für ein Feld

Momentan mag das noch wie Ressourcenverschwendung aussehen, da wir hier einen Funktionsaufruf nutzen, obwohl wir direkt auf die Liste zugreifen könnten, nämlich den Lebensraum. Dass die Indirektion über die Funktion `isAlive()` kein unnötiger Overkill ist, werden Sie im Verlaufe dieses Kapitels erkennen, sobald wir einige Anpassungen vorgenommen haben, die sich dank dieser Indirektion leichter durchführen lassen.

Um ungültige Zugriffe – also einen Zugriff außerhalb der Listengröße – zu vermeiden, werden zunächst `xpn` und `ypn` berechnet, die für »positiver Nachbar von x (xPositiveNeighbor)« und »positiver Nachbar von y (yPositiveNeighbor)« stehen. Somit ist der Zugriff nach oben abgesichert, da hier Modulo der Obergrenze gerechnet wird. Wenn zum Beispiel bei einem Lebensraum mit den Ausmaßen 9 × 9 das untere rechte Feld überprüft wird, nähme `xpn` den Wert 9 an, da wir die Liste von 0 bis 8 durchlaufen, und läge außerhalb des für die Liste gültigen Bereichs. Da aber für jeden Wert Modulo Listenlänge gerechnet wird, enthält `xpn` statt des Wertes 9 den Wert 0.

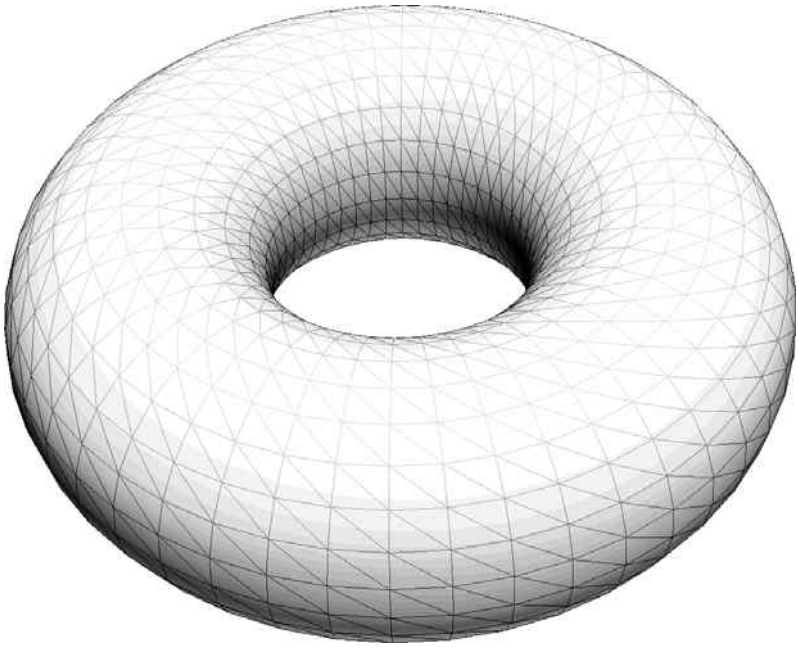


Abbildung 2.8 Eine Donut-Welt (in der Mathematik ein »Torus«)

Es ist natürlich beabsichtigt, hier mit einer sogenannten »Donut-Welt« zu arbeiten, also mit einer Welt, die keine Grenzen hat, da sie wie ein Donut aufgebaut ist. Eine »Donut-Welt« können Sie zwar komplett umrunden, aber Sie werden weder einen Anfang noch ein Ende festlegen können. Stellen Sie sich unser »Game of Life« auf einem Blatt Papier vor, bei dem Sie zwei gegenüberliegende Spielfeldkanten zusammenkleben. Um diesen Effekt zu erreichen, »verbinden« wir entsprechend die Ränder des Spielfeldes mit der genannten Modulo-Arithmetik.

Würden wir den Wert nicht mittels Modulo reduzieren, so müssten Sie zusätzlich eine Überprüfung einbauen, die Werte außerhalb des Listenbereichs um die Länge der Liste reduziert. Die Überprüfung fällt dank Modulo aber unter den Tisch, denn zu große Werte werden so automatisch auf den Listenbereich reduziert. Nach unten ist keine Absicherung notwendig, denn es kommt maximal zu einem Index von -1, den Python automatisch als Index auf das letzte Element auswertet. Dass es nur zu einem Index von -1 kommen kann, liegt daran, dass wir bei den Berechnungen für die nächste Generation nur die **direkten** Nachbarn für jedes Feld betrachten. Die Absicherung nach unten bleibt uns also dank eines Sprachfeatures von Python erspart.

Alles in allem wirkt sich durch diese Maßnahmen das Leben in den Feldern am rechten Rand auf das Leben in den Feldern am linken Rand aus. Analog verhält es sich auch mit den Feldern am oberen und unteren Rand des Spielfeldes.

Der Aufruf von `isAlive()` liefert zwar einen booleschen Wert, aber für diesen gilt, dass er numerisch als 1 ausgewertet wird, falls es sich um `True` handelt. Da `False` entsprechend als 0 ausgewertet wird, funktioniert der Aufruf auch für das Zählen der Nachbarn anstandslos. Grundsätzlich sollte man solche Tricks natürlich nicht zu sehr überstrapazieren und gut kommentieren.

Die kürzeste Variante ist nicht immer die beste **[!]**

Dieser kleine Trick mit der Auswertung der booleschen Werte sollte nicht zur Regel werden. Sie müssen bedenken, dass `True` und `False` in späteren Pythonversionen auch anders umgewandelt werden könnten. Jede Zahl ungleich 0 ist als `True` definiert, es wäre also auch denkbar, `True` in jede beliebige Zahl umzuwandeln.

Die nächste Generation lässt sich aufbauend auf diese Funktion wie folgt errechnen:

```
def calculateNextGeneration():
    neighborCount = []
    for column in range(livingSpaceWidth):
        neighborCount.append([])
        for row in range(livingSpaceHeight):
            neighborCount[column].
```

```
            append(getNeighborCount(column, row))

    for column in range(livingSpaceWidth):
        for row in range(livingSpaceHeight):
            if 2 <= neighborCount[column][row] <= 3:
                if neighborCount[column][row] == 3:
                   # Geburt eines Lebewesens
                    livingSpace[column][row] = 1
            else:
               # Tod eines Lebewesens
                livingSpace[column][row] = 0
```

Listing 2.10 Berechnung der Folgegeneration

Zunächst wird in einer verschachtelten Liste die Anzahl der Nachbarn für jedes Feld des Lebensraumes in der Liste `neighborCount` hinterlegt. Erst danach werden die einzelnen Felder entsprechend neu gesetzt.

[!] Dieses zweistufige Vorgehen ist so auch notwendig, denn wenn der Lebensraum der Folgegeneration bereits jeweils beim Ermitteln der Nachbarn für die einzelnen Felder gesetzt würde, so würde das Setzen die Anzahl der Nachbarn für die noch folgenden Felder verändern und damit das Ergebnis verfälschen. Deshalb ist ein Vorgehen in zwei getrennten Schritten hier die richtige Wahl.

Die aus den Regeln resultierenden Bedingungen wurden hier durch geschickte Verkettung und Verschachtelung auf ein Minimum reduziert. Überprüfen Sie doch einmal den Code auf Herz und Nieren – es werden alle Regeln exakt eingehalten.

Zeit für einen Probelauf – und schon wird es auf dem Bildschirm lebendig! Sind Sie überrascht von so viel Dynamik?

Allerdings lässt sich die Darstellung noch etwas aufpeppen, indem wir die gerade verstorbenen Lebewesen nicht direkt ausblenden. Diesem Vorhaben widmen wir uns im nächsten Abschnitt.

2.6 Langsames Sterben

Unsere Simulation ist beeindruckend, allerdings laufen die Berechnungen für die jeweils nächste Generation in so hohem Tempo ab, dass viele Generationen gar nicht wahrgenommen werden können. Darum ist es besser, die Kreaturen »langsam sterben« zu lassen. Dies erreichen wir, indem wir die Felder nicht abrupt schwarz zeichnen, sobald sie nicht mehr belebt sind, sondern die dort anzutreffende Farbe je Generation langsam in Schwarz übergehen lassen. Durch

diesen Trick wird sehr schön sichtbar, wie viel Dynamik das »Game of Life« tatsächlich enthält.

Um dieses Ziel zu erreichen, initialisieren wir den Lebensraum nun nicht mehr mit `1` und `0`, sondern mit `1000` und `0`. Ein Feld gilt also ab jetzt erst dann als belebt, wenn es mit `1000` befüllt ist.

Den Quelltext zu diesem Beispiel finden Sie auf der CD-Rom im Verzeichnis *Kapitel03/gameOfLife_v2.py*. [●]

```
def initLivingSpace():
    for x in range(livingSpaceWidth):
        livingSpace.append([])
        for y in range(livingSpaceHeight):
            if random.randint(0, 1) == 1:
                livingSpace[x].append(1000)
            else:
                livingSpace[x].append(0)
```

Listing 2.11 Initialisierung des Lebensraumes

Mit der Anpassung der Initialisierungswerte ist natürlich auch die Änderung der Funktionen zum Zeichnen notwendig, weshalb wir die als belebt zu initialisierenden Felder nun mit `1000` befüllen. Diesen neuen Sachverhalt übernehmen Sie auch in den Aufruf von `isAlive()`:

```
def isAlive(x, y):
    return livingSpace[x][y] == 1000
```

Listing 2.12 Anpassung der Feldprüfung

Die aufwendigste Anpassung ist in der Funktion `draw()` nötig, denn wir müssen ab sofort nicht nur die Zustände lebendig und tot darstellen, sondern auch die Übergangszustände dazwischen. Um den Übergang zu visualisieren, setzen wir einfach die Intensität der Farbe des Feldes abhängig vom Feldinhalt und reduzieren diesen schrittweise von 1.000 auf 0. Eine lineare Reduzierung wäre eine Möglichkeit, die aber nicht so schön »ausblendet«, weshalb wir nicht einfach um gleiche Anteile reduzieren, sondern teilen. Durch das Teilen erhalten wir eine degressive Reduzierung der Farbintensität, die sehr schön ausblendet.

Da alle lebendigen Felder nun mit `1000` belegt werden, ergeben sich folgende Änderungen:

```
def draw():
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
    glLoadIdentity()
    glTranslatef(0.0, 0.0, 3.0)
```

```
    glBegin(GL_QUADS)
    for column in range(livingSpaceWidth):
        for row in range(livingSpaceHeight):
            healthStatus = float(livingSpace[column][row]) / 1000.0
            # die Farbe für die belebten Felder wird
            # nun in Abhängigkeit vom Wert des Feldes bestimmt
            glColor4f(healthStatus, 0.0, 0.0, 1.0)
            x = column * 10.0
            y = row * 10.0
            glVertex3f(x, y, 0.0)
            glVertex3f(9.0 + x, y, 0.0)
            glVertex3f(9.0 + x, 9.0 + y, 0.0)
            glVertex3f(x, 9.0 + y, 0.0)
    glEnd()
```

Listing 2.13 Das Zeichnen von langsam sterbenden Individuen

Eine weitere Anpassung besteht darin, dass die Felder immer gezeichnet werden und nicht wie zuvor nur dann, wenn sie belebt waren. Zudem wird die Farbe, die für das Zeichnen verwendet wird, je nach aktuellem Wert des Feldes gesetzt. Dies geschieht nun dementsprechend innerhalb der verschachtelten Schleifen. Der Aufruf von `glColor()` ist im obigen sowie im folgenden Quelltext, der die alte Version noch einmal zum Vergleich darstellt, fett markiert. Der bislang nötige Aufruf von `isAlive()` entfällt nun und ist zum Vergleich im Folgenden ebenfalls entsprechend **fett** markiert. Der Rest der Funktion `draw()` hat sich nicht verändert.

```
def draw():
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
    glLoadIdentity()
    glTranslatef(0.0, 0.0, 3.0)
    glColor4f(1.0, 0.0, 0.0, 1.0)
    glBegin(GL_QUADS)
    for column in range(livingSpaceWidth):
        for row in range(livingSpaceHeight):
            if isAlive(column, row):
                x = column * 10.0
                y = row * 10.0
                glVertex3f(x, y, 0.0)
                glVertex3f(9.0 + x, y, 0.0)
                glVertex3f(9.0 + x, 9.0 + y, 0.0)
                glVertex3f(x, 9.0 + y, 0.0)
    glEnd()
```

Listing 2.14 Die Ausgangsversion ohne langsam sterbende Individuen

Nach diesen Änderungen werden tote Lebewesen nicht mehr sofort ausgeblendet, indem das entsprechende Feld auf die Hintergrundfarbe gesetzt wird, sondern sie verschwinden langsam. In der Variablen `healthStatus` wird der prozentuale Restwert des jeweils zu zeichnenden Feldes hinterlegt und als Maß für die Intensität der Farbe gewählt. Für die Berechnung der Folgegeneration gelten ja nur jene Felder als belebt, die den Wert `1000` zugewiesen bekommen haben. Würden wir hier alle Felder mit Werten größer Null berücksichtigen, so würden alle Lebewesen sehr schnell aussterben, da sich eine allgemeine Überbevölkerung einstellen würde. Grafisch würde das durch eine langsam die roten Felder ausblendende Animation sichtbar werden.

Die nun noch fehlende Anpassung der Berechnung der nächsten Generation ist im Quelltext wieder **fett** hervorgehoben:

```
def calculateNextGeneration():
    neighborCount = []
    for column in range(livingSpaceWidth):
        neighborCount.append([])
        for row in range(livingSpaceHeight):
            neighborCount[column].
             append(getNeighborCount(column, row))

    for column in range(livingSpaceWidth):
        for row in range(livingSpaceHeight):
            if 2 <= neighborCount[column][row] <= 3:
                if neighborCount[column][row] == 3:
             # Geburt eines Lebewesen
                 livingSpace[column][row] = 1000
            else:
               # Langsames Sterben eines Lebewesen
                livingSpace[column][row] =
                 livingSpace[column][row]/1.1

            if livingSpace[column][row] < 200:
                livingSpace[column][row] = 0
```

Listing 2.15 Anpassungen zur Berechnung der Folgegeneration

Hier werden jetzt Felder, die neu geborene Lebewesen kennzeichnen, auf `1000` gesetzt, und Felder für verstorbene Lebewesen auf einen Wert kleiner `1000`. Die oben bereits angesprochene degressive Reduzierung erreichen wir durch das Teilen des Feldwertes durch `1.1`. Würden wir allerdings immer nur durch `1.1` teilen, so würde ein Feld sehr lange mit geringer Intensität weitergezeichnet. Um diesen Effekt zu verhindern, setzen wir ab einem Wert unter `200` den Wert direkt auf `0`. Falls Sie zufällig ein wenig von Buchführung verstehen, dann können Sie sich das

wie eine degressive Abschreibung vorstellen, die zu einem festgelegten Zeitpunkt in eine lineare Abschreibung überführt wird. Sogar die Bedeutung ist in diesem Fall ziemlich ähnlich, denn wir wollen sicherstellen, dass das entsprechende Lebewesen nach einer gewissen Zeit komplett »abgeschrieben« ist und sozusagen keinen »Restbuchwert« mehr hat. Man könnte sagen, Sie leisten hier ein wenig Sterbehilfe.

Damit wären alle notwendigen Anpassungen umgesetzt, und wir können die neue Darstellung bewundern.

2.7 »Game of Life« in 3D

Das »Game of Life« können Sie nicht nur in zwei, sondern auch in drei Dimensionen spielen. Da durch die dritte Dimension auch neue Nachbarn hinzukommen, müssen wir die Regeln etwas anpassen. Zuvor gab es genau 8 Nachbarn, jetzt sind es sage und schreibe 26. Das ist eine ganze Menge, die wir jetzt in den Griff bekommen wollen.

2.8 Die neuen Regeln

Setzen wir die alten Regeln in das Verhältnis für die neue Umgebung und probieren ein wenig herum, so kommen wir auf die folgenden Regeln:

- Eine tote Zelle mit genau 8 lebenden Nachbarn wird in der Folgegeneration neu geboren.
- Lebende Zellen mit weniger als 6 lebenden Nachbarn sterben in der Folgegeneration an Einsamkeit.
- Eine lebende Zelle mit 6 bis 11 lebenden Nachbarn bleibt in der Folgegeneration lebend.
- Lebende Zellen mit mehr als 11 lebenden Nachbarn sterben in der Folgegeneration an Überbevölkerung.

Zunächst hatte ich versucht, die Werte in Relation zur 2D-Variante durch folgende Formel zu ermitteln:

$$\frac{Wert_{2D}}{AnzahlNachbarn_{2D}} * AnzahlNachbarn_{3D}$$

Allerdings funktionierten die Werte bei zufälliger Startinitialisierung nicht besonders gut, weshalb ich ausgehend von den so ermittelten Werten per »Trial

and Error« die oben genannten Regeln ermittelt habe. Eine andere Möglichkeit wäre, die Werte mittels genetischer Algorithmen zu testen. Näheres zum Thema genetische Algorithmen erfahren Sie in Kapitel 8, »Evolution im Computer«.

2.9 Der 3D-Lebensraum

Um die dritte Dimension des Lebensraums zu ermöglichen, fügen wir einfach eine Verschachtelungstiefe mehr in die Liste für den Lebensraum ein.

Den Quelltext zu diesem Beispiel finden Sie auf der CD-Rom im Verzeichnis *Kapitel02/gameOfLife_v3.py*. [●]

```
def initLivingSpace():
    for x in range(livingSpaceWidth):
        livingSpace.append([])
        for y in range(livingSpaceHeight):
            livingSpace[x].append([])
            for z in range(livingSpaceDepth):
                if random.randint(0, 1) == 1:
                    livingSpace[x][y].append(1000)
                else:
                    livingSpace[x][y].append(0)
```

Listing 2.16 Die Initialisierung der 3D-Variante

Auch für das Zeichnen in 3D ist nicht viel zu ergänzen. Um jedoch die Darstellung etwas schöner zu gestalten, werden wir die Lebewesen nicht als flache Quadrate – kurz: Flächen – zeichnen, sondern für jedes Lebewesen einen kleinen Würfel anzeigen. Die `draw()`-Funktion sieht nun wie folgt aus:

```
def draw():
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
    glBegin(GL_QUADS)
    for column in range(livingSpaceWidth):
        for row in range(livingSpaceHeight):
            for depth in range(livingSpaceDepth):
                if livingSpace[column][row][depth] > 0:
                    healthStatus =
                     float(livingSpace[column][row][depth])
                     / 1000.0

                    if depth % 2 == 0:
                        glColor4f(1.0, 0.0, 0.0, healthStatus)
                    elif depth % 3 == 0:
                        glColor4f(0.0, 1.0, 0.0, healthStatus)
```

```
                    else:
                        glColor4f(0.0, 0.0, 1.0, healthStatus)

                    x = column * 20.0
                    y = row * 20.0
                    z = depth * 20.0
                    drawCube(x, y, z, 15.0)
    glEnd()
```

Listing 2.17 Das »Game of Life« in 3D

Da die Darstellung nun im 3D-Raum stattfindet, ist eine dritte Dimension hinzugekommen, die durch die Variable `depth` zum Ausdruck kommt. Um den Würfel etwas bunter zu machen und die Dynamik der Veränderung klarer herauszustellen, zeichnen wir die Lebewesen abhängig von der Tiefe `depth` in verschiedenen Farben, wobei jedes erste, zweite und dritte Element jeweils in der gleichen Farbe erscheint. Die Bestimmung erfolgt dabei anhand der Tatsache, ob der Wert der Tiefe durch zwei oder drei teilbar ist. Alle Werte, die weder durch zwei noch durch drei teilbar sind – also weder nur zweiten noch zur dritten Ebene gehören, sondern zu ersten – erhalten die dritte Farbe. Der Einfachheit halber habe ich die drei Grundfarben verwendet, so dass die entsprechenden Farbanteile jeweils auf `1.0` gesetzt sind. Die Würfel werden also rot, grün und blau gefärbt.

```
if depth % 2 == 0:
    glColor4f(1.0, 0.0, 0.0, healthStatus)
elif depth % 3 == 0:
    glColor4f(0.0, 1.0, 0.0, healthStatus)
else:
    glColor4f(0.0, 0.0, 1.0, healthStatus)
```

Listing 2.18 Verschiedene Farben für mehr Abwechslung

Sterbende Lebewesen blenden wir diesmal durch Ändern der Farbtransparenz langsam aus. Im oberen Quelltextausschnitt erkennen Sie dies am letzten Parameter des Aufrufs `glColor4f()`. Der für `healthStatus` ermittelte Wert muss entsprechend zwischen `0.0` und `1.0` liegen, was wir durch die folgende Berechnung kurz vor dem Aufruf `glColor4f()` sicherstellen:

```
healthStatus = float(livingSpace[column][row][depth]) / 1000.0
```

Listing 2.19 Skalierung der Farbintensität anhand des Lebenszustandes

Da jedes Feld maximal einen Wert von 1.000 annimmt, ermitteln wir durch Division durch `1000` die relative Zahl zwischen `0.0` und `1.0`.

Um die Transparenz zu aktivieren und die Szene schön auszuleuchten, passen wir `init()` wie in Listing 2.20 an; zuvor existierte lediglich der Aufruf von `glClearColor()`, der nun Weiß statt Schwarz als Hintergrundfarbe setzt.

```
def init():
    glClearColor(1.0, 1.0, 1.0, 0.0)
    glClearDepth(1.0)
```

Listing 2.20 Weißer Hintergrund

Bis auf den ersten Aufruf von `glClearColor(1.0, 1.0, 1.0, 0.0)` ist alles weitere neu hinzugekommen, wobei `glEnable(GL_DEPTH_TEST)` die Tiefenprüfung aktiviert. Damit stellen wir sicher, dass OpenGL die Objekte entsprechend ihrer Tiefe in der richtigen Reihenfolge zeichnet. Ohne diese Prüfung müssten Sie selbst dafür sorgen, dass Objekte, die weiter vorn liegen, auch später gezeichnet werden. Der Aufruf `glDepthFunc(GL_LEQUAL)` bestimmt dabei die Art der Tiefenprüfung. `GL_LEQUAL` bedeutet, dass alles, was niedriger oder gleich dem aktuellen Wert ist, gezeichnet wird.

```
    # Tiefenpruefung aktivieren
    glEnable(GL_DEPTH_TEST)
    # Art der Pruefung festlegen
    glDepthFunc(GL_LEQUAL)
```

Listing 2.21 Tiefenprüfung mittels OpenGL

`glEnable(GL_BLEND)` aktiviert die Transparenz. Der darauffolgende Aufruf von `glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)` gibt an, wie die Deckkraft für die einzelnen Objekte anhand der Deckkraftkomponente der dazugehörigen Farbe zu ermitteln ist.

```
# Transparenz aktivieren
    glEnable (GL_BLEND);
    # Art der Transparenzberechnung festlegen
    glBlendFunc (GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

Listing 2.22 Transparenz aktivieren

Im Anschluss aktivieren wir die Beleuchtung mit `glEnable(GL_LIGHTNING)`. Damit die Beleuchtung funktioniert, müssen Sie außerdem Lichtquellen definieren. Uns genügt an dieser Stelle eine Lichtquelle, die wir in den folgenden Zeilen generieren. Da es verschiedene Arten von Licht gibt, werden diese auch in OpenGL unterschieden und sind entsprechend zu definieren.

```
# Aktivierung der Beleuchtung
glEnable(GL_LIGHTING)
# eine Lichtquelle erstellen
glLightfv(GL_LIGHT0, GL_AMBIENT, [0.6, 0.6, 0.6, 1.0])
glLightfv(GL_LIGHT0, GL_DIFFUSE, [0.4, 0.4, 0.4, 1.0])
glLightfv(GL_LIGHT0, GL_POSITION,
 [(livingSpaceWidth * 20.0)/2.0,
 (livingSpaceHeight * 20.0)/2.0, -500.0, 1.0])
# Aktivierung der erstellten Lichtquelle
glEnable(GL_LIGHT0)
```

Listing 2.23 Individuelle Beleuchtung

Der Aufruf `glLightfv(GL_LIGHT0, GL_AMBIENT, [0.6, 0.6, 0.6, 1.0])` setzt zum Beispiel ein schummeriges Licht, das alle Farben gleichmäßig enthält und einem weißen Licht entspricht. Die Intensität der gewählten Lichtart lässt sich dabei entweder über die Farbanteile regeln oder über die Deckkraft. Mit dem Wert `0.6` für alle Farbkomponenten erhalten Sie das gleiche Ergebnis wie mit einem Wert von `1.0` für alle Farben bei einer Deckkraft von `0.6`. Analog dazu wird durch den Aufruf von `glLightfv(GL_LIGHT0, GL_DIFFUSE, [0.4, 0.4, 0.4, 1.0])` der diffuse Anteil des Lichts bestimmt. Da sich beide Aufrufe auf die gleiche Lichtquelle beziehen, müssen wir sicherstellen, dass die Anteile je Farbe möglichst nicht 100 Prozent übersteigen, wobei 100 Prozent `1.0` entsprechen.

`glLightfv(GL_LIGHT0, GL_POSITION, [...])`, der letzte Aufruf für die Erstellung unserer Lichtquelle, setzt die Position des Lichtes im Raum. Die Lichtquelle wird mittig vor dem Lebensraum platziert, so dass der Betrachter alles perfekt ausgeleuchtet sieht. Mit der Erstellung einer Lichtquelle ist es aber noch nicht getan, wir müssen sie zum Abschluss noch mit `glEnable(GL_LIGHT0)` aktivieren.

Nun fehlen uns noch die Materialeigenschaften. OpenGL erlaubt es, verschiedene Oberflächeneigenschaften zu definieren. Damit wird es möglich, jedes in der Natur vorkommende Material entsprechend realistisch darzustellen. Sie können also spiegelnde Flächen genauso animieren wie matte Flächen und alle Abstufungen dazwischen.

```
# Aktivierung von Materialeigenschaften
glEnable(GL_COLOR_MATERIAL)
# diffuses, ambientes Licht für Vorder- und Rueckseite
glColorMaterial(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE)
```

Listing 2.24 Materialeigenschaften festlegen

Mit `glEnable(GL_COLOR_MATERIAL)` veranlassen wir OpenGL, Materialeigenschaften zu berücksichtigen. Das verwendete Material wird daraufhin durch den

Aufruf von `glColorMaterial(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE)` näher spezifiziert. Der Parameter `GL_FRONT_AND_BACK` legt hierbei fest, dass sowohl die Ober- als auch die Unterseite des Materials entsprechend zu behandeln sind. Hier wäre es auch möglich, für die Oberseite andere Eigenschaften zu definieren als für die Unterseite. Ein Spiegel ist in der Regel zum Beispiel auf der Unterseite matt, während die Oberseite spiegelt. Die Unterscheidung zwischen Ober- und Unterseite beziehungsweise Vorder- und Rückseite erfolgt dabei – wie bereits in Kapitel 1, »Es schlägt 12«, erläutert – durch das sogenannte Winding von OpenGL. Nähere Informationen dazu entnehmen Sie bitte dem Anhang A.2, »Culling und Winding«.

```
# automatische Korrektur der Normalen aktivieren
glEnable(GL_NORMALIZE)
# bestmoeglich rendern
glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST)
```

Listing 2.25 Normalenskalierung und bestmögliche Darstellung aktivieren

Der Aufruf `glEnable(GL_NORMALIZE)` wäre streng genommen nicht notwendig, da wir bereits eine auf `1.0` festgelegte Normale verwenden. Sinn des Aufrufes ist es, Normalen, die nicht bereits eine Länge von 1 haben, auf diese Länge zu reduzieren beziehungsweise sie zu normalisieren.

Zum Abschluss sorgt `glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST)` für die bestmögliche Darstellung. Die Unterstützung von `glHint()` ist jedoch optional, weshalb nicht jeder OpenGL-Anbieter hierauf reagieren muss. Die meisten Anbieter tun das jedoch und versuchen entsprechend, die bestmögliche Darstellung zu rendern.

Das eigentliche Zeichnen des Würfels ist jetzt etwas umfangreicher, da wir die sechsfache Menge von Punkten benötigen. Aufgrund dieser Veränderung habe ich das Zeichnen der Würfel in die Funktion `drawCube(x, y, z, CubeSize)` ausgelagert, die wie folgt aufgebaut ist:

```
def drawCube(x, y, z, cubeSize):
   # vordere Seitenflaeche
    glNormal3f(0.0, 0.0, -1.0)
    glVertex3f(x, y, z)
    glVertex3f(cubeSize + x, y, z)
    glVertex3f(cubeSize + x, cubeSize + y, z)
    glVertex3f(x, cubeSize + y, z)

   # hintere Seitenflaeche
    glNormal3f(0.0, 0.0, +1.0)
    glVertex3f(x, y, z + cubeSize)
```

```
    glVertex3f(cubeSize + x, y, z + cubeSize)
    glVertex3f(cubeSize + x, cubeSize + y, z + cubeSize)
    glVertex3f(x, cubeSize + y, z + cubeSize)

  # linke Seitenflaeche
    glNormal3f(-1.0, 0.0, 0.0)
    glVertex3f(x, y, z)
    glVertex3f(x, cubeSize + y, z)
    glVertex3f(x, cubeSize + y, z + cubeSize)
    glVertex3f(x, y, z + cubeSize)

  # rechte Seitenflaeche
    glNormal3f(+1.0, 0.0, 0.0)
    glVertex3f(cubeSize + x, y, z)
    glVertex3f(cubeSize + x, cubeSize + y, z)
    glVertex3f(cubeSize + x, cubeSize + y, z + cubeSize)
    glVertex3f(cubeSize + x, y, z + cubeSize)

  # obere Seitenflaeche
    glNormal3f(0.0, +1.0, 0.0)
    glVertex3f(x, cubeSize + y, z)
    glVertex3f(cubeSize + x, cubeSize + y, z)
    glVertex3f(cubeSize + x, cubeSize + y, z + cubeSize)
    glVertex3f(x, cubeSize + y, z + cubeSize)

  # untere Seitenflaeche
    glNormal3f(0.0, -1.0, 0.0)
    glVertex3f(x, y, z)
    glVertex3f(cubeSize + x, y, z)
    glVertex3f(cubeSize + x, y, z + cubeSize)
    glVertex3f(x, y, z + cubeSize)
```

Listing 2.26 Einen Würfel zeichnen

Im Großen und Ganzen entspricht das Zeichnen unserer Vorgehensweise für die Quadrate. Allerdings zeichnen wir jetzt sechs Quadrate, die zusammen den Würfel ergeben.

Nun fehlt nur noch die Ermittlung der Nachbarn, wofür wir zunächst die Funktion `isAlive()` um die dritte Dimension erweitern müssen:

```
def isAlive(x, y, z):
    return livingSpace[x][y][z] == 1000
```

Listing 2.27 Den Zustand eines Feldes prüfen

Das eigentliche Zählen der noch lebenden Nachbarn erfolgt wie gehabt in der Funktion `getNeighborCount()`, in der wir nun wesentlich mehr Nachbarn berücksichtigen müssen. Die Überprüfung der Nachbarn ist vom Prinzip her gleich geblieben, es gehen lediglich mehr Nachbarn in die Summe der belebten Nachbarfelder ein.

```
def getNeighborCount(x, y, z):
    count = 0

    xpn = (x + 1) % livingSpaceWidth
    ypn = (y + 1) % livingSpaceHeight
    zpn = (z + 1) % livingSpaceDepth

    count += isAlive(x  , ypn, z - 1)
    count += isAlive(xpn, ypn, z - 1)
    count += isAlive(xpn, y, z - 1)
    count += isAlive(xpn, y - 1, z - 1)
    count += isAlive(x  , y - 1, z - 1)
    count += isAlive(x - 1, y - 1, z - 1)
    count += isAlive(x - 1, y, z - 1)
    count += isAlive(x - 1, ypn, z - 1)

    count += isAlive(x  , ypn, z)
    count += isAlive(xpn, ypn, z)
    count += isAlive(xpn, y, z)
    count += isAlive(xpn, y - 1, z)
    count += isAlive(x  , y - 1, z)
    count += isAlive(x - 1, y - 1, z)
    count += isAlive(x - 1, y, z)
    count += isAlive(x - 1, ypn, z)

    count += isAlive(x  , ypn, zpn)
    count += isAlive(xpn, ypn, zpn)
    count += isAlive(xpn, y, zpn)
    count += isAlive(xpn, y - 1, zpn)
    count += isAlive(x  , y - 1, zpn)
    count += isAlive(x - 1, y - 1, zpn)
    count += isAlive(x - 1, y, zpn)
    count += isAlive(x - 1, ypn, zpn)

    count += isAlive(x, y, zpn)
    count += isAlive(x, y, z - 1)

    return count
```

Listing 2.28 Die Anzahl der Nachbarn ermitteln

Zunächst addieren wir die Anzahl der Nachbarn vor dem aktuellen Würfel, dann die Anzahl der Nachbarn rund um den Würfel und zu guter Letzt die Anzahl der Nachbarn hinter dem Würfel. Dabei habe ich jeweils die gleichen Nachbarn verwendet wie bei der 2D-Variante – es ist lediglich die dritte Dimension hinzugekommen, die durch den Wert von `z` ausgedrückt wird. Deshalb ist ganz zum Schluss noch das Addieren der beiden Nachbarn direkt vor und direkt hinter dem aktuellen Lebewesen notwendig.

Die Folgegeneration wird ebenfalls fast identisch ermittelt. Zusätzlich zur Anpassung für die dritte Dimension habe ich die Degression ein wenig angehoben, um einen schnelleren Einblick in das Innere der 3D-Welt zu ermöglichen, wenn Lebewesen sterben. Dementsprechend wird die Farbe schneller verblassen.

```
def calculateNextGeneration():
    neighborCount = []
    for column in range(livingSpaceWidth):
        neighborCount.append([])
        for row in range(livingSpaceHeight):
            neighborCount[column].append([])
            for depth in range(livingSpaceDepth):
                neighborCount[column][row].
                 append(getNeighborCount(column, row, depth))

    for column in range(livingSpaceWidth):
        for row in range(livingSpaceHeight):
            for depth in range(livingSpaceDepth):
                if 6 <= neighborCount[column][row][depth] <= 11:
                    if neighborCount[column][row][depth] == 8:
                       # creature gets born
                        livingSpace[column][row][depth] = 1000
                else:
                   # creature dies slowly
                    livingSpace[column][row][depth] =
                     livingSpace[column][row][depth] / 1.5

                if livingSpace[column][row][depth] < 200:
                    livingSpace[column][row][depth] = 0
```

Listing 2.29 Die nächste Generation berechnen

Das waren alle Änderungen, die notwendig sind, um aus dem 2D- ein 3D-Game-of-Life zu zaubern. Bei der Berechnung der Folgegeneration ist es auch denkbar, nur die direkt anliegenden sechs Nachbarn zu berücksichtigen. Die entsprechende Implementierung hierfür zeigt Listing 2.30.

Den Quelltext zu diesem Beispiel finden Sie auf der CD-Rom im Verzeichnis *Kapitel02/gameOfLife_v4.py*. [●]

```
def getNeighborCount(x, y, z):
    count = 0

    xpn = (x + 1) % livingSpaceWidth
    ypn = (y + 1) % livingSpaceHeight
    zpn = (z + 1) % livingSpaceDepth

    count += isAlive(x  , ypn, z)
    count += isAlive(xpn, y, z)
    count += isAlive(x  , y - 1, z)
    count += isAlive(x - 1, y, z)

    count += isAlive(x, y, zpn)
    count += isAlive(x, y, z - 1)

    return count
```

Listing 2.30 Alternative zur Zählung der Nachbarzellen

2.10 War das schon alles?

Bis hierher ist unser »Game of Life« ganz nett anzuschauen, aber ein bisschen mehr Interaktivität wäre sicher nicht schlecht. Denkbar wäre zum Beispiel die Möglichkeit, eigene Ausgangskonfigurationen für den Lebensraum festzulegen. So könnten Sie leicht verschiedene bereits bekannte Strukturen ausprobieren und sich auch einmal darin versuchen, eigene interessante Kreationen zu erzeugen. Eine interessante Ausgangskonfiguration sehen Sie in Abbildung 2.9. Wenn Sie wissen möchten, was sich dahinter versteckt, dann erweitern Sie doch einfach unseren bisherigen Code so, dass Sie diese Anordnung als eigene Startkonfiguration vorgeben können.

Als ich zum ersten Mal das »Game of Life« programmierte, versuchte ich, Strukturen zu finden, die möglichst lange dynamisch bleiben, ohne periodisch zu werden. Ein während des Spiels vorgekommener Spielfeldzustand sollte also nicht zweimal erreicht werden. Dabei entdeckte ich hin und wieder auch Ausgangskonfigurationen, die sehr lange dynamisch bleiben und dabei auch Phasen sehr geringer Population durchschreiten, um dann wieder in eine sehr starke Population überzugehen. Dabei machte ich mir allerdings nie die Mühe, selbst Startkonfigurationen zu kreieren, sondern probierte so lange verschiedene zufällige

Startkonfigurationen aus, bis eine tolle Kombination dabei war. Dabei passte ich nur den Parameter für die Dichte der Ausgangspopulation an.

Solche Tests werden aber mit der Zeit schnell sehr stupide, weshalb es sich natürlich empfiehlt, das Ganze zu automatisieren. Für die Automatisierung legte ich einfach eine Untergrenze an sich verändernden Feldern fest, die das Abbruchkriterium für oszillierende Endstrukturen darstellte. Wenn also eine Ausgangskonfiguration in eine fast nicht mehr dynamische oszillierende Konfiguration übergegangen war, so wurde neu gestartet. Um bei den Testläufen die besten Konfigurationen zu speichern, sicherte ich einfach den Initialisierungswert des Zufallsgenerators vor der Erzeugung der Ausgangskonfiguration.

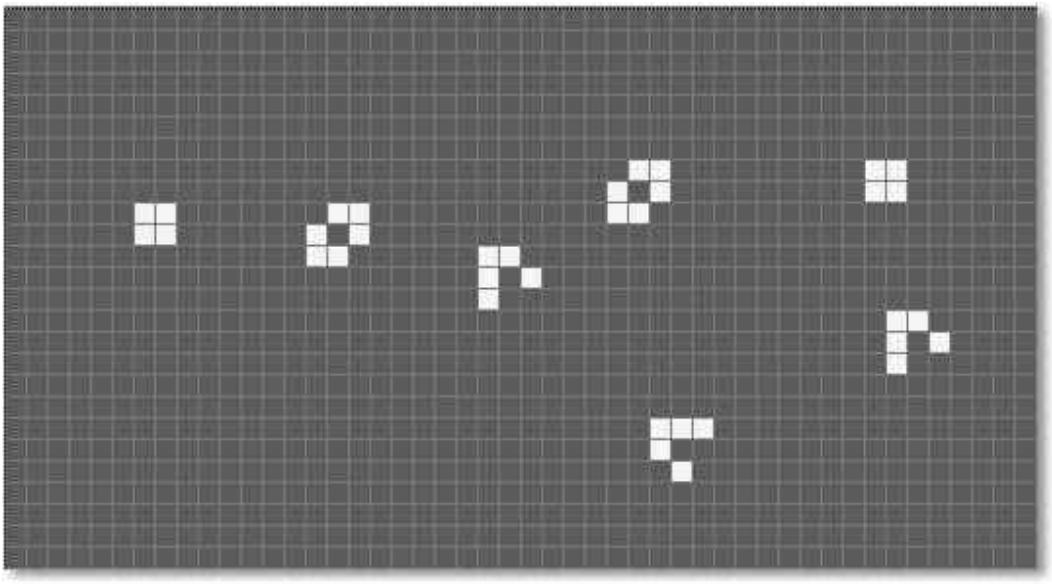


Abbildung 2.9 Eine interessante Startkonfiguration

Probieren Sie einmal, möglichst dynamische und viele Generationen überlebende Ausgangskonfigurationen zu finden. Sie werden überrascht sein, wie viele interessante Abläufe Sie dabei entdecken – hin und wieder bricht scheinbar alles in sich zusammen, um kurz darauf wieder in voller Blüte aufzuerstehen.

Eine andere interessante Möglichkeit besteht darin, die Simulation möglichst stark zu parallelisieren. Hierfür könnten Sie das Spielfeld in Quadranten unterteilen, die jeweils von einem eigenen Thread berechnet werden und für die Überführung von Lebewesen miteinander kommunizieren. Der Vorteil einer solchen Implementierung wäre die hohe Performance bei der Simulation sehr großer Spielfelder, da die Threads für die einzelnen Quadranten nur dann Prozessorzeit verbrauchen, wenn im Quadranten Veränderungen berechnet werden müssen. Dementsprechend legen sich die einzelnen Threads für nicht belebte oder statische Quadranten so lange schlafen, bis neue Dynamik aus einem Nachbarquadrant in den eigenen Quadranten überwechselt.

Ein schönes Beispiel für eine Implementierung dieser Idee finden Sie unter *http://www.sts.tu-harburg.de/ ax.wienberg/life/lifeIntern.html*.

Darüber hinaus gibt es beliebig viele Möglichkeiten zum Experimentieren. Verändern Sie einfach kurzerhand die Bedingungen Ihres »Spiel des Lebens« – probieren Sie Neues aus, oder stellen Sie Ihre ganz eigenen Regeln auf. Interessant wäre sicher auch eine Simulation von verschiedenen Lebewesen, die sich vielleicht nicht in jeder Runde gegen die direkten Nachbarn behaupten, sondern einfach umherziehen und nach anderen suchen. Dabei muss es nicht unbedingt beim Konzept eines zellulären Automaten bleiben; es wäre genauso denkbar, dass jedes Lebewesen für sich einen Automaten darstellt. Spielen Sie einfach ein bisschen – hier lässt sich auch künstliche Intelligenz mit ins Spiel bringen –, dazu erfahren Sie später noch mehr.

Falls Ihnen eine tolle Idee eingefallen ist, dann lassen Sie es mich wissen. An dieser Stelle haben Sie sehr viel Raum für Ihre eigene Kreativität. Seien Sie einmal ganz ehrlich – Gott spielen ist doch gar nicht so übel ... Natürlich wird es mit zunehmender Komplexität der Lebewesen und der Lebensbedingungen immer schwerer, ein interessantes Gleichgewicht herzustellen, aber alles, was richtig interessant wird, ist eben auch nicht total trivial – sonst wäre es ja langweilig.

»Der Utopist sieht das Paradies,
der Realist das Paradies plus Schlange.«

Friedrich Hebbel

3 Snake

In diesem Kapitel werden Sie das Spiel »Snake« programmieren. Es handelt sich dabei um ein einfaches 2D-Spiel, in dem Sie eine Schlange steuern. Ziel des Spiels ist es, alle verfügbaren Äpfel zu essen, ohne mit einem Hindernis zu kollidieren. Die Schlange selbst gilt hierbei ebenfalls als Hindernis und verlängert sich mit jedem verzehrten Apfel. Es wird also mit zunehmendem Verzehr von Äpfeln immer schwieriger, nicht mit sich selbst zu kollidieren.

3.1 Wie ist Snake zu gestalten?

Für dieses Spiel benötigen wir ein rechteckiges Spielfeld, in dem wir die Schlange der Einfachheit halber durch aneinandergehängte Quadrate darstellen. Zum Navigieren werten wir die Pfeiltasten aus und erlauben nur die vier Himmelsrichtungen als Richtungswahl. Entsprechend sieht die grafische Darstellung des Spiels aus wie in Abbildung 3.1.

Natürlich sind auch nicht rechteckige Spielfelder denkbar, genauso wie Sie die Fortbewegung der Schlange auch in einem beliebigen Winkel erlauben könnten. Ersteres erhöht den Spielspaß sicherlich etwas, während Letzteres in der Bedienung vermutlich zu kompliziert wird, so dass es die Freude am Spiel eher reduzieren würde. Falls Sie Gefallen an dem Spiel finden, so probieren Sie die beiden Alternativen doch einmal aus. Die Grundform des Spielfeldes wird allerdings wohl immer rechteckig sein und nur durch entsprechende Hindernisse davon abweichen.

Für die Hindernisse ist eine Kollisionserkennung notwendig. Wir werden als Hintergrundfarbe Schwarz verwenden und jeden anderen Farbton als Kollision werten. Dabei reservieren wir einen Farbton für die Äpfel, so dass in diesem Fall die Kollision die Schlange verlängert. Jede andere Kollision führt zum Spielende.

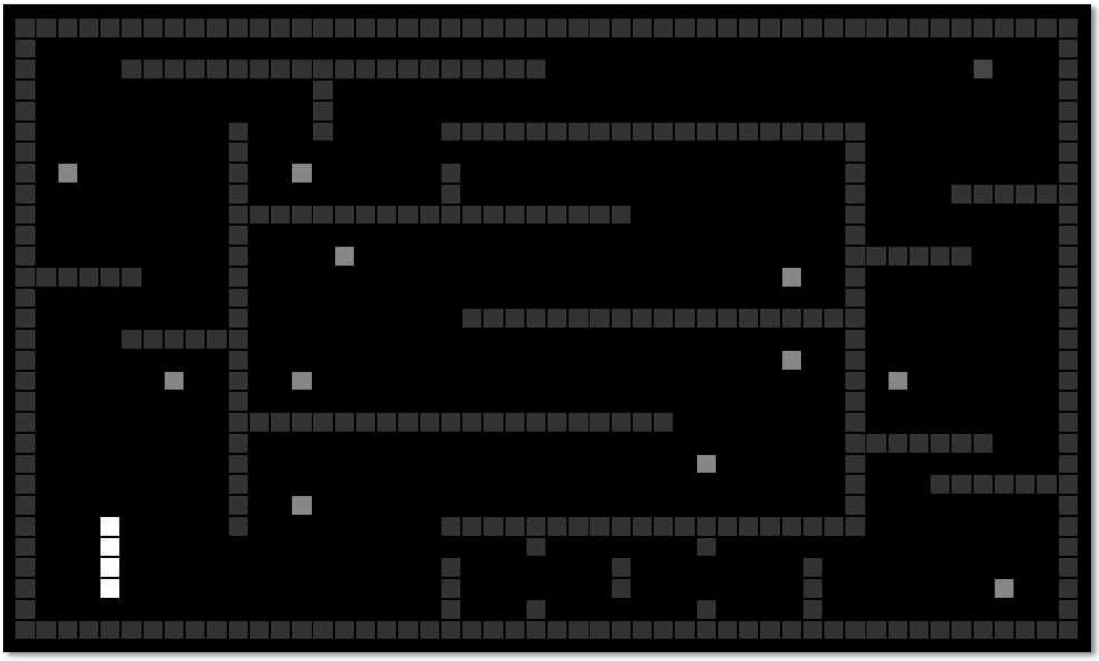


Abbildung 3.1 »Snake«

Um dem Spiel einen gewissen Reiz zu geben, fehlt nun noch eine Bewertung der erbrachten Leistung – oder kurz: eine Highscore-Tabelle. Hierfür eignet sich die Anzahl der gegessenen Äpfel in Kombination mit der abgelaufenen Zeit. Je Level ist eine gewisse Anzahl Äpfel zu vertilgen, bevor ein Ausgang erscheint, der in das nächste Level führt. Die Anzahl der zu vertilgenden Äpfel sowie der Hindernisse steigt je Level. Der Ausgang lässt sich ebenfalls als Spezialfall einer Kollision auswerten.

3.2 Definition des Levelformats

Für die Darstellung des Spielfeldes legen Sie am besten eine verschachtelte Liste an, die die einzelnen Felder genauer spezifiziert. Wie in den Vorüberlegungen bereits definiert wurde, repräsentieren die einzelnen Felder Hindernisse, Äpfel, begehbaren Weg oder den Ausgang. Alle passierbaren Felder – also der begehbare Weg – werden schwarz dargestellt. Äpfel sollen rot angezeigt werden und Hindernisse grau. Die Schlange selbst gilt ebenfalls als Hindernis, soll aber nicht grau, sondern grün gefärbt sein.

Festgelegter Wert	Farbe	Bedeutung	Kollisionsbehandlung
<Leerzeichen>	Schwarz	Weg	Schlange bewegen
A	Rot	Apfel	verlängern + bewegen
S	Grün	Schlange	Game over
H	Grau	Hindernis	Game over
E	Blau	Ausgang/Exit	next Level

Tabelle 3.1 Werte, Farben und deren Bedeutung

```
HHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHH
H                                                           H
H     HHHHHHHHHHHHHHHHHHHH                            E     H
H                H                                          H
H                H                                          H
H           H    H       HHHHHHHHHHHHHHHHHHHHH             H
H           H                                  H            H
H A         H   A        H                     H            H
H           H            H                     H     HHHHHHH
H           HHHHHHHHHHHHHHHHHHHH               H            H
H           H                                  H            H
H           H     A                            HHHHHHH      H
HHHHHHH     H                                A   H            H
H           H                                    H            H
H           H              HHHHHHHHHHHHHHHHHHHH              H
H     HHHHHHH                                    H            H
H           H                                A   H            H
H        A  H   A                                 H A          H
H           H                                    H            H
H           HHHHHHHHHHHHHHHHHHHHH                H            H
H           H                                    HHHHHHHH     H
H           H                          A         H            H
H           H                                    H    HHHHHHHH
H           H   A                                H            H
H    S      H            HHHHHHHHHHHHHHHHHHHH              H
H    s                        H         H                   H
H    s                   H         H          H             H
H    s                   H         H          H          A  H
H                        H    H         H     H             H
HHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHH
```

Listing 3.1 Die Levelkodierung als Text

Die Zustände der einzelnen Felder werden entsprechend des Feldwertes im Spielfeld gespeichert. Zur Darstellung des Spielfeldes zeichnen wir entsprechend der

Tabelle 3.1 jedes Feld in der definierten Farbe. Das Spielfeld soll zunächst 50 × 30 Felder groß sein, wobei für die grafische Darstellung pro Kästchen 10 Pixel reichen sollten. Die Definition der Feldwerte nehmen wir in einer Textdatei vor, die zum Beispiel *Level1.txt* heißen könnte. Diese und auch jede weitere Leveldefinition laden wir jeweils beim Levelwechsel aus der entsprechenden Datei. Als Vereinfachung soll sich die Schlange beim Starten eines Levels Richtung Norden bewegen, also auf dem Bildschirm nach oben kriechen.

Es handelt sich jeweils um die Anfangsbuchstaben des darzustellenden Objektes, also `H` für ein Hindernis, `S` für die Schlange – wobei das große `S` für den Kopf der Schlange steht, ein kleines `s` für den Körper –, `A` für den Apfel und `E` für »Exit« (da das A für »Ausgang« schon durch den »Apfel« belegt ist). In einem Texteditor in Proportionalschrift angezeigt, erscheint das Level hoch und schmal, da die Buchstaben höher als breit sind. Als Quadrate dargestellt ergibt sich jedoch ein breites und flaches Level, das mir in Zeiten von Breitbildschirmen angemessener erscheint. Der Einfachheit halber verzichten wir auf die Erstellung eines Leveleditors, da ein Texteditor für so einfache Spiele vollkommen genügt.

[●] Alle Quelltexte zu Snake finden Sie auf der CD unter *Kapitel03/snakeGame_v1.py*, *Kapitel03/snakeGame_v2.py*, *Kapitel03/snakeGame_v3.py*, *Kapitel03/snake.py* und *Kapitel03/playground.py*.

Zu Beginn des Spiels wird die Leveldatei geladen und das Spielfeld angezeigt. Die Schlange startet mit der Bewegung nach oben und ändert die Richtung entsprechend der Befehle des Spielers. Es werden nur die vier Himmelsrichtungen akzeptiert, wobei die aktuelle Richtung der Schlange jeweils nur um 90 Grad verändert werden kann. Der folgende Quelltext sorgt dafür, dass das Spielfeld eingelesen wird:

```
livingSpaceWidth    = 60
livingSpaceHeight   = 80

def loadLevel(fileName):
    livingSpace = []
    datei = open(fileName, "r")

    x = -1
    y = -1
    for zeile in datei:
        y += 1
        livingSpace.append([])
        for zeichen in zeile:
            if zeichen != '\n':
                x += 1
                livingSpace[y].append(zeichen)
```

```
    print "Level loaded... (", len(livingSpace[0]),
     " x ", len(livingSpace), ")"

    return livingSpace
```

Listing 3.2 Einlesen des Spielfeldes

Zunächst wird eine leere Liste angelegt, welche die das Spielfeld definierenden Hindernisse, Äpfel und sonstige Felder mit besonderer Bedeutung, aufnehmen wird. In den Variablen `x` und `y` sind die jeweils aktuellen Spielfeldkoordinaten gespeichert. Das Spielfeld wird zeilenweise geladen, wobei eine Zeile jeweils durch einen Zeilenumbruch gekennzeichnet ist. Zu guter Letzt wird das erfolgreiche Laden des Spielfeldes als Nachricht in der Konsole ausgegeben, wobei auch die Spielfeldgröße mit angegeben wird.

3.3 Darstellung des Spielfeldes

Die Darstellung des Spielfeldes übernimmt die Funktion `draw()`, wobei sie sich der Funktion `setRegardingColor()` bedient, um die für das jeweilige Feld korrekte Farbe auszuwählen.

```
def setRegardingColor(column, row):
    if      livingSpace[row][column] == "H":
            glColor4f(0.5, 0.5, 0.5, 1.0)
    elif    livingSpace[row][column] == "S" or
            livingSpace[row][column] == "s":
            glColor4f(0.0, 1.0, 0.0, 1.0)
    elif    livingSpace[row][column] == "A":
            glColor4f(1.0, 0.0, 0.0, 1.0)
    elif    livingSpace[row][column] == "E":
            glColor4f(0.0, 0.0, 1.0, 1.0)
    else:   glColor4f(0.0, 0.0, 0.0, 1.0)
```

Listing 3.3 Farbe entsprechend der Bedeutung des Feldes setzen

Die Buchstaben entsprechen den Buchstaben, die in der Leveldatei hinterlegt sind. Je nachdem, um welche Art von Feld es sich gerade handelt, wird eine entsprechende Farbe für das Feld gesetzt.

```
def draw():
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
    glLoadIdentity()
    glTranslatef(0.0, 0.0, 3.0)
```

```
    glBegin(GL_QUADS)
    for column in range(livingSpaceWidth):
        for row in range(livingSpaceHeight):
            setRegardingColor(column, row)
            x = column * 10.0
            y = row * 10.0
            glVertex3f(x, y, 0.0)
            glVertex3f(9.0 + x, y, 0.0)
            glVertex3f(9.0 + x, 9.0 + y, 0.0)
            glVertex3f(x, 9.0 + y, 0.0)
    glEnd()
```

Listing 3.4 Das Spielfeld zeichnen

Hier wird das Feld gezeichnet, wobei wir für die Felder kleine Quadrate zeichnen. Mehr zu OpenGL finden Sie im Anhang A, »OpenGL«.

3.4 Implementierung der Spielsteuerung

Jetzt fehlt uns nur noch die Steuerung des Spielablaufs. Dazu gehört zum einen das Einlesen der Benutzerbefehle, zum anderen deren Umsetzung während des Spielverlaufs. Da wir Richtungswechsel nur nach oben, unten, links und rechts erlauben, ist der Schlange keine 180-Grad-Wendung möglich. Entsprechend kann die Schlange nur nach links oder rechts abbiegen. Die Steuerung soll über die Pfeiltasten erfolgen und wird in der Funktion `handleEvent()` behandelt.

```
def handleEvent(event):
    if event.type == QUIT or
     (event.type == KEYDOWN and event.key == K_ESCAPE):
        return False
    global snakeDirection
    if event.type == KEYDOWN:
        if event.key == K_RIGHT:
            snakeDirection = ( +1,  0)
        if event.key == K_LEFT:
            snakeDirection = ( -1,  0)
        if event.key == K_UP:
            snakeDirection = (  0, -1)
        if event.key == K_DOWN:
            snakeDirection = (  0, +1)

    return True
```

Listing 3.5 Die Ereignisbehandlung

Hier wird – je nach Tastendruck – die Richtung `snakeDirection` unserer Schlange verändert. Die dabei zugewiesenen Tupel geben die neue Richtung in Form von Vektoren an, welche die Bewegung der Schlange für die x- und y-Koordinate des Spielfeldes repräsentieren.

Anhand der Benutzereingaben und des vorherigen Zustandes wird nun die neue Spielsituation berechnet. Hierfür wird die neue Position der Schlange ermittelt und eine Kollisionsüberprüfung durchgeführt. Falls es zu keiner Kollision kommt, so kriecht die Schlange ein Kästchen weiter in die vom Spieler vorgegebene Richtung; wäre allerdings eine Kollision die Folge, so wird das entsprechende Ereignis verarbeitet. Mögliche Ereignisse sind, wie bereits definiert, das Vertilgen eines Apfels und damit die Verlängerung der Schlange oder die Kollision mit einem Hindernis. Sind alle Äpfel verzehrt, gibt es noch die Möglichkeit, den Ausgang zu erreichen, was wir ebenfalls als Kollisionsereignis behandeln.

Doch setzen wir zunächst das Verrücken der Schlange ohne Kollisionsbehandlung um. Dies ist in der Funktion `moveSnake()` realisiert:

```
def moveSnake():
    for snakePart in snakePosition:
        x, y = snakePart
        livingSpace[y][x] = ' '

    print snakePosition
    x, y            = snakePosition[0]
    dx, dy          = snakeDirection
    newHeadPosition = (x + dx, y + dy)
    print newHeadPosition
    snakePosition.insert(0, newHeadPosition)
    snakePosition.pop()
    print snakePosition

    for snakePart in snakePosition:
        y, x = snakePart
        livingSpace[x][y] = 'S'
```

Listing 3.6 Die Schlange bewegen

Die Position des Schlangenkopfes ist das erste Element in der Liste `snakePosition` und dient als Ausgangspunkt für das Verschieben der Schlange. Die Kopfposition wird ausgelesen und die Zielposition durch Addition der Komponenten des Bewegungsvektors – der in `snakeDirection` hinterlegt ist – ermittelt und am Anfang der Liste als neues Tupel hinzugefügt. Danach wird das letzte Tupel der Liste entfernt. Durch diese Vorgehensweise ersparen wir uns ein aufwendiges und unnötiges Verschieben aller Elemente der Schlange. Hier nutzen wir den Vorteil,

dass eine virtuelle Schlange sich auch dann bewegt, wenn wir ein Teil vom Ende abtrennen und vorn ein neues Teil hinzufügen.

Die Schlange bewegt sich jetzt noch vollkommen ungebremst, denn wir müssen natürlich erst noch eine Kollisionskontrolle einbauen.

3.5 Kollisionen erkennen und behandeln

Was tun, wenn es kracht? Ganz einfach: In den meisten Fällen ist das Spiel dann vorbei; in einigen Fällen wird es einfach nur schwieriger für den Spieler, und zwar immer dann, wenn die Schlange an Länge gewinnt. Um den Schwierigkeitsgrad beim Verzehr von Äpfeln weiter zu erhöhen, könnten wir auch die Spielgeschwindigkeit anheben.

Um Kollisionen zu bestimmen, werten wir beim Verschieben der Schlange das Zielfeld aus. Sofern dieses nicht leer ist, ist eine besondere Behandlung notwendig, die in Tabelle 3.1 am Anfang des Kapitels beschrieben wurde. Zunächst hinterlegen wir alle interessanten Positionen bereits beim Laden der Koordinaten der Felder in verschiedenen Listen. Dazu passen Sie die Funktion `loadLevel()` wie folgt an:

```
def loadLevel(fileName):
    global appleCount, livingSpace,
      livingSpaceWidth, livingSpaceHeight
    appleCount = 0
    livingSpace = []
    datei = open(fileName, "r")

    y = -1
    for zeile in datei:
        x = -1
        y += 1
        livingSpace.append([])
        for zeichen in zeile:
            if zeichen != '\n':
                x += 1
                livingSpace[y].append(zeichen)
                if zeichen == 'A':
                    applePositions.append((x, y))
                elif zeichen == 'S':
                    snakePosition.append((x, y))
                elif zeichen == 's':
                    snakePosition.insert(0, (x, y))
                elif zeichen == 'H':
```

```
                barPositions.append((x,y))
            elif zeichen == 'E':
                exitPosition.append((x,y))

    livingSpaceWidth    = len(livingSpace[0])
    livingSpaceHeight   = len(livingSpace)
```

Listing 3.7 Ein Level laden

Die einzelnen Listen definieren wir bereits ganz zu Beginn im Zusammenhang mit `snakePosition` und `applePositions`. Darüber hinaus legen wir noch ein paar Konstanten für die verschiedenen Ereignisse an. Die globalen Definitionen sehen nun wie folgt aus.

```
NOCOLLISION =  0
COLLISION   = -1
EATAPPLE    = -2
PASSEXIT    = -3

livingSpace = []
livingSpaceWidth = 0
livingSpaceHeight = 0
creatureSize = 20
snakeDirection = ( 0, 0)

applePositions = []
barPositions   = []
snakePosition  = []
exitPosition   = []

fileName = "level1.txt"
```

Listing 3.8 Initialisierung von »Snake«

Um Kollisionen behandeln zu können, definieren wir jetzt die neue Funktion `collisionDetection()`. Ihr übergeben wir nur einen Parameter, der die Koordinaten der Position als Tupel enthält. In diesem Stadium erzeugt diese Funktion zunächst nur Ausgaben auf der Konsole und gibt bei keiner Kollision den Wert `NOCOLLISION` zurück, der global als `0` definiert ist.

```
def collisionDetection(position):
    if position in snakePosition:
        print "self collision"
        return COLLISION

    if position in barPositions:
```

```
        print "bar collision"
        return COLLISION

    if position in applePositions:
        print "apple collision"
        return EATAPPLE

    if position in exitPosition:
        print "exit collision"
        return PASSEXIT

    return NOCOLLISION
```

Listing 3.9 Kollisionen erkennen

Die neue Funktion wird direkt nach der Berechnung der `newHeadPosition` innerhalb von `moveSnake()` aufgerufen:

```
def moveSnake():
    for snakePart in snakePosition:
        x, y = snakePart
        livingSpace[y][x] = ' '

    print snakePosition
    x, y            = snakePosition[0]
    dx, dy          = snakeDirection
    newHeadPosition = (x + dx, y + dy)
    collisionDetection(newHeadPosition)
    snakePosition.insert(0, newHeadPosition)
    snakePosition.pop()

    for snakePart in snakePosition:
        y, x = snakePart
        livingSpace[x][y] = 'S'
```

Listing 3.10 Die Schlange bewegen

Die Kollisionen an sich werden noch immer nicht behandelt, aber sind nun schon auf der Konsole sichtbar. Als Nächstes werden wir anhand der Rückgabewerte eine Behandlung der verschiedenen Kollisionen implementieren. Zuerst kümmern wir uns um den Apfelverzehr. Dazu passen wir `moveSnake()` wie folgt an:

```
def moveSnake():
    for snakePart in snakePosition:
        x, y = snakePart
        livingSpace[y][x] = ' '
```

```
x, y            = snakePosition[0]
dx, dy          = snakeDirection
newHeadPosition = (x + dx, y + dy)

collisionEvent  = collisionDetection(newHeadPosition)
if collisionEvent == EATAPPLE:
    global snakeExpansion
    snakeExpansion += 4

snakePosition.insert(0, newHeadPosition)
if snakeExpansion == 0:
    snakePosition.pop()
else:
    snakeExpansion -= 1

for snakePart in snakePosition:
    y, x = snakePart
    livingSpace[x][y] = 'S'
```

Listing 3.11 Anpassung zum Verzehr von Äpfeln

Die Variable `snakeExpansion` fügen wir ebenfalls den globalen Definitionen hinzu und initialisieren sie mit dem Wert `0`. Mit jedem Verzehr eines Apfels wird dieser Wert um `4` erhöht, wodurch sich die Schlange um genau vier Blöcke verlängert. Das letzte Element wird nur noch dann freigegeben, wenn `snakeExpansion` den Wert `0` enthält. Beim Verschieben wird die Schlange so automatisch länger, weil das jeweils letzte Element gesperrt ist, bis `snakeExpansion` aufgrund der Dekrementierung wieder den Wert `0` enthält.

Inzwischen ist das Spiel wegen der zunächst einfacheren, aber auf lange Sicht nicht überschaubaren prozeduralen Programmierung unübersichtlich geworden. Darum führen wir vor der weiteren Behandlung von Kollisionsfällen die Klasse `Snake` ein, die alle für die Schlange relevanten Daten und Operationen zusammenführt. So werden wir auch einige der globalen Definitionen los, indem wir sie durch eine globale Definition einer Instanz der neuen Klasse `Snake` ersetzen. Diesen wichtigen Refactoring-Schritt erläutere ich im nächsten Abschnitt als Exkurs.

Die Umsetzung bis hierher finden Sie auf der CD unter *Kapitel03/snakeGame_v1.py*. **[●]**

3.6 Exkurs: Refactoring von Software

Software wird immer inkrementell entwickelt; auch wenn ursprünglich alles als vorausgeplant gilt, wird sich immer die eine oder andere unvorhergesehene Änderung ergeben. Deshalb ist es auch nicht schlimm, zunächst den einfachsten Weg zu gehen, um dann bei Gelegenheit mal wieder aufzuräumen. Denn ohne bereits eine lauffähige Software zu haben, ist es meist nicht möglich, überhaupt den richtigen Weg einzuschlagen – es fehlt einfach das Feedback durch die Software, die es ja noch nicht gibt.

Entsprechend haben wir Snake zunächst ohne weitgehende Designüberlegungen so einfach wie möglich programmiert, bis wir den Zeitpunkt erreicht haben, an dem jede Änderung ein bisschen komplizierter geworden ist. Jetzt ist es also an der Zeit, ein bisschen aufzuräumen und wie bereits angekündigt die Klasse `Snake` zu erstellen. Die Klasse umfasst alle Operationen für die darzustellende Schlange und sieht wie folgt aus:

```
class Snake(object):
    '''
    represents a snake in our snake game
    '''
    NOCOLLISION =  0
    COLLISION   = -1
    EATAPPLE    = -2
    PASSEXIT    = -3

    def __init__(self, playground):
        self.playground = playground
        self.expansion = 0

    def move(self, direction):
        x, y = self.playground.snakePosition[0]
        dx, dy = direction
        newHeadPosition = (x + dx, y + dy)
        collisionEvent = self.playground.
         collisionDetection(newHeadPosition)
        if collisionEvent == Snake.EATAPPLE:
            self.expansion += 4
        self.playground.snakePosition.insert(0, newHeadPosition)
        if self.expansion == 0:
            self.playground.snakePosition.pop()
        else:
            self.expansion -= 1
```

Listing 3.12 Die Klasse »Snake«

Parallel zur Klasse `Snake` haben wir auch gleich eine Klasse `Playground` definiert, die sich um alle Umgebungsdaten und Operationen kümmert. Die Klasse `Playground` übergeben wir bei der Erzeugung der Klasse `Snake` als Parameter. Außerdem haben wir die verschiedenen parallel stattfindenden Behandlungen separiert. So wird nun nicht mehr während des Verschiebens gezeichnet, sondern im Anschluss an alle zustandsändernden Verarbeitungsschritte. Auch den Speicherverbrauch konnten wir reduzieren, indem wir nicht alle Felder des Spielfeldes speichern, sondern nur noch jene, die nicht leer sind. Die Klasse `Playground` sieht nun so aus:

```
from coding4fun.kapitel3.snake import Snake

class Playground(object):
    '''
    represents the playground in our snake game
    '''

    def __init__(self):
        self.width          = 0
        self.height         = 0
        self.applePositions = []
        self.barPositions   = []
        self.exitPosition   = []
        self.snakePosition  = []

    def loadLevel(self, fileName):
        datei = open(fileName, "r")
        y = -1
        for zeile in datei:
            x = -1
            y += 1
            for zeichen in zeile:
                if zeichen != '\n':
                    x += 1
                    if zeichen == 'A':
                        self.applePositions.append((x, y))
                    elif zeichen == 'S':
                        self.snakePosition.append((x, y))
                    elif zeichen == 's':
                        self.snakePosition.insert(0, (x, y))
                    elif zeichen == 'H':
                        self.barPositions.append((x,y))
                    elif zeichen == 'E':
                        self.exitPosition.append((x,y))
```

```
        self.width  = x
        self.height = y
        print self.width, " -> ", self.height

    def collisionDetection(self, newHeadPosition):
        if newHeadPosition in self.snakePosition:
            print "self collision"
            return Snake.COLLISION

        if newHeadPosition in self.barPositions:
            print "bar collision"
            return Snake.COLLISION

        if newHeadPosition in self.applePositions:
            print "apple collision"
            return Snake.EATAPPLE

        if newHeadPosition in self.exitPosition:
            print "exit collision"
            return Snake.PASSEXIT

        return Snake.NOCOLLISION
```

Listing 3.13 Die Klasse »Playground«

Hier kommen die oben bereits beschriebenen Funktionen alle unter ein Dach: Aus den Funktionen sind jetzt Methoden geworden, welche zusammen mit den relevanten Daten die Klasse `Playground` bilden.

Wie Ihnen sicher aufgefallen ist, wird in den Klassen `Playground` und `Snake` überhaupt nicht gezeichnet. Jeglicher OpenGL-Code ist in unserem ursprünglichem Modul verblieben, das sich nun unter Verwendung der beiden neuen Klassen hauptsächlich in der Funktion `main()` verändert hat:

```
def main():
    pygame.init()
    video_flags = OPENGL | HWSURFACE | DOUBLEBUF

    playground.loadLevel("level1.txt")
    screenSize = (playground.width * 20, playground.height * 20)
    pygame.display.set_mode(screenSize, video_flags)
    resize(screenSize)

    init()
```

```
    while True:
        if not handleEvent(pygame.event.poll()):
            break

        clearScreen()
        drawElement(playground.barPositions, (0.5, 0.5, 0.5))
        drawElement(playground.applePositions, (1.0, 0.0, 0.0))
        drawElement(playground.exitPosition, (0.0, 0.0, 1.0))
        drawElement(playground.snakePosition, (0.0, 1.0, 0.0))

        if direction != (0, 0):
            snake.move(direction)

        pygame.display.flip()
        pygame.time.delay(500)
```

Listing 3.14 Der Hauptablauf unter Verwendung der neuen Klassen

Darüber hinaus entfallen viele der globalen Definitionen und wurden durch die Initialisierung der neuen Klassen ersetzt:

```
from coding4fun.kapitel3.snake import Snake
from coding4fun.kapitel3.playground import Playground

direction   = (0, 0)
playground  = Playground()
snake       = Snake(playground)
```

Listing 3.15 Die Inititialisierung mit den neuen Klassen

Die Funktion `draw()` wurde – wie Sie in Listing 3.14 sehen – durch `drawElement()` und die Funktion `clearScreen()` ersetzt. Zudem sind einige obsolete Funktionen entfallen – so werden zum Beispiel die Farben fix an `drawElement()` übergeben, anstatt über eine separate Funktion gesetzt zu werden.

```
def clearScreen():
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
    glLoadIdentity()
    glTranslatef(0.0, 0.0, 3.0)
```

Listing 3.16 Das Löschen des Bildschirmes

```
def drawElement(element, color):
    r, g, b = color
    glColor3f(r, g, b)
    glBegin(GL_QUADS)
```

```
    for part in element:
        x, y    = part
        x       = x * 10.0
        y       = y * 10.0
        glVertex3f(x, y, 0.0)
        glVertex3f(9.0 + x, y, 0.0)
        glVertex3f(9.0 + x, 9.0 + y, 0.0)
        glVertex3f(x, 9.0 + y, 0.0)
    glEnd()
```

Listing 3.17 Zeichnen einzelner Quadrate für die Darstellung von Snake

So weit zum Refactoring – jetzt werden Sie sich wieder mit der Kollisionsbehandlung auseinandersetzen, die wir im folgenden Abschnitt ausbauen.

3.7 Tödliche Kollisionen

Nun fehlen uns nur noch die Kollisionen, die das Spiel schlagartig beenden. Dabei unterscheiden wir im Folgenden zwischen tödlichen Kollisionen und jenen, die zum Erreichen des nächsten Levels führen, also Kollisionen mit dem Feld für den Ausgang. Außerdem werden wir den Code so anpassen, dass der Ausgang erst nach Verzehr aller Äpfel sichtbar wird. Diese Aufgabe lässt sich leicht durch folgende Anpassung innerhalb von `main()` bewerkstelligen:

```
if not playground.applePositions:
    drawElement(playground.exitPosition,    (0.0, 0.0, 1.0))
```

Listing 3.18 Den Ausgang nach Verzehr aller Äpfel darstellen

Hier nutzen wir die Tatsache, dass Python leere Listen als `False` auswertet. Bei der Gelegenheit ist Ihnen vielleicht aufgefallen, dass der Ausgang nie angezeigt werden wird, denn momentan verlängern wir zwar die Schlage, aber vertilgen die Äpfel nicht korrekt: Sie werden nicht vom Bildschirm gelöscht. Diese kleine Ungenauigkeit beim Verzehr ist aber leicht gelöst und erfolgt durch Anpassung der Methode `collisionDetection` unserer neuen Klasse `Playground`:

```
if newHeadPosition in self.applePositions:
    # der Verzehr von Äpfeln
    print "apple collision"
    self.applePositions.remove(newHeadPosition)

    return Snake.EATAPPLE
```

Listing 3.19 Die Äpfel restlos verzehren

Wir mussten nichts weiter tun, als die Position des vertilgten Apfels aus der entsprechenden Liste zu löschen. Nun können Sie gleich eine kleine Testrunde einlegen, um zu sehen, ob nach Vertilgen aller Äpfel auch der Ausgang sichtbar wird.

Falls Ihnen das momentane Tempo etwas zu träge ist, so reduzieren Sie einfach die Zeitspanne des Wartens innerhalb von `main()`.

```
pygame.time.delay(200)
```

Listing 3.20 Die Geschwindigkeit reduzieren

Auch ohne große Spielerfahrung stört Sie vermutlich schon die träge Reaktion auf Ihre Tastendrücke. Der Grund hierfür ist zum einen, dass nur der jeweils letzte Tastendruck nach dem Warten verarbeitet wird. Um hier schon vorab Befehle zu erlauben, die nicht einfach so im Rauschen untergehen, stellen wir im Folgenden alle Tastendrücke in eine Queue ein, die der Reihe nach abgearbeitet wird. Listing 3.21 zeigt Ihnen wie das geht:

```
def handleEvent(event):
    if event.type == QUIT or
     (event.type == KEYDOWN and event.key == K_ESCAPE):
        return False

    global commands
    if event.type == KEYDOWN:
        if event.key == K_RIGHT:
            commands.append(( +1,  0))
        if event.key == K_LEFT:
            commands.append(( -1,  0))
        if event.key == K_UP:
            commands.append((  0, -1))
        if event.key == K_DOWN:
            commands.append((  0, +1))

    return True
```

Listing 3.21 Puffern der Eingabe

Die Befehle nehmen wir in die Liste `commands` auf, die wir als Queue verwenden und global definieren. Daraufhin füllen wir in `main()` aus dieser Queue die globale Variable `direction`:

```
global direction
if commands:
    direction = commands.pop(0)
```

```
if direction != (0, 0):
    snake.move(direction)
```

Listing 3.22 Die Richtung entsprechend der Eingabe setzen

Sie werden langsam merken, dass es sinnvoll wäre, auch die Spielsteuerung in eine Klasse zu verlagern, aber im Augenblick werden wir darauf verzichten, da die Komplexität noch sehr gut zu bewältigen ist. Das volle Ausschöpfen aller Möglichkeiten für Entwurfsmuster ist nicht zu empfehlen, da diese ansonsten nicht vereinfachen, sondern sogar verkomplizieren. Nicht alles, was prinzipiell über Model-View-Controller machbar ist, sollte auch wirklich entsprechend umgesetzt werden.

Trotz Queue ist immer noch eine gewisse Trägheit spürbar, die darauf beruht, dass wir die Befehle im gleichen Takt einlesen wie den Spielverlauf. Darum werden wir im Folgenden beide Arbeitsschritte entkoppeln und den Spielverlauf nur alle vier Takte fortsetzen. Dafür definieren wir global die Variable `gameCycle` und passen `main()` wie folgt an:

```
while True:
    if not handleEvent(pygame.event.poll()):
        break

    global gameCycle
    gameCycle += 1
    gameCycle %= 4
    if gameCycle == 0:
        clearScreen()
        drawElement(playground.barPositions, (0.5, 0.5, 0.5))
        drawElement(playground.applePositions, (1.0, 0.0, 0.0))
        drawElement(playground.snakePosition, (0.0, 1.0, 0.0))
        if not playground.applePositions:
            drawElement(playground.exitPosition, (0.0, 0.0, 1.0))

        global direction
        if commands:
            direction = commands.pop(0)
        if direction != (0, 0):
            snake.move(direction)

        pygame.display.flip()
    pygame.time.delay(50)
```

Listing 3.23 Eine höhere Abtastrate für Eingaben

Durch die Änderung wird das Spielfeld nur bei jedem vierten Durchlauf, also für die im Listing verwendete Zeitspanne von 50ms, alle 200ms aktualisiert. Entsprechend werden die Tastatureingaben viermal häufiger entgegengenommen, als das Spiel fortschreitet.

Das Ergebnis ist zufriedenstellend, und wir können uns jetzt den noch verbleibenden Kollisionen widmen. Bei Kollisionen mit Hindernissen soll einfach »Game over« eingeblendet werden. Für diese Aufgabe werden wir nun innerhalb der Methode `move()` unserer Klasse `Snake` das `collisionEvent` zurückgeben, um es in der Hauptschleife `main()` weiterverarbeiten zu können.

```
if direction != (0, 0):
    collisionEvent = snake.move(direction)
    if collisionEvent == Snake.COLLISION:
        showGameOver()
```

Listing 3.24 Eine tödliche Kollision

Die Funktion `showGameOver()` zeigt für drei Sekunden »Game over« an und startet das Spiel dann neu. Dazu wird innerhalb von `showGameOver()` die Funktion `showASCIIArt()` mit der darzustellenden Datei und 3000 aufgerufen, wobei 3000 für 3000ms seht.

```
def showGameOver():
    showASCIIArt("GameOver.txt", 3000)
    restartGame()
```

Listing 3.25 »Game Over« einblenden und das Spiel neu starten

Die Funktion `showASCIIArt()` lädt die Anzeige aus der Datei *GameOver.txt* und stellt den Inhalt für drei Sekunden – also den übergebenen 3000ms – auf dem Bildschirm dar, bevor das Spiel neu gestartet wird.

```
def showASCIIArt(fileName, delay):
    datei = open(fileName, "r")
    asciiArt = []
    y = -1
    for zeile in datei:
        x = -1
        y += 1
        for zeichen in zeile:
            if zeichen != '\n':
                x += 1
                if zeichen == 'X':
                    asciiArt.append((x, y))
```

```
    clearScreen()
    drawElement(asciiArt, (1.0, 0.0, 0.0))
    pygame.display.flip()
    pygame.time.delay(delay)
```

Listing 3.26 ASCII-Art darstellen

Die Funktion `showASCIIArt()` zeigt eine in ASCII-Art definierte Grafik an. Sie verwenden sie für den »Game over«- und »You won«-Screen. Für den Neustart setzen wir lediglich alle Zustände zurück und laden die Leveldatei neu.

```
def resetGameState():
    # reinitialize the game (restart)
    global direction, commands, playground, snake
    playground  = Playground()
    snake       = Snake(playground)
    direction   = (0, 0)
    commands    = []
    pygame.event.clear()
```

Listing 3.27 Einen Neustart einleiten

Wenn Sie bereits ein bisschen getestet haben, dann ist Ihnen sicher aufgefallen, dass die Schlange erst gezeichnet wird und dann verschoben, was dazu führt, dass die Befehle des Benutzers häufig ein bisschen zu spät erteilt werden. Entsprechend schwer dürfte es Ihnen beim Spielen bisher gefallen sein, alle Äpfel zu essen und den Ausgang zu erreichen – die Darstellung ist dem Spielzustand bisher immer ein Feld zurück. Das ist schnell korrigiert, indem wir einfach die Schlange verschieben, bevor wir sie zeichnen:

```
global direction
if commands:
    direction = commands.pop(0)
if direction != (0, 0):
    collisionEvent = snake.move(direction)
    if collisionEvent == Snake.COLLISION:
        showGameOver()

clearScreen()
drawElement(playground.barPositions,    (0.5, 0.5, 0.5))
drawElement(playground.applePositions,  (1.0, 0.0, 0.0))
drawElement(playground.snakePosition,   (0.0, 1.0, 0.0))
if not playground.applePositions:
    drawElement(playground.exitPosition,    (0.0, 0.0, 1.0))
```

Listing 3.28 Reihenfolge der Aktualisierung von »Snake« anpassen

Was nun noch fehlt, ist das Erreichen des nächsten Levels nach Vertilgen aller Äpfel und Betreten des Ausgangs. Die Implementierung gestaltet sich analog zu den bereits vorgenommenen Kollisionsbehandlungen.

Programmieren wir zuerst die Kollision mit dem Ausgang, wofür wir `main()` wie folgt anpassen:

```
if direction != (0, 0):
    collisionEvent = snake.move(direction)
    if collisionEvent == Snake.COLLISION:
        showGameOver()
    if collisionEvent == Snake.PASSEXIT:
        if not playground.applePositions:
            showYouWon()
```

Listing 3.29 Kollision mit dem Ausgang

Hierfür war zu prüfen, ob alle Äpfel vertilgt wurden, bevor der Ausgang passiert wird. Denn dem Spiel ist der Ausgang immer bekannt – er wird, wenn er auch passierbar ist, für den Spieler lediglich eingeblendet.

Damit haben Sie Snake weitgehend vollständig implementiert und können es bei Interesse beliebig erweitern. Sofern Sie jedoch vorhaben, das Spiel stark auszubauen, empfiehlt es sich, ab jetzt wieder ein bisschen aufzuräumen und einen Controller für das Spiel zu implementieren. Je nachdem, wie fortgeschritten Ihre Version am Ende werden soll, ist es auch sinnvoll, die grafische Darstellung besser zu entkoppeln.

Ein Hinweis noch: Momentan ist es möglich, direkt in die entgegengesetzte Richtung zu steuern und somit sofort ein »Game over« zu provozieren. Das ist weder schön noch üblich für Snake und lässt sich durch die folgende Änderung leicht beheben:

```
global direction
if commands:
    validCommand = False
    while not validCommand and commands:
        new_x, new_y = commands.pop(0)
        old_x, old_y = direction
        if not ((abs(new_x) == abs(old_x)) and
         (abs(new_y) == abs(old_y))):
            validCommand  = True

    if validCommand:
        direction = (new_x, new_y)
```

Listing 3.30 Hier wird die 180-Grad-Wende verboten

Die Änderung funktioniert ganz einfach, indem sie die absoluten Beträge der beiden Richtungskomponenten vergleicht. Wenn die Beträge gleich sind, dann war es entweder die gleiche Richtung oder die exakt entgegengesetzte Richtung. In beiden Fällen wird der Befehl aus der Liste entfernt und der nächste geladen.

Falls Sie das Spiel zu einfach finden, so reduzieren Sie schlicht die Verzögerung durch `pygame.time.delay(25)` in der Funktion `main()`. Der angegebene Wert von 25 ms Verzögerung war für mich das maximal Schaffbare, ein Wert von 30 ms lag noch in der Komfortzone – wie schnell sind Sie? Falls Sie das Spiel ausbauen, aber auf andere Leveldesigns verzichten möchten, so heben Sie einfach die Level durch Reduzierung der Verzögerung an. Für diese Vorgehensweise würde sich eine schrittweise Reduzierung um 5 oder 10 ms gut eignen – wobei Sie fairerweise bei 50 ms anfangen sollten. Für einen Highscore könnten Sie entweder die zurückgelegte Strecke – was auch der benötigten Zeit entspräche – oder die Anzahl der abgegebenen Befehle sowie das erreichte Level einbeziehen.

3.8 Das Paradies erweitern

Ein paar Anregungen für Erweiterungen habe ich Ihnen bereits im letzten Abschnitt gegeben. Jetzt wollen wir Snake um ein Level ergänzen, wobei wir auch die Spielgeschwindigkeit erhöhen wollen. Das neue Level legen wir allerdings noch klassisch über das Textformat an. Im Anschluss an die »klassische« Erweiterung können Sie sich gerne auch an einem Levelgenerator versuchen, der rein zufällige Level erzeugt. Der Vorteil eines Generators wäre sicherlich, dass es nie zu langweiligen Wiederholungen kommt. Dafür ist für die Generierung neuer Level allerdings einiges zu beachten – denn es wäre schade, wenn Sie feststellen müssten, dass der letzte Apfel oder der Ausgang leider von Mauern eingezäunt ist.

Aber beginnen wir zunächst mit der einfachen Variante. Verwenden Sie entweder das folgende Level als fertige Textdatei, oder lassen Sie Ihrer Fantasie freien Lauf und entwickeln ein eigenes Level mit den für unsere Implementierung passenden Größenverhältnissen. Mit ein paar kleinen Anpassungen können Sie aber auch beliebige Spielfeldgrößen verwenden. Falls Ihnen die Seitenwände nicht zusagen, so wandeln Sie den Lebensraum doch einfach in eine Donut-Welt (siehe Kapitel 2, »Game of Life«) um, so dass die Schlange, wenn sie das Fenster auf einer Seite verlässt, auf der gegenüberliegenden Seite wieder auftaucht.

```
HHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHH
H     H                                                     H
H     HHHHHHHHHHHHHHHHHHHH                        E         H
H       A       H                                           H
H               H                                           H
H           H   H      HHHHHHHHHHHHHHHHHHHHH                H
H           H                                 AH            H
H           H           H                       H           H
H           H           H      HHHHHHH          H    HHHHHHH
H           H           H      H      H         H           H
H           H           H      H      H         H    HHH    H
H           H     A     H      HHHHHHH          H    H      H
HHHHHH      H           H                  A    H    H      H
H A         H   HHHH    HA                       H    H      H
H           H       H   HHHHHHHHHHHHHHHHHHHHH    H    H      H
H     HHHHHH        H                            H    HHH    H
H           H       H              H             H      H    H
H           H   A   H              H       A     H      H    H
H           H                      H             H    A H    H
HHHHHHH     HHHHHHHHHHHHH          HHHHHHHHHHHHH       H    H
H       H   H                      H             HHHHHHH    H
H       H   H                      H     A       H          H
H           H                      H             H    HHHHHHH
H           H   A                                 H          H
H    S      H           HHHHHHHHHHHHHHHHHHHHH              H
H    S      H        HHH     H    A     H                  H
H    S      HHHH    H        H          H                  H
H    S                       HHH    HHHH                   H
H                                                          H
HHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHH
```

Listing 3.31 Das zweite Level in ASCII-Art

Um in das nächste Level zu gelangen, ist ein erfolgreiches Absolvieren des aktuellen Levels notwendig. Entsprechend ließe sich das nächste Level nach dem »You won!«-Screen laden. Hier ergibt sich nun eine Fallunterscheidung zwischen der Wiederholung des letzten Levels und dem Laden des nächsten Levels. Für diese Unterscheidung führen wir die Variable `currentLevel` ein, die die Nummer des aktuellen Levels enthält. Zudem wird wie bereits oben beschrieben der Spielzustand jeweils durch Aufruf von `resetGameState()` zurückgesetzt.

Die Aufrufe kennen Sie bereits aus der Funktion `restartGame()`, die sich jetzt unter Zuhilfenahme der neuen Funktion so verkürzt:

```
def restartGame():
    currentLevel = 1
    loadNextLevel(currentLevel)
```

Listing 3.32 Ein neues Spiel starten

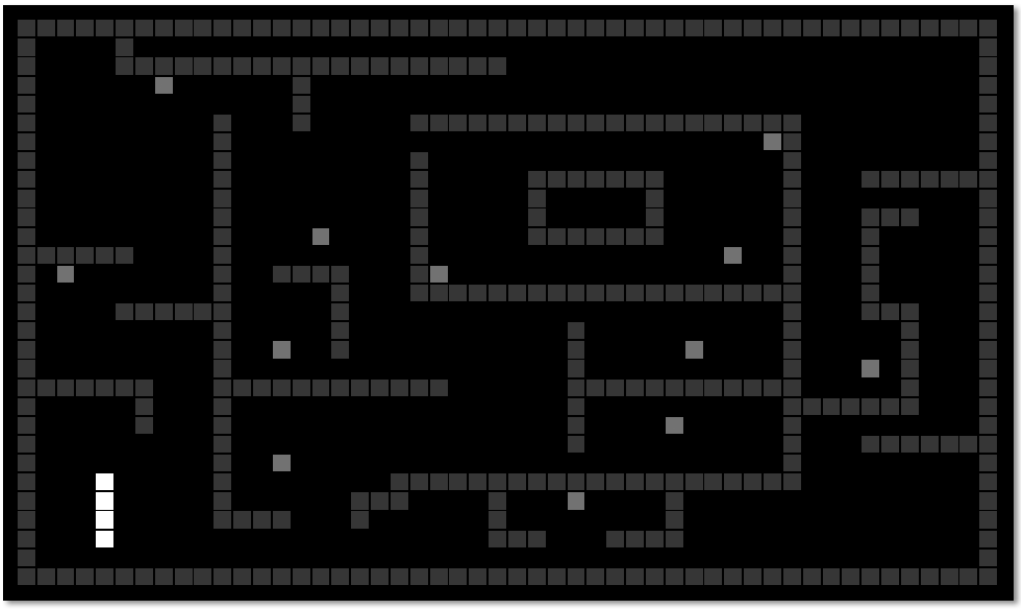


Abbildung 3.2 Die Bildschirmdarstellung des zweiten Levels

Um im Falle des Erfolgs das nächste Level zu laden, rufen wir nun die ebenfalls neu hinzugekommene Funktion `loadNextLevel()` auf, die nun auch in der angepassten Variante von `restartGame()` auftaucht:

```
def loadNextLevel(nextLevel):
    resetGameState()
    playground.loadLevel("level%d.txt" % nextLevel)
```

Listing 3.33 Das nächste Level laden

Sobald der Spieler nun die positive Mitteilung seines Sieges sieht, wird im Nachhinein direkt das nächste Level geladen. Hier könnten wir natürlich auch die Meldung in »Loading next Level« ändern. Zudem fehlt in der angegebenen Implementierung noch die Überprüfung, ob weitere Level zur Verfügung stehen. Sollten alle vorhandenen Level durchgespielt worden sein, so könnten Sie entweder bei Level 1 neu starten oder dem Spieler eine Punktwertung geben, die Sie optional in einer Art Highscore oder »Hall of Fame« verzeichnen. Die zwei Funktionen für die Ergebnisrückmeldung gestalten sich nun wie folgt:

```
def showGameOver():
    showASCIIArt("GameOver.txt", 3000)
    restartGame()

def showYouWon():
    global currentLevel
    currentLevel += 1
    showASCIIArt("YouWon.txt", 5000)
    loadNextLevel(currentLevel)
```

Listing 3.34 Darstellung des Spielergebnisses

Also dann viel Spaß beim Spielen und Ausbauen des Spiels!

»If privacy is outlawed, only outlaws will have privacy.«

Phil Zimmermann

4 Lszqupmphjf?!

In diesem Kapitel werden Sie etwas über die Geschichte der Kryptographie, Kryptologie und Steganographie erfahren und daraufhin ein paar einfache kryptographische Verfahren implementieren. Die verwendeteten Verfahren werden dabei im Laufe des Kapitels sukzessive komplexer. Zum Abschluss des Themas werden Sie das RSA-Verfahren umsetzen. Vielleicht soviel vorab: Bei RSA-Verfahren handelt sich um ein asymmetrisches Verschlüsselungsverfahren, also ein Verfahren, das ohne Schlüsselaustausch auskommt, weil der Schlüssel zum Verschlüsseln öffentlich bekannt sein darf, während der zum Entschlüsseln geheim gehalten wird. Ein symmetrisches Verfahren verwendet im Gegensatz dazu den gleichen Schlüssel für die Verschlüsselung sowie für die Entschlüsselung, weshalb ein Austausch des Schlüssels erforderlich ist. Wenn die kommunizierenden Parteien aber nicht ihren öffentlichen Schlüssel verifizieren, so ist es gegen eine sogenannte Man-in-the-Middle-Attack nicht gerüstet. Die Gefahr besteht einfach darin, dass der öffentliche Schlüssel des Angreifers verwendet wird, so dass jener die Daten mit dem ihm bekannten privaten Schlüssel entschlüsseln kann. Die entschlüsselten Daten kann der Angreifer dann an den eigentlichen Empfänger verschicken, wofür er die Daten zuvor mit dem öffentlichen Schlüssel des eigentlichen Empfängers verschlüsselt. Natürlich steht dem Angreifer hier auch die Möglichkeit offen, die Daten beliebig zu manipulieren.

Das klingt ein wenig nach Neuland? Keine Sorge, es ist alles viel einfacher, als es sich anhört.

4.1 Wie man Geheimnisse bewahrt

Zu jeder Zeit gab es Botschaften, die nicht für jeden bestimmt waren und entsprechend geheim gehalten werden mussten. Dennoch war die Übermittlung der Botschaften nicht immer durch direkten persönlichen Kontakt denkbar, weshalb im Laufe der Jahre Verfahren entwickelt wurden, die Botschaften so verschlüsseln, dass diese sicher transportiert werden können.

4.1.1 Kryptographie

Wenn es um das reine Verschlüsseln von Daten geht, so spricht man in diesem Zusammenhang von *Kryptographie*. Der Haken an reiner Verschlüsselung ist jedoch, dass es entsprechend Gegner geben wird, die sich der Herausforderung der Entschlüsselung stellen werden. So knackten die Engländer im Zweiten Weltkrieg zum Beispiel die Chiffriermaschine Enigma der Deutschen und konnten so wertvolle Informationen frühzeitig entziffern und nutzen.

Abbildung 4.1 »Enigma« (Quelle: Verkehrshaus der Schweiz, Luzern)

Um jedoch die erfolgreiche Dechiffrierung möglichst lange geheim zu halten, wurden die meisten Informationen nicht für Evakuierungen genutzt, da der Gegner sonst sehr schnell das eingesetzte Verfahrens gewechselt hätte. Aufgrund dieser Taktik wussten die Deutschen bis zum Kriegsende nicht einmal, dass die Enigma ihren Wert zur Verschlüsselung verloren hatte. Abbildung 5.2 stellt ist die Funktionsweise der Enigma schematisch dar.

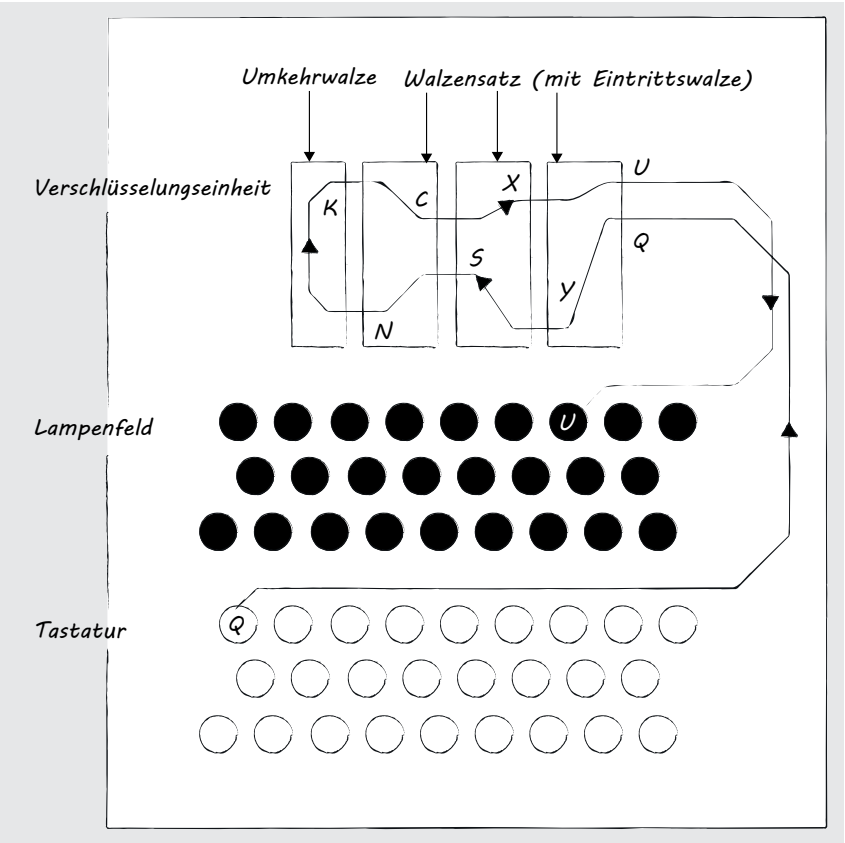


Abbildung 4.2 Schematische Darstellung des Funktionsprinzips der »Enigma«

4.1.2 Steganographie

Eine andere Möglichkeit der sicheren Datenübertragung besteht im Verschleiern der Daten, das heißt, die Daten selbst sind nicht als solche erkennbar. Die entsprechende Disziplin nennt sich auch *Steganographie*. Ein Vorteil der Steganographie ist, dass sie den Gegner nicht reizt, etwas knacken zu wollen, da nicht einmal das Vorhandensein der Informationen offensichtlich ist. Die Informationen werden sozusagen versteckt, wobei es sich ein bisschen wie mit der »Nadel im Heuhaufen« verhält. Natürlich sind solche Verfahren nur so lange sicher, wie nicht tatsächlich jemand nach der geheimen Botschaft sucht.

Als Trägerkanal eignen sich verschiedenste Medien, wie zum Beispiel Fernsehen, Radio, Bilder oder Textdateien, wobei diese Liste bei weitem keinen Anspruch auf Vollständigkeit erhebt. Die geheimzuhaltenden Daten werden dabei in das Rauschen der Signale verrechnet, so dass sie nicht weiter auffallen. Ohne die Kenntnis über die Art der Verrechnung ist es somit Außenstehenden nicht möglich, die relevanten Daten aus dem Rauschen zu extrahieren. Da das Rauschen selbst als bedeutungslos empfunden wird und keinen festgelegten Regeln folgt, ist eine Mustererkennung innerhalb des Rauschens sehr schwer umzusetzen. Obwohl die Kodierung innerhalb des Rauschens von Trägerkanälen sehr gut möglich

ist, ist auch die Kodierung ohne vorhandenes Rauschen des Mediums möglich. So ist mir bei Textdateien noch nie ein »Rauschen« aufgefallen, wobei diese Aussage sicher auch ein wenig Interpretationsspielraum hat. Seltsame Absätze und Einrückungen in einem Text könnten genauso als Rauschen interpretiert werden, wie das Farbrauschen innerhalb eines Bildes. Beim Schreiben eines Briefes kommt man aber auch ohne die Kodierung durch »unmotivierte« Absätze aus, da man zuvor ein Muster beziehungsweise eine Regel für die Art der Kodierung festlegen kann. So ist es zum Beispiel denkbar, dass jeder dritte Anfangsbuchstabe einen Teil der geheimen Nachricht kodiert.

```
ES GEHT MIR
NICHT UM AUFGABEN,
WELCHE KEINEN REIZ
ZUR WEITEREN SUCHE
AUSLÖSEN.
```

Listing 4.1 Beispiel für Steganographie in Texten, wobei die Buchstaben natürlich im Anwendungsfall nicht fett gedruckt wären

Die eigentlich irrelevanten Informationen im Brief verschleiern also die brisanten Informationen. Mit der nötigen Geduld lassen sich so Informationen in Texten verbergen, welche auch für sich allein genommen sehr gut lesbar und sinnvoll erscheinen. Eine einfache Möglichkeit besteht auch darin, dass das anzuwendende Muster im Nachhinein zu einem vorhanden Text festgelegt wird.

4.1.3 Die richtige Mischung machts

Zur zusätzlichen Sicherheit im Falle des Auffindens der verstecken Informationen können diese zusätzlich chiffriert sein. Kryptographie und Steganographie lassen sich sehr gut kombinieren. So ist sichergestellt, dass auch beim Auffinden der Botschaft der Sinn der Nachricht noch nicht entschlüsselt werden kann. In unserem Beispielsatz ist nur ein Wort versteckt, welches vorher mit einer bestimmten Bedeutung zwischen Sender und Empfänger festgelegt worden ist. Durch die Verwendung vorher vereinbarter Codewörter kann so die Informationssicherheit weiter gesteigert werden. Die angesprochene zusätzliche Chiffrierung der zu übermittelnden Daten geht aber über die Verwendung von Codewörtern hinaus. So ist es auch gut denkbar, dass die Daten mit einem beliebigen Verfahren verschlüsselt werden, bevor sie mittels Steganographie in einem beliebigen Trägerkanal untergebracht werden.

Die Verfahren beider Bereiche sind über die Zeit und vor allem mit dem Erscheinen der Computer sehr viel komplexer und leistungsfähiger geworden, so dass heutzutage jeder in den Genuss sehr starker Kryptographie und auch Steganographie kommt. Beliebte Medien zur Datenverschleierung sind vor allem Videos,

Bilder und Musikdateien. Diese Medien weisen immer ein scheinbar willkürliches Grundrauschen auf, das ohne sichtbaren beziehungsweise hörbaren Unterschied als Datenkanal dienen kann. Gekoppelt mit guten kryptographischen Verfahren können Daten somit extrem sicher geheim gehalten werden.

Die Steganographie bietet dabei den zusätzlichen Nutzen, dass der Angreifer weder weiß, wo noch wann er suchen muss. Denn bei Praktizierung würde man beliebig oft Bilder, Videos und Musikdateien versenden, wobei nur ganz selten einmal etwas Verschlüsseltes im Grundrauschen kodiert worden wäre. Der sich damit ergebende Aufwand für die Kryptoanalyse, welche das Themenfeld des Knackens von Verschlüsselung umfasst, wäre so groß, dass ein Erfolg nahezu auszuschließen ist, vor allem, wenn man bedenkt, dass die Geheimhaltung der Daten in der Regel nur für einen begrenzten Zeitraum notwendig ist. Hinzu kommt das Problem, dass man schnell Informationen in das Grundrauschen hineininterpretiert, die gar nicht existieren. Ein gutes Beispiel für die Möglichkeit der falschen Interpretation von angeblich vorhandenen Mustern ist das Buch »Der Bibelcode«, in dem Ereignisse der Gegenwart als in der Bibel vorhergesehen beschrieben werden. Hier kann man erkennen, dass eine grundlegende Eigenschaft des menschlichen Gehirns – die Fixierung auf Mustererkennung – nicht immer von Vorteil ist, beziehungsweise leicht zu falschen Annahmen führen kann. Wenn Sie das nächste mal Wolkenstrukturen anschauen, dann überlegen Sie sich einfach mal, wonach sie aussehen – es findet sich nahezu immer eine Bedeutung.

4.2 Ein paar Konventionen vorab

Bevor wir jetzt ans Codieren machen, vorab noch ein paar Konventionen, die im gesamten Kapitel eingehalten werden. Da Sie in diesem Kapitel immer wieder einen Klartext und einen Geheimtext sehen werden, unterscheide ich beide Arten von Text zur Vereinfachung durch Groß- und Kleinschreibung voneinander. Der Klartext wird jeweils in Kleinbuchstaben abgedruckt sein, der Geheimtext hingegen in Großbuchstaben. Das sieht dann in etwa wie folgt aus:

```
LSZQUPMPHJF?! <=> kryptologie?!
```

Zudem werde ich häufiger von der sogenannten Modulo-Arithmetik Gebrauch machen, die beim Rechnen mit Restklassen zum Einsatz kommt. Hierfür stelle ich jeweils den Modulo-Operator durch `MOD` dar, der in Python durch das Prozentzeichen symbolisiert wird. `MOD` ist dabei wie folgt definiert:

```
14 MOD 3 = 2 (<= 4 mal 3 => 12 + 2 => 14)
 5 MOD 5 = 0 (<= 5 mal 1 =>  5 + 0 =>  5)
```

Der Operator gibt also jeweils den Rest der Division zurück und sorgt somit dafür, dass das Ergebnis stets eine Zahl zwischen 0 und x - 1 ist. Dabei steht x für die Zahl vor dem Operator.

4.3 Seit wann wird chiffriert?

Kürzestmögliche Antwort: seit einer Ewigkeit. Mindestens schon seit der Zeit der Pharaonen. Allerdings waren die Mittel, um Nachrichten zu verschlüsseln, denkbar einfach im Vergleich zu den heutigen ausgeklügelten Systemen. Selbst die Verfahren der Römer sind für heutige Verhältnisse sehr leicht zu knacken – auch für Sie.

Eines der ältesten überlieferten Verfahren ist unter dem Namen *Skytale von Sparta* bekannt. Es wurde von den Griechen verwendet, um geheime Botschaften auf Papier mittels Transposition der Buchstaben zu verschlüsseln. Es handelt sich also um einen *Transpositionsalgorithmus*, bei dem die einzelnen Buchstaben des Klartextes nicht verändert, sondern nur ihre Positionen innerhalb der Nachricht vertauscht werden. Die Skytale von Sparta basierte darauf, dass sowohl Absender als auch Empfänger einen Stock gleichen Durchmessers besaßen, um den spiralförmig ein Pergament – ein schmaler Papierstreifen – gewickelt wurde, das daraufhin längs zum Stock beschriftet wurde. Zum Ablesen musste das Pergament erneut um einen Stock gleichen Durchmessers gewickelt werden. Der Durchmesser des Stocks entsprach somit dem Schlüssel und war entsprechend geheim zu halten.

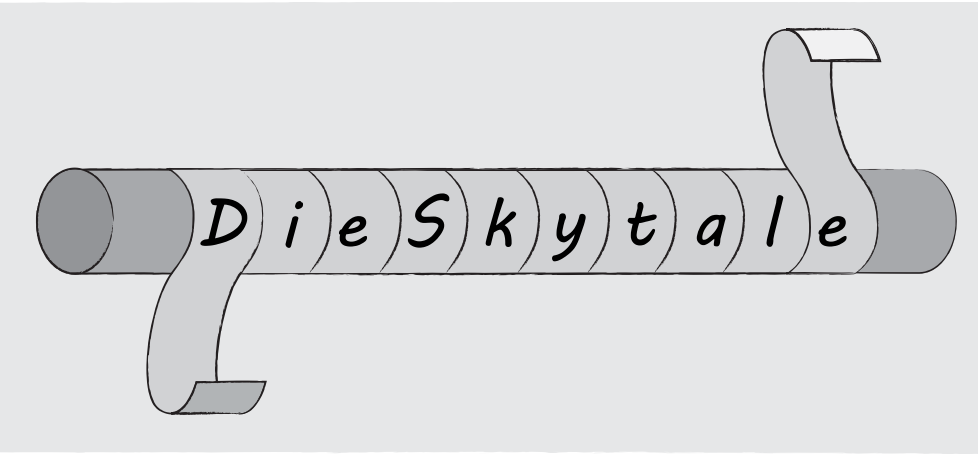


Abbildung 4.3 Skytale von Sparta

Die Verschiebung der einzelnen Zeichen lässt sich auch durch Untereinanderschreiben des entsprechenden Textes nachvollziehen:

```
heute ist sehr
schönes wetter

HSECUHTÖEN EISS TW ESTETHERR
```

Listing 4.2 Beispiel zur Skytale von Sparta

Zur verschlüsselten Form des Textes, also zum Geheimtext, gelangen Sie, indem Sie die beiden Zeilen spaltenweise lesen. Zum Entziffern müssen Sie in diesem Fall nicht den Durchmesser des Stocks, sondern die Anzahl der Zeichen pro Zeile kennen. Der Text ist mit dem entsprechenden Wissen leicht zu entziffern, aber auch ohne ist das Verfahren aus heutiger Sicht alles andere als sicher.

Transpositionsalgorithmen sind nach wie vor ein wichtiger Bestandteil vieler moderner Verfahren zur Verschlüsselung. Wie es mit der Umsetzung in Python aussieht, schauen wir uns im nächsten Abschnitt an.

4.4 Die Skytale von Sparta

Die Skytale von Sparta lässt sich mit Python sehr einfach nachprogrammieren und auch leicht mit Papier und Bleistift simulieren. Beides werden wir uns in diesem Kapitel genauer anschauen. Beginnen wir mit der Implementierung in Python.

Den Quelltext zur Skytale von Sparta finden Sie auf der beiliegenden CD-Rom im Verzeichnis *Kapitel04/skytale.py*. **[●]**

```
def skytale_encrypt(text, umfang):
    text    = ensureSideCondition(text, umfang)
    length  = len(text)
    cipher  = ""
    for x in range(0, umfang):
        for y in range(x, length, umfang):
            cipher += text[y]

    return cipher.upper()
```

Listing 4.3 Verschlüsseln mit der Skytale

Hier ist zunächst die Funktion `skytale_encrypt()` dargestellt, die sich um die Verschlüsselung des Textes kümmert. Dabei ist eine Nebenbedingung zu beachten: Die Textlänge muss glatt, also ohne Rest, durch den Umfang teilbar sein.

Die Überprüfung dieser Bedingung ist in die Funktion `checkSideCondition()` ausgelagert, die bei Bedarf den Text entsprechend um Leerzeichen erweitert. Der Parameter `umfang` gibt die Zeichen pro Zeile für die Verschlüsselung vor. Entsprechend werden die Zeichen genau in dem durch den Parameter `umfang` vorgegebenen Abstand aneinander gehängt.

```
# stellt sicher, dass die Länge des Textes
# ohne Rest durch den Umfang teilbar ist
def ensureSideCondition(text, umfang):
    length      = len(text)
    remainder   = length % umfang
    while remainder != 0:
        text       += " "
        length      = len(text)
        remainder   = length % umfang

    return text
```

Listing 4.4 Überprüfung der geraden Teilbarkeit des Textes durch die vorgegebene Zeilenlänge

Wenn der die Länge des Textes nicht gerade durch den vorgegebenen Umfang teilbar ist, so werden entsprechend viele Leerzeichen an das Textende angehängt.

Die Entschlüsselung muss den Vorgang der Umwandlung von zeilenweises in spaltenweises Lesen wieder rückgängig machen. Hierfür wird einfach die Anzahl der für das Verschlüsseln verwendeten Spalten ermittelt und als Parameter `umfang` dem Funktionsaufruf zum Verschlüsseln mitgegeben. Die Entschlüsselung nutzt somit das gleiche Verfahren wie die Verschlüsselung, nur mit einem anderen Schlüssel als Parameter. Die zwei sich hierbei gegenüberstehenden Schlüssel entsprechen also der Breite und der Höhe der Tabelle in Zeichen. Für das Verschlüsseln verwenden wir die Anzahl der Zeichen pro Zeile als Umfang, während wir für das Entschlüsseln die Anzahl der Zeilen als Umfang verwendet haben. Die beiden Werte lassen sich aber sehr leicht in den jeweils anderen konvertieren, da hierfür lediglich die Gesamtanzahl von Zeichen durch den jeweiligen Umfang dividiert werden muss. Dieses Umwandlung ist im folgenden Listing zu sehen.

```
def skytale_decrypt(text, umfang):
    length      = len(text)
    umfang      = length/umfang
    print umfang, length
    plaintext   = skytale_encrypt(text, umfang)

    return plaintext.lower()
```

Listing 4.5 Entschlüsseln mit der Skytale

Die jeweiligen Aufrufe von `lower()` beziehungsweise `upper()` werden nur durchgeführt, um den Konventionen, die am Anfang dieses Kapitels beschrieben worden sind, zu entsprechen. Zur Erinnerung: Diese besagen, dass Klartext kleingeschrieben und die Chiffre in Großbuchstaben erscheint. Sie verwenden die Funktionen dabei wie folgt:

```
key         = 8
plaintext   = "heute ist sehr schönes wetter"
cipher      = skytale_encrypt(plaintext, key)

print plaintext
print cipher
print skytale_decrypt(cipher, key)
```

Listing 4.6 Verwendung der Funktionen für die Skytale

```
heute ist sehr schönes wetter
HTCEE HTUSöTTENEEHER RS I   SSW
heute ist sehr schönes wetter
```

Listing 4.7 Konsolenausgabe zum Listing

Die Skytale von Sparte ist in ihrer Anwendung so einfach, dass Sie sie leicht auch unterwegs verwenden können. Statt eines Stocks mit bestimmtem Umfang benötigen Sie lediglich die Information über die Anzahl der verwendeten Spalten, also Zeichen je Zeile.

Zum Verschlüsseln tragen Sie dann einfach den Text in die Zeilen Ihrer Tabelle mit der festgelegten Anzahl an Spalten ein, wie in Abbildung 4.4 dargestellt.

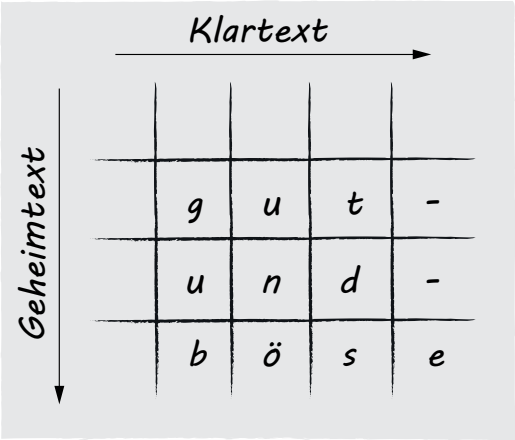


Abbildung 4.4 Skytale von Sparta für unterwegs – verschlüsseln

Für das Entschlüsseln wechseln Sie einfach zwischen Zeilen und Spalten, so dass die Anzahl der Zeilen aus der oberen Tabelle nun der Anzahl der Spalten entspricht und umgekehrt. Wie das genau aussieht, sehen Sie in Abbildung 4.5.

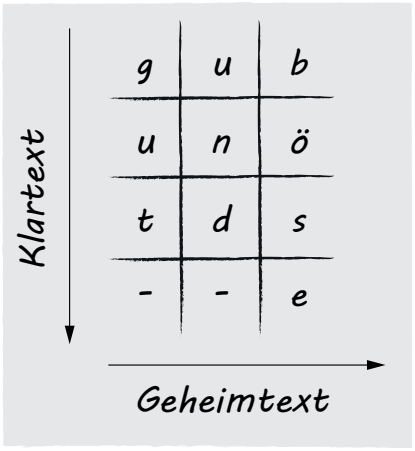


Abbildung 4.5 Skytale von Sparta für unterwegs – entschlüsseln

Die Pfeile in den Darstellungen verdeutlichen noch einmal, in welcher Richtung der jeweilige Text zu lesen ist.

4.5 Die Caesar-Chiffre

Ein weiterer Bestandteil moderner Algorithmen sind die sogenannten *Substitutionsalgorithmen*, die nicht die Position der Klartextzeichen verändern, sondern die Zeichen an sich durch andere Zeichen ersetzen. Die Substitutionsverfahren sind auch unter dem Namen *Verschiebechiffren* bekannt. Einer der ersten und gleichzeitig einfachsten Vertreter dieser Form ist die Caesar-Chiffre, die von Julius Caesar angewandt wurde, um Botschaften an seine Feldherren zu verschicken.

Bei der Caesar-Chiffre werden die einzelnen Buchstaben im Alphabet verschoben, so dass jedes A zum Beispiel durch D ersetzt wird. Bei allen anderen Buchstaben wird der gleiche Abstand, hier also drei Positionen im Alphabet, angewendet.

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
d e f g h i j k l m n o p q r s t u v w x y z a b c
```

Listing 4.8 Caesar-Chiffre für den Schlüssel 3

Um wie viele Stellen die einzelnen Buchstaben verschoben werden, ist dabei beliebig. Die Anzahl der Stellen für die Verschiebung ist entsprechend der verwendete Schlüssel. Da es bei diesem einfachen Verfahren allerdings lediglich 25

sinnvolle Varianten gibt, muss ich hier nicht viele Worte bezüglich der Sicherheit verlieren.

Es ist klar, dass selbst der naivste Ansatz – das Durchprobieren aller Möglichkeiten – sehr schnell zur Entschlüsselung des Geheimnisses führen würde. Im Anschluss sehen Sie alle möglichen Substitutionen für die verschiedenen Schlüssel.

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
b c d e f g h i j k l m n o p q r s t u v w x y z a
c d e f g h i j k l m n o p q r s t u v w x y z a b
d e f g h i j k l m n o p q r s t u v w x y z a b c
e f g h i j k l m n o p q r s t u v w x y z a b c d
f g h i j k l m n o p q r s t u v w x y z a b c d e
g h i j k l m n o p q r s t u v w x y z a b c d e f
h i j k l m n o p q r s t u v w x y z a b c d e f g
i j k l m n o p q r s t u v w x y z a b c d e f g h
j k l m n o p q r s t u v w x y z a b c d e f g h i
k l m n o p q r s t u v w x y z a b c d e f g h i j
l m n o p q r s t u v w x y z a b c d e f g h i j k
m n o p q r s t u v w x y z a b c d e f g h i j k l
n o p q r s t u v w x y z a b c d e f g h i j k l m
o p q r s t u v w x y z a b c d e f g h i j k l m n
p q r s t u v w x y z a b c d e f g h i j k l m n o
q r s t u v w x y z a b c d e f g h i j k l m n o p
r s t u v w x y z a b c d e f g h i j k l m n o p q
s t u v w x y z a b c d e f g h i j k l m n o p q r
t u v w x y z a b c d e f g h i j k l m n o p q r s
u v w x y z a b c d e f g h i j k l m n o p q r s t
v w x y z a b c d e f g h i j k l m n o p q r s t u
w x y z a b c d e f g h i j k l m n o p q r s t u v
x y z a b c d e f g h i j k l m n o p q r s t u v w
y z a b c d e f g h i j k l m n o p q r s t u v w x
z a b c d e f g h i j k l m n o p q r s t u v w x y
```

Listing 4.9 Caesar-Chiffre – Verschiebetabelle

Für die Entschlüsselung kehren Sie den Vorgang einfach um. Das heißt, dass Sie die Buchstaben in die andere Richtung verschieben.

Eine besondere Bedeutung hat die Verschiebung um 13 Stellen, denn in diesem Fall ist die Verschlüsselung mit der Entschlüsselung identisch. Hierfür hat sich entsprechend dieser Besonderheit auch ein eigener Name etabliert: *ROT13*, was für Rotation um 13 Stellen steht.

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
n o p q r s t u v w x y z a b c d e f g h i j k l m
```

Listing 4.10 ROT13 – ein Spezialfall der Caesar-Chiffre

Die Caesar-Chiffre ist noch einfacher zu implementieren als die Skytale von Sparta. Es ist lediglich jedes Zeichen durch den um die vorgesehene Anzahl von Stellen weiter rechts vorkommenden Nachfolger zu ersetzen. Dafür gibt es natürlich eine Vielzahl von Möglichkeiten, eine Variante sehen Sie im folgenden Quelltext.

[●] Sie finden den entsprechenden Code auf der CD unter *Kapitel04/caesar.py*.

```
def caesar(text, key):
    cipher = ""
    for letter in text:
        # Index des aktuellen Klartextbuchstabens
        # im Symbolvorrat bestimmen
        index   = symbols.index(letter)
        # Addition des Schlüssels zum Index und Sicherstellung,
        # dass der Index im gültigen Bereich bleibt
        index   = (index + key) % len(symbols)
        cipher += symbols[index]

    return cipher
```

Listing 4.11 Funktion zur Umsetzung der Caesar-Chiffre

Verwenden Sie die Funktion `caesar()` wie folgt:

```
key         = 4
symbols     = "abcdefghijklmnopqrstuvwxyz "
plaintext   = "diese nachricht kann im prinzip jeder knacken"
encrypted = caesar(plaintext,  key)
decrypted = caesar(encrypted, -key)
print plaintext
print encrypted
print decrypted
```

Listing 4.12 Verwendung der Funktion für die Caesar-Chiffre

Auf der Konsole wird beim Aufruf der Befehle Folgendes ausgegeben:

```
diese nachricht kann im prinzip jeder knacken
hmiwidreglvmglxdoerrdmqdtvmrcmtdnihivdoregoir
diese nachricht kann im prinzip jeder knacken
```

Listing 4.13 Konsolenausgabe

Wie Sie sehen, zeigt die Konsolenausgabe in der verschlüsselten Form der Botschaft auch keine Leerzeichen mehr. Dies liegt daran, dass die von mir gewählte Implementierung auch die Leerzeichen um die als Schlüssel festgelegte Anzahl von Zeichen verschiebt. Dieser Sachverhalt ist Ihnen vielleicht schon bei der Definition der `symbols` im Listing 4.13 aufgefallen. Natürlich können Sie die Leerzeichen auch durch eine geringfügige Änderung des Codes unverändert lassen, was die Verschlüsselung zwar noch stärker schwächt, aber bei einem so unsicheren Verfahren auch nicht mehr viel schaden kann. Die zu Beginn angesprochenen 25 sinnvollen Verschiebemöglichkeiten sind durch das Hinzufügen des Leerzeichens entsprechend um eine Möglichkeit gestiegen.

Um die Caesar-Chiffre zu knacken, muss man tatsächlich kein Genie sein, denn es reicht ja sogar ein so naives Vorgehen wie das Ausprobieren der 25, beziehungsweise mit Leerzeichen 26 möglichen Schlüssel. Allerdings wäre diese Vorgehensweise nicht wirklich sinnvoll, denn Sprache ist sehr redundant, und die einzelnen Buchstaben treten mit sehr unterschiedlicher Wahrscheinlichkeit auf. So kommt im Deutschen das »e« sehr viel häufiger vor als zum Beispiel das »z«. Diese Tatsache lässt sich leicht dazu verwenden, um die Caesar-Chiffre zu knacken. Da die Buchstaben alle um die gleiche Anzahl Stellen verrückt werden, ändert sich an der Häufigkeitsverteilung der Symbole nichts. Sie müssen also lediglich den im chiffrierten Text am häufigsten vorkommenden Buchstaben ausfindig machen und durch den in der entsprechenden Sprache häufigsten Buchstaben ersetzen. Der Abstand der beiden Buchstaben zueinander entspricht dabei dem Schlüssel.

Je kürzer ein Text ist, desto weniger aussagekräftig ist natürlich seine Statistik. Allerdings muss man sich nicht nur auf die Häufigkeitsverteilung eines Buchstaben stürzen, sondern kann auch die Häufigkeitsverteilung von Buchstabenpaaren betrachten. Auch innerhalb einer Sprache sind die Häufigkeiten der einzelnen Buchstaben nicht immer identisch, denn die Häufigkeitsverteilung der Buchstaben hängt natürlich auch von der Art des Textes ab. So enthält ein Aufsatz über die neuesten technologischen Errungenschaften der Automobilindustrie ganz andere Wörter – unter anderem viele Fremdwörter aus dem Englischen – als ein Werk von Friedrich Nietzsche. Im Großen und Ganzen ist die Sprache aber bereichsübergreifend so redundant, dass statistische Auswertungen auch deutlich fortschrittlichere Systeme knacken können.

Bei einer so geringen Anzahl von Schlüsselmöglichkeiten wie im Fall der Caesar-Chiffre ist es jedoch noch viel einfacher den Code zu knacken, da man in sehr kurzer Zeit alle Schlüssel ausprobieren kann. Denkbar ist zum Beispiel das folgende kleine Skript, das alle Verschiebungen ausgibt.

```
for x in range(1, 25):
    decrypted = caesar(encrypted, -x)
    print decrypted
```

Listing 4.14 »Brute Force« für die Caesar-Chiffre

Die Ausgabe sähe nach dem Aufruf dabei wie folgt aus:

```
glhvhcqdfkulfkwcndqqclpcsulqblscmhghucnqdfnhq
fkgugbpcejtkejvbmcppbkobrtkpakrblgfgtbmpcemgp
ejftfaobdisjdiualbooajnaqsjo jqakfefsalobdlfo
diese nachricht kann im prinzip jeder knacken
chdrdzm bgqhbgszj mmzhlzoqhmyhozidcdqzjm bjdm
bgcqcylzafpgafryizllygkynpglxgnyhcbcpyilzaicl
afbpbxky eof eqxhykkxfjxmofkwfmxgbaboxhky hbk
 eaoawjxzdnezdpwgxjjweiwlnejvelwfa anwgjxzgaj
zd n viwycmdycovfwiivdhvkmdiudkve z mvfiwyf i
yczmzuhvxblcxbnuevhhucgujlchtcjudzyzluehvxezh
xbylytguwakbwamtduggtbftikbgsbitcyxyktdguwdyg
waxkxsftv jav lsctffsaeshjafrahsbxwxjscftvcxf
v wjwresuzi uzkrbseer drgi eq grawvwirbesubwe
uzvivqdrtyhztyjqarddqzcqfhzdpzfq vuvhqadrtavd
tyuhupcqsxgysxip qccpybpegycoyepzutugp cqs uc
sxtgtobprwfxrwhozpbboxaodfxbnxdoytstfozbprztb
rwsfsnaoqvewqvgnyoaanw ncewamwcnxsrsenyaoqysa
qvrerm npudvpufmxn  mvzmbdv lvbmwrqrdmx npxr
puqdqlzmotcuotelwmzzluylacuzkualvqpqclwzmowqz
otpcpkylnsbtnsdkvlyyktxk btyjt kupopbkvylnvpy
nsobojxkmrasmrcjukxxjswjzasxiszjtonoajuxkmuox
mrnaniwjlq rlqbitjwwirviy rwhryisnmn itwjltnw
lqm mhvikpzqkpahsivvhquhxzqvgqxhrmlmzhsviksmv
kplzlguhjoypjo grhuugptgwypufpwgqlklygruhjrlu
```

Listing 4.15 Ausgabe aller Verschiebungen

Es ist kein großer Aufwand, jetzt nach der Zeile mit dem Klartext zu suchen. Deshalb sollte man beim Verschlüsseln immer sicherstellen, dass die Anzahl möglicher Schlüssel so groß ist, dass ein »Brute Force«-Angriff nicht in annehmbarer Zeit durchführbar ist.

4.6 Exkurs: Redundanz der Sprache

Wie redundant Sprache tatsächlich ist, möchten ich Ihnen in diesem kleine Exkurs etwas näher erläutern. Hierfür werden wir einen Text analysieren, um die

Häufigkeitsverteilung der einzelnen Buchstaben begutachten zu können. Dabei wird sich eine Ungleichverteilung zeigen, welche Ausdruck der Redundanz ist. Diese Ungleichverteilung lässt sich aber durch Kompression stark reduzieren. Der Vorteil hierbei ist, dass Sie durch eine vorherige Komprimierung der Daten den Dechiffrierungsaufwand beliebig steigern, da der entscheidende Angriffspunkt vieler Verfahren aufgrund der ungleichmäßigen Häufigkeitsverteilung von Buchstaben zum Tragen kommt. Nach dem Komprimieren ist jedoch die Verteilung in den resultierenden Daten nahezu gleich – je nach Güte des Kompressionsverfahrens versteht sich.

Je höher die Datenkompression, desto stärker ist in der Regel danach die Gleichverteilung der möglichen Symbole. Dieser Sachverhalt verdeutlicht auch, warum komprimierte Daten nicht mehr komprimiert werden können: Kompression ist nur möglich, solange die einzelnen Zeichen des Datenstroms nicht gleichverteilt sind und sich somit Muster extrahieren lassen. Ein Fachmann würde sagen: Die Redundanz der Daten ist umso geringer, je stärker die Entropie zunimmt. Kompressionsverfahren reduzieren die Redundanz beträchtlich und sorgen somit für eine relativ gute Gleichverteilung der im Datenstrom vorkommenden Zeichen.

Um die Redundanz der Sprache etwas besser zu verstehen, schreiben wir nun ein kurzes Programm, das die Buchstabenhäufigkeit in einem Text ermittelt und als prozentuale Anteile ausgibt. Zunächst ermitteln wir also die Anzahl des Auftretens einzelner Buchstaben. Danach teilen wir diese Anzahl durch die Summe aller im Text enthaltenen Buchstaben.

Sie finden den entsprechenden Quelltext auf der CD unter *Kapitel04/statistics.py*. **[●]**

```
def countLetters(fileName):
    # festlegen der für die
    # Statistik relevanten Zeichen
    letters = "abcdefghijklmnopqrstuvwxyz"
    mydict  =
    # öffnen der Datei, welche
    # statistisch ausgewertet werden soll
    fobj    =   open(fileName, "r")
    counter = 0

    for line in fobj:
        for character in line:
            # keine Unterscheidung von Groß-
            # und Kleinbuchstaben notwendig
            character = character.lower()
            if character in letters:
                # Anzahl eingelesener Buchstaben
                # um eins erhöhen
```

```
                counter += 1
                # Buchstabe wurde schonmal gelesen
                if mydict.has_key(character) == 1:
                    mydict[character]=mydict.get(character)+1
                # erstes Auftreten des Buchstabens
                else:
                    mydict[character]=1

    fobj.close()

    # Umwandlung des Dictionary in eine
    # Liste, um die Sortierung zu ermöglichen
    letterList = mydict.items()
    # Sortierung der Liste anhand der
    # Häufigkeit des Auftretens der Buchstaben
    letterList = sorted(letterList, key=operator.itemgetter(1),
     reverse=True)

    return counter, letterList
```

Listing 4.16 Die Funktion zur Ermittlung der Statistik

Die Kommentare im Quelltext beschreiben die Funktion `countLetters()` schon recht genau: Es wird zunächst der Zeichenvorrat, der für die zu erstellende Statistik relevant ist, festgelegt. Daraufhin wird ein leeres Dictionary erzeugt, in dem die Buchstaben mit der jeweiligen Häufigkeit eingetragen werden sollen. Die auszuwertende Datei wird der Funktion als Parameter `fileName` übergeben und in den beiden Schleifen durchlaufen. Die äußere Schleife durchläuft dabei die einzelnen Zeilen der Datei, die innere Schleife die Zeichen einer Zeile. Da eine Datei nicht nur Buchstaben enthält, sondern auch Sonderzeichen wie zum Beispiel Zeilenumbrüche und Kommas, wird als Nächstes überprüft, ob das aktuelle Zeichen im relevanten Symbolvorrat `letters` vorkommt. Wenn es sich um ein relevantes Zeichen handelt, also einen Buchstaben, dann wird überprüft, ob es bereits einen Eintrag im Dictionary für diesen Buchstaben gibt. In diesem Fall wird diese Anzahl ausgelesen und um eins erhöht wieder gespeichert. Gab es noch keinen Eintrag, so wird ein neuer Eintrag für den Buchstaben erzeugt und mit eins belegt, da es sich um das erste Auftreten des Buchstabens handelt.

Im Anschluss an die Ermittlung der Auftretenshäufigkeit der einzelnen Buchstaben innerhalb der Datei wird die Datei wieder geschlossen und aus dem erstellten Dictionary eine Liste extrahiert. Die Liste benötigen wir zum Sortieren der Einträge. Die Sortierung erfolgt hierbei anhand der Auftretenshäufigkeit der einzelnen Buchstaben. Zum Abschluss der Funktion `countLetters()` werden die Anzahl gefundener Buchstaben sowie die Liste der Buchstaben mit den einzelnen Auf-

tretenshäufigkeiten zurückgegeben. Die Nutzung der Funktion `countLetters()` schauen wir uns jetzt genauer an.

```
letterCount1, letterList1 = countLetters("FaustTrag1.txt")
letterCount2, letterList2 = countLetters("FaustTrag2.txt")
letterCount3, letterList3 = countLetters("GoethePPmd.7z")

allLists = zip(letterList1, letterList2, letterList3)

for letterList in allLists:
    for entry in letterList:
        print entry[0]+" -> " +
         str(float(entry[1])/letterCount2*100)+"\t",
    print
```

Listing 4.17 Nutzung der Funktion zur Ermittlung der Buchstabenverteilung

Die ersten drei Zeilen des Listings ermitteln für die als Parameter übergebene Datei die Statistik der Buchstabenhäufigkeiten. Danach werden die drei resultierenden Listen zu einer Liste zusammengeführt und im Anschluss innerhalb der Schleife nebeneinander ausgegeben. Das Ergebnis der Ausgabe sehen Sie im nachfolgenden Listing 4.18.

```
e -> 10.356417823 ^e -> 15.5542998263 ^z -> 0.00125572926477
n -> 6.33013122371 ^n -> 9.34388145916 ^a -> 0.00083715284318
i -> 5.61143550784 ^i -> 7.84161068207 ^n -> 0.00083715284318
r -> 4.60182917896 ^r -> 7.03166530629 ^t -> 0.00083715284318
t -> 4.48504635734 ^s -> 6.74368472824 ^g -> 0.00041857642159
s -> 4.39463385028 ^t -> 6.68843264059 ^i -> 0.00041857642159
h -> 4.20501873129 ^h -> 6.26525187836 ^q -> 0.00041857642159
a -> 3.45116259601 ^a -> 4.93124882275 ^u -> 0.00041857642159
d -> 3.18787802683 ^d -> 4.67549862916 ^w -> 0.00041857642159
u -> 2.6889349323 ^l -> 4.30631422532 ^v -> 0.00041857642159
c -> 2.64498440803 ^c -> 4.06186559511 ^x -> 0.00041857642159
```

Listing 4.18 Konsolenausgabe

Bei den verwendeten Dateien handelt es sich um Goethes *Faust I* und *Faust II* sowie um eine komprimierte Datei, die beide Teile enthält. Anhand der prozentualen Auftretenshäufigkeit erkennen Sie sehr schön, dass der Buchstabe »e« sowohl im ersten als auch im zweiten Teil von Goethes Werk am häufigsten auftaucht. Nach einer Kompression hingegen kommen alle Buchstaben in etwa gleich oft vor. Zudem ist die relative Häufigkeit nach der Kompression deutlich gesunken, was vor allem daran liegt, dass jetzt natürlich viele Buchstaben durch andere ASCII-Zeichen repräsentiert werden. Eine Kompression von Daten führt

nahezu zu einer Gleichverteilung über den gesamten möglichen Wertebereich des Zeichensatzes. Da außerdem Sprache von Haus aus sehr redundant ist, sind gerade bei Textdateien sagenhafte Kompressionsergebnisse möglich, die weit über neunzig Prozent liegen – in der Regel sogar bei neunundneunzig Prozent. Allein die Tatsache, dass die sechsundzwanzig Buchstaben des Alphabets in etwa nur ein Zehntel des darstellbaren Wertebereichs eines Bytes ausmachen, führt bei gleichverteilter Nutzung des gesamten Wertebereichs zu einer Kompression von neunzig Prozent.

Jetzt fragen Sie sich vermutlich, wie man überhaupt Daten in eine Gleichverteilung bringen kann beziehungsweise wie man überhaupt Daten komprimiert. Die Verfahren hierfür sind ganz unterschiedlicher Natur, aber eines der leicht verständlicheren Verfahren möchte ich Ihnen kurz vorstellen. Es handelt sich dabei um die Hufmann-Codierung, die die Zeichen entsprechend ihrer Häufigkeitsverteilung in einem Binärbaum abspeichert. Je häufiger ein Zeichen im Datenstrom vorkommt, desto näher an der Wurzel des Baumes wird es gespeichert. Nachdem der komplette Baum für den Datenstrom erstellt wurde, wird der Datenstrom entsprechend der jeweiligen Position des aktuell zu kodierenden Zeichens durch kürzere Bitfolgen ersetzt. Dabei wird von der Wurzel des Binärbaumes für jeden Schritt nach rechts eine Eins kodiert und für jeden Schritt nach links eine Null. So wird für jedes Zeichen der Pfad zum Zeichen innerhalb des Binärbaumes als Bitfolge in den neuen, komprimierten Datenstrom geschrieben. Das Ergebnis dieser Vorgehensweise ist, dass sehr häufig vorkommende Zeichen durch weniger als acht Bit repräsentiert werden, weniger häufige Zeichen hingegen durch mehr als acht Bit. Hierdurch ergibt sich die Kompression der Daten. Für die im oberen Listing ermittelten Verteilungen könnte ein Huffman-Baum zum Beispiel wie in Abbildung 4.6 aussehen, wobei hier nur ein Ausschnitt des Baumes dargestellt ist.

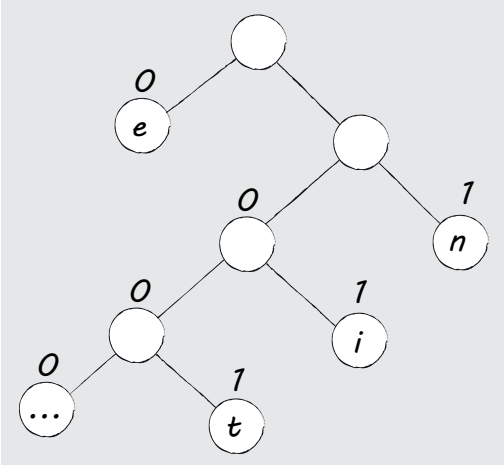


Abbildung 4.6 Huffman Baum

Da der Baum mit abgespeichert wird, lässt sich das Prinzip beim Dekodieren wieder anwenden, um die Daten zu entpacken.

In Kombination mit der Burrows-Wheeler-Transformation und einer Lauflängenkodierung wird eine sehr hohe Kompressionsrate ermöglicht. Dabei sorgt die Burrows-Wheeler-Transformation dafür, dass gleiche Zeichen möglichst nah beieinander liegen. Im Anschluss lassen sich Wiederholungen von Zeichen durch die Lauflängenkodierung zusammenfassen. Lauflängenkodierung ist nichts weiter als das Speichern des Zeichens und der Anzahl der Wiederholungen; so wird aus »aaaa« zum Beispiel das kürzere »4a«, wobei die Vier natürlich als die Zahl Vier in Byteform kodiert wird. Erst nach diesen beiden Transformationen würde man die Huffman-Kodierung anwenden, um häufiger auftretende Symbole beziehungsweise Symbolkombinationen entsprechend durch weniger Bits zu repräsentieren.

Die momentan stärksten Algorithmen zur Kompression von Daten verwenden arithmetische Verfahren, die versuchen, eine dem Datenstrom möglichst nahekommende Funktion zu finden. Hier ist mitunter auch bei scheinbar vollkommener Entropie – was soviel wie totale Gleichverteilung heißt – noch Kompression möglich – jedoch steigt die Laufzeit für diese Verfahren exorbitant an.

4.7 Vigenère-Chiffre

In diesem Abschnitt werden wir die denkbar einfachste symmetrische Verschlüsselung implementieren, die natürlich auch einen Haken hat: Sie ist nicht sicher. Die beiden vorherigen Verfahren gelten zwar auch als symmetrische Verfahren, bieten aber gar keinen Schutz und sind nicht unbedingt leichter umzusetzen. Trotz des Hakens können Sie allerdings davon ausgehen, dass die meisten Menschen nicht in der Lage sind, diese Verschlüsselung zu knacken – wenn Sie ganz sicher gehen wollen, dann komprimieren Sie die Daten noch vor der Verwendung dieser Chiffre.

Doch kommen wir nun zur Vigenère-Chiffre, die einfach jeden Buchstabenwert eines Passwortes und den des Klartextes addiert. Da der Klartext in der Regel länger ist als das Passwort, wird nach Erreichen des letzten Passwortbuchstabens wieder beim ersten begonnen. Die benannte Schwäche des Verfahrens ergibt sich nun dadurch, dass jeder x. Buchstabe des Klartextes mit dem selben Passwortbuchstaben addiert wird, wobei x für die Länge des Passwortes plus eins steht.

Zunächst fragen Sie sich vielleicht, wie man überhaupt Buchstaben addieren kann. Genaugenommen werden hierbei nicht die Buchstaben addiert, sondern die Werte der jeweiligen Bytes der verwendeten Zeichen. Die Problematik der mehrfachen Verwendung des gleichen Passwortbuchstabens sehen Sie im

folgenden Listing. Noch deutlicher zum Vorschein kommt das Problem, wenn Sie sich einen Text vorstellen, der über eine lange Passage das gleiche Zeichen enthält. Denn hier sieht der Angreifer sofort das Passwort im Geheimtext beziehungsweise zumindest die einzelnen Abstände der Passwortbuchstaben zueinander.

```
Klartext = "unverschluesselter Text"
++++++++++++++++++++++
Passwort = "geheimgeheimgeheimgehei"
======================
kodiert  = " rbizdilsymdyisxmcfxlaa"
```

Listing 4.19 Prinzip der Vigenère-Chiffre

Für die Darstellung wurde eine Version der Implementierung verwendet, die nur innerhalb der Kleinbuchstaben rechnet und im Folgenden dargestellt ist.

[●] Sie finden den entsprechenden Quelltext zur Vigenère-Chiffre auf der CD-Rom unter *Kapitel04/vigenere_v1.py*.

```
symbols     = "abcdefghijklmnopqrstuvwxyz "
plainText   = "unverschluesselter text"
password    = "geheim"
```

Listing 4.20 Globale Definitionen

```
def encrypt(plainText, password):
    pwdLength   = len(password)
    # die aktuelle Position innerhalb
    # des Passwortes wird in keyIndex vermerkt
    keyIndex    = 0
    encrypted   = []
    for letter in plainText:
        # der aktuelle Klartextbuchstabe wird zum
        # aktuellen Passwortbuchstaben addiert, um Überläufe
        # abzufangen wird zum Schluss Modulo der
        # Anzahl der Symbole gerechnet
        encryptedLetter = symbols[(symbols.index(letter) +
         symbols.index(password[keyIndex])) %
         len(symbols)]
        # der Index innerhalb des Passwortes
        # wird um eins erhöht
        keyIndex += 1
        # es wird sichergestellt, dass der keyIndex
        # innerhalb der Passwortlänge bleibt
```

```
        keyIndex %= pwdLength
        encrypted.append(encryptedLetter)

    # es wird ein Leerstring erzeugt, um die Methode
    # join der entstandenen String-Instance für die Umwandlung
    # der Liste in einen String zu verwenden
    return "".join(encrypted)
```

Listing 4.21 Verschlüsseln mittels Vigenère-Chiffre

Das Vorgehen für die Verschlüsselung der Daten ist analog zum Vorgehen bei der Caesar-Chiffre, nur dass nun die Verschiebung vom jeweilig aktuellen Passwortbuchstaben abhängt. Sobald die Passwortlänge verbraucht ist, wird wieder beim ersten Buchstaben des Passwortes angefangen. Um im Wertebereich des Symbolvorrats zu bleiben, wird wieder der Modulo-Operator herangezogen. Das Entschlüsseln erfolgt durch die Subtraktion der einzelnen Passwortbuchstaben vom verschlüsselten Text.

```
def decrypt(encryptedText, password):
    pwdLength   = len(password)
    keyIndex    = 0
    decrypted   = []
    for letter in encryptedText:
        # der aktuelle Passwortbuchstabe wird vom
        # aktuellen Geheimtextbuchstaben subtrahiert,
        # um negative Werte abzufangen, wird zum Schluss
        # die Symbolanzahl addiert und Modulo der Anzahl
        # der Symbole gerechnet, um Überläufe abzufangen
        decryptedLetter = symbols[(symbols.index(letter) -
         symbols.index(password[keyIndex]) +
         len(symbols)) % len(symbols)]
        keyIndex += 1
        keyIndex %= pwdLength
        decrypted.append(decryptedLetter)

    return "".join(decrypted)
```

Listing 4.22 Entschlüsseln mittels Vigenère-Chiffre

Waren die Klartextdaten vor dem Verschlüsseln mit dieser Chiffre noch nicht komprimiert, so lässt sich anhand der für die Sprache üblichen Buchstabenverteilung die Passwortlänge ermitteln. Diese Chiffre ist unter anderem mittels Kasiski-Test und Friedmann-Test knackbar – »Brute Force«-Angriffe sind natürlich immer denkbar. »Brute Force« steht hierbei einfach für das Ausprobieren aller möglichen Passwörter, bis eine sinnvolle Lösung entdeckt wird. Die beiden an-

deren Verfahren bedienen sich hingegen den bekannten Häufigkeitsverteilungen für Buchstaben und Buchstabenpaare, um die Chiffre zu knacken.

Allerdings kann die Vigenère-Chiffre natürlich auch extrem sicher sein – und zwar genau dann, wenn das Passwort genauso lang ist wie der zu verschlüsselnde Klartext. In diesem speziellen Fall spricht man allerdings vom *One-Time-Pad* – wobei sich die Frage stellt, wie man so lange Schlüssel austauscht ... One-Time-Pad ist auch nur dann sicher, wenn für jede Nachricht ein anderer Schlüssel verwendet wird; ansonsten ist es wieder möglich, über Häufigkeitsverteilungen eine Analyse durchzuführen.

Eine allgemeinere Form der Implementierung der Vigenère-Chiffre berücksichtigt nicht nur Buchstaben, sondern alle möglichen Werte eines Bytes. Bei dem Summieren der Werte von Bytes kommt es unter Umständen zu Überläufen, also zu einer Summe aus Klartextbuchstabe und Passwortbuchstabe, die größer als 256 ist. Dies ist analog zu den Überläufen zu sehen, die außerhalb des Symbolvorrats liegen. Diese Überläufe fangen Sie wie bereits bekannt durch den Modulo-Operator ab – modulo 256 stellt somit sicher, dass die Zahl innerhalb des für Zeichen gültigen Bereichs bleibt. Beim Entschlüsseln können negative Werte auftreten, die Sie durch Addition von 256 abfangen können. Der Einfachheit halber wird dabei immer um 256 erhöht und modulo 256 gerechnet; damit ersparen Sie sich die Überprüfung, ob der Wert kleiner null ist – Sie addieren einfach wahllos 256 und bleiben dank modulo 256 im gültigen Wertebereich.

Schauen wir uns zunächst die Verschlüsselungsfunktion `encrypt()` an.

[o] Sie finden den entsprechenden Quelltext zur Vigenère-Chiffre auf der CD-Rom unter *Kapitel04/vigenere_v2.py*.

```
def encrypt(plainText, password):
    pwdLength   = len(password)
    # die aktuelle Position innerhalb
    # des Passwortes wird in keyIndex vermerkt
    keyIndex    = 0
    encrypted   = []
    for letter in plainText:
        # der aktuelle Klartextbuchstabe wird mit dem
        # aktuellen Passwortbuchstaben addiert, um Überläufe
        # abzufangen wird zum Schluss modulo 256 gerechnet
        encryptedLetter = chr((ord(letter) +
         ord(password[keyIndex])) % 256)
        # der Index innerhalb des Passwortes
        # wird um eins erhöht
        keyIndex += 1
        # es wird sichergestellt, dass der keyIndex
        # innerhalb der Passwortlänge bleibt
```

```
        keyIndex %= pwdLength
        encrypted.append(encryptedLetter)

    # es wird ein Leerstring erzeugt, um die Methode
    # join der entstanden String-Instance für die Umwandlung
    # der Liste in einen String zu verwenden
    return "".join(encrypted)
```

Listing 4.23 Verschlüsseln mittels Vigenère-Chiffre

Die Funktion `encrypt()` addiert zunächst wie bereits geschildert den Wert des aktuellen Passwortbuchstabens und den des aktuellen Buchstabens des Klartextes. Um einen Überlauf zu vermeiden – das heißt, um die Werte bei der Addition dieser beiden Buchstaben immer innerhalb des Wertebereichs von 0 bis 255 zu halten –, wird im Anschluss an die Addition modulo 256 gerechnet. Die entsprechende Zeile ist im Quelltext (vgl. Listing 4.21) oben fett gedruckt. Der Index innerhalb des Passwortes wird je Iteration um eins erhöht, wobei auch hier wie bereits geschildert einem Überlauf vorgebeugt wird, indem nach jeder Indexerhöhung modulo Passwortlänge gerechnet wird. Die bei dieser Berechnung sich ergebende Summe wird an die Liste `encrypted` angehängt.

Zum Abschluss der Funktion `encrypt()` wird die Liste `encrypted` in einen String konvertiert, indem einfach alle Elemente der Liste konkateniert werden. Hierbei wird zunächst ein Leerstring erzeugt, um auf die für die Umwandlung der Liste in einen String nützliche Methode `join()` der Klasse `str` zugreifen zu können.

Die Entschlüsselung unterscheidet sich im Wesentlichen nicht groß von der Verschlüsselung, außer dass wir jetzt subtrahieren und nicht addieren:

```
def decrypt(encryptedText, password):
    pwdLength   = len(password)
    keyIndex    = 0
    decrypted   = []
    for letter in encryptedText:
        decryptedLetter = chr((ord(letter) -
         ord(password[keyIndex]) +
         256) % 256)
        keyIndex += 1
        keyIndex %= pwdLength
        decrypted.append(decryptedLetter)

    return "".join(decrypted)
```

Listing 4.24 Entschlüsseln mittels Vigenère-Chiffre

Um wiederum sicherzustellen, dass der Wertebereich von 0 bis 255 nicht verlassen wird, müssen wir nun auch die Möglichkeit von negativen Ergebnissen bei der Berechnung berücksichtigen. Ganz genau genommen müssten eigentlich nur negative Ergebnisse behandelt werden. Es würde genügen, nach jeder Subtraktion zu überprüfen, ob das Ergebnis negativ ist, und in diesem Fall einfach 256 zu addieren. Durch einen kleinen Trick bleibt uns hier jedoch die Überprüfung erspart, da wir einfach in jedem Fall prophylaktisch 256 addieren können, wenn wir hinterher zusätzlich sicherstellen, dass der Wertebereich nach oben nicht verlassen wird. Wie uns das gelingt, wissen Sie bereits seit der Implementierung der Verschlüsselung durch die Funktion `encrypt()`: Modulo 256 hilft uns weiter.

[+] Modulo-Berechnungen

Eine Besonderheit beim Rechnen mit Modulo ist, dass die Zahl, die für die Modulo-Reduzierung verwendet wird, das neutrale Element darstellt. Damit ist für diese Berechnungen die Zahl 256 so etwas wie die Zahl Null, weshalb wir hier wahllos immer Modulo 256 rechnen können, ohne vorher überprüfen zu müssen, ob wir momentan tatsächlich eine Zahl größer 256 vorliegen haben. Zahlen kleiner 256 bleiben unverändert erhalten. Aufgrund dieser Tatsache ersparen wir uns eine Überprüfung der tatsächlichen Größe der Zahl, da wir größenunabhängig immer modulo 256 rechnen können.

Kommen wir zu guter Letzt zur Anwendung der beiden Funktionen `encrypt()` und `decrypt()`.

```
plainText   = "Dieser Text wird verschluesselt werden"
password    = "Coding4Fun"

print "plain Text =", plainText
print "password   =", password

encryptedText = encrypt(plainText, password)
decryptedText = decrypt(encryptedText, password)

print "encrypted Text =", encryptedText
print "decrypted Text =", decryptedText
```

Listing 4.25 Anwendung der Funktionen zur Vigenère-Chiffre

Ich denke, dass sich die Anwendung weitgehend selbst erklärt und entsprechend keinerlei Erläuterung an dieser Stelle bedarf. Die Ausgabe der oberen Aufrufe sehen Sie im folgenden Listing.

```
plain Text = Dieser Text wird verschluesselt werden
password   = Coding4Fun
encrypted Text = <enthält nicht darstellbare Zeichen>
decrypted Text = Dieser Text wird verschluesselt werden
```

Listing 4.26 Die Ergebnisse der Vigenère-Chiffre

Die Einfachheit dieses Verfahrens wird nur noch durch die der Caesar-Chiffre übertroffen, die einen Spezialfall darstellt. Sie ist im Prinzip eine Vigenère-Chiffre, bei der die Länge des Passwortes genau eins beträgt. Die sich daraus ergebende Verschlüsselung ist allerdings nicht mehr zu retten – von sicherer Verschlüsselung kann keine Rede sein. Dennoch fand diese Art der Verschlüsselung im alten Rom regen Gebrauch.

Ein Nachteil unserer zweiten Implementierung der Vigenère-Chiffre ist, dass es bei der Verschlüsselung zu nicht druckbaren Zeichen kommen kann. Wenn Sie sich hier mittels der ersten Variante auf Buchstaben beschränken, so ließe sich die Nachricht auch in gedruckter Form weiterleiten. Natürlich wäre es denkbar, die Nachricht einfach in hexadezimaler Darstellung zu drucken.

Wie bereits gesagt ist diese Verschlüsselung sehr einfach und nicht besonders sicher. Die Sicherheit lässt sich jedoch, wie schon erwähnt, durch Komprimierung des Klartextes erhöhen. Noch sicherer wird es, wenn Sie das Passwort komprimieren, das dann allerdings entsprechend lang sein muss. Beim Passwort spricht man in der Regel nicht von Kompression, da es hier auch genügen würde, einen Hash-Wert für das Passwort zu ermitteln.

Was ist ein Hash-Wert? **[+]**

Unter einem Hash-Wert versteht man einen Wert der von einer Hashfunktion – auch Streuwertfunktion genannt – ermittelt worden ist. Hierbei würde man zum Beispiel das Passwort als Parameter der Funktion übergeben und als Ergebnis den Hash-Wert zurückgeliefert bekommen. Hashfunktionen sind sogenannte Einwegfunktionen, was heißt, dass aus dem Hashwert nicht auf den Parameter geschlossen werden kann. Zu einem einzelnen Hashwert existieren mehrere gültige Eingangsparameter, wobei eine gute Hashfunktion dafür sorgt, dass es schwer ist, zwei Parameter zu finden, welche den gleichen Hash liefern.

Der Unterschied zwischen Kompression und der Berechnung eines Hash-Wertes liegt dabei darin, dass aus dem Hashwert nicht wieder auf das Passwort geschlossen werden kann, während eine verlustfreie Kompression stets umkehrbar ist. Falls generell nicht komprimiert werden soll, so ist es auch möglich, die Stärke des Verfahrens durch Verknüpfen der einzelnen Datenblöcke zu erhöhen. So könnten Sie zum Beispiel jeden Block mit dem jeweiligen vorhergehenden Block per XOR verknüpfen, wobei die Länge des Blocks der Länge des Passwortes ent-

sprechen sollte. Diese Art der Verknüpfung würde ebenfalls zum Verschleiern der Häufigkeiten führen; am besten ist natürlich die Kombination aus allen Verfahren zur Verschleierung der Häufigkeitsverteilungen. Ein Nachteil der Verknüpfung der aufeinanderfolgenden Blöcke ist jedoch, dass bei Übertragungsfehlern im Datenstrom alle nach dem Fehler folgenden Daten verloren sind, denn es besteht eine Abhängigkeit der Daten zu allen Vorgängerblöcken. Zudem wäre es zusätzlich sinnvoll, den ersten Block zufällig zu initialisieren, denn so lassen sich auch keine Annahmen über den Ausgangsblock treffen. Im Fachjargon ist die Verknüpfung der Blöcke als *CBC* bekannt, was für »Cipher Block Chaining« steht. CBC ist ein sehr beliebter Modus bei allen gängigen Blockchiffren, den Sie im nächsten Abschnitt näher kennenlernen.

4.8 CBC – Cipher Block Chaining

Um die Häufigkeitsverteilung der Zeichen zu verschleiern, wird sehr gerne CBC verwendet. CBC ist wie bereits erwähnt die Verknüpfung aufeinanderfolgender Blöcke mittels XOR. Die Blockgröße kann dabei variieren, wobei sie in der Regel der der verwendeten Blockchiffre entspricht. Zwei Blöcke beliebiger Größe lassen sich mit der folgenden Funktion leicht miteinander verknüpfen.

[●] Sie finden den entsprechenden Quelltext zum Cipher Block Chaining auf der CD-Rom unter *Kapitel04/cbc.py*.

```
def blockXOR(a, b):
    result = []
    for x in xrange(len(a)):
        result.append(chr(ord(a[x]) ^ ord(b[x])))

    return "".join(result)
```

Listing 4.27 XOR-Verknüpfung zweier Blöcke

Die Funktion ist so ausgelegt, dass sie zwei gleich große Blöcke erwartet, weshalb auf eine Überprüfung der Größe von `b` verzichtet wurde. Falls Sie etwas mehr Sicherheit wünschen, so fügen Sie noch eine Überprüfung in die Methode ein. Eine andere Möglichkeit wäre, dass bei zwei unterschiedlich großen Blöcken der kleinere die Anzahl der mit XOR zu verknüpfenden Stellen vorgibt, wobei dann die darüber hinaus reichenden Daten verlorengehen. Alternativ könnte der kleinere Block aber auch wieder vom Start aus verwendet werden, so wie Sie es bei der Vigenère-Chiffre beim Passwort schon erlebt haben.

Die Blockverknüpfung verwenden Sie nun in den folgenden Funktionen, die eine größere Datenmenge in Blöcke teilen und verknüpfen.

```
def cbcForward(blockLength, data, initVector):
    dataLength  = len(data)
    blockCount  = dataLength / blockLength
    vector      = initVector
    index       = 0
    cbcResult   = []

    while index < blockCount:
        start   = index * blockLength
        end     = start + blockLength
        block   = data[start:end]
        vector  = blockXOR(block, vector)
        index  += 1

        cbcResult.append(vector)

    remainder   = dataLength % blockLength
    if remainder > 0:
        block   = data[-remainder:]
        plain   = blockXOR(block, vector[0:remainder])

        cbcResult.append(plain)

    return "".join(cbcResult)
```

Listing 4.28 Cipher Block Chaining beim Verschlüsseln

Zunächst ermittelt die Funktion `cbcForward()` die Gesamtlänge `dataLength` der zu verarbeitenden Daten, worauf die Anzahl der ganzen Blöcke durch Integer-Division der Datenlänge `dataLength` durch die Blocklänge `blockLength` ermittelt wird. Der dabei möglicherweise anfallende Rest wird aber nicht unter den Tisch fallen, sondern noch separat zu einem späteren Zeitpunkt behandelt werden.

Die Variable `vector` enthält den jeweils vorherigen Block, der als Nächstes verknüpft werden soll. Wie bereits erwähnt werden immer zwei direkt aufeinander folgende Blöcke verknüpft. Abbildung 4.7 veranschaulicht die Vorgehensweise.

Die Variable `index` enthält den jeweils aktuell zu verknüpfenden Block innerhalb der Gesamtdatenmenge. Das Ergebnis der Verknüpfung wird in der hierfür gedachten Variablen `cbcResult` referenziert. Für die Anzahl der geraden Blöcke wird die `while`-Schleife durchlaufen, in der jeweils zunächst der `start` des aktuellen Blockes ermittelt wird. Für die Berechnung wird die Blocklänge mit dem aktuellen Index multipliziert. Das Ende (`end`) des aktuellen Blocks ermittelt sich durch Addition der Blocklänge und des aktuellen Startindex innerhalb der Ge-

samtdatenmenge. Der zu verarbeitende Block wird als *Slice* in die Variable `block` kopiert.

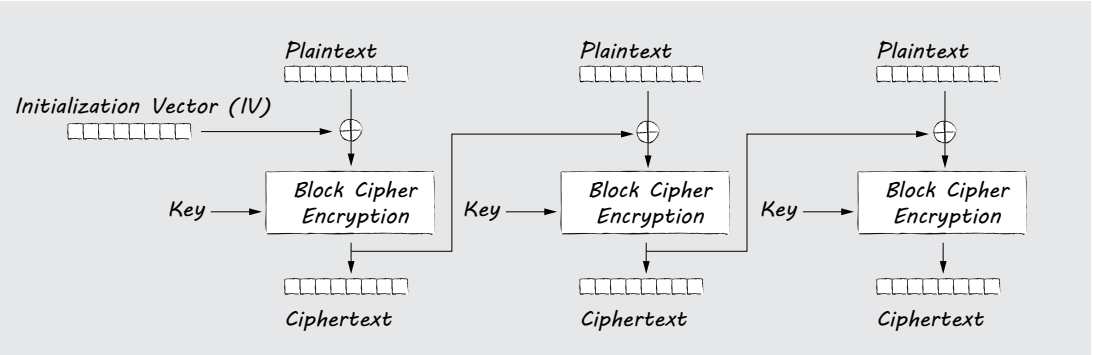


Abbildung 4.7 Cipher Block Chaining

Die zuvor besprochene Funktion `blockXOR()` wird mit dem aktuellen Block und dem Inhalt von `vector` aufgerufen. In `vector` befindet sich das Ergebnis aller vorherigen Blockverknüpfungen. Die Rückgabe von `blockXOR()` wird wieder der Variablen `vector` zugewiesen, wodurch die vorherige Aussage bezüglich dieser Variablen auch für den nächsten Block valide bleibt. Zum Schluss der `while`-Schleife wird der aktuelle Inhalt von `vector` an `cbcResult` angehängt und der Index um eins erhöht.

Im Anschluss an die `while`-Schleife wird noch ein eventuell zu behandelnder Rest verarbeitet. Mit Rest ist hierbei eine Datenmenge gemeint, die keinen gesamten Block mehr füllen würde. Für die eventuell noch verbliebenen Daten wird das letzte Verknüpfungsergebnis, das in `vector` hinterlegt ist, nur in dem Umfang angewendet, wie es für den Datenrest notwendig ist.

Die gesamte Liste in `cbcResult` wird mittels `join()` der Klasse `str` als String konkateniert zurückgegeben.

Die Umkehrung des Vorganges übernimmt die Funktion `cbcBackward()`.

```
def cbcBackward(blockLength, data, initVector):
    dataLength  = len(data)
    blockCount  = dataLength / blockLength
    vector      = initVector
    index       = 0
    cbcResult   = []

    while index < blockCount:
        start   = index * blockLength
        end     = start + blockLength
        block   = data[start:end]
        plain   = blockXOR(block, vector)
```

```
        vector  = block
        index  += 1

        cbcResult.append(plain)

    remainder   = dataLength % blockLength
    if remainder > 0:
        block   = data[-remainder:]
        plain   = blockXOR(block, vector[0:remainder])

        cbcResult.append(plain)

    return "".join(cbcResult)
```

Listing 4.29 Cipher Block Chaining beim Entschlüsseln

Die Initialisierung der lokalen Variablen für die Funktion `cbcBackward()` erfolgt analog zur Funktion `cbcForward()`. Auch die `while`-Schleife sieht auf den ersten Blick identisch zur `while`-Schleife der Funktion `cbcForward()` aus, aber hier gibt es einen Unterschied – diese Zeilen sind **fett** im Quelltext hervorgehoben. Der Unterschied besteht darin, dass der verknüpfte Block nicht als Block für die nächste Operation zu verwenden ist. Für die nächste Verknüpfung ist der Block so zu verwenden, wie er in den Daten vorhanden ist. Aus diesem Grund wird der Variablen `vector` in der Funktion `cbcBackward()` der Slice `block` anstelle des Funktionsergebnisses `plain` zugewiesen. Alles andere ist identisch, so auch die Behandlung eines eventuell vorhandenen Restes, falls die Datenmenge nicht glatt durch die Blockgröße teilbar war. Die Daten selbst werden immer von vorn durchlaufen – insofern ist der Name eventuell etwas irreführend, aber er verdeutlicht dennoch sehr gut, dass es sich um zwei entgegengesetzte Funktionen handelt.

Die zwei Funktionen berücksichtigen wie erwähnt auch, dass es bei einer vorgegebenen Blockgröße zu einem Rest kommen kann, und zwar genau dann, wenn die Gesamtgröße nicht glatt durch die vorgegebene Blockgröße teilbar ist. Die dann verbleibenden Zeichen werden analog zur Verknüpfung der gesamten Blöcke mit den entsprechenden Zeichen des Vorgängerblocks verknüpft.

Des Weiteren ist Ihnen sicher aufgefallen, dass ein `initVector` übergeben wird. Dabei handelt es sich um einen zufälligen Vektor, der als Verknüpfungsblock für den ersten Block verwendet wird. Dieser Vektor wird entsprechend der verketteten Verknüpfung über die gesamte Datenmenge »gestempelt« und wird auch bei der Umkehroperation benötigt.

Der Vollständigkeit halber sei hier erwähnt, dass diese Art, einen Zufallsvektor zu erzeugen, nicht sicher ist, da es sich nicht wirklich um zufällige Werte han-

delt. Computer können ohne Spezialhardware oder Eingaben vom Benutzer keine echten Zufallszahlen erzeugen, dann das liegt außerhalb der Möglichkeiten der exakten Vorgänge eines Rechners. Entsprechend werden Zufallszahlengeneratoren in Software meist durch Schieberegister realisiert, die nicht wirklich zufällige Zahlenfolgen erzeugen und einer festen Regel folgen. Die resultierenden Zufallszahlen werden auch als *Pseudozufallszahlen* bezeichnet.

Da echte Zufallszahlen nur schwer automatisiert zu erzeugen sind, gibt es aber auch besonders ausgereifte Zufallszahlengeneratoren, die sich durchaus für kryptografische Zwecke eignen. Da die Funktion `randint()` des Moduls `random` nicht kryptografisch sicher ist und deswegen nicht für ernsthafte Kryptographie eingesetzt werden sollte, bietet sich die folgende Funktion an, um einen pseudozufälligen Vektor zu erzeugen:

```
import random
def generateInitVector(blockLength):
    vector = []
    for x in xrange(blockLength):
        randomByte = chr(random.randint(0, 256) ^ x)
        vector.append(randomByte)

    return "".join(vector)
```

Listing 4.30 Erzeugung eines zufällig befüllten Vektors

Die XOR-Verknüpfung der Zufallszahlen mit `x` ist nicht von Bedeutung – mir missfiel einfach, dass Eclipse die nicht verwendete Variable `x` bemängelte. Allerdings hatte ich natürlich vorher überlegt, ob diese Verknüpfung die Zufallszahlen verschlechtern könnte, und beschloss intuitiv, dass sie das nicht tut – zudem sind wie gesagt diese Pseudozufallszahlen nicht kryptografisch sicher. Der Zufallszahlengenerator wird vor Aufruf in der Regel mit der Systemzeit in Millisekunden initialisiert. Echte Zufallszahlen können bei Software nicht entstehen, da Computer immer deterministisch arbeiten. Hier lässt sich durch Einbeziehung von Benutzereingaben Abhilfe schaffen. Mit der entsprechenden Hardware geht es auch ganz ohne Benutzer.

Die oben definierten Funktionen können Sie mit den folgenden Aufrufen testen:

```
a = "ABC ABC ABC ABC ABC ABC"
b = "123 456 789 123 456 789"
c = blockXOR(a, b)
print "a XOR b = c =", c
print "c XOR a = b =", blockXOR(c, a)
print "c XOR b = a =", blockXOR(c, b)
```

```
blockLength = 8
initVector  = generateInitVector(blockLength)
cbcForward  = cbcForward(blockLength, a, initVector)
print "cbcForward  =", cbcForward
print "cbcBackward =", cbcBackward(blockLength,
 cbcForward, initVector)
```

Listing 4.31 Verwendung der CBC-Funktionen

Jetzt sind wir bei den symmetrischen Verschlüsselungssystemen schon ein gutes Stück vorangekommen. Sie haben jetzt genug Informationen, um ein bisschen zu basteln und weiterzuentwickeln. Interessant ist sicherlich ein kleiner Wettkampf unter Kollegen, bei dem jeder ein einfaches Verfahren entwickelt, das von den anderen Mitstreitern zu knacken ist. Um die Komplexität der Algorithmen möglichst einfach zu halten, sollten Sie sich für so einen Wettstreit ein paar Einschränkungen bei den erlaubten Operationen einfallen lassen – am interessantesten sind sehr einfache Verfahren, deren Funktionsweise klar nachvollziehbar ist, die aber dennoch nicht leicht zu knacken sind.

Bevor wir nun einen Schritt weitergehen, möchte ich kurz an etwas erinnern, was ich bereits zu Anfang des Kapitels sagte, nämlich: Ein Problem der symmetrischen Verfahren besteht in der Notwendigkeit, den Schlüssel austauschen zu müssen. Dieses Problem wird durch die asymmetrischen Verfahren aufgegriffen, für welche kein Schlüsselaustausch notwendig ist. Was genau »asymmetrisch« in diesem Zusammenhang bedeutet, wird im nächsten Abschnitt erläutert.

4.9 Was genau heißt »asymmetrisch«?

Im letzten Abschnitt habe ich die Vigenère-Chiffre vorgestellt, die zu den symmetrischen Verschlüsselungsverfahren gehört. Symmetrische Verfahren verwenden für die Verschlüsselung den gleichen Schlüssel wie für die Entschlüsselung der Daten. Entsprechend muss der vom Versender verwendete Schlüssel dem Empfänger irgendwie bekannt gemacht werden oder bereits vor Verwendung bekannt gewesen sein. Aus diesem Grund ist eine sichere Übertragung des Schlüssels von entscheidender Bedeutung für die Sicherheit des Nachrichtenkanals. Der Schlüsselaustausch ist also ein nicht zu unterschätzendes Problem, da hierfür ein sicherer Übertragungskanal benötigt wird.

Bei symmetrischen Verschlüsselungsverfahren müssen sich Sender und Empfänger also zuvor auf eine Liste von Schlüsseln einigen, die sie der Reihe nach verwenden. Unterschiedliche Schlüssel für verschiedene Nachrichten sind noch sicherer, als nur einen einzigen Schlüssel zu verwenden, da Rückschlüsse an-

hand der Häufigkeitsverteilung der Buchstaben möglich wären. Die bei der Vigenère-Chiffre angesprochene Möglichkeit der Datenkompression ist auch keine echte Lösung, da hier immer noch Rückschlüsse möglich wären, sobald das Kompressionsverfahren bekannt ist.

Bei asymmetrischen Verschlüsselungsverfahren ist der vorherige Austausch der zu verwendenden Schlüssel hingegen nicht notwendig, denn für die Verschlüsselung wird ein öffentlicher, jedem bekannter Schlüssel verwendet, der jedoch nicht für die Entschlüsselung der Daten vewendet werden kann. Da es also nicht nur einen Schlüssel für Entschlüsselung und Verschlüsselung gibt, sondern zwei voneinander verschiedene Schlüssel existieren, spricht man in diesem Fall von asymmetrischer Verschlüsselung.

Diese asymmetrische Verschlüsselung wird durch sogenannte Einwegfunktionen ermöglicht. Das sind Funktionen, die sehr schnell berechnet werden können, während die Umkehrfunktionen nur sehr schwer oder im Idealfall gar nicht zu berechnen sind. Für die in der Kryptographie gängigen Verfahren gilt Ersteres. So ist es zum Beispiel sehr schwer, den diskreten Logarithmus in der Gruppe der Punkte einer elliptischen Kurve zu berechnen, worauf die Elliptische-Kurven-Kryptographie basiert. Ähnlich schwierig ist es, eine Zahl in Primfaktoren zu zerlegen, da es hierfür noch kein schnelles allgemeingültiges Verfahren gibt. Wir werden nun mittels großer Primzahlen verschlüsseln – was durchaus seinen Reiz hat und mathematisch nicht ganz so viele Vorkenntnisse voraussetzt wie Verfahren, die auf elliptischen Kurven basieren.

4.10 Das RSA-Kryptosystem

Im diesem Abschnitt werden wir das RSA-Kryptosystem implementieren, das von den drei Kryptologen Ronald L. Rivest, Adi Shamir und Leonard Adleman am *MIT (Massachusetts Institute of Technology)* entwickelt wurde. Die Bezeichnung RSA ergibt sich dabei aus den Anfangsbuchstaben der Nachnamen der drei beteiligten Kryptologen. Dieses System macht sich die Schwierigkeit der Primfaktorzerlegung für Produkte sehr großer Primzahlen zunutze. Es wird wie bei allen asymmetrischen Verfahren ein öffentlicher sowie ein privater Schlüssel generiert. Bevor wir uns die mathematischen Hintergründe genauer anschauen, möchte ich Ihnen das Vorgehen an einem kleinen Beispiel verdeutlichen. Wenn Sie nicht alles verstehen, dann ist das an dieser Stelle kein Problem – schauen Sie sich einfach die verschiedenen Schritte an, und Sie werden feststellen, dass am Ende wieder das Gleiche herauskommt. Wobei hier mit dem »Gleichen« die im Beispiel verwendete Zahl gemeint ist, welche zunächst durch die Verschlüsselung in eine andere Zahl umgerechnet wird, um zu guter Letzt – nach erfolgreicher Entschlüsselung –

wieder aufzutauchen. Den Grund dafür werde ich Ihnen natürlich erläutern, kann Ihnen aber hier schon einmal versichern, dass es kein Zufall ist, sondern astreine Mathematik.

Seien Sie ganz beruhigt: Wenn beispielsweise Ihr Abend etwas kürzer ist als sonst und Sie vielleicht aus diesem Grund (oder aus ganz anderen) gerade keine Lust haben, die Mathematik hinter dem RSA-Kryptosystem vollständig nachzuvollziehen, dann können Sie den nächsten Abschnitt getrost überspringen und am Ende trotzdem den richtigen Code schreiben. Für alle, die es dann aber doch genau wissen wollen, gibt es nur eins: Mitten hinein in die Mathematik mit dem folgenden Abschnitt 4.11.

4.11 Ausflug in die Mathematik des RSA-Kryptosystems

Beginnen werden wir den Abstecher in das Reich der Mathematik mit einem anschaulichen Beispiel. Für dieses kleine Beispiel wählen wir die zwei Primzahlen Elf und Dreizehn. Diese sollen für uns als die Variablen `p` und `q` gelten. Nun ermitteln wir für beide Zahlen das Produkt `N`. Das Produkt `N` wird auch als Modul bezeichnet, da es die Zahl ist, welche für die Modulo-Operationen verwendet wird, die den Restklassenring bilden werden.

Was ist ein Restklassenring? [+]

Ein Restklassenring bezeichnet in der Mathematik eine Menge ganzer Zahlen, die als Rest bei der Division durch eine den Ring klassifizierende Zahl n auftreten. Für den Restklassenring modulo 2 zum Beispiel, enthält die dazugehörige Menge lediglich die Zahlen 0 und 1, weil jede natürliche Zahl modulo 2 entweder 0 oder 1 als Ergebnis zurückliefert. Mathematikbegeisterte Leser finden ausführlichere Informationen, wenn sie dem Literaturverweis [Bundschuh 2008] im Literaturverzeichnis folgen.

```
p = 11
q = 13
N = 143 <=  p*q
phi(N) = 120 <= (p-1)*(q-1)
```

Listing 4.32 Wählen der Parameter

Als privater Schlüssel dienen uns `p` und `q`, als öffentlicher Schlüssel `N`. Nun bestimmen wir mittels der Eulerschen Phi-Funktion die Anzahl der ganzen Zahlen, die teilerfremd zur eingesetzten Zahl sind. Für `N` berechnet sich die Eulersche Phi-Funktion phi(N) = (p-1)*(q-1). Diese Formel ergibt sich aus dem Sachverhalt, dass zu jeder Primzahl p genau p -1 teilerfremde ganze Zahlen existieren, die kleiner als die Primzahl p sind. Also als Formel ausgedrückt gilt für jede Primzahl

p: phi(p) = p - 1. Nun ist anhand dieser Anzahl teilerfremder ganzer Zahlen ein Exponent e zu finden, der größer als eins und kleiner als die bekannte Anzahl teilerfremder ganzer Zahlen ist, also kurz 1 < e < phi(N).

Für die spätere Umkehrung der Rechnung benötigen wir den Entschlüsselungsexponenten d, der das multiplikativ Inverse von e bei Verwendung des Moduls Phi(N) repräsentieren muss. Das können Sie sich vereinfacht so vorstellen, als ob Sie einmal mit zwei multiplizieren und als Umkehrrechnung mit einhalb multiplizieren, nur dass wir in diesem Fall nicht multiplizieren, sondern innerhalb des Restklassenrings potenzieren. Allerdings wird im Restklassenring nur mit ganzen Zahlen gerechnet. Eine einfache Multiplikation, ohne in einem Restklassenring zu arbeiten, wäre natürlich überhaupt nicht sicher, weil das Inverse der Multiplikation zu offensichtlich wäre und entsprechend leicht bestimmt werden könnte. Entsprechend ist für die Zahl d also folgende Bedingung zu erfüllen:

$$e * d \equiv 1 \pmod{\Phi(N)}$$

Die Bedingung können Sie sich wieder vereinfacht anhand der Multiplikation vorstellen. Denn wenn e zum Beispiel 2 wäre, dann wäre natürlich e * d, wobei d dann einhalb sein müsste, gleich eins. Jetzt stellt sich nur die Frage: Wie ermittelt man das Multiplikativ Inverse, also die Zahl d innerhalb des Restklassenrings?

Die Zahl d wird mit dem erweiterten Euklidischen Algorithmus bestimmt. Den Euklidischen Algorithmus kennen Sie vielleicht schon, es handelt sich dabei um ein Verfahren zur Bestimmung des größten gemeinsamen Teilers. Falls Sie den euklidischen Algorithmus noch nicht kennen, so finden Sie eine Beschreibung in [Beutelspacher 2007] und [Bundschuh 2008]. Beim erweiterten Euklidischen Algorithmus werden zusätzlich ein paar Zwischenergebnisse nicht einfach verworfen, sondern gespeichert und zum Schluss ausgewertet. Für die Auswertung gilt der folgende Zusammenhang:

$$e * d + k * \phi(N) = 1 = ggT(e, \phi(N))$$

Für unser kleines Beispiel ergeben sich hieraus die folgenden Parameter:

```
e = 23
d = 47
```

Listing 4.33 Zwei passende Komplementäre

$$23 * 47 \equiv 1 \pmod{120}$$

$$1080 \bmod 120 = 1$$

Der erweiterte Euklidische Algorithmus zur Bestimmung von d lässt sich in Python wie folgt formulieren:

```
def xggt(a,b):
    x1, x2, y1, y2 = 1, 0, 0, 1
    while b:
        q = a // b
        x1, x2, y1, y2 = x2, x1 - q * x2, y2, y1 - q * y2
        a, b = b , a % b

    return a, x1, y1
```

Listing 4.34 Erweiterter Euklidischer Algorithmus

Alles in allem haben wir bis hierher die untenstehenden Parameter ermittelt, wobei wir die Parameter p, q und phi(N) nicht mehr benötigen. In der Praxis werden diese Parameter aber häufig dennoch mit dem privaten Schlüssel gespeichert, da sie eine schnellere Entschlüsselung ermöglichen. Die schnellere Entschlüsselung wird durch Anwendung der Erkenntnisse des Chinesischen Restsatzes möglich, der eine Aussage über simultane Kongruenzen für den Fall, dass die Moduln teilerfremd sind, trifft. Wenn Sie keine Faible für Zahlentheorie haben, dann sagt ihnen das jetzt vermutlich noch gar nichts. Falls Sie die Details dazu jedoch interessieren, so finden Sie im Literaturverzeichnis weiterführenden Lesestoff unter den Verweisen [Bundschuh 2008] und [Scharlau 1980]. Allerdings werden die Erkenntnisse dieses Satzes im Laufe dieses Kapitels ohnehin nicht verwendet, so dass Sie die Information zum chinesischen Restsatz auch einfach ignorieren können.

```
p = 11
q = 13
N = 143 = p*q
phi(N) = 120 = (p-1)*(q-1)
e = 23  = frei gewählt, aber 1 < e < phi(N)
d = 47 = multiplikativ Inverses von e
```

Listing 4.35 Zusammenfassung der Parameter

So weit, so gut, aber wie ver- und entschlüsseln wir nun anhand dieser Parameter? Angenommen, wir möchten die Zahl Sieben verschlüsselt übertragen, dann ergibt sich die verschlüsselte Nachricht durch die folgende Rechnung:

$$C \equiv 7^{23} \bmod 143 = 2$$

Entsprechend ist die verschlüsselte Botschaft die Zahl Zwei. Der Empfänger kann diese Nachricht nach dem Empfang wie folgt entschlüsseln:

$$K \equiv 2^{47} \bmod 143 = 7$$

Die sehr hohen Exponenten führen zu extrem großen Zahlen, auch schon bei diesen sehr kleinen Parametern. Um bei den in der Praxis üblichen Zahlen nicht zu viel Rechenzeit zu beanspruchen, bietet es sich an, das Potenzieren in Teilschritten zu erledigen, wobei nach jedem dieser Teilschritte das Teilergebnis durch Modulo 143 wieder reduziert werden kann.

Wie wir bereits in unserem kleinen Beispiel gesehen haben, entspricht der private Schlüssel beim RSA-Verfahren im einfachsten Fall zwei Primzahlen (p und q) und der öffentliche – der als *RSA-Modul* bezeichnet wird – dem Produkt der Primzahlen. Natürlich reicht es auch, das multiplikativ Inverse d als privaten Schlüssel zu speichern und p und q zu löschen.

Falls man aber von p und q ausgeht, so ist zusätzlich für das Entschlüsseln auch der Verschlüsselungsexponent e nötig, um d zu bestimmen. Die Vorgehensweise in diesem Fall ist im Folgenden noch einmal kurz widerholend zusammengefasst. Mittels der zwei Primzahlen wird jetzt die Eulersche Phi-Funktion berechnet. Wenn der RSA-Modul (N) sich aus der Multiplikation der beiden Primzahlen (p und q) ergibt, so ist $\Phi(N) = (p - 1) * (q - 1)$, also wie folgt:

$$N = p * q$$

$$\Phi(N) = (p - 1) * (q - 1)$$

$$1 < e < \Phi(N)$$

$$e \not| \Phi(N)$$

Der Verschlüsselungsexponent e ist zudem teilerfremd zu $\Phi(N)$ zu wählen. Für die Entschlüsselung können wir nun auch d als multiplikativ Inverses zu $\Phi(N)$ ermitteln, wobei wie erwähnt der erweiterte Euklidische Algorithmus zum Einsatz kommt:

$$e * d \equiv 1 \pmod{\Phi(N)}$$

Die Zahlen p, q und Φ(N) werden nach dem Ermitteln der eigentlichen Schlüssel, die nur aus (d, e) und (N, e) bestehen, gelöscht. Die Zahl d lässt sich anhand von (N, e) nur sehr schwer ermitteln, da hierfür N in seine Primfaktoren zerlegt werden müsste, um danach Φ(N) ausrechnen zu können. Dies ist wie bereits erwähnt für das Produkt zweier sehr großer Primzahlen nicht in angemessener Zeit möglich.

Möglicherweise raucht Ihnen jetzt der Kopf, falls diese Thematik vollkommenes Neuland gewesen sein sollte. Falls dem so ist, dann ignorieren Sie diesen Abschnitt ruhig ersteinmal.

4.12 RSA in Python

Für die im vorherigen Abschnitt beschriebenen Schritte werden wir nun entsprechende Funktionen implementieren, wobei wir an verschiedenen Stellen auf Bibliotheksfunktionen zurückgreifen werden. Die erste Hürde des Verfahrens ist das Bestimmen zweier sehr großer Primzahlen. Um zweifelsfrei zwei sehr große Primzahlen zu ermitteln, wird jedoch sehr viel Zeit benötigt. Deshalb wird in der Praxis mit Pseudoprimzahlen gearbeitet, die mit sehr hoher Wahrscheinlichkeit Primzahlen sind, aber eben nicht mit absoluter Sicherheit. Um Pseudoprimzahlen zu erzeugen, wird einfach eine sehr große Zufallszahl generiert, die dann auf ihre Eigenschaften hin untersucht wird. Dies wird in der Praxis mit dem Miller-Rabin-Test bewerkstelligt. Dieser Test ist ein Monte-Carlo-Algorithmus, also ein auf Zufall basierender Algorithmus, dessen Aussage mit einer nach oben beschränkten Wahrscheinlichkeit falsch sein kann. Hin und wieder werden auch verschiedene Primzahltests kombiniert, was die Wahrscheinlichkeit für das Fehlschlagen noch einmal reduziert. Für alle, die keine Pseudoprimzahlen verwenden wollen und lieber ganz sicher gehen, gibt es Primo. Einen Screenshot des Programmes sehen Sie in Abbildung 4.8.

Echte Primzahlen nachweisen: Primo! [CD]

Für diejenigen unter Ihnen, die die doch lieber auf Nummer sicher gehen, gibt es das Programm *Primo*, das sehr große Zahlen auf ihre Primeigenschaften testet und echte Primzahlen nachweist. Dieses Programm kann nach einer gewissen Laufzeit mit absoluter Sicherheit ermitteln, ob eine Zahl eine echte Primzahl ist. Primo finden Sie unter *http://www.ellipsa.eu/* und auf der beiliegenden CD-Rom.

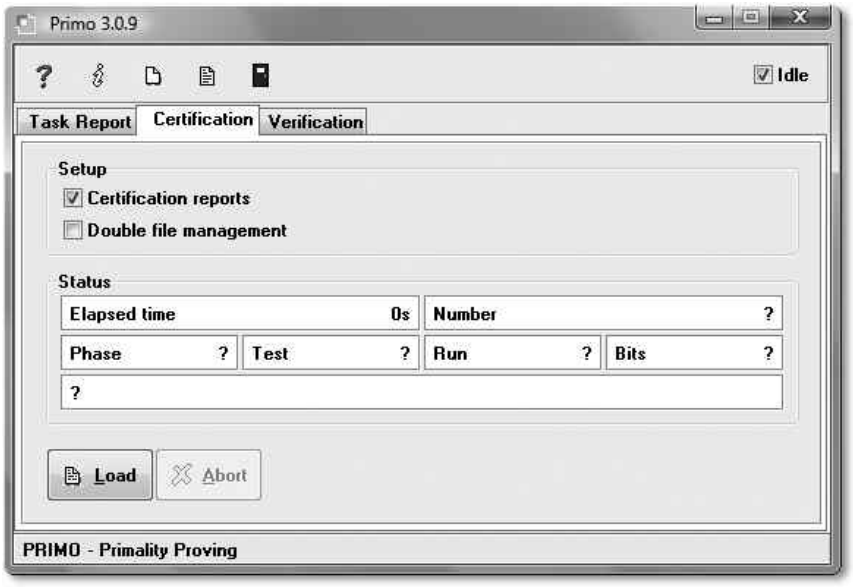


Abbildung 4.8 »Primo« – für alle, die auf Nummer sicher gehen wollen

Des Weiteren benötigen Sie schnelles modulares Rechnen für die jeweiligen Rechenschritte zum Ver- und Entschlüsseln. Damit ist gemeint, dass zum Beispiel beim Potenzieren Teilschritte vollzogen werden, nach denen jeweils reduziert wird. Modulares Rechnen kommt im Alltag sehr häufig vor, zum Beispiel beim Darstellen der Uhrzeit, wo die Stunden im Prinzip die Minuten modulo 60 angeben. Es handelt sich also nur um Rechnen mit Rest. Für die jeweiligen Schritte interessiert in der Regel nur der Rest, weshalb modulares Rechnen zum Einsatz kommt.

Beginnen wir zunächst mit einer interessanten Rechenart – der sogenannten »Russischen Bauernmultiplikation«, die auch unter dem Namen »Ägyptisches Multiplizieren« bekannt ist. Für diese Art des Multiplizierens schreiben Sie die beiden Faktoren nebeneinander auf und halbieren die linke Zahl, während Sie die rechte Zahl verdoppeln. Die sich beim Halbieren ergebenden Reste werden dabei abgerundet. Die einzelnen Ergebnisse schreiben Sie unter die beiden Faktoren und streichen für die rechte Zahl alle Zwischenergebnisse, für die die linke Zahl gerade ist. Die Summe der nicht gestrichenen, rechts noch verbleibenden Zahlen ist das gesuchte Ergebnis.

Angenommen, Sie haben die Zahlen 35 und 77 zu multiplizieren, dann sähe die Berechnung wie in Tabelle 4.1 aus.

1. Zahl	2. Zahl	zu addieren
34	77	—
17	154	154
8	308	—
4	616	—
2	1232	—
1	2464	2464

Tabelle 4.1 Russische Bauernmultiplikation

Diese Vorgehensweise lässt sich in Python wie folgt nachvollziehen.

Sie finden den entsprechenden Quelltext zur RSA-Chiffre auf der beiliegenden CD-Rom unter *Kapitel04/RSA.py*. **[◉]**

```
def multi(x, y):
    '''Multiplizieren durch Addieren und Verdoppeln'''
    result = 0
    while y > 0:
        if y % 2 == 1:
           # bei ungerade Werten für die 2. Zahl wird
           # das Zwischenergebnis der 1. Zahl summiert
            result += x
           # die Reduzierung um 1 macht die 2. Zahl
           # wieder gerade und verhindert einen Rest
           # bei der Division durch 2, wobei es hier
           # nur darum geht, in den else-Zweig zu wechseln
            y -= 1
        else:
            x *= 2
            y /= 2

    return result
```

Listing 4.36 Multiplizieren durch Addieren und Verdoppeln

Wenn an Stelle der Verdoppelung eine Quadrierung erfolgt, so erhalten wir den »Square and Multiply«-Algorithmus, der für das RSA-Verfahren von Nutzen sein wird. Der Algorithmus gestaltet sich in Python wie folgt:

```
def pot(x,y):
    '''Potenzieren durch Quadrieren und Multiplizieren'''
    result = 1
    while y > 0:
        if y % 2 == 1:
```

```
            result *= x
            y -= 1
        else:
            y /= 2
            x *= x

    return result
```

Listing 4.37 Potenzieren durch Quadrieren und Multiplizieren

Für die Verschlüsselung und Entschlüsselung können die Zwischenergebnisse des »Square and Multiply«-Algorithmus noch modulo n reduziert werden; die Zahl, die wir für n einsetzen, betrachten wir später noch genauer. Die sich daraus ergebende Funktion ist denkbar einfach:

```
def modpot(x,y,n):
    '''
    Schnelles Potenzieren x^y (mod n)
    nach der Methode "square and multiply".
    Zwischenergebnisse werden modulo n reduziert
    '''
    result = 1
    while y > 0:
        if y % 2 == 1:
            result *= x
            result %= n
            y -= 1
        else:
            x *= x
            x %= n
            y /= 2

    return result
```

Listing 4.38 Schnelles Potenzieren

Für die Verschlüsselung und Entschlüsselung kommt unsere neue Funktion `modpot()` zum Einsatz.

Durch das Reduzieren der Zwischenergebnisse bleibt die Berechnung sehr leicht und kann entsprechend schnell ausgeführt werden. Die eigentliche Verschlüsselung beziehungsweise Entschlüsselung des RSA-Verfahrens erfolgt nach Ermittlung der Schlüsselparameter wie folgt:

$$C \equiv M^e \bmod n (C = R_n(M^e))$$

$$2^{27} \equiv 18 \bmod 55$$

$$M \equiv C^d \bmod n (M = R_n(C^d))$$

$$18^3 \equiv 2 \bmod 55$$

Nun beschäftigen wir uns mit dem etwas schwierigeren Teil, der Schlüsselgenerierung. In dem Zusammenhang geht es zunächst um die Erzeugung von großen Pseudoprimzahlen. Zu diesem Problem haben Sie bereits früher etwas in diesem Kapitel gelesen, als das Programm Primo vorgestellt worden ist. Nun möchten wir aber lediglich Pseudoprimzahlen finden, also Zahlen, die mit großer Wahrscheinlichkeit Primzahlen sind. Die Überprüfung auf echte Primzahlen ist zwar auch denkbar, aber einfach zu zeitintensiv für genügend große Zahlen.

Ein echter Primzahltest wäre wie folgt möglich, wobei dies die einfachste, aber zugleich eine sehr zeitaufwändige Vorgehensweise wäre:

```
def isprime(n):
    if n % 2 == 0 and not n == 2:
        return (False)
    else:
        limit = int(math.sqrt(n)) + 1

    for l in range(3, limit, 2):
        if n % l == 0:
            return (False)

    return (True)
```

Listing 4.39 Primzahltest – Sieb des Eratosthenes

Aufgrund der Laufzeitbeschränkungen verwenden Sie aber einen nicht absolut sicheren Test.
Im Folgenden lernen Sie den Miller-Rabin-Test kennen, dessen Fehlerwahrscheinlichkeit sich mit jedem Testdurchlauf halbiert, weshalb 50 Testdurchläufe absolut ausreichend sein sollten. 50 Tests entsprechen dann einer Fehlerwahrscheinlichkeit von 2^{-50} = 8,88178419700125232338905334472 66*10^{-16}.

```
def toBinary(n):
    """
    toBinary(n)
    wandelt n in eine Liste, welche die Binärdarstellung
    von n repräsentiert - diese Darstellung vereinfacht
    die folgenden Schritte bzgl. der Primzahlprüfung
    """
    r = []
    while (n > 0):
        r.append(n % 2)
        n = n / 2
    return r

def test(a, n):
    """
    test(a, n)
    überprüft, ob n (einfach) zusammengesetzt ist,
    also als Produkt anderer Zahlen entsteht
    """
    b = toBinary(n - 1)
    d = 1
    for i in xrange(len(b) - 1, -1, -1):
        x = d
        d = (d * d) % n
        if d == 1 and x != 1 and x != n - 1:
            return True
        if b[i] == 1:
            d = (d * a) % n
        if d != 1:
            return True
    return False

def MillerRabin(n, s = 50):
    """
    MillerRabin(n, s = 1000)
    überprüft, ob n prim ist
    """
    for j in xrange(1, s + 1):
        a = random.randint(1, n - 1)
        if (test(a, n)):
            return False # n ist zusammengesetzt
    return True # n is prime
```

Listing 4.40 Miller-Rabin-Test: suchen von Pseudoprimzahlen

Für den Miller-Rabin-Test ist es günstig, wenn die zu testende Zahl zunächst in ihre Binärdarstellung umgewandelt wird, was die Funktion `toBinary()` übernimmt. Der eigentliche Test findet dann in der Funktion `test()` statt, die durch die Funktion `MillerRabin()` entsprechend der angestrebten Genauigkeit mehrmals aufgerufen wird. So finden wir die für das Verfahren benötigten Primzahlen, wobei die Wahrscheinlichkeit, dass es keine Primzahlen sind, sehr gering ist. Alternativ verwenden Sie das bereits früher erwähnte Programm Primo, wenn Sie ganz sicher sein wollen, dass die verwendete Zahl eine Primzahl ist.

Aus den Primzahlen wird das Produkt gebildet, was den Modul darstellt, mit dem gerechnet wird. Für die Berechnungen kommt die Funktion `modPot()` zum Einsatz. Als kleines Beispiel verschlüsseln Sie jetzt einen kurzen Text mittels RSA. Dazu müssen Sie den Text zunächst in eine große Zahl umwandeln, die dann entsprechend der Formeln weiter oben verrechnet wird. Die dabei ermittelte Zahl wandeln Sie einfach wieder in einen Text um, der dann jedoch im Binärformat vorliegt. Natürlich wäre es mit minimalem Aufwand auch möglich, die Zahl in einen Text umzuwandeln, der nicht binär, sondern gut lesbar ist. Hierfür wäre dann aber eine weitere Funktion notwendig, die dieses gut lesbare Format wieder in eine Zahl umwandeln kann. Die beiden Funktionen zum Umwandeln in eine Zahl und wieder zurück in Text sehen wie folgt aus:

```
def strToNumber(text):
    x       = 1
    number  = 0
    for zeichen in text:
        number  += ord(zeichen)*x
        x       *= 256

    return number

def numberToStr(number):
    text = []
    while number > 0:
        zeichen = chr(number % 256)
        number  = number / 256
        text.append(zeichen)

    return "".join(text)
```

Listing 4.41 Umwandlung von Strings in Zahlen

Die Funktion `strToNumber()` wandelt einen Text in eine große Zahl um, wobei einfach jeder Buchstabe als Wert interpretiert wird, der in einem Zahlensystem zur Basis 256 existiert. Um zu einer Zahl zu gelangen, wird aufgrund die-

ser Annahme in eine Dezimalzahl umgewandelt, und zwar durch Multiplikation des Stellenwertes mit dem Basiswert hoch der aktuellen Position. Die Funktion `numberToStr()` konvertiert eine Dezimalzahl in einen Text, was man wiederum ebenfalls als eine Umwandlung in eine Zahl eines Zahlensystems mit Basis 256 betrachten kann. Die Umwandlung erfolgt über den Euklidischen Algorithmus. Die Ver- sowie Entschlüsselung geschieht nun ganz einfach durch die entsprechenden Rechenschritte, die bereits weiter oben erwähnt wurden.

```
import rsa

q=864006630282221617546607165505132768833442927403087253437971070579738052430368997434168187774249241951614756886898778257056066247808371060434399495780120519992351353974141025134838110883240469955211816819125032764542995315946722704756735945014229730497747034468419619537240011450094095310929293646545904289036986543253037388962051682151266850800780321722541

p=203760457035785707274440765779841571954960905020216438093288697556051306412361010381948211478739481724728871392770550387381532382660420658266741318271699868487291176814463322229620222245833529237754749958472949097604722412462711929816226562448624147196460229785274621791738101798102616023157685119198323119793328819060209462062125952867409315980899161557

n   = p * q
phi = (p - 1) * (q - 1)
e   = 1210475170735953725411
d   = rsa.findMultiplicativeInverse(p, q, e)

text    =   "Dieser Text soll in eine Zahl umgewandelt werden"
number  =   rsa.strToNumber(text)
print number
print rsa.numberToStr(number)

cipher  =   rsa.modPow(number, e, n)
cipher  =   rsa.numberToStr(cipher)
print cipher

cipher  =   rsa.strToNumber(cipher)
cipher  =   rsa.modPow(cipher, d, n)
plain   =   rsa.numberToStr(cipher)
print plain
```

Listing 4.42 RSA in der Anwendung

Die Funktion `findMultiplicativeInverse()` aus diesem Code ist im folgenden Listing kurz dargestellt. Zunächst überprüft die Funktion die Bedingung der Teilerfremdheit und verändert bei Bedarf den Exponent e. Danach wird die Funktion `inverse_mod()` aufgerufen, die die eigentliche Arbeit leistet.

```
def findMultiplicativeInverse(p, q, e):
    phi = (p - 1) * (q - 1)
    while gcd(phi, e) > 1:
        e += 2

    d = inverse_mod(e, phi)

    return d

def inverse_mod(a, m):
    x, q, gcd = extended_gcd(a, m)

    if gcd == 1:
       # x ist das Inverse, aber es soll zusätzlich
       # sichergestellt werden, das x als positive Zahl
       # zurückgegeben wird
        return (x + m) % m
    else:
       # wenn gcd ungleich eins ist,
       # dann existiert kein Inverses,
       # da a und m teilerfremd sind
        return None
```

Listing 4.43 Ermittlung des multiplikativ Inversen

Damit wäre das RSA-Verfahren soweit vollständig implementiert. Eine Schwäche der aktuellen Umsetzung ist jedoch, dass die zu verschlüsselnden Texte nicht mehr Bits aufweisen dürfen als das RSA-Modul. Falls der Text also zu lang wird, so wird er beim Verschlüsseln vernichtet. Das liegt einfach daran, dass die Ausgangszahl bereits bei der ersten Modulrechnung viele Informationsbits verliert. Dieses Problem lässt sich aber leicht lösen, indem Sie einfach nur einen Schlüssel per RSA verschlüsseln, den Sie dann für eine symmetrische Verschlüsselung des Textes verwenden. So lässt sich der Schlüssel später wieder per RSA entschlüsseln, und die eigentliche Nachricht wird dann symmetrisch entschlüsselt. Hierfür wird der Schlüssel für das symmetrische Verfahren mittels RSA verschlüsselt und mit den ansonsten symmetrisch verschlüsselten Daten übertragen. Der Empfänger entschlüsselt dann zunächst den Schlüssel für das symmetrische Verfahren mittels RSA und verwendet daraufhin den symmetrischen Schlüssel mit dem entsprechenden Verfahren zum Entschlüsseln der restlichen Daten. Dieses Vorge-

hen hat auch deutliche Geschwindigkeitsvorteile, da der Rechenaufwand für das RSA-Verfahren etwa um den Faktor eintausend größer ist als für symmetrische Verfahren. Wenn Sie jedoch trotz dieser Performanceeinbußen nur mit RSA verschlüsseln wollen, so können Sie die Daten auch einfach in Blöcke aufteilen, die in Bit kodiert kleiner sind als der RSA-Modul.

Durch die Kombination von symmetrischen und asymmetrischen Verfahren wird ein schnelles und dennoch komfortables Verschlüsselungssystem möglich, bei dem der Schlüsselaustausch weniger problematisch ist.

4.13 Die Welt ist nicht so einfach, wie es scheint

Obwohl der Austausch der Schlüssel nun anscheinend kein offensichtliches Problem mehr darstellt, ist die Sache dennoch nicht ganz so einfach zu handhaben. Es gibt trotz der asymmetrischen Verfahren immer noch ein Problem, nämlich das der Authentizität – kurz: Wie stelle ich sicher, dass ich tatsächlich mit demjenigen kommuniziere, mit dem ich kommunizieren möchte? So besteht zum Beispiel die Gefahr, dass ein unbekannter Angreifer uns einfach seinen öffentlichen Schlüssel als den Schlüssel unseres eigentlichen Gesprächspartners verkaufen möchte. Dieses Szenario ist in Fachkreisen auch unter dem Begriff »Man-in-the-Middle-Attack« bekannt. Diese Art des Angriffs basiert auf einer Unterbrechung der Kommunikation der beiden eigentlichen Gesprächspartner, wobei die eigentlichen Gesprächspartner den öffentlichen Schlüssel vom Angreifer erhalten. Entsprechend kann der Angreifer alle Botschaften lesen und muss diese im Anschluss nur mit dem eigentlichen öffentlichen Schlüssel des tatsächlichen Empfängers verschlüsseln und weiterleiten. Dabei kann er zuvor natürlich auch die Nachricht beliebig manipulieren. Da die meiste Kommunikation über das Internet erfolgt, ist ein solches Angriffsszenario durchaus realistisch und mehr als pure Paranoia.

Eine mögliche Absicherung ergäbe sich durch den Vergleich des *Fingerprints* des öffentlichen Schlüssel. Der Fingerprint stellt eine Art Hash über den Schlüssel dar, der den Schlüssel eindeutig identifiziert. Allerdings wäre damit das ursprüngliche Problem des Schlüsselaustausches einfach nur verlagert worden, so dass nun eben der Fingerprint ausgetauscht werden müsste ... Somit bleibt also immer noch die Frage der Authentizität zu klären, also die Frage, ob mein Kommunikationspartner auch tatsächlich derjenige ist, für den ich ihn halte. Um hier einen Ausweg zu schaffen, gibt es sogenannte »Trust Center«, welche die Authentizität eines Kommunikationspartners bestätigen, indem sie den Schlüssel des Kommunikationspartners signieren. Um hier eine gültige Vertrauenskette sicherzustellen, muss sich jeder dem »Trust Center« gegenüber authentifizieren.

4.14 Soviel zur Kryptologie, aber was nun?

Für das Verschlüsseln und Entschlüsseln sind Ihnen jetzt einige Verfahren bekannt. Was nun noch fehlt ist eine nützliche Anwendung der neuen Verfahren. Mal davon abgesehen, dass Sie vermutlich nicht befürchten müssen abgehört zu werden, wäre ein verschlüsselter Chat-Client doch eine schöne kleine Applikation. Falls Sie mit Chatten nicht viel anfangen können, dann bietet sich als einfachste Anwendung ein Verschlüsselungsprogramm für Dateien an. Hierfür ist fast nichts mehr zu tun, außer eine schöne GUI zu bauen, welche die Benutzerinteraktion ermöglicht. Für die Spartaner unter Ihnen genügt hier sicherlich auch eine Kommandozeilenvariante, die entsprechend ohne GUI auskommt.

»Die Statistik ist wie eine Laterne im Hafen.
Sie dient dem betrunkenen Seemann mehr zum Halt
als zur Erleuchtung.«

Hermann Josef Abs

5 Statistik: das Ende jeglicher Intuition

Kennen Sie das »Ziegenproblem«? Es geht dabei um eine Aufgabe aus der Wahrscheinlichkeitstheorie, die der amerikanischen Fernsehshow »Let's make a Deal« entstammt und sehr schön aufzeigt, dass der menschliche Verstand beim Schätzen von Wahrscheinlichkeiten zu Trugschlüssen neigt.

Damit Sie selbst nicht Opfer eines Trugschlusses werden, entwickeln Sie in diesem Kapitel eine einfache Simulation des Ziegenproblems, mit deren Hilfe die Wahrscheinlichkeiten der verschiedenen Optionen sofort unmissverständlich sichtbar werden. Die dabei gewonnenen Erkenntnisse lassen sich dann auch leicht auf andere Fragestellungen anwenden, welche ebenfalls nicht auf den ersten Blick zu lösen sind. Eine dieser Fragestellungen betrachten wir direkt im Anschluss an das Ziegenproblem: Da beschäftigen wir uns nämlich gleich noch ein bisschen mit Roulette.

5.1 Was ist das »Ziegenproblem«?

Beim Ziegenproblem geht es um eine spezielle Situation in der oben genannten Spielshow, in der der Kandidat die Möglichkeit hat, seine bereits getroffene Entscheidung noch einmal zu revidieren. Der Spielverlauf sieht dabei wie folgt aus: Zunächst gibt es drei verschlossene Tore; hinter einem der Tore versteckt sich ein Preis. Der Kandidat wählt ein Tor, worauf der Moderator aus den zwei noch verbleibenden Tores eines (das keinen Preis enthält) öffnet und dem Kandidaten die Chance gibt, jetzt auf das nicht gewählte noch geschlossene Tor zu wechseln. Hinter den Toren ohne Preis steht immer eine Ziege – was auch den Namen erklärt.

Abbildung 5.1 So eine Ziege ist doch auch ganz nett – wo liegt da das Problem?

Für uns interessant ist dabei die Frage: Hat der Kandidat im Durchschnitt eine höhere Gewinnwahrscheinlichkeit, wenn er das Tor wechselt, oder ist es egal, was er tut? Zunächst denken Sie vermutlich, dass sich an der ursprünglichen Wahrscheinlichkeit nichts verändert haben kann. Dementsprechend wäre ein Wechsel absolut egal – 33 Prozent bleiben auch beim Wechsel einfach 33 Prozent. Aber ist es wirklich so? Durch reine Überlegung ist das nicht ganz so einfach zu klären und führte schon beim ersten Auftreten dieser Fragestellung zu starken Kontroversen. Denn es wäre auch denkbar, dass beim Wechsel eine Chance von 50 Prozent bestehen könnte, aber sind Sie sich sicher?

Wir werden uns aber zunächst nicht den Kopf darüber zerbrechen, sondern die beschriebene Situation einfach mit Python simulieren, denn Computer sind nicht anfällig für Trugschlüsse – zumindest solange sich diese nicht in die Implementierung einschleichen.

5.2 Wie lässt sich das Problem abstrahieren?

Zur Simulation des Problems werden Sie zwei Spieler ins Rennen schicken, die die zwei gegensätzlichen Strategien verfolgen. Spieler 1 wird immer bei seiner ursprünglichen Entscheidung bleiben, während Spieler 2 nach Öffnen eines Tors immer auf das noch nicht geöffnete Tor wechseln wird. Diese zwei Spieler werden den gesamten Prozess einfach parallel für 1.000 Runden durchführen und am Ende die jeweilige Erfolgsquote quittieren. Klingt einfach? Ist es auch. Das Schöne

daran ist, dass wir so die Frage absolut sicher beantworten können, ohne das Risiko, einen Irrtum zu begehen.

Wir benötigen also zur Lösung zwei voneinander unabhängige Spieler und eine dritte Instanz, die das Öffnen der Tore übernimmt – also sozusagen den Moderator. Die Spieler lassen sich wie in Listing 5.1 darstellen.

Den Quelltext zum Ziegenproblem finden Sie auf der beiliegenden CD-Rom im Verzeichnis *Kapitel05/ziegenproblem.py*. **[o]**

```
import random

class Player(object):
    def __init__(self):
        self.guess = 0
        self.wins = 0

    def newGuess(self):
        self.guess = random.randint(1, 3)
```

Listing 5.1 Die Klasse »Player«

```
    def changeMind(self, knownNumber):
        allNumbers = set((1, 2, 3))
        allNumbers.remove(self.guess)
        allNumbers.remove(knownNumber)
        self.guess = allNumbers.pop()

    def saveResult(self, winning):
        if winning == self.guess:
            self.wins += 1
```

Listing 5.2 Der Meinungswechsel und die Ergebnissicherung

Anfänglich hat keiner der Spiele einen Gewinn zu Buche stehen, da noch kein Spiel stattgefunden hat. Entsprechend sieht die Initialisierung der beiden dafür vorgesehenen Werte innerhalb der Klasse `Player` aus. Bei jedem Spiel wird für beide Spieler die Methode `newGuess()` aufgerufen, die dafür sorgt, dass sich der jeweilige Spieler ein Tor aussucht und entsprechend merkt. Die Instanz der Klasse `Player`, die den zweiten Spieler repräsentiert, wird nach Öffnen eines Tors durch den Moderator das verbliebene Tor durch Aufruf der Methode `changeMind()` wählen. Dazu übergeben wir der Methode das Tor, das durch den Moderator geöffnet wurde, als Parameter. Zum Schluss wird für beide Spieler das Tor bekannt gegeben, das gewonnen hat – hierfür dient der Aufruf von `saveResult()`, der als Parameter das Tor übergibt. Die Methode vergleicht daraufhin das selbst gewählte Tor der Spieler-Instanz mit dem Tor, das gewonnen hat, und verbucht

im günstigen Fall einen Sieg für den Spieler. Die dritte Instanz – der Moderator – habe ich der Einfachheit halber nicht extra als Klasse umgesetzt, sondern sie wird durch den Hauptablauf dargestellt, der wie folgt aussieht:

```
if __name__=='__main__':
    player1 = Player()
    player2 = Player()

    x = 0
    while x < 1000:
        player1.newGuess()
        player2.newGuess()

        winning = random.randint(1, 3)
        allNumbers = set((1, 2, 3))
        numbersToRemove = set((winning, player1.guess))
        allNumbers -= numbersToRemove
        knownNumber = allNumbers.pop()

        player1.changeMind(knownNumber)

        player1.saveResult(winning)
        player2.saveResult(winning)
        x += 1

    print "player 1 won ", player1.wins, " times"
    print "player 2 won ", player2.wins, " times"
```

Listing 5.3 Hauptablauf der Simulation

Es werden zunächst zwei Instanzen der Klasse `Player` erzeugt, welche bei der Initialisierung automatisch ein Tor wählen. In der Variablen `winning` wird daraufhin das Siegtor hinterlegt. Da Python sehr schön Mengen unterstützt, werden diese hier für die Auswahl des zu öffnenden Tors verwendet. Es werden hierfür das vom ersten Spieler gewählte Tor und das Siegtor aus der Menge aller Tore entfernt. Das danach verbleibende Tor in `knownNumber` entspricht dem vom Moderator geöffneten Tor. Der erste Spieler verfolgt hierbei die Strategie immer zu wechseln, weshalb auf der Instanz des Spielers die Methode `changeMind()` aufgerufen wird. Das vom Moderator geöffnete Tor, welches durch die Variable `knownNumber` repräsentiert wird, wird der Methode `changeMind()` übergeben, so dass innerhalb der Methode nur noch vom gewählten, zum nicht gewählten noch geschlossenen Tor gewechselt werden kann. Zum Abschluss werden die Ergebnisse durch Aufruf der Methode `saveResult()` auf die beiden Spielerinstanzen gespeichert. Der ganze Ablauf wird 1000 mal durchgeführt, um hier genug Sicherheit bezüglich der Aussagekraft der Simulation zu garantieren. Außerhalb der Schleife werden

die Siege für die beiden Spieler ausgegeben, wobei der erste Spieler immer das Tor gewechselt hat, während der zweite Spieler immer bei seiner Wahl geblieben ist.

Bevor Sie das Programm jetzt aber starten, sollten Sie ruhig versuchen, die Frage im Vorhinein selbst zu beantworten. Dabei können Sie gerne auch entsprechende Formeln heranziehen und ein bisschen herumprobieren. Machen Sie dann am besten eine klare Aussage über die Anzahl der Siege für die jeweilige Strategie. Also zum Beispiel: Egal, ob gewechselt wird oder nicht – jeder der beiden Spieler gewinnt ungefähr in 33 Prozent der Fälle. Die Art und Weise, wie die Simulation vorgeht, verrät zwar schon ein wenig über den Ausgang, aber es ist vermutlich noch nicht alles absolut klar. Viel Spaß mit der Statistik, und geben Sie nicht zu früh auf – denn wenn das Geheimnis erst gelüftet ist, dann ist jeder Anreiz, es selbstständig zu belegen, sofort verpufft.

Analog zur Vorgehensweise für diese Simulation können Sie natürlich auch jede andere Problemstellung simulieren, was häufig zu verblüffenden Resultaten führt. Wenn es um Wahrscheinlichkeiten geht, ist die Intuition meist nicht wirklich hilfreich. Wobei die geringe Einsicht in die Unwahrscheinlichkeit eines Gewinns nur zum Teil die große Beteiligung am Lotto erklärt – bei Lotto geht es für viele nicht wirklich ums Gewinnen, sondern um den Dopamin-Ausstoß beim Vergleichen der Zahlen, oder genauer gesagt kurz vor dem Vergleichen der Zahlen, also direkt während der Ziehung.

5.3 Ein weitverbreiteter Irrglaube

Wie im letzten Abschnitt soll es auch in diesem Abschnitt um die Simulation einer Spielstrategie gehen. Wir bleiben also weiterhin im Reich der Stochastik und beschäftigen uns in diesem Abschnitt mit einer Roulettesimulation, wobei es mir nur um einen kleinen Ausschnitt des Spiels geht. Wir werden also keine komplette Roulette-Simulation entwickeln, sondern nur einen Teil des Spiels betrachten, der für die Strategie maßgeblich ist.

Eigentlich sollte ich das vermutlich besser nicht schreiben, aber als ich 2004 für ein halbes Jahr in Kalifornien in San José war, verschlug es mich auch einmal nach Las Vegas. Zu meiner Entschuldigung gebe ich gleich im Vorhinein zu, dass ich bei den Themen Statistik und Wahrscheinlichkeitsrechnung ziemlich schwach auf der Brust bin. Trotzdem fand ich mich in Las Vegas wieder und wollte unbedingt eine Strategie beim Roulette ausprobieren, die mir ein Kollege aus San José vorher verraten hatte. Zu dem Zeitpunkt war mir allerdings noch nicht klar, dass er das nicht wirklich ernst gemeint, sondern dass er sich eher einen Scherz mit mir erlaubt hatte.

Die Strategie ist ziemlich weit verbreitet und im Grunde genommen nicht sonderlich schwer zu verstehen – wobei sie allerdings auch nicht wirklich funktioniert. Die Idee dahinter ist, immer auf eine Farbe zu setzen und nach jedem Verlust einfach den Einsatz zu verdoppeln. Nach dem Gesetz der großen Zahlen wird die Farbe auch irgendwann einmal folgen, aber mit »großen Zahlen« sind natürlich Zahlen weit jenseits der beispielsweise 13 gemeint.

Abbildung 5.2 Roulette (Quelle: Udo Kroener, Fotolia.com)

Trotzdem klang für mich die Strategie ganz plausibel, so dass ich mir vornahm, insgesamt 20 Dollar einzusetzen. Tatsächlich machte ich aus den 20 Dollar innerhalb von ca. ein bis zwei Stunden 200 Dollar! Wobei ich mir da noch einbildete, dass es besonders schlau sei, wenn ich warte, bis viermal Rot kommt und dann anfange, immer auf Schwarz zu setzen ... Tatsache ist natürlich, dass es dem Roulettetisch völlig egal ist, welche Farbe vorher aktuell war.

Ich dachte aber weniger über die Befindlichkeiten des Roulettetischs nach, sondern beschloss, einfach weitere 300 Dollar abzuheben, damit ich häufiger verdoppeln kann. Kaum hatte ich ein größeres Budget von insgesamt ungefähr 500 Dollar, ging die Strategie auf einmal nicht mehr ganz auf. Der Mindesteinsatz betrug 5 Dollar, wobei das Maximalgebot für den Rand bei 4.000 Dollar lag, was also bei meinen insgesamt 500 Dollar keine relevante Grenze hinsichtlich des Einsatzes setzte. Im nächsten Spiel kam dann geschlagene 13 Mal die Farbe Rot in Folge, was meiner Verdopplungsstrategie ein jähes Ende setzte – und statt in Las Vegas zu übernachten, fuhr ich kurz entschlossen weiter ins Death Valley ...

Warum diese kleine Anekdote? Ich dachte mir, die 500 Dollar können Sie sich sparen – nur für den Fall, dass Sie bereits Strategien parat haben, die Sie bei Ihrem nächsten Casinobesuch ausprobieren möchten. Falls Sie davon aber nicht die Finger lassen können und gerade die obengenannte Strategie noch einmal durchdenken – so sieht es tatsächlich aus: Wenn nach einigen Verdopplungen Ihres Einsatzes die für Sie korrekte Farbe erscheint, dann haben Sie den ersten Einsatz verdoppelt und den Verlust ausgeglichen. Entsprechend können Sie pro Spiel, wenn Sie mit 5 Dollar Ersteinsatz spielen, nur 5 Dollar gewinnen. Wäre nicht das Limit für den äußeren Rand, so könnte die Strategie mit beliebig viel Geld zwar immer noch funktionieren, aber sie wäre sehr mühselig. Zusätzlich müssen Sie berücksichtigen, dass die Wahrscheinlichkeit für Rot oder Schwarz nicht jeweils 50 Prozent beträgt, da es die Null und die Doppelnull gibt, die beide die Farbe Grün haben.

Wenden wir uns nach diesem Ausflug in die Welt der Jackpots und Jetons aber wieder unserer kleinen Simulation zu, für die wir folgende Vereinfachungen einführen: Es lässt sich nur auf eine Farbe oder eine Zahl setzen, wobei der Einfachheit halber gerade Zahlen rot sein sollen und ungerade Zahlen schwarz. Die 0 bleibt natürlich wie üblich grün; die Doppelnull lassen wir in unserer Simulation noch außen vor. Der Einwurf, dass es sinnvoller ist, auf gerade oder ungerade zu setzen, wäre hier fehl am Platze. Auf eine Farbe zu setzen hat nämlich keinen Nachteil gegenüber dem Setzen auf »Gerade« oder »Ungerade«, da in beiden Fällen die Null nicht dazuzählt, weshalb die Erfolgsaussichten in jedem Fall unter fünzig Prozent liegen. Aber nun zur Simulation.

Den Quelltext zum Roulettespiel finden Sie auf der CD unter *Kapitel05/Roulette.py*. **[◉]**

```
print "Welcome to Las Vegas!"
budget = 20

while True:
    print
    print "Choose a number: ",
    number      = readInput()
    numberBet   = askForBet(number)

    print "Red(1) or Black(0):",
    color       = readInput()
    colorBet    = askForBet(color)

    if number == -1 and color == -1:
        break
```

Listing 5.4 Die Simulation mit manueller Eingabe

Sie starten das Spiel mit den besagten 20 Dollar und dürfen auf eine Zahl und auf eine Farbe setzen, wobei Sie beim Setzen auf eine Farbe in etwa eine Siegchance von fünzig Prozent haben – das stimmt nicht ganz, da es noch die Null und Doppelnull gibt. Die Simulation überprüft der Einfachheit halber keine Sonderbedingungen wie negative Einsätze oder anderweitige Schummelversuche, also bleiben Sie fair.

Die Funktion `readInput()` kümmert sich um die Annahme Ihrer Eingaben und gestaltet sich wie folgt:

```
def readInput():
    guess = raw_input()
    number = -1
    if guess != '':
        try:
            number = int(guess)
        except Exception, arg:
            print "no valid number", arg

    return number
```

Listing 5.5 Beträge einlesen

Ungültige Eingaben werden nur insofern abgefangen, als es sich nicht um Zahlen handelt. Negative Beträge sind dagegen möglich und erlauben Ihnen, zur Abwechslung einmal die Perspektive der Bank zu betrachten, denn wenn Sie negativ setzen, erhalten Sie bei falschen Tipps den Einsatz als Belohnung. Sollten Sie jedoch richtig liegen, so werden Sie genauso hart bestraft wie sonst die Bank. (In diesen Fällen ist es natürlich von Vorteil, wenn Sie ausnahmsweise auf eine Zahl setzen.)

Die Funktion `askForBet()` nimmt Ihren Wetteinsatz entgegen und wäre für den gerade geschilderten Sachverhalt der negativen Beträge anfällig:

```
def askForBet(number):
    bet = 0
    if number != -1:
        print "Your bet:",
        bet = readInput()

    return bet
```

Listing 5.6 Wette abschließen

Sobald Sie keine Einsätze mehr tätigen, ist das Spiel beendet. Also vorausgesetzt, Sie scheiden nicht automatisch aus, weil Sie Ihr Budget komplett ausgegeben haben ...

Der Stand der Dinge wird Ihnen nach jedem Spielzug geschildert, wobei auch Ihr aktuelles Budget angezeigt wird, das in der Funktion `calculateBudget()` berechnet wurde:

```
def calculateBudget(budget, number, numberBet, color,
colorBet, realityNumber, realityColor):
    if number != -1:
        budget -= numberBet
        if number == realityNumber:
         # 35-fach + Einsatz
            budget += numberBet*36
            print budget, "number"

    if color != -1:
        budget -= colorBet
        if color == realityColor:
            budget += colorBet*2

    return budget
```

Listing 5.7 Berechnung des Budgets

Es wird zunächst überprüft, ob auf eine Zahl gewettet wurde, wobei ein Sieg mit dem 35-Fachen des Einsatzes vergolten würde. Danach erfolgt die Prüfung auf eine Farbwette, die bei Erfolg lediglich mit dem Einsatz belohnt würde, also eine reine Verdopplung des Einsatzes.

Doch lassen wir das Spiel beginnen und die Kugel rollen! Rien ne va plus ... Auf welche Nummer wird die Kugel fallen? Die zufällige Zahl wird in der Funktion `checkReality()` erzeugt:

```
def checkReality():
# hier definieren wir die Realität,
# wobei die Farbzuordnung auch
# anders aussehen könnte und hier
# rein willkürlich gewählt wurde
    realityNumber = random.randint(0, 36)
    if realityNumber % 2 == 0:
        if realityNumber == 0:
            realityColor = 3 # 3 == green
        else:
            realityColor = 1 # 1 == red
```

```
    else:
        realityColor = 2 # 2 == black

    return realityNumber, realityColor
```

Listing 5.8 Ergebnisauswertung

Den einzelnen Farben ordnen wir Zahlen zu, welche ersatzweise für die Farben stehen sollen. Hier könnten wir auch eine Enumeration verwenden, diese ist aber nicht unbedingt notwendig. Falls Sie einen leichten Drang zu Refactoring verspüren, so tun Sie sich keinen Zwang an. Bei größeren Programmen führt die Verwendung von sogenannten »Magic Numbers« – also Zahlen, welche nicht näher erläutert werden und im Quelltext auftauchen – schnell zu schwer auffindbaren Fehlern. Ein Kommentar im Quelltext ist das Mindeste, wobei der noch nicht sicherstellt, dass an allen Stellen auch mit den gleichen Zahlen die gleichen Farben referenziert werden. Bei dieser sehr kleinen Anwendung wird es jedoch übersichtlich genug bleiben.

Mittels der von `checkReality()` zurückgelieferten Werte wird im Hauptablauf entsprechend die Auswertung durchgeführt. Was nun noch fehlt, ist ein Feedback an den Spieler, der gespannt auf das Ergebnis seines kleinen Kasinoabenteuers wartet. Das Feedback liefert in diesem Fall die Funktion `showReality()` – meine Kreativität bei der Namensfindung beeindruckt mich manchmal selbst ...

```
def showReality(realityNumber, realityColor, budget):
    colorDef = 2: "black", 1: "Red", 3: "Green"
    print
    print "number:", realityNumber
    print "color:", colorDef.get(realityColor)
    print "budget:", budget
```

Listing 5.9 Ergebnispräsentation

Als Ergebnis wird die tatsächlich Zahl und die entsprechende Farbe ausgegeben, worauf das sich aus der Wette ergebende Budget ausgegeben wird.

Die genannten Funktionen fügen sich dann zusammen wie in Listing 5.10. Die Bedingung am Ende sorgt dafür, dass der Spieler freundlich hinauskomplimentiert wird, sobald sein Budget komplett aufgebraucht ist. Das ist die geldbörsenfreundliche Version; die für die Finanzen gefährlichere Version ließe hier auch einen Kredit zu.

```
if number == -1 and color == -1:
    break

realityNumber, realityColor = checkReality()
```

```
budget = calculateBudget(budget, number, numberBet, color, color-
Bet, realityNumber, realityColor)
showReality(realityNumber, realityColor, budget)

if budget <= 0:
    break
```

Listing 5.10 Das Zusammenspiel der Funktionen unserer Simulation

Am Ende des Spiels dürfen Sie sich über die Gratulation freuen – Sie haben alles richtig gemacht, auch wenn Sie eventuell pleite sind. Zum Glück nur virtuell!

```
print "Congratulations, your budget is:", budget, "Dollar"
print "Ciao!"
```

Listing 5.11 Die Gesamtbilanz ausgeben

Zu Las Vegas gibt es diesbezüglich auch einen passenden Spruch: »Diese Stadt ist nicht auf Gewinner gebaut.« Das Hauptproblem der meisten Menschen, die in Las Vegas – oder anderswo – Geld verlieren, ist das Problem, rechtzeitig aufzuhören. Solange man gut fährt und gewinnt, spielt man weiter; sobald die Strategie nicht mehr aufgeht, kann es schon zu spät sein. Zumindest die mir »empfohlene« Strategie lässt einen sehr rabiat alles am Stück verlieren – da gibt es keine Chance mehr, den Wendepunkt zu erkennen. Steigen Sie also am besten aus, wenn es super läuft – bloß nicht zu früh, aber genau da liegt der Hase im Pfeffer. Insofern hat die Börse psychologisch betrachtet schon eine gewisse Ähnlichkeit mit Glücksspiel: Wer gewinnt, steigt in der Regel nicht aus, und irgendwann kommt der Crash.

5.4 Automatisierung der Roulettesimulation

Nachdem Sie sich jetzt selbst ein bisschen mit Roulette beschäftigen konnten, möchte ich in diesem Abschnitt eine automatisierte Fassung erstellen, die lediglich auf eine Farbe setzt. Dabei wird sie auch wie viele Menschen die zuletzt aufgetretenen Farben berücksichtigen. Der Computerspieler soll also auf die schon länger nicht mehr erschienene Farbe setzen und, wenn er verliert, immer den Einsatz verdoppeln. Sie werden sehen, dass die Strategie gar nicht so schlecht ist, aber eben auch nicht so genial, wie im ersten Moment anzunehmen wäre. Wir werden auch ein Limit nach oben setzen, aber zunächst ohne Limit starten. Der Hauptablauf gestaltet sich jetzt etwas kürzer, wie die folgenden Listings zeigen.

Den Quelltext zum automatisieren Roulette finden Sie auf der beiliegenden CD unter *Kapitel05/AutomatedRoulette.py*. [CD]

```
lastColor       = 0
howManyTimes    = 0
lossCounter     = 0
numberOfGamesToPlay = 1000
while numberOfGamesToPlay > 0:
    numberOfGamesToPlay -= 1
```

Listing 5.12 Automatisierung der Simulation

Zunächst benötigen wir ein paar Variablen, die den Spielverlauf über ein paar Runden Revue passieren lassen. Entscheidend sind nun jedoch nur die letzte Farbe und die Anzahl der Wiederholungen dieser Farbe sowie die Anzahl der eventuell schon verbuchten Verluste. Um nicht ewig spielen zu müssen, begrenzen wir die Anzahl der Spiele auf `1000`.

```
if howManyTimes >= 3 and lastColor != 3:
    if lastColor == 1:
        color = 2
    else:
        color = 1

    colorBet = 5*(2**lossCounter)
else:
    color    = 0
    colorBet = 0
```

Listing 5.13 Kodierung der Strategie

Zu Beginn beobachtet der Spieler den Spielverlauf so lange, bis die gleiche Farbe dreimal in Folge aufgetreten ist. Das ist die konsequente Umsetzung eines menschlichen Denkfehlers, nämlich der Annahme, dass das Gesetz der großen Zahlen hier bereits erste Wirkung zeigen müsste. In der Realität kennt der Roulettetisch die letzte Farbe jedoch nicht, und sie wäre ihm auch relativ egal. Das heißt, dass die gleiche Farbe auch sehr häufig hintereinander auftreten kann. Unser Spieler »menschelt« jedoch, und sobald dreimal hintereinander Rot oder Schwarz aufgetaucht ist, beginnt er, auf genau die andere Farbe zu setzen. Dabei verdoppelt er den Einsatz nach jedem Verlust, um bei dem noch zu erwartenden Gewinn alle Verlust ausgleichen zu können und den Starteinsatz zu verdoppeln.

```
realityNumber, realityColor = checkReality()
budget = calculateBudget(budget, color, colorBet,
 realityNumber, realityColor)

if realityColor != color and colorBet != 0:
    lossCounter += 1
```

```
else:
    lossCounter = 0
```

Listing 5.14 Ergebnisüberprüfung

Falls es zu einem Spiel kam, bei dem der Spieler etwas eingesetzt hat, so wird sein Budget neu berechnet. Die Funktion `calculateBudget()` habe ich wie nachstehend an die vereinfachten Spielbedingungen angepasst:

```
def calculateBudget(budget, color, colorBet,
realityNumber, realityColor):
    if color != -1:
        budget -= colorBet
        if color == realityColor:
            budget += colorBet*2

    return budget
```

Listing 5.15 Budgetberechnung

Im Prinzip ist alles beim Alten geblieben – ich habe lediglich die Teile entfernt, die sich um Wetten auf eine Zahl gekümmert haben.

Die Misserfolge des Spielers bei seinen Farbwetten werden in der Variablen `lossCounter` protokolliert. Dies ist notwendig, um beim nächsten Spiel den neuen Einsatz anhand der bereits verlorenen Einsätze zu ermitteln.

```
if lastColor == realityColor:
    howManyTimes += 1
else:
    howManyTimes = 0

lastColor = realityColor

if budget <= 0:
    break
```

Listing 5.16 Ermittlung des nächsten Schrittes

In der Variablen `howManyTimes` werden die Wiederholungen einer Farbe gezählt. Damit wird für den Spieler der Spieleinstieg bestimmt – nur noch einmal zur Erinnerung: Das Gesetz der großen Zahlen findet hier keine Anwendung.

Mit einem Budget von 20 Dollar kommt unser Computerspieler nicht sonderlich weit, also testen Sie ruhig einmal 2.000 Dollar; dann werden Sie überrascht sein, wie gut die Strategie scheinbar funktioniert. Hin und wieder wird am Ende des

Spiels ein negatives Budget ausgewiesen – in diesen Fällen konnte nicht oft genug verdoppelt werden, um im Spiel zu bleiben, das heißt, Sie wären mit 0 Dollar schon eine Runde früher ausgeschieden. Spielen Sie doch einfach ein bisschen mit den Parametern Budget und Anzahl der Farbwiederholungen.

Natürlich geht der Spieler nicht immer als Gewinner hervor, aber dennoch relativ häufig – allerdings soll das jetzt keine Ermutigung sein, die Strategie mal selbst auf Herz und Nieren zu überprüfen. Denn es gibt ein ganz einfaches Problem, das immer zum Vorteil des Kasinos wirkt: Sie werden sehr wahrscheinlich nicht aufhören, solange Sie gewinnen. Zudem ist ein Spieler, der statistische Überlegungen betreibt, die womöglich auch noch halbwegs funktionieren, nicht so gern gesehen. Bei Black Jack gibt es tatsächlich noch bessere Möglichkeiten, auf Sieg zu spielen, allerdings natürlich ebenfalls nicht im Sinne des Kasinos und hart überwacht.

5.5 Rien ne va plus?

Ein Punkt, den ich zwar erwähnt, aber nicht umgesetzt habe, ist das Tischlimit. Versuchen Sie sich selbst einmal daran, und setzen Sie für den Randbereich – also für Wetten auf die Farbe – des Roulettetisches ein Limit, um zu sehen, wie sich das auf die Ergebnisse auswirkt. Momentan würde das Limit nicht viel bewirken, da das Budget lediglich 2.000 Dollar erlaubt. Wenn das Budget jedoch auf eine Million anwächst, dann steht einem Sieg des Spielers nur noch das Limit für den Tischrand im Wege, denn bei 5 Dollar Einsatz lässt sich mit einer Million sehr häufig verdoppeln. Um genau zu sein, etwa 17 Mal, denn $2^{17} * 5 = 655.360$; hoch 18 ergäbe in Summe also bereits über eine Million, da sich ja der Wert dann noch einmal verdoppeln würde. Dies zeigt, wie riskant und nicht sonderlich sinnvoll diese Strategie ist, denn Sie müssen bedenken, dass Sie nur 5 Dollar gewinnen könnten, um mit einer Million 17 Mal verdoppeln zu können – da stehen die Chancen nicht gerade gut! Außerdem gibt es nur sehr wenige Kasinos ohne Randlimit.

Solange Sie jedoch am Computer simulieren, spielt es keine Rolle, wie lukrativ – oder eben nicht – Ihre Strategie wohl sein mag. Vielleicht haben Sie ja auch Ihre eigene Strategie für Roulette, die Sie gerne mal ausprobieren wollen. Auch eine komplette Simulation des Spiels erfordert keine Hexerei und lässt sich mit wenig Aufwand durchführen. Zudem lassen sich auch beim Roulette sehr schön verschiedene Strategien miteinander vergleichen. Hierfür können Sie analog zum Ziegenproblem einfach verschiedene Spieler ins Feld schicken. Falls Ihnen jedoch Roulette gar nicht zusagt, so versuchen Sie sich doch einfach mal an Black Jack.

Mein Fazit zur erlebten Realität: die meisten Menschen spielen genau so lange weiter, bis sie ihr Geld endlich komplett verspielt haben. Dieser Zusammenhang wirkt sich gleich doppelt positiv für die Kasinos aus: Wer gewinnt, wird risikofreudiger und spielt weiter; wer verliert und sein Geld entsprechend schon abgegeben hat, verlässt automatisch das Kasino, da er ohnehin nichts mehr einsetzen könnte. Was die Rendite angeht, dürften also Spielkasinos in guter Lage nicht mehr zu schlagen sein, jedoch ist das ein Geschäft mit hohen Investitionskosten und einer sehr unangenehmen Konkurrenz, die neue Mitstreiter nicht ohne weiteres duldet.

»Experience teaches you to see the trees;
game theory helps you to see the forest.«

John MacMillan

6 Tic Tac Toe

Tic Tac Toe kennen Sie ziemlich sicher – es ist ein einfaches, aber dennoch sehr beliebtes Spiel, das zudem einen leichten Einstieg in die Spieltheorie ermöglicht. Aufgrund des einfachen Spielprinzips ist es für erste Spielereien mit der Zugberechnung bei Nullsummenspielen sehr gut geeignet. Ich verspreche Ihnen: Vor diesem Hintergrund ist Tic Tac Toe – trotz seiner geringen Komplexität – ein durchaus interessantes Spiel.

Wenn Menschen gegeneinander Tic Tac Toe spielen, dann geht das Spiel sehr häufig unentschieden aus. Gegen einen Computer, der es perfekt spielt, verlieren einige aber dennoch – zumindest am Anfang. Wir werden in diesem Kapitel Tic Tac Toe programmieren, und zwar so, dass der Computer nicht besiegt werden kann. Gleichzeitig erfahren Sie einiges über Nullsummenspiele und Algorithmen, die in diesem Zusammenhang nützlich sind. Auf Basis der von Ihnen in diesem Kapitel implementierten Algorithmen lässt sich dann aber jedes Nullsummenspiel realisieren. Was genau sich hinter dem Begriff »Nullsummenspiel« verbirgt, erfahren Sie im Anschluss an den Abschnitt über die Spielregeln.

6.1 Die Spielregeln

Für den Fall, dass Sie die Spielregeln von Tic Tac Toe noch nicht kennen, werden wir uns diese jetzt anschauen. Wie bereits geschrieben, handelt es sich bei Tic Tac Toe um ein Spiel für zwei Spieler, die abwechselnd am Zug sind. Jeder der beiden Spieler hat sein eigenes Zeichen, mit dem er seine Züge darstellt.

Tic Tac Toe wird auf einem Spielfeld mit drei × drei Feldern gespielt, also auf einem Spielfeld mit insgesamt neun Feldern. Die Spieler nehmen abwechselnd ein beliebiges noch nicht besetztes Feld durch Kennzeichnung mit ihrem Zeichen ein. Sobald es einem der beiden Spieler gelingt, drei Felder in einer Linie zu besetzen, hat er das Spiel für sich entschieden. Dabei sind sowohl horizontale und vertikale als auch diagonale Linien möglich. In der Spielstellung in der folgenden

Abbildung 6.1 hat der Spieler, der die Kreuze setzt, gewonnen, da er sein Zeichen dreimal diagonal aneinandergereiht hat.

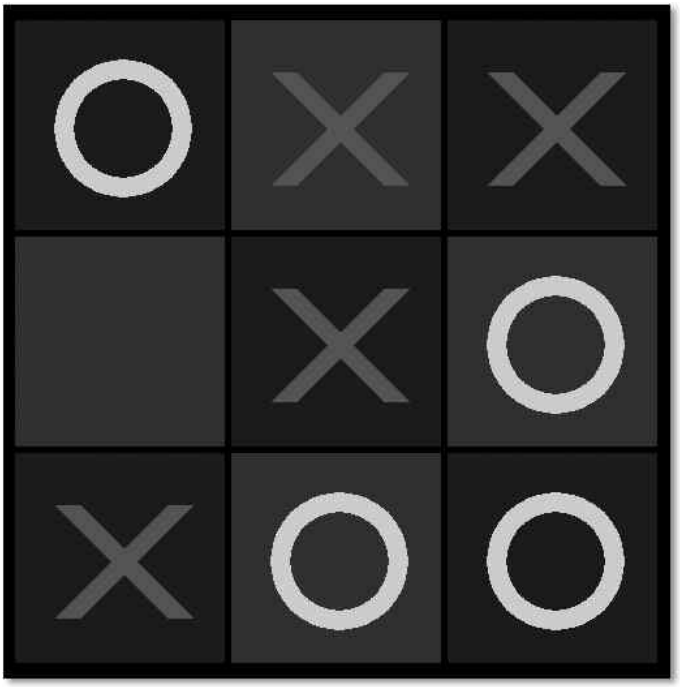


Abbildung 6.1 Der Spieler mit den Kreuzen hat gewonnen.

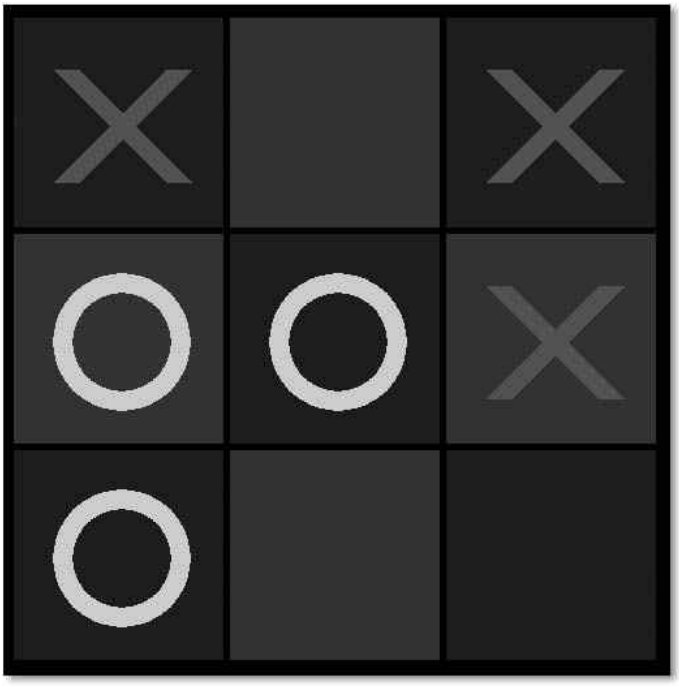


Abbildung 6.2 Der Spieler mit den Kreuzen wird gewinnen.

6.2 Was sind Nullsummenspiele?

Nullsummenspiele beschreiben in der Spieltheorie Situationen, also im verallgemeinerten Sinne Spiele, bei denen die Summe der Gewinne und Verluste aller Spieler zusammengenommen gleich null ist. Grob formuliert also Spiele, die ausgeglichen sind und keinen der beteiligten Spieler bevorzugen, solange die Spieler halbwegs auf gleichem Niveau spielen, also z. B. kein Sechsjähriger gegen einen Mathematik-Professor. Spiele wie »Tic Tac Toe«, »Vier gewinnt«, »Othello« oder Schach haben zusätzlich das Merkmal der vollständigen Information, das heißt, dass alle Spieler alle für den Spielverlauf bedeutenden Informationen besitzen und folglich den Verlauf des Spiels zumindest theoretisch komplett vorausberechnen könnten.

Dies ist in der Tat möglich, wenn es sich um Spiele handelt, die nicht zu komplex sind, das heißt, bei denen nicht zu viele Verzweigungen der Spielentscheidungen je Zug möglich sind. Ein Spiel wie Schach ist entsprechend nicht vollständig vorausberechenbar, da seine Spielkomplexität viel zu hoch ist; es gibt einfach zu viele Zugalternativen in einem zu langen Spielverlauf. Deshalb scheint die Anzahl der möglichen Spielverläufe im Prinzip nahezu unendlich groß – wobei das natürlich nicht stimmt; es handelt sich einfach um eine extrem große Zahl, die aufgrund der kombinatorischen Explosion zustande kommt.

Spiel mit Tiefgang: Schach **[+]**

Beim Schach hat jeder Spieler zunächst sechzehn Figuren und zu Beginn die Auswahl zwischen zwanzig Zügen, wobei die beiden Springer sowie die acht Bauern jeweils zwischen zwei Feldern wählen können. Mit jedem weiteren Zug steigt in der Regel die Anzahl der zur Verfügung stehenden Züge. Entsprechend gibt es, wenn man von im Schnitt zweiundzwanzig Zugmöglichkeiten pro Zug ausgeht, schon nach sechs Zügen – wobei ein Zug als ein Zug von Weiß und Schwarz gerechnet wird – 22^{12} verschiedene Spielabläufe: 12.855.002.631.049.216. Beeindruckend, nicht wahr?

Natürlich ist nicht jeder dieser Abläufe auch sinnvoll, deshalb werden verschiedene Heuristiken zur Stellungsbewertung verwendet, um einige der Äste in dem sich aufspannenden Spielbaum zu kappen. Heuristik heißt in diesem Fall, dass anhand einer statischen Stellungsbewertung versucht wird, die aktuelle Position zu bewerten, ohne dabei innerhalb des Spielbaums weiter in die Tiefe zu gehen. Im Falle von Schach geht in einer solchen Bewertung in der Regel zunächst die Materialverteilung ein, also wie viele Figuren jeder Spieler noch zur Verfügung hat. Darüber hinaus wird die Beweglichkeit der Figuren – also wie viele Felder sie kontrollieren – beurteilt sowie ihre »Kraft« beziehungsweise ihr Einfluss, also wie agil sie aufgestellt sind. So ist es zum Beispiel oft vorteilhaft, wenn eine Figur möglichst zentrumsnah postiert ist, da sie von dort aus die meiste Kraft in alle Richtungen entfalten kann. Vom Zentrum aus werden in der Regel auch die meisten Felder durch die Figuren beherrscht.

6.3 Wie »denkt« ein Computer?

Die einfachste Möglichkeit für einen Computer ist die Vorausberechnung aller möglichen Spielabläufe. Diese Möglichkeit bietet sich aber nur in sehr wenigen Fällen an, in denen es nicht zu einer zu starken Aufspaltung des Spielbaums kommt. Dies sind in der Regel Spiele mit kurzem Verlauf. Tic Tac Toe gehört zu diesen Spielen, da es hier maximal neun Züge vorauszuberechnen gibt, wobei viele der Spielverläufe bereits in weniger als neun Zügen ein Ende finden.

Aus diesem Grund reicht es, bei der Stellungsbewertung zu überprüfen, ob das Spiel für Spieler 1 oder Spieler 2 gewonnen ist, da immer bis zum Spielende vorausberechnet werden kann. Bei Tic Tac Toe verringert sich die Anzahl der noch verbleibenden Züge mit jedem Zug, was zu einer geringeren Aufspaltung des Spielbaums führt als zum Beispiel beim Schach, wo Felder mehrfach durch Figuren besetzt und wieder freigegeben werden können.

Was machen wir jetzt in der Praxis daraus? Wenn wir die Fälle mit vorzeitigem Spielende durch Sieg einer Seite vernachlässigen, so ist die Anzahl der Spielverläufe gleich neun Fakultät: 362.880 – eine für einen Computer nicht wirklich große Zahl. Entsprechend könnten wir Tic Tac Toe einfach durch Berechnen aller Spielverläufe lösen. Wir werden aber eine etwas elegantere Variante wählen, die sich auch auf komplexere Spiele anwenden lässt.

Grundlage für die Auswahl des nächsten vorauszuberechnenden Zuges wird eine einfache Stellungsbeurteilung sein. Für Tic Tac Toe reicht uns hierfür eine Funktion, die +1 bei für den ersten Spieler gewonnenen Stellungen ausgibt, -1 bei für den zweiten Spieler gewonnenen Stellungen und 0 bei noch nicht gewonnenen Stellungen.

6.3.1 Der Minimax-Algorithmus

Beginnen werden wir mit dem *Minimax-Algorithmus*, der die einfachste Suche zur Zugauswahl darstellt. Er hat seinen Namen aufgrund seiner Optimierungsstrategie – je nachdem, welcher Spieler gerade am Zug ist, wird entweder der Zug mit der höchsten beziehungsweise niedrigsten Bewertung gewählt. Entsprechend wird der maximierende Spieler den jeweils am höchsten bewerteten Zug wählen. Genau umgekehrt wird sich der minimierende Spieler verhalten.

Aber wie sieht der Algorithmus nun aus? Es handelt sich im Grunde genommen um einen Algorithmus, der jede mögliche Zugabfolge durchläuft und die beste aller Zugabfolgen zurückliefert. Dabei traversiert – eine Erklärung des Begriffs folgt im nächsten Kasten – der Mini-Max-Algorithmus immer den gesamten Spielbaum, was für komplexe Spiele enormen Rechenaufwand bedeutet.

Was heißt »Traversierung«? [+]

Mit »Traversierung« bezeichnet man in der Graphentheorie Verfahren, bei der jeder Knoten und jede Kante eines baumförmigen Graphen genau einmal besucht wird. Im Zusammenhang mit der Suche in Spielbäumen kann es bei den komplexeren Verfahren auch zu mehreren Besuchen des gleichen Knoten kommen. Für die betreffenden Algorithmen werden deshalb die Ergebnisse der Bewertungen zwischengespeichert, so dass beim zweiten Besuch nur aus dem Cache gelesen werden muss.

Für die Bewertung der einzelnen Positionen sind Kriterien zu definieren, wobei wir uns für die reine Bewertung von Endpositionen entscheiden. Entsprechend werden für den ersten Spieler gewonnene Positionen mit positiven Zahlen bewertet, während die für den zweiten Spieler gewonnenen Positionen mit negativen Zahlen bewertet werden. Da wir hier nur Endpositionen bewerten werden, welche entweder für den ersten oder zweiten Spieler gewonnen sind, werden wir entweder +1 oder -1 als Bewertung zurückgeben. Es ist aber durchaus auch möglich, die Zwischenstellungen, welche noch keinen Gewinner ausweisen, zu bewerten. In diesem Fall werden die verschiedenen Züge entsprechend mit einer größeren oder kleineren Zahl bewertet, um zwischen verschiedenen Zügen differenzieren zu können. Schauen wir uns doch das Ganze einmal im folgenden Diagramm in Abbildung 6.3 an.

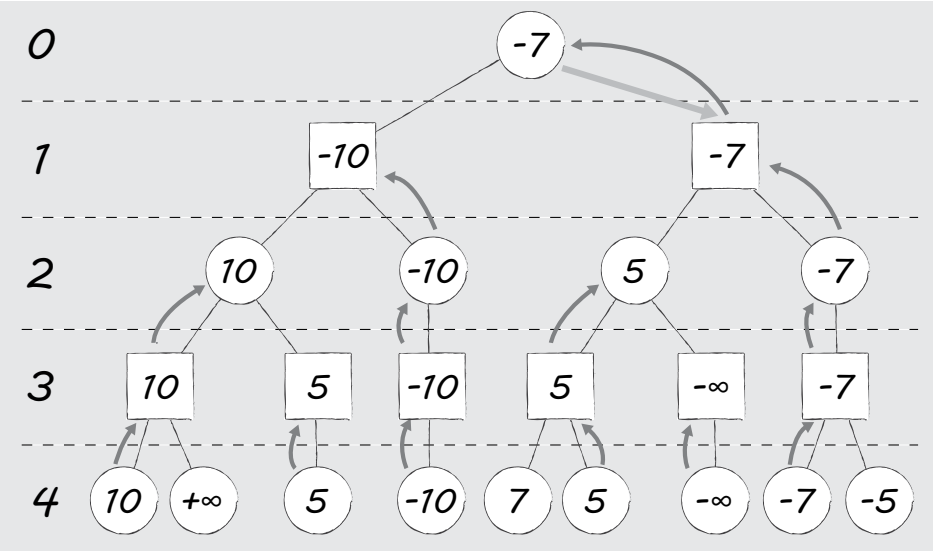


Abbildung 6.3 Minimax-Algorithmus

Mit jeder Ebene im Baum wechselt der Spieler, der am Zug ist. Entsprechend werden von den in der Ebene zur Verfügung stehenden alternativen Zügen jeweils abwechselnd mal die am höchsten und mal die niedrigsten weitergereicht. Jede zweite Ebene ist entsprechend in der Auswahl konsequent und bleibt immer bei der Wahl der höchsten oder niedrigsten Bewertungen, da es sich bei jeder zweiten

Ebene immer um den gleichen Spieler handelt. Die Zahlen im Diagramm stellen die Bewertung der einzelnen Positionen dar, wobei die Pfeile verdeutlichen, welche Züge bei dieser Bewertungskonstellation ausgewählt werden würden.

Unsere sehr einfache Bewertungsfunktion bedingt zwangsläufig, dass wir bis zum Ende rechnen müssen, da nur die Endpositionen bewertet werden. Alle nicht entschiedenen Stellungen werden wir im Folgenden als ausgeglichen bewerten, auch wenn dies nicht stimmt. Dieser Makel ließe sich durch eine prognostizierende Bewertungsfunktion – also eine Heuristik – etwas in den Griff bekommen. Bei komplexen Spielen wie Schach liegt die eigentliche Intelligenz des Spiels hauptsächlich in der Bewertungsfunktion.

6.3.2 Implementierung des Minimax-Algorithmus

Gehen wir nun ein wenig ins Detail. Der Algorithmus teilt sich in verschiedene Teilschritte auf, so dass es eine Einstiegsmethode für den Start der Baumsuche sowie eine Maximierungs- und eine Minimierungsmethode gibt. Der Minimax-Algorithmus geht auf das Minimax-Theorem zurück, das einen Spezialfall des Existenzsatzes für Nash-Gleichgewichte für Nullsummenspiele mit zwei Spielern darstellt.

Das Nash-Gleichgewicht stellt einen zentralen Begriff der Spieltheorie dar und beschreibt in nicht-kooperativen Spielen einen strategischen Gleichgewichtszustand, von dem aus kein Spieler für sich einen Vorteil erzielen kann, indem er einseitig von seiner Strategie abweicht. Kein Spieler kann sozusagen von einem solchen Gleichgewicht ausgehend einen Vorteil für sich erzwingen, solange der andere Spieler keine groben Fehlentscheidungen trifft. Das Nash-Gleichgewicht wurde von John Forbes Nash sowohl definiert als auch als existent bewiesen und stellt ein grundlegendes Lösungskonzept der Spieltheorie dar.

[◉] Den Quelltext zum Tic Tac Toe-Spiel finden Sie auf der beiliegenden CD im Verzeichnis *Kapitel06*. Der Hauptablauf des Spiels ist jeweils in *TicTacToe.py* implementiert, wobei Sie in dem Verzeichnis verschiedene Versionen des Spiels finden. Die unterschiedlichen Versionen stellen die inkrementellen Tuning-Maßnahmen dar, die in den folgenden Abschnitten noch besprochen werden.

Den Minimax-Algorithmus zum Durchsuchen des sich aus den Zugmöglichkeiten ergebenden Baums kapseln wir für die weitere Verwendung in anderen Spielen in der folgenden Klasse:

```
class MiniMax(object):
    infinity = 200
```

```
    def __init__(self, gameObject):
        self.gameObject = gameObject
```

Listing 6.1 Die MiniMax-Klasse

Das Listing zeigt einen Ausschnitt der für den Minimax-Algorithmus definierten Klasse, die mit einer Instanz eines `gameObject`s initialisiert wird. Das `gameObject` abstrahiert für uns die Spiellogik und stellt eine einheitliche Schnittstelle zur Verfügung, die es erlaubt, verschiedene Spiele mit der `MiniMax`-Klasse zu behandeln. Die unterschiedlichen Spiele müssen lediglich die Schnittstellenvereinbarung einhalten, die folgende Methoden vorsieht:

- `gameObject.getMoves()`
- `gameObject.makeMove()`
- `gameObject.undoMove()`
- `gameObject.getSide()`
- `gameObject.evaluate()`

Die Methode `getMoves()` gibt dabei alle aus der momentan im `gameObject` hinterlegten Position möglichen Züge zurück. Die Methode `makeMove()` führt den als Parameter übergebenen Zug aus, wobei das Zugformat für die Klasse `MiniMax` vollkommen unbedeutend ist, da die Züge der von `getMoves()` zurückgegebenen Liste entnommen werden. Die Methode `undoMove()` macht den zuletzt ausgeführten Zug rückgängig. Die Methode `getSide()` liefert den am Zug befindlichen Spieler zurück, was notwendig ist, um zu bestimmen, ob die Bewertung der Stellung maximiert oder minimiert werden soll. Die Methode `evaluate()` gibt eine Bewertung der aktuellen Stellung aus.

Im Anschluss an die Initialisierung der Klasse `MiniMax` generiert die Methode `getEvaluatedMoves()` eine Liste von Zügen mit der dazugehörigen Bewertung, wobei als Parameter die Tiefe der für die Bewertung zugrundeliegenden Suche übergeben wird. Die Züge innerhalb der Liste sind entsprechend ihrer Bewertung sortiert, so dass die besten Züge weiter vorn in der Liste stehen.

```
    def getEvaluatedMoves(self, depth):
        possibleMoves   = self.gameObject.getMoves()
        evaluatedMoves  = []
        for move in possibleMoves:
            self.gameObject.makeMove(move)
            evaluation = self.explore(depth)
            self.gameObject.undoMove(move)
            evaluatedMoves.append((evaluation, move))
```

```
        evaluatedMoves.sort()
        return evaluatedMoves
```

Listing 6.2 Die Einstiegsmethode für die Zugbewertung

Die Methode `getEvaluatedMoves()` ermittelt zunächst alle möglichen Züge der aktuellen Stellung durch Aufruf der Methode `gameObject.getMoves()`. Für jeden dieser Züge wird die entsprechende Stellung nach Ausführung des Zugs durch `gameObject.makeMove()` erzeugt und danach durch `explore()` bewertet. Wie bereits erwähnt wird dem Aufruf von `explore()` die Tiefe der Suche als Parameter mit übergeben, so dass nicht nur die nach dem Zug erreichte Stellung statisch bewertet wird, sondern bis in die vorgegebene Tiefe weitere Züge für die Bewertung mit einbezogen werden.

Nach der Bewertung der neuen Stellung wird der Zug dann durch den Aufruf von `gameObject.undoMove()` zurückgenommen und der bewertete Zug inklusive der Bewertung in der Liste `evaluatedMoves` als Tupel gespeichert. Die Liste `evaluatedMoves` wird zum Abschluss nach Bewertung sortiert zurückgegeben. Die eigentliche Tiefensuche wird durch den Aufruf von `explore()` eingeleitet.

```
    def explore(self, depth):
        if self.gameObject.getSide() == 1:
            return self.max(depth)
        else:
            return self.min(depth)
```

Listing 6.3 Die Einleitung der Tiefensuche

`explore()` bestimmt dabei lediglich den Spieler, der am Zug ist, und ruft je nachdem entweder die Methode `max()` oder die Methode `min()` auf, die für die aktuelle Stellung den Positionswert ermitteln. Den beiden genannten Methoden wird jeweils wie bereits der Methode `explore()` die Tiefe vorgegeben, in der maximal gesucht werden soll.

Die eigentliche »Intelligenz« des Minimax-Algorithmus liegt hauptsächlich in den beiden Methoden `max()` und `min()`. Die Unterschiede der beiden Methoden sind sehr gering und in den folgenden Listings **fett** hervorgehoben.

```
    def max(self, depth):
        alpha = -MiniMax.infinity
        moves = self.gameObject.getMoves()

        if depth <= 0 or not moves:
            return self.gameObject.evaluate()
```

```
        for move in moves:
            self.gameObject.makeMove(move)
            value = self.min(depth-1)
            self.gameObject.undoMove(move)
            if value > alpha:
                alpha = value

        return alpha
```

Listing 6.4 Die Maximierung des Minimax-Algorithmus

```
    def min(self, depth):
        alpha = MiniMax.infinity
        moves = self.gameObject.getMoves()

        if depth <= 0 or not moves:
            return self.gameObject.evaluate()

        for move in moves:
            self.gameObject.makeMove(move)
            value = self.max(depth-1)
            self.gameObject.undoMove(move)
            if value < alpha:
                alpha = value

        return alpha
```

Listing 6.5 Die Minimierung des Minimax-Algorithmus

Die Methoden unterscheiden sich also im Vergleich der aktuellen Bewertung in `value` mit dem Wert `alpha`. Hier wird, je nachdem, ob maximiert oder minimiert werden soll, entsprechend gegenteilig verglichen. Eine weitere Unterscheidung ist die Initialisierung von `alpha`, die jeweils mit dem für den aktuellen Spieler ungünstigsten Wert erfolgt und entsprechend auch exakt gegensätzlich ist.

Das Ganze noch einmal kurz zusammengefasst:

Als Einstieg in den Algorithmus wird die Methode `explore()` der Klasse `MiniMax` aufgerufen. Dabei wird als Parameter `depth` übergeben, der als Abbruchkriterium dient. Der Spielbaum wird entsprechend nur bis zur angegebenen Tiefe durchsucht. Um den Algorithmus beliebig oft wiederverwenden zu können, wird der Klasse ein `gameObject` übergeben, das die möglichen Züge aus der momentanen Position zurückgeben und mögliche Züge ausführen und rückgängig machen kann. Da das `gameObject` die Zugeingabe und das Zurücknehmen von Zügen erlaubt, wird das »Klonen« beziehungsweise Kopieren der Spielstellung nicht

notwendig. Entsprechend spart diese Vorgehensweise einiges an Arbeitsspeicher. Unter Verwendung des übergebenen `gameObject`s kann der Algorithmus so den ganzen Spielbaum durchsuchen und den für beide Spieler besten Pfad zurückgeben. Dabei wird je nach Spieler der Stellungswert entweder maximiert oder minimiert. Unabhängig davon gibt es aber auch sehr viel effizientere Algorithmen als Minimax. Allein die Feststellung, dass die Unterschiede zwischen der Methode `min()` und der Methode `max()` nur sehr geringfügig sind, deutet darauf hin, dass es hier noch einiges an Optimierungspotenzial zu geben scheint.

Aber kommen wir zunächst zur Darstellung unseres Tic Tac Toe-Spiels.

6.4 Darstellung von Tic Tac Toe

Zu Beginn ist es immer schön, etwas auf dem Bildschirm zu sehen, deshalb fangen wir zunächst mit der Visualisierung des Spielfeldes an. Dazu ist nicht viel nötig; wir müssen lediglich ein paar Linien ziehen und je nach Spieler ein anderes Symbol für den gemachten Zug darstellen. Wir werden der Tradition folgen und die Spielerzüge mit einem Kreis beziehungsweise einem Kreuz ins Spielfeld einzeichnen.

Die komplette Darstellung des Spielfeldes werden wir mit Hilfe der hierfür entwickelten Klasse `PlayingField` abkapseln, die gleichzeitig den Spielzustand speichert. Bei einer ganz sauberen Trennung der Verantwortlichkeiten würden wir darüber hinaus natürlich auch den Spielzustand in eine separate Klasse auslagern.

Innerhalb von `PlayingField` definieren wir zunächst die Methode `drawPlayerSign()` und übergeben ihr als Parameter die Position des Zeichens im Spielfeld und den Spieler. Abhängig vom jeweiligen Spieler wird dann als Markierung des jeweiligen Zuges entweder ein O oder ein X in das Spielfeld eingetragen.

```
def drawPlayerSign(self, startPosition, player):
    startX, startY  =   startPosition
    startPosition   =  (startX+self.padding,
                        startY+self.padding)
    endPosition     =  (startX+self.fieldWidth-self.padding,
                        startY+self.fieldHeight-self.padding)
    if player == 1:
        self.drawPlayerOne(startPosition, endPosition)
    else:
        self.drawPlayerTwo(startPosition, endPosition)
```

Listing 6.6 Darstellung der Züge

Anhand des als Parameter übergebenen Spielers wird die entsprechende Methode zum Darstellen des Zuges aufgerufen. Hierbei wird die Position des Zeichens mit übergeben. Diese wurde bereits in ein für das Zeichnen vorteilhaftes Wertepaar – hinsichtlich der den Zeichenfunktionen zu übergebenden Parameter – umgewandelt, das die Startposition sowie die Endposition enthält.

```
def drawPlayerOne(self, startPosition, endPosition):
    x1, y1      = startPosition
    x2, y2      = endPosition
    x2, y2      = x2-x1, y2-y1
    rect        = x1, y1, x2, y2
    innerRect   = x1+15, y1+15, x2-30, y2-30
    color       = 0, 180, 0
    backColor   = self.screen.get_at(startPosition)
    pygame.draw.ellipse(self.screen, color, rect, 0)
    pygame.draw.ellipse(self.screen, backColor, innerRect, 0)
```

Listing 6.7 Zeichen des ersten Spielers darstellen

Für den ersten Spieler werden Ringe als Zeichen im Feld dargestellt. Die Umsetzung der Ringe erfolgt über das Zeichnen einer großen ausgefüllten Ellipse, die eine etwas kleinere ausgefüllte Ellipse in der Hintergrundfarbe enthält. Dieser kleine Umweg hat einen einfachen Grund: Die 2D-API von Pygame bietet zwar auch eine Funktion zum Zeichnen eines Kreises beziehungsweise Rings, aber es erfolgt kein Anti-Aliasing. Unter *Anti-Aliasing* ist die Glättung der Darstellung zu verstehen, also zum Beispiel die Glättung von diagonalen Linien, so dass diese nicht wie kleine Treppenstufen dargestellt werden.

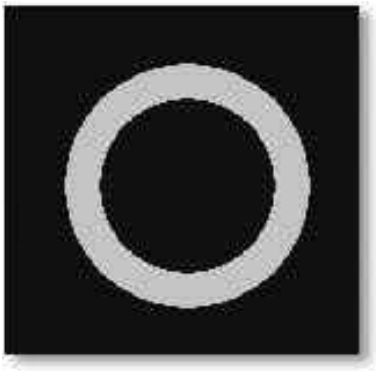


Abbildung 6.4 Zeichen des ersten Spielers

Dass Sie jetzt aber gleich eine Ellipse statt eines ausgefüllten Kreises zeichnen, hat auch den Vorteil, dass wir anhand eines Rechtecks die Größe der Ellipse bestimmen können. Wir übergeben in diesem Fall ein Quadrat, das durch die übergebenen Positionsdaten beschrieben wird.

Das Prinzip, um das Zeichen für den zweiten Spieler darzustellen, ist ähnlich:

```
def drawPlayerTwo(self, startPosition, endPosition):
    color       = 180, 0, 0
    width       = 20
    adjustion   = 10
    startX, startY  = startPosition
    endX, endY      = endPosition
    startPosition   = startX+adjustion, startY+adjustion
    endPosition     = endX-adjustion, endY-adjustion
    pygame.draw.line(self.screen, color, startPosition,
      endPosition, width)
    startX, startY  = startPosition
    startX, startY  = (startX-self.padding-adjustion,
                       startY-self.padding-adjustion)
    startPosition   = (startX+self.padding+adjustion,
                       startY+self.fieldHeight-self.padding-adjustion)
    endPosition     = (startX+self.fieldWidth-self.padding-adjustion,
                       startY+self.padding+adjustion)
    pygame.draw.line(self.screen, color, startPosition,
      endPosition, width)
```

Listing 6.8 Zeichen des zweiten Spielers darstellen

Anstelle eines Rings zeichnen wir hier ein Kreuz.

Abbildung 6.5 Zeichen des zweiten Spielers

Die in den Methoden verwendeten Bezeichner wurden als Instanzvariablen während der Erzeugung des Spielfeldes innerhalb des Konstruktors definiert wie in Listing 6.9 dargestellt. Die Variable `computerSide` bestimmt hierbei den Spieler, der vom Computer gesteuert wird. Die `2` steht hierbei für Spieler Nummer 2, so dass der menschliche Spieler den ersten Zug ausführen darf. Wenn Sie allerdings lieber den zweiten Zug ausführen wollen, so klicken Sie einfach mit der rechten Maustaste auf das Spielfeld; dann übernimmt der Computer den nächsten Zug.

```
def __init__(self, size, gameObject, compObject):
    self.gameObject     = gameObject
    self.compObject     = compObject
    self.computerSide   = 2
    sizeX, sizeY        = size
    self.field          = self.gameObject.field
    self.fieldsY        = len(self.field)
    self.fieldsX        = len(self.field[0])
    self.fieldWidth     = sizeX / self.fieldsX
    self.fieldHeight    = sizeY / self.fieldsY
    self.margin         = 50
    self.padding        = 30
    self.seperator      = 5
    self.size           = sizeX+self.margin*2, sizeY+self.margin*2
    self.screen         = pygame.display.set_mode(self.size)
    self.running        = True
    self.start()
```

Listing 6.9 Initialisierung der Klasse »PlayingField«

Nun fehlt uns nur noch die Darstellung der eigentlichen Spielfelder. Wir visualisieren sie durch das Zeichnen von entsprechenden Quadraten. Um das Spielfeld etwas ansprechender zu gestalten, färben wir die Felder, ähnlich wie bei einem Schachbrett, abwechselnd leicht unterschiedlich ein. Dazu verwenden wir ein helles und ein dunkles Blau. Zwischen den einzelnen Quadraten lassen wir ein wenig Platz frei, so dass ein Gitter sichtbar sein wird. Das Gitter an sich zeichnen wir also nicht explizit, sondern erhalten es aufgrund des schwarzen Hintergrundes, der an den freien Stellen sichtbar bleibt.

```
def drawField(self):
    for y in range(self.fieldsY):
        for x in range(self.fieldsX):
            position    = (self.margin+x*self.fieldWidth,
                           self.margin+y*self.fieldHeight)
            size        = (self.fieldWidth-self.seperator,
                           self.fieldHeight-self.seperator)
            if (y*3+x) % 2 == 0:
                color = 0, 0, 140
            else:
                color = 0, 0, 220
            self.drawRect(position, color, size)
            if (self.field[y][x] != 0):
                self.drawPlayerSign(position, self.field[y][x])yy
```

Listing 6.10 Zeichnen des Spielfeldes

Zum Zeichnen werden hier zwei verschachtelte Schleifen durchlaufen, wobei die Spalten und Zeilen des Spielfeldes durch die Variablen `x` und `y` beschrieben werden. Der Wechsel der Feldfarbe erfolgt durch die Überprüfung, ob sich die jeweilige Feldnummer durch 2 teilen lässt; die Stelle ist im Quelltext **fett** hervorgehoben. Die Feldnummer ergibt sich hierbei innerhalb der Bedingung durch einfache Linearisierung der Position im Feld, die durch das Wertepaar x und y angegeben ist. Linearisierung heißt, dass wir einfach die aktuelle Zeilennummer mit der Anzahl der Spalten multiplizieren und am Ende die aktuelle Spaltennummer addieren.

Die Methode `drawRect()` müssen Sie als Hilfsmethode wie folgt implementieren:

```
def drawRect(self, position, color, size=(10, 10)):
    x, y            = position
    width, height   = size
    rect            = (x, y, width, height)
    pygame.draw.rect(self.screen, color, rect)
```

Listing 6.11 Ein Rechteck zeichnen

Um das Feld nach jedem Zug entsprechend neu zu zeichnen, rufen Sie einfach die Methode `refresh()` auf. Der Auf Aufruf von `refresh()` erfolgt aus der Methode `start()`, wobei `refresh()` wie folgt definiert ist:

```
def refresh(self):
    self.screen.fill((0, 0, 0))
    self.drawField()
    pygame.display.flip()
```

Listing 6.12 Den Bildschirm neu zeichnen

Zunächst wird der Bildschirm komplett mit Schwarz gefüllt, was dem Löschen des Bildschirmes entspricht. Im Anschluss wird das Spielfeld gezeichnet. Alle Zeichenoperationen erfolgen dabei zunächst in einem nicht sichtbaren Hintergrundpuffer, der am Ende schlagartig in den Vordergrund geschaltet wird. Das Umschalten des Puffers erfolgt durch den Aufruf von `pygame.display.flip()`. Diese Vorgehensweise ist unter dem Begriff »Double Buffering« bekannt und dient bei schnellen grafischen Darstellungen der Vorbeugung von Flackern.

6.5 Die abstrakte Spiellogik

Die folgende unabhängige Spiellogik unserer Tic Tac Toe Implementierung ist so auch für andere Spiele weiterverwendbar, bei denen zwei Spieler abwechselnd ziehen.

In der Methode `start()` ist zusätzlich zum Zeichnen des Spielfeldes die komplette Ablauflogik definiert. Entsprechend prüft diese Methode den Spielzustand und reagiert je nachdem, welcher Zustand aktuell ist. Sie gibt also den Sieg eines Spielers genauso aus wie den Zug des Computers oder Menschen. Die Hauptablaufschleife ist sozusagen die `while`-Schleife, die so lange durchlaufen wird, wie die Variable `self.running` auf `True` gesetzt ist.

```
def start(self):
    while self.running:
#       das Spielfeld zeichnen
         self.refresh()
#       wer ist am Zug?
         if self.gameObject.getSide() == self.computerSide:
# der Computer ist am Zug
             self.makeCompMove()
         else:
#           der Mensch ist am Zug
             event = pygame.event.poll()
             if event.type == pygame.QUIT:
                 self.running = False
             elif event.type == pygame.MOUSEBUTTONDOWN:
                 self.makeHumanMove(event.pos, event.button)
```

Listing 6.13 Der Hauptablauf

Nach dem Zeichnen der aktuellen Spielsituation wird überprüft, ob der Computer am Zug ist. Falls dem so ist, wird der Computer durch den Aufruf der Methode `makeCompMove()` zur Berechnung des besten zur Verfügung stehenden Zuges veranlasst, den er sofort nach der Ermittlung auch ausführt. Die angesprochene Methode `makeCompMove()` gestaltet sich dabei folgendermaßen:

```
def makeCompMove(self):
    bestMove = self.compObject.getBestMove(9)
    self.gameObject.makeMove(bestMove)
    self.checkForResult()
```

Listing 6.14 Einen Computerzug ausführen

Das `compObject` repräsentiert den zu verwendenden Algorithmus. Das könnte zum Beispiel der Minimax-Algorithmus sein oder eine beliebige Optimierung auf Basis desselben. `compObject` ist in unserem Fall eine Instanz der Klasse `MiniMax`.

Die Methode `getBestMove()` startet den Suchalgorithmus mit vorgegebener maximaler Suchtiefe von `9`; somit ist sichergestellt, dass alle Berechnungen bis zum Spielende ausgeführt werden. Hier sehen Sie wieder, dass es aufgrund unserer sehr einfachen Bewertungsfunktion notwendig ist, den kompletten Baum zu

durchsuchen; nach maximal neun Zügen ist Tic Tac Toe beendet, da es lediglich neun Spielfelder gibt.

Falls der Mensch am Zug ist, so wird auf eine Mauseingabe gewartet. Dieser Mausklick wird dann in der Methode `makeHumanMove()` entsprechend der maximal neun zur Verfügung stehenden Felder in Spielfeldkoordinaten umgewandelt. Diese Umwandlung der Mauskoordinaten in Spielfeldkoordinaten heißt *Diskretisierung*. Wenn das Spielfeld beispielsweise eine Größe von 900 × 900 Pixel aufweist und 3 × 3 Felder groß sein soll, so ist ein Mausklick mit den Pixelkoordinaten (200, 150) nach der Diskretisierung das Feld (1, 1). Bei der Diskretisierung werden die nicht gezeichneten Zwischenräume dem jeweils näher anliegenden Feld zugewiesen.

Die Diskretisierung ist sehr einfach umzusetzen, denn es ist lediglich die relative Pixelkoordinate durch die Feldbreite beziehungsweise durch die Feldhöhe in Pixeln zu teilen. Das heißt, die x-Koordinate der Pixelposition wird durch die Feldbreite in Pixeln geteilt, die y-Koordinate der Pixelposition hingegen durch die Feldhöhe in Pixeln. Vor dem Teilen ist jedoch der obere beziehungsweise linke Rand neben dem Spielfeld von der Pixelkoordinate abzuziehen. Die entsprechende Zeile ist im Quelltext **fett** hervorgehoben.

```
def makeHumanMove(self, position, button):
    x, y    = position
    if x > self.margin and y > self.margin:
        x, y = (x-self.margin)/self.fieldWidth,
          (y-self.margin)/self.fieldHeight
```

Listing 6.15 Diskretisierung der Klickkoordinaten

Als Nächstes werden die resultierenden Spielfeldkoordinaten auf Gültigkeit überprüft. Mausklicks, die außerhalb des Spielfeldes liegen, werden einfach ignoriert.

Um einen Seitenwechsel – beziehungsweise Spielerwechsel – zu ermöglichen, habe ich innerhalb der Methode `makeHumanMove()` eine Sonderbehandlung für die rechte Maustaste eingebaut. Bei jedem Rechtsklick werden die Seiten getauscht. Entsprechend kann der Mensch jedes Mal, wenn er am Zug ist, mit der rechten Maustaste den Computer ziehen lassen und damit den Platz des Computers einnehmen. Nach jedem Zug wird der Spielzustand überprüft, um gegebenenfalls festzustellen, ob das Spiel für einen der beiden Spieler gewonnen ist oder in einem Unentschieden resultierte. Der Aufruf von `checkForResult()` sorgt für die Überprüfung des Spielzustandes. Falls die rechte Maustaste zum Seitenwechsel verwendet wurde, so wird diese Information entsprechend in der Variablen `computerSide` gespeichert.

```
        if len(self.field) > y and len(self.field[0]) > x:
            move = x, y
            if button == 3:
# hier werden die Seiten gewechselt und
# der aktuelle Zug wird dem Computer überlassen
                self.computerSide = self.gameObject.getSide()
                self.makeCompMove()
            elif move in self.gameObject.getMoves():
# der Spieler führt seinen Zug aus,
# wobei zunächst überprüft wird,
# ob der Zug überhaupt möglich ist
                self.gameObject.makeMove(move)
                self.checkForResult()
```

Listing 6.16 Überprüfung der Gültigkeit der Koordinaten

Sollte das Spiel beendet sein, so wird eine entsprechende Nachricht ausgegeben. Die Überprüfung des Spielzustandes und das Ausgeben der Nachricht erfolgen in der Methode `checkForResult()`, die wir uns im Folgenden näher anschauen.

```
def checkForResult(self):
    state = self.gameObject.checkGameState()
    if state != -1:
```

Listing 6.17 Überprüfung des Spielzustandes

Zunächst wird der aktuelle Spielzustand durch Aufruf von `checkGameState()` ermittelt. Innerhalb von `checkGameState()` wird zunächst die aktuelle Stellung durch Aufruf von `evaluate()` bewertet, wobei wie bereits erwähnt nur Endpositionen bewertet werden, und das Ergebnis in der Variablen `eval` gespeichert wird. Danach überprüft `getMoves()`, ob weitere Spielzüge möglich sind. Ist das nicht der Fall, so wird der Zustand des Spiels zunächst als unentschieden deklariert, was hier mit `0` definiert ist. Im Anschluss wird die Bewertung überprüft. Sollte sie von `0` abweichen, so hat einer der Spieler gewonnen – bei positiver Bewertung der erste Spieler, bei negativer Bewertung geht der Sieg an den zweiten Spieler.

```
    def checkGameState(self):
        # unbekannter Zustand als Initialwert
        state   = -1
        # aktuelle Postionsbewertung
        eval = self.evaluate()
        # sind noch Züge möglich?
        if not self.getMoves():
            # keine Züge mehr möglich
            state   = 0
        if eval > 0:
```

```
            # Spieler Nr. 1 hat gewonnen
            state   = 1
        if eval < 0:
            # Spieler Nr. 2 hat gewonnen
            state   = 2

        return state
```

Listing 6.18 Prüfung des Spielzustandes

Als Nächstes wird der Spielzustand dargestellt, wobei nicht sofort nach dem Spielende das Ergebnis erscheint, sondern zunächst eine Sekunde lang die Endposition sichtbar bleibt. Dann wird sichergestellt, dass nach der Anzeige des Spielzustandes und dem Neustarten des Spiels zunächst der menschliche Spieler am Zug ist. Dies geschieht durch die Zuweisung von `2` an die Variable `computerSide`.

```
# Darstellung der aktuellen Position
self.refresh()
time.sleep(1)
# nach Neustart soll der Computer wieder
# für den zweiten Spieler agieren
self.computerSide = 2
```

Listing 6.19 Endposition kurz beibehalten und Computer als zweiten Spieler festlegen

Da das Spiel einen Endzustand erreicht hat, wird es nun neu gestartet. Vorher jedoch wird das Ergebnis des alten Spiels für zwei Sekunden angezeigt. Hier könnten Sie bei Bedarf auch eine Abfrage einbauen, die explizit in Erfahrung bringt, ob ein neues Spiel gestartet werden soll. Bei aufwendigeren Spielen wäre an dieser Stelle auch eine Verewigung in einem Highscore denkbar; bei Tic Tac Toe ist diese Art der Würdigung der spielerischen Leistung aber vermutlich etwas übertrieben.

```
self.gameObject.newGame()
# den Sieger anzeigen
self.screen.fill((0, 0, 0))
if state == 1:
    msg = 'Player 1 won!'
elif state == 2:
    msg = 'Player 2 won!'
elif state == 0:
    msg = 'The Game is drawn'

font = pygame.font.Font(None, 80)
# den Text darstellen
text = font.render(msg, True,
```

```
 (255, 255, 255), (159, 182, 205))
# ein Rechteck erstellen
textRect = text.get_rect()
# das Rechteck zentrieren
textRect.centerx = self.screen.get_rect().centerx
textRect.centery = self.screen.get_rect().centery
# den Text im Rechteck darstellen
self.screen.blit(text, textRect)
pygame.display.flip()
time.sleep(2)
```

Listing 6.20 Ausgabe des Spielergebnisses

Übersichtlich zusammengefasst geschieht in diesem Code Folgendes:

- Spielzustand ermitteln
- Spielzustand darstellen
- Computer wieder als zweiten Spieler festlegen
- Spielzustand zurücksetzen
- neues Spiel starten

6.6 Die Suche im Spielbaum

Die eigentliche Berechnung der Züge erfolgt wie bereits erläutert über den Minimax-Algorithmus. Jedoch können Sie hier noch eine kleine Optimierung vornehmen, denn es ist nicht notwendig, zwei Funktionen zu verwenden, die sich jeweils abwechselnd aufrufen. Statt entweder zu minimieren oder zu maximieren, können wir auch kontinuierlich entweder maximieren oder minimieren und den entsprechenden Parameter je Aufruf negieren.

Das Maximieren des negierten Wertes entspricht ja dem Minimieren des nicht negierten Wertes. Der sich daraus ergebene Algorithmus ist im Grunde identisch zum Minimax-Algorithmus – er verwendet jedoch nur noch eine Funktion zur Optimierung und ist unter dem Namen *NegaMax* geläufig. Darauf aufbauend lässt sich jedoch noch einiges optimieren.

Einen Fortschritt erreichen wir, indem wir das Suchfenster so klein wie möglich gestalten und somit schneller Äste des Baumes abtrennen. Das Abtrennen vermeintlich schlechter Äste des Baumes wird *Alpha-Beta-Prunning* bezeichnet. Dabei werden jene Äste abgetrennt, welche Züge aufweisen, die in Stellungen mit schlechterer Bewertung resultieren, als der bis zu diesem Zeitpunkt bestbewer-

tete Zug. Die Obergrenze und Untergrenze des Suchfensters wird entsprechend der ausgewerteten Züge kontinuierlich verkleinert. Unter dem »Suchfenster« versteht man in dem Zusammenhang die Spannweite der Stellungsbewertung. Das Abtrennen vermeintlich schlechte Äste des Spielbaumes reduziert somit den Rechenaufwand erheblich. Voraussetzung für die Anwendung dieser Optimierung ist das Sortieren der bewerteten Züge. Es sollten zunächst die Züge ausgewertet werden, welche zur größtmöglichen Reduzierung des Suchfensters führen. Es sollten also einfach die laut Heuristik vermeintlich besten Züge zuerst bewertet werden.

Eine weitergehende Optimierung ist die Reduzierung des Suchfensters ohne auf die Auswertung der Züge zu warten. Hierbei wird von Anfang an ein sehr kleines Suchfenster festgelegt, wodurch alle stark von der momentanen Bewertung abweichenden Züge aussortiert werden. Der daraus resultierende Algorithmus nennt sich *NegaScout*, auch unter dem Namen *Principal Variation Search* bekannt, und läuft in den meisten Fällen performanter als der NegaMax-Algorithmus. Die Reduzierung des Suchfensters hat zur Folge, dass vermeintlich deutlich schlechtere Züge bereits sehr früh verworfen werden. Es kommt aber dennoch nicht zu einem Fehler, weil zu einem späteren Zeitpunkt erneut überprüft wird, ob es tatsächlich schlechtere Züge waren. Ein voreiliges Aussortieren eines vermeintlich schlechten Zuges kann also immer revidiert werden.

Sollte der Algorithmus einen Zug vorschnell verworfen haben, obwohl sich die Alternativen als nicht besser herausgestellt haben, so wird der Zug reevaluiert. Natürlich ergibt diese Form der Optimierung nur dann Sinn, wenn die Bewertungsfunktion beziehungsweise die zugrundeliegende Heuristik so genau ist, dass eine Neubewertung des Zuges nur sehr selten notwendig wird. Sollten Neubewertungen an der Tagesordnung sein, so ist die Performance des NegaScout-Algorithmus sogar deutlich schlechter als die von NegaMax.

Der Großteil des Algorithmus läuft in der Methode `explore()` innerhalb der Klasse `NegaScout` ab:

```
def explore(self, depth,
            alpha   =   -NegaScout.infinity,
            beta    =   +NegaScout.infinity):

    moves = self.gameObject.getMoves()
    if depth <= 0 or not moves:
        if self.gameObject.sideToPlay == 1:
            return +self.gameObject.evaluate()
        else:
            return -self.gameObject.evaluate()

    pvFound = False
```

```
    self.treeNodeCounter += 1
    for move in moves:
        self.gameObject.makeMove(move)
        if pvFound:
# suche mit kleinem Fenster starten
            value = -self.explore(depth-1, -alpha-1, -alpha)
            if value > alpha and value < beta:
# neue Hauptvariante gefunden,
# Suchlauf ist deshalb zu wiederholen
                value = -self.explore(depth-1, -beta, -alpha)
        else:
            value = -self.explore(depth-1, -beta, -alpha)
        self.gameObject.undoMove(move)

        if value >= beta:
# beta-">cut off"<
            return beta

        if value > alpha:
            alpha = value
            pvFound = True

    return alpha
```

Listing 6.21 Die Tiefensuche mittels »NegaScout«

Die Methode zählt unter anderem auch die besuchten Knoten des Spielbaumes in der Variable `treeNodeCounter` mit. Die Variablen `alpha` und `beta` stellen die Fenstergrenzen dar. Die Algorithmen haben sich wie bereits erwähnt schrittweise weiterentwickelt, wobei es vom MiniMax-Algorithmus über den NegaMax-Algorithmus zum *Alpha-Beta-Pruning* kam. Das heißt nichts weiter als das Beschneiden des Baumes anhand von bereits bekannten Ober- und Untergrenzen für den zu erreichenden Positionswert. Die beiden konkurrierenden Spieler optimieren gegeneinander, weshalb es zu zwei Werten kommt. Das sich durch die Grenzwerte definierende Fenster sorgt für die Vorauswahl der zu beurteilenden Züge.

NegaScout stellt jedoch sogar eine Verbesserung gegenüber diesem normalen Alpha-Beta-Pruning dar, indem er das Fenster fix auf Breite 1 festlegt. Diese Festlegung schränkt die möglichen Züge noch stärker ein und führt zu mehr abgespalteten Ästen im Baum, die nicht mehr berechnet werden müssen. Sie schadet jedoch nicht, solange es Züge gibt, die trotz dieser extremen Fensterverkleinerung innerhalb des Suchfensters liegen. Falls kein Zug mehr innerhalb der Grenzen liegt, so wird das Fenster wieder vergrößert.

Um den größtmöglichen Vorteil aus dem frühen Abtrennen vermeintlich schlechter Äste im Baum zu ziehen, müssen die in der Stellung möglichen Züge nach ihrer Bewertung sortiert werden. Diese Sortierung erfolgt innerhalb der Methode `getEvaluatedMoves()`:

```
def getEvaluatedMoves(self, depth):
    possibleMoves   = self.gameObject.getMoves()
    evaluatedMoves  = []
    for move in possibleMoves:
        self.gameObject.makeMove(move)
        self.treeNodeCounter = 0
        evaluation = self.explore(depth-1)
        self.gameObject.undoMove(move)
        evaluatedMoves.append((evaluation, move,
          self.treeNodeCounter))

    evaluatedMoves.sort()
    return evaluatedMoves
```

Listing 6.22 Die Zugbeurteilung

Da die Klasse `NegaScout` als `compObject` dienen soll, definiert sie auch die Methode `getBestMoves()`, die wie folgt aussieht:

```
def getBestMove(self, depth):
    moves = self.getEvaluatedMoves(depth)
    print '\nEvaluated Moves:'
    for move in moves:
        print move
    print '----------------'
    bestMove = moves[0][1]
    print 'best move = ', bestMove
    return bestMove
```

Listing 6.23 Den besten Zug ermitteln

Zur Veranschaulichung des Verfahrens gibt der Computer mit jeder Berechnung auf der Konsole die entsprechenden Züge inklusive der jeweiligen Bewertung aus.

Zusätzlich zu den genannten Methoden wird der Klasse `NegaScout` ein entsprechendes Spielobjekt im Konstruktor übergeben:

```
class NegaScout(object):
#  die Bewertung darf nie größer als
#  der Vorgabewert für Unendlich werden
    infinity    = 200
```

```
    def __init__(self, gameObject):
        self.gameObject = gameObject
```

Listing 6.24 Die Klasse »NegaScout«

Bis jetzt ist alles noch nicht konkret – die eigentlichen Regeln folgen im nächsten Abschnitt.

6.7 Konkrete Spiellogik für Tic Tac Toe

Die für Tic Tac Toe geltenden Regeln werden durch die Positionsbewertung und Zugerzeugung innerhalb der Klasse `TicTacToe` festgelegt. Die Klasse nutzt die bereits definierten Klassen aus den vorherigen Abschnitten und wird so initialisiert:

```
from playingfield import PlayingField
from negamax import NegaMax

class TicTacToe(object):
    def __init__(self):
        self.field = [[0, 0, 0],
                      [0, 0, 0],
                      [0, 0, 0]]
        self.newGame()
```

Listing 6.25 Die Klasse »TicTacToe«

Das Spielfeld wird durch eine verschachtelte Liste dargestellt. Mit Hilfe der Methode `newGame()` sorgen Sie für eine Reinitialisierung des Spielzustandes:

```
def newGame(self):
    self.player = 1
# das Spielfeld muss hier je Feld
# zurückgesetzt werden, da sonst eine
# neue Instanz erzeugt wird,
# die nicht referenziert ist
    for y in range(3):
        for x in range(3):
            self.field[y][x] = 0
```

Listing 6.26 Ein neues Spiel starten

Anstelle von verschachtelten Listen könnten Sie auch NumPy-Arrays verwenden, die deutlich schneller verarbeitet werden als Listen. NumPy ist die Basis für SciPy und steht kostenlos für Python zur Verfügung. SciPy ist eine Bibliothek, die zur Unterstützung von Scientific Computing entwickelt wurde.

Wie im Kommentar zur Methode geschildert, ist das Spielfeld in der Klasse `PlayingField` referenziert. Somit ist der Zustand des Spielfeldes nicht per Parameter zu übergeben, da die Klasse zur Darstellung direkten Zugriff auf das Spielfeld hat.

Die Bewertung der aktuellen Spielsituation erfolgt durch die Ihnen schon bekannte Methode `evaluate()`. Diese ist, wie bereits anfangs des Kapitels erläutert, sehr einfach gehalten und bewertet nur die Endpositionen, weshalb immer der komplette Baum durchsucht werden muss. Aus dem gleichem Grund wird sich durch Anwendung von NegaScout kein großer Vorteil bei der Suche im Baum ergeben.

```
def evaluate(self):
# kleine Verbesserung um den schnellsten
# Siegweg zu wählen werden die Züge gezählt
    moveCount = 0
    for y in range(3):
        for x in range(3):
            if self.field[y][x] == 0:
                moveCount += 1

    for i in range(3):
        if self.field[0][i] == self.field[1][i] == self.field[2][i]:
            if self.field[0][i] == 1:
                return + 10 + moveCount
            if self.field[0][i] == 2:
                return - 10 - moveCount

        if self.field[i][0] == self.field[i][1] == self.field[i][2]:
            if self.field[i][0] == 1:
                return + 10 + moveCount
            if self.field[i][0] == 2:
                return - 10 - moveCount

    if  (self.field[0][0] == self.field[1][1] == self.field[2][2]) or
     (self.field[0][2] == self.field[1][1] == self.field[2][0]):
        if self.field[1][1] == 1:
            return + 10 + moveCount
        if self.field[1][1] == 2:
            return - 10 - moveCount

    return 0
```

Listing 6.27 Die Stellungsbewertung

Die Bewertung erfolgt durch Überprüfung aller Horizontalen, Vertikalen und Diagonalen. Sollte es bei der Überprüfung irgendwo drei Zeichen des gleichen Spielers in Reihe geben, so wird die Bewertung zu dessen Gunsten ausfallen, denn

er hat somit gewonnen. Um sicherzustellen, dass der Computer den kürzesten Siegweg wählt, wird zusätzlich die Anzahl der bis zur Endposition ausgeführten Züge abgezogen. Entsprechend sind gewonnene Stellungen, die später erreicht werden, geringer bewertet.

6.8 So weit, so gut – was geht da noch?

Mit den in diesem Kapitel erarbeiteten Algorithmen können Sie nun auch leicht andere Nullsummenspiele wie »Othello«, »Vier gewinnt« oder »Schach« implementieren. Sie haben die freie Wahl: Alles was dazu nötig ist, ist die Ermittlung der möglichen Züge und die Bewertung einer Stellung. Allerdings erhöht sich der Schwierigkeitslevel dadurch, dass Sie für die komplexeren Spiele eine Heuristik – also eine prognostizierende Bewertung – implementieren müssen, wenn Sie einen vernünftigen Gegner schaffen wollen. Der einfachste nächste Schritt bestünde in der Implementierung von »Vier Gewinnt«. »Othello«, auch unter dem Namen »Reversi« bekannt, ist schon etwas schwieriger umzusetzen, aber liegt immer noch leicht im Bereich des Möglichen. Die größte Herausforderung ist sicherlich die Entwicklung einer guten »Schachsoftware«. Falls Ihnen diese Spiele allerdings gar nicht gefallen, so sind auch Klassiker wie »Dame« und »Mühle« durchaus denkbar.

»Sucht nur die Menschen zu verwirren,
Sie zu befriedigen, ist schwer.«

Johann Wolfgang von Goethe

7 Ich verstehe nur Bahnhof – der Goethe-Generator

In diesem Kapitel lernen Sie ein mächtiges Verfahren zum Generieren von Text kennen. Das Erstaunliche daran ist, dass der generierte Text gut lesbar und auf den ersten Blick nicht sinnlos erscheinen, sondern sogar nach Goethe klingen wird. Goethe?! Ja, richtig, wir schreiben eine Software, die uns einen Text »zaubert«, der sich stilistisch sehr an Goethe anlehnt, um nicht zu sagen sogar stark »abkupfert«. Um das zu erreichen, werden wir einfache statistische Verfahren verwenden, die sich aufgrund der enormen Redundanz der Sprache relativ leicht anwenden lassen. Als Grundlage für die Generierung werden uns einige der bekanntesten Werke von Johann Wolfgang von Goethe dienen, denn schließlich muss unser Generator den Stil erst einmal erlernen.

7.1 Was soll das Ganze?

Zunächst einmal stellt sich hier natürlich die Frage, warum man überhaupt Texte generieren sollte. Die wohl naheliegendste Antwort wäre sicherlich, dass es im Prinzip gar keinen Sinn ergibt, Texte zu generieren, die nicht wirklich etwas aussagen. Einige Politiker mögen hier widersprechen, aber diese Klientel gehört sicher nicht zu den Hauptinteressenten dieses Buches. Dennoch bleibt ein Kapitel ohne echte Motivation nicht wirklich reizvoll, weshalb mir doch noch ein paar gute Gründe für einen Textgenerator eingefallen sind:

- Aufsätze, Reports und Facharbeiten auf die Schnelle
- Suchmaschinenoptimierung durch Textgenerierung für den Crawler
- das Wort zum Sonntag für die eigene Homepage
- automatisierte E-Mail-Antworten für unliebsame Absender
- einfach nur zum Spaß

Sicher fällt Ihnen aber noch eine ganze Menge anderer sehr sinnvoller Einsatzzwecke ein. Ein sehr schönes Beispiel für die Möglichkeiten, die ein einfacher Textgenerator bietet, ist der »Scientific Paper Generator«. Bevor Sie jetzt mit diesem Kapitel fortfahren, sollten Sie sich ihn auf jeden Fall einmal anschauen. Sie finden ihn unter: *http://pdos.csail.mit.edu/scigen/*. Die generierten Texte dieser Software sind sogar so gut, dass sie bereits als Paper bei Konferenzen eingereicht und akzeptiert wurden. Sie durften sogar vorgetragen werden – also wenn das keine Motivation ist!

7.2 Was steht dahinter?

Das Prinzip hinter dem in diesem Kapitel entwickelten Textgenerator basiert auf der Redundanz der Sprache und der statistischen Abhängigkeit von Wörtern untereinander. Bereits Buchstaben haben eine unterschiedliche Auftretenswahrscheinlichkeit in Abhängigkeit von dem jeweils vorhergehenden Buchstaben. Diese Abhängigkeiten werden umso stärker, je mehr Vorgänger berücksichtigt werden. Entsprechend ergibt sich schon bei zwei vorhergehenden Buchstaben eine gute Abhängigkeit zum Nachfolger.

Analysiert man nun einen Text auf diese Abhängigkeiten zwischen den Buchstaben oder sogar auf die Abhängigkeiten zwischen den Wörtern, so erhält man eine Tabelle mit den Auftretenswahrscheinlichkeiten der Buchstaben oder Wörter in Abhängigkeit von den Vorgängern. Nutzt man die in der Tabelle ermittelten Verteilungen zur Erzeugung eines Textes anhand eines Zufallsgenerators, so erhält man einen dem analysierten Text stilistisch ähnlichen Text, der allerdings oft jedweder Bedeutung entbehrt. Dennoch sind die so generierten Texte gut lesbar und oft nicht auf den ersten Blick als generierte Texte zu erkennen.

7.3 Woher nehmen, wenn nicht stehlen?

Grundlage für die Generierung von Texten ist, wie bereits angeschnitten, eine statistische Analyse bereits vorhandener Texte. Da stellt sich natürlich zunächst einmal die Frage, woher wir die Texte zur Erstellung der Statistik bekommen. Für unseren Goethe-Generator benötigen wir natürlich auch Goethe-Texte, die zum Glück keinem Copyright mehr unterliegen und außerdem in digitalisierter Form vorliegen. Die digitalisierte Form haben wir dem Projekt Gutenberg zu verdanken, das Sie unter *http://www.gutenberg.org* finden. Dort können Sie viele bekannte Werke der Weltliteratur wiederfinden und für eigene Zwecke verwenden, solange Sie die Lizenzbedingungen entsprechend wahren.

Leider sind in jedem digitalisierten Text aus dem Projekt Gutenberg mehrfach die Lizenzbedingungen zu Beginn eingefügt, weshalb wir die Daten für die eigentliche Statistikerhebung vorbereiten müssen. Unser Ziel ist ja, Goethe-Texte statistisch zu analysieren, nicht irgendwelche Lizenzabkommen. Deshalb finden Sie auf der Buch-CD eine Auswahl der für die Statistik heranzuziehenden Werke in einer einzelnen Datei gebündelt. Diese Datei wurde um alle Lizenzinhalte reduziert, so dass sie nicht mehr die Statistik beeinflussen. Die für die Statistik relevante Datei habe ich *allGoethe.txt* genannt, um sie von den anderen abzuheben und dank alphabetischer Sortierung gleich an erster Stelle erscheinen zu lassen.

Falls Sie sich jetzt vielleicht schon fragen, was in der ominösen *allGoethe.txt* Einzug gehalten hat, so schauen Sie einfach in der folgenden Auflistung nach.

- »Wilhelm Meisters Lehrjahre« 1 bis 8
- »Wilhelm Meisters Wanderjahre« 1 bis 3
- »Faust« 1 und 2
- »Italienische Reise« 1 und 2
- »Iphigenie auf Tauris«
- »Herman und Dorothea«
- »Goetz von Berlichingen«
- »Die Wahlverwandtschaften«
- »Die natürliche Tochter«
- »Die Mitschuldigen«
- »Die Leiden des Jungen Werthers« 1 und 2
- »Die Geschwister«
- »Die Aufgeregten«

Die nächste mögliche Frage wäre sicherlich, was bei einer so bunten Mischung von Texten so alles zustande kommt. Ein paar nette Beispiele:

- »Der Alten fehlte es niemals, aber einer solchen Repräsentation müssen Männer sein, daher entsteht die Notwendigkeit, sie anzuziehen, sie festzuhalten.«
- »Der Alten fehlte es niemals, aber einer ihrer ältesten Diener hatte so gar kein Verhältnis zu fragen.«
- »Doch schwur ich mir, was du so davon; denn freilich mag ich am bequemsten, am liebsten.«

- »Mann von Verstande, der nicht merkwürdig war.«
- »Es ist eine falsche Nachgiebigkeit gegen die Menge, wenn man ihnen die Empfindungen erregt, die sie haben sollen.«
- »Vorstellung, die mich quält, so angenehm, so liebenswürdig, daß ich gern wiederkam.«

Falls Sie nun neugierig sind, wie die Passagen aussehen, die den Generator hierbei inspiriert haben, dann schafft die folgende Auflistung Abhilfe, die die einleitende Passage mit der Fortsetzung im Original enthält:

- »Der Alten fehlte es nicht an Gegenvorstellungen und Gründen; doch da sie in fernerem Wortwechsel heftig und bitter ward, sprang Mariane auf sie los und faßte sie bei der Brust.«
- »Doch schwur ich mir, wenn ich nur einmal aus dieser Verlegenheit gerettet wäre, mich nie, als mit der größten Überlegung, an die Vorstellung eines Stücks zu wagen.«
- »Er ist ein Mann von Verstande, aber von ganz gemeinem Verstande; sein Umgang unterhält mich nicht mehr, als wenn ich ein wohl geschriebenes Buch lese.«
- »Es ist eine entsetzliche Sekkatur, andere toll zu sehen, wenn man nicht selbst angesteckt ist.«
- »Dabei ist das Bild, die Vorstellung, die mich quält, so angenehm, so liebenswürdig, daß ich gern dabei verweile.«

Hin und wieder lesen sich die generierten Texte auf Deutsch ein wenig holprig, was mit den durch den Generator eingeführten Grammatikfehlern zusammenhängt. Im Englischen ist bezüglich der Generierung von Texten diesbezüglich mit weniger Fehlern zu rechnen, weshalb Sie das Ganze ruhig auch einmal mit Shakespeare ausprobieren sollten.

7.4 Häufigkeitsverteilung von Wörtern

Wir beginnen zunächst wieder ganz einfach und ermitteln nur die Häufigkeitsverteilung der einzelnen Wörter, die in einem Text auftreten. Um dies zu bewerkstelligen, implementieren wir die folgende Funktion, die die Anzahl des Auftretens der einzelnen Wörter im Text zählt:

```
def countWords1(fileName):
    mydict  =
    fobj    =   open(fileName, "r")

    for line in fobj:
        zuordnung = line.split()
        for word in zuordnung:
                if mydict.has_key(word) == 1:
                    mydict[word]=mydict.get(word)+1
                else:
                    mydict[word]=1

    fobj.close()

    wordList = mydict.items()
    wordList = sorted(wordList, key=operator.itemgetter(1),
     reverse=True)

    for word in wordList:
        anzahl, wort = word
        print "%s ->%s" % (wort, anzahl)
```

Listing 7.1 Wörter zählen

Den Quelltext zum Beispiel finden Sie wie üblich auf der beiliegenden CD-Rom unter *Kapitel07/wordcount.py*. **[●]**

Die Funktion öffnet eine Datei und durchschreitet sie dann Zeile für Zeile, um die in den einzelnen Zeilen auftretenden Wörter in ein Dictionary einzutragen. Dabei wird im Dictionary die Anzahl des Auftretens des jeweiligen Wortes hinterlegt. Nach Erstellen des Dictionarys wird die Liste noch nach Auftretenshäufigkeit sortiert und danach ausgegeben.

Ein erster sehr primitiver Ansatz zur Generierung wäre nun diese Liste. Wir könnten entsprechend der Auftretenswahrscheinlichkeit der Wörter diese wieder in den Text einfließen lassen und so einen neuen Text generieren. Dieser Text wäre allerdings nicht schwer als generierter Text zu erkennen – er wäre nicht einmal gut lesbar. Um es jedoch nicht bei Mutmaßungen zu belassen, folgt nun ein kleines Testprogramm, das anhand der Anzahl des Auftretens einzelner Wörter einen Text generiert:

```
def generateTextByWordCount(wordList):
    overallCount = 0
    for entry in wordList:
        overallCount += entry[1]
```

```
    for x in range(0, 100):
        choice = random.randint(0, overallCount)
        currentNumber   = 0
        selectedWord    = None
        for entry in wordList:
            word, count = entry
            currentNumber += count
            if currentNumber >= choice:
                selectedWord = word
                break

        print word,
```

Listing 7.2 Text anhand der Worthäufigkeit generieren

Die Methode `generateTextByWordCount()` ermittelt zunächst die Summe der Anzahl aller Wörter, um dann entsprechend der Häufigkeitsverteilung einzelne Wörter auszuwählen. Die Wortliste ist sortiert und beginnt mit den am häufigsten vorkommenden Wörtern. Es wird einfach eine Zufallszahl innerhalb der Gesamtanzahl der Wörter ausgewählt, die dann als Zielwert fungiert. Die Anzahl des Auftretens der jeweiligen Wörter wird sequentiell addiert, bis die Zufallszahl erreicht ist. Das dann aktuelle Wort wird als Nächstes ausgegeben. Das Verfahren arbeitet analog zu dem im Kapitel 8 über genetische Algorithmen. Es handelt sich auch hierbei im Prinzip um eine sogenannte »Roulette-Wheel-Selection«. Die häufiger auftretenden Wörter werden entsprechend durch größere Zahlenbereiche abgedeckt, was zu einer Auswahl entsprechend der Häufigkeitsverteilung der Wörter führt.

7.5 Texten mit Markov

Um einen besseren Text generieren zu können als mit der im vorherigen Abschnitt beschriebenen Methode, müssen wir die Auftretenswahrscheinlichkeit in Abhängigkeit von den Vorgängern der Wörter ermitteln. Wie bereits erläutert, ließe sich das Verfahren auch über Buchstabenabhängigkeiten bewerkstelligen – anhand von Wörtern ist es jedoch deutlich besser. Wenn jeweils nur der vorherige Zustand beziehungsweise für unseren Anwendungsfall die vorherigen Wörter für den Folgezustand von Bedeutung sind, dann spricht man in dem Zusammenhang auch von *Markov-Ketten*. Etwas wissenschaftlicher formuliert kann dieser Zusammenhang wie folgt beschrieben werden:

»Wenn die Wahrscheinlichkeit eines Zustands nur direkt vom vorangegangenen Zustand abhängt, so spricht man von Gedächtnislosigkeit oder auch von der sogenannten »Markov-Eigenschaft«.«

Je mehr Vorgänger berücksichtigt werden, desto besser liest sich der aus der Generierung erzeugte Text. Wenn jedoch zu viele Vorgänger einbezogen werden, so werden große Passagen des Textes einfach komplett übernommen. Um diesen Zusammenhang deutlich zu machen, habe ich mehrere Versionen für die Textgenerierung erstellt, die Sie entsprechend einzeln ausprobieren können.

Zunächst betrachten wir eine Version, die jeweils die beiden letzten Wörter berücksichtigt, also die beiden Vorgänger des aktuell zu wählenden Wortes. Die Dateibezeichnung der Quelltexte der entsprechenden Versionen besitzt als Postfix eine Zahl, die die Anzahl der zu berücksichtigenden Vorgänger angibt. Entsprechend beginnen wir mit der Version, die in der Datei *markov2.py* gespeichert ist und somit zwei Vorfahren für die Generierung berücksichtigt.

Den Quelltext zum Beispiel finden Sie wie üblich auf der beiliegenden CD unter **[◉]** *Kapitel07/markov2.py*.

Für die Generierung habe ich die Klasse `TextGenerator` definiert, die zunächst bei der Instantiierung den Text der übergebenen Datei aufbereitet und die enthaltenen Wörter in eine Liste speichert:

```
class TextGenerator(object):
    ''' Text-Generierung mit Markov-Ketten '''

    def __init__(self, file):
        self.cache      =
        self.words      = self.loadWords(file)
        self.startWords = filter(lambda x: x[0].isupper(),
          self.words)
        self.word_size  = len(self.words)
        self.createStatistics()
```

Listing 7.3 Die Klasse »TextGenerator«

Der Aufruf der Methode `loadWords()` erstellt die Liste der im Text enthaltenen Wörter. Als Nächstes werden die Wörter, die mit einem Großbuchstaben anfangen, in der Liste `startWords` gespeichert. Diese Liste verwenden wir später, um zufällig ein großgeschriebenes Wort als Text und Satzanfang auszuwählen. Da im noch folgenden Quelltext häufig die Länge der Wortliste benötigt wird, wird diese in der Variablen `self.word_size` hinterlegt. Der Aufruf von `createStatistics()` legt weitere Listen an, die jeweils aufeinanderfolgende Wörter enthalten. Diese

Statistiken werden wir später einfach für die Generierung wieder heranziehen, um mögliche Wortkombinationen auszuwählen.

Kommen wir zunächst zur Methode `loadWords()`.

```
def loadWords(self, file):
    file.seek(0)
    text    = file.read()
    file.close()
    text    = text.replace('\n', ' ')
    words   = text.split(' ')
    words   = filter(lambda x: x not in "", words)
    return words
```

Listing 7.4 Wörter laden

Um sicherzugehen, dass wir die gesamte Datei auslesen, wird zunächst an den Dateianfang gesprungen. Dieser Schritt ist natürlich nicht unbedingt notwendig, wenn immer sichergestellt ist, dass die übergebene Datei gerade erst geöffnet wurde. Im nächsten Schritt wird die komplette Datei als String eingelesen und über die Variable `text` referenziert. Das Dateiobjekt kann somit direkt nach dem Einlesen wieder geschlossen werden.

Auf den eingelesen String wird jetzt ein `replace()` aufgerufen, um alle Zeilenumbrüche in Leerzeichen umzuwandeln. Danach werden die Wörter des Textes anhand der sie trennenden Leerzeichen ermittelt und in der Liste `words` gespeichert. Viele aufeinanderfolgende Leerzeichen oder Zeilenumbrüche führen dabei zu »leeren« Listeneinträgen, die durch den Aufruf von `filter(lambda x: x not in "": words)` entfernt werden.

Die Funktion `filter()` ist Ausdruck der funktionalen Programmierung mittels Python und erzeugt eine neue Liste, die nur die Elemente enthält, die für die übergebene Funktion `True` zurückliefern. Entsprechend wird für jedes Element der Liste `words`, die als zweiter Parameter übergeben wurde, die anonyme Funktion aufgerufen, um zu überprüfen, ob die Bedingung `x not in ""` wahr oder falsch ist.

Jetzt kommen wir mit der Methode `createStatistics()` zum interessantesten Teil des Generators.

```
def createStatistics(self):
    for w1, w2, w3 in self.triples():
        key = (w1, w2)
        if key in self.cache:
            self.cache[key].append(w3)
```

```
        else:
            self.cache[key] = [w3]
```

Listing 7.5 Die Statistik erstellen

Um die statistische Verteilung der Dreierpaare zu erfassen, wird jedes Dreierpaar von Wörtern in einem Dictionary hinterlegt. Die ersten beiden Wörter dienen dabei als Index, das dritte Wort als hinterlegter Wert. Es werden also alle Zweierpaare von Wörtern zu jedem auftretenden Wort gespeichert, die irgendwo im Text einmal vor dem Wort auftauchen. Für den Fall, dass das gleiche Zweierpaar von Wörtern mehrmals auftaucht, erfolgen entsprechend auch mehrere Einträge in das Dictionary für das betreffende Nachfolgewort.

Die Bildung der Dreiertupel erfolgt durch die Methode `triples()`:

```
def triplets(self):
    """ Erzeugt Wort-Paare zu einem vorgegebenen Text,
        so dass für den String "Hallo, wie geht es dir?"
        Paare wie folgt erzeugt werden:
        (Hallo, wie, geht), (wie, geht, es), ...
    """

    # stellt sicher, dass noch mindestens drei
    # Wörter folgen, bevor das Textende erreicht ist
    if len(self.words) < 3:
        return

    # gibt für den gesamten Text die jeweils nächste
    # Dreierkombination von benachbarten Wörtern zurück
    for i in range(len(self.words) - 2):
        yield (self.words[i], self.words[i+1], self.words[i+2])
```

Listing 7.6 Dreierpaar von Wörtern zurückgeben

Die Kommentare im Quelltext beschreiben die Methode schon sehr gut; es werden jeweils Wortpaare gebildet, wobei einfach immer die nächsten drei benachbarten Wörter verwendet werden. Allerdings wird das »Fenster« für die Auswahl der nächsten drei Wörter jeweils nur um ein Wort verschoben. Entsprechend sind alle Wörter Bestandteil mehrerer Gruppen.

Um den eigentlichen Text zu generieren, benötigen wir nun noch die Methode `generate_markov_text()`, die sich aus dem Dictionary mit den Wortgruppen der vorherigen Wörter bedient. Klingt kompliziert – ist es aber nicht.

```
def generate_markov_text(self, size=25):
    seed = self.words.index(random.choice(self.startWords))
    seed_word, next_word = self.words[seed], self.words[seed+1]
    w1, w2 = seed_word, next_word
    gen_words = []
    for i in xrange(size):
        gen_words.append(w1)
        w1, w2 = w2, random.choice(self.cache[(w1, w2)])
    gen_words.append(w2)
    text = ' '.join(gen_words)
    return text[:text.rfind('.')+1]
```

Listing 7.7 Mit Markov texten

Der Parameter `size` gibt an, wie viele Wörter wir maximal generieren wollen, also aus wie vielen Wörtern der zu generierende Text maximal bestehen soll. In der Variablen `seed` wird ein Anfangswort festgelegt, das wir zufällig aus der Liste der großgeschriebenen Wörter auswählen. Normalerweise würde es natürlich reichen, wenn wir einfach das Wort wählen, aber in diesem Fall benötigen wir auch den Index des ausgewählten Wortes in der Liste aller Wörter. Den Index verwenden wir gleich darauf, um auch das darauffolgende Wort zu setzen. Bis jetzt haben wir lediglich einen Teil des Ausgangsmaterials zufällig ausgewählt und kopiert.

Was heißt nun »Ausgangsmaterial«? Ganz einfach: Der Textgenerator muss zunächst eine Datei mit genug Text einlesen, die uns die Wortgruppenbildung erlaubt. Für diesen Zweck benutzen wir die von mir zusammenkopierte Datei mit einigen bekannten Goethe-Werken und übergeben sie beim Start des Programms als Eingabe.

Zurück zur Initialisierung – nach dem Festlegen der beiden ersten Wörter werden alle weiteren Wörter zufällig aus unserer Statistik gewählt. Das Dictionary wurde hierfür extra so gewählt, dass als Zugriffsschlüssel ein Tupel bestehend aus zwei Worten übergeben werden muss. Als Rückgabe erhalten Sie eine Liste, die alle in der Datei auftauchenden nachfolgenden Wörter enthält. Aus dieser Liste wird irgendein Eintrag durch den Aufruf von `random.choice(self.cache[(w1, w2)]` ausgewählt und an den zu generierenden Text angehängt. Zum Schluss wird die Liste der Wörter, die unseren neuen Text darstellen, in einen durch Leerzeichen getrennten String umgewandelt. Der String wiederum wird durch den Aufruf `text[:text.rfind('.')+1]` nur bis zum letzten Punkt ausgegeben.

Am besten probieren Sie jetzt gleich einmal das Programm aus und versuchen, den generierten Text zu verstehen. Vermutlich gefällt Ihnen das Resultat noch nicht so gut, aber wenn Sie mehrere Durchläufe probieren, dann erhalten Sie

ganz sicher schon ein paar lustige Passagen. Testen Sie darüber ruhig auch die zwei anderen Versionen, die noch mehr Vorgängerwörter berücksichtigen. Sie werden feststellen, dass bei zu vielen Vorgängern der Text einfach nur zufällig herauskopiert wird. Diesem Phänomen können Sie durch sehr viel mehr Ausgangsmaterial entgegenwirken. Allerdings sind bereits Texte, bei denen für die Statistik die drei vorhergehenden Wörter berücksichtigt werden, sehr gut lesbar.

7.6 Was denn noch?

Googeln Sie doch einfach einmal nach Markov-Ketten. Sie werden überrascht sein, wie viele Anwendungsmöglichkeiten es gibt. Ein Bereich, in dem ebenfalls mit Markov-Ketten gearbeitet wird, ist die Spracherkennung. Dafür wird allerdings das Modell der *Hidden-Markov-Chains* verwendet, bei welchem die Zustände der Markov-Kette während der Ausführung verborgen sind. Nähere Informationen dazu finden Sie hierzu, wenn Sie im Literaturverzeichnis dem Verweis [Elliott 1995] folgen.

Der nächste einfache Schritt wäre die Verwendung einer anderen Datenbasis für die Erstellung der Statistik. Diesbezüglich hatte ich bereits die Verwendung von Shakespeare-Texten empfohlen, da sich Englisch etwas besser eignet als Deutsch. Falls Sie also noch keinen Shakespeare-Generator haben, so probieren Sie das doch gleich einmal aus. Sie werden überrascht sein – die Texte sehen dabei schon sehr viel korrekter aus als beim deutschen Äquivalent. Grundsätzlich sollte zunächst allerdings genug Material vorhanden sein, denn zu kurze Texte führen zu relativ schlechten Ergebnissen.

Markov-Ketten lassen sich aber auch in ganz anderen Kontexten einsetzen. So könnten Sie beispielsweise einen Computergegner mittels statistischer Analysen der menschlichen Spieler füttern. Ein denkbarer Anwendungsfall diesbezüglich sind zum Beispiel Beat-'em-up-Spiele, also Spiele wie »Tekken« oder »Street Fighter«, in denen sich zwei Spieler in einer straßenkampfähnlichen Situation bekämpfen. Der Computergegner könnte so die vom Menschen in der Regel verwendeten Angriffsmanöver auswerten und identisch selbst zum Einsatz bringen. Wobei hier natürlich eine ähnliche Varianz eintreten würde wie beim Textgenerator – es würden neue Kombinationen von bewerteten Abläufen entstehen.

Viel Spaß beim Probieren und finden neuer Anwendungsmöglichkeiten.

»Alles, was gegen die Natur ist,
hat auf die Dauer keinen Bestand.«

Charles Darwin

8 Evolution im Computer

Kennen Sie noch den »Lunar Lander«? Dieses Spiel habe ich noch zu Zeiten von Windows 3.11 gespielt, wobei es sicher schon sehr viel früher existierte. Die Spielregeln sind denkbar einfach: Alles, was zu tun ist, ist eine Landung auf dem Mond. Dabei muss der Spieler die Raumfähre mittels gezielter Raketenstöße langsam auf dem Mond absetzen. Eine nicht ganz triviale Aufgabe, wenn zusätzlich der Treibstoff begrenzt ist.

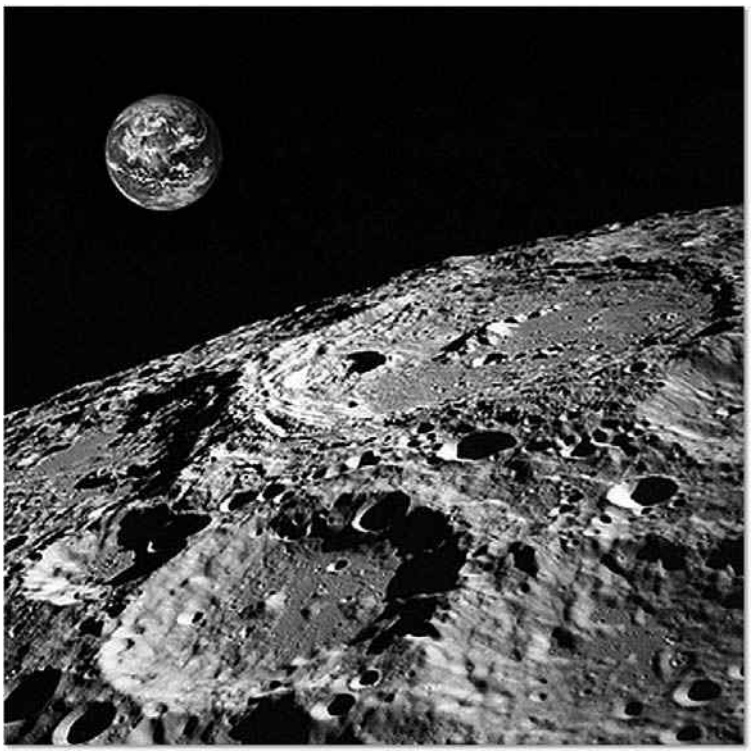


Abbildung 8.1 Der Mond – nicht gerade eine Landebahn (Quelle: Jim Mills, Fotolia.com)

Um dieses Spiel und diese Aufgabe wird sich dieses Kapitel drehen, wobei die Aufgabe am Ende nicht von Ihnen, sondern vom Computer gelöst werden wird. Ein so einfacher Job ist für den Computer an sich natürlich kein Problem und

wäre auch direkt lösbar. Aber um die Sache etwas spannender zu gestalten, werden wir für die Lösung genetische Algorithmen verwenden, die Sie später auch zur Lösung sehr komplexer Probleme einsetzen können.

Der Vorteil von genetischen Algorithmen oder allgemeiner von evolutionären Algorithmen ist die Anpassungsfähigkeit an sich verändernde Bedingungen. Zudem müssen Sie für die Lösung des Problems nur gewisse Rahmenbedingungen vorgeben, worauf wie von Zauberhand eine Lösung angenähert wird, ohne dass Sie eine exakte Lösungsstrategie vorgeben müssen.

Ausnahmsweise geht es in diesem Kapitel mal nicht um die Möglichkeiten von OpenGL, sondern viel mehr um künstliche Intelligenz. Ihr Computer wird in diesem Kapitel in die Lage versetzt werden, selbständig landen zu können – durchaus keine einfache Aufgabe.

Doch bevor wir zum eigentlichen Programm übergehen, beschäftigen wir uns zunächst einmal mit den biologischen Grundlagen von genetischen Algorithmen.

8.1 Das Gesetz des Stärkeren

Erinnern Sie sich noch an den Biologieunterricht? Als Stichwort wäre hier Charles Darwin ganz passend, der entgegen der Lehre der Kirche die Evolutionstheorie verkündete. Anfangs als Hypothese aufgestellt, hat sich seine Theorie inzwischen als treibende Kraft bei der Entwicklung allen Lebens herausgestellt.

Das Grundprinzip der Evolutionstheorie ist das Überleben des Stärkeren beziehungsweise des an die Umgebung besser angepassten Lebewesens. Dieses Prinzip erfüllt sich schon allein deshalb, weil alle benachteiligten Individuen mit der Zeit aufgrund ihres Nachteils sterben. Dabei wird keine Weiterentwicklung einzelner Lebewesen betrachtet, sondern die Weiterentwicklung über viele Generation hinweg. Manche Entwicklungen ergeben sich einfach durch Mutationen und sind nicht wirklich gewollt – stellen sich aber letzten Endes als entscheidender Vorteil heraus oder verschwinden genauso schnell, wie sie aufgetaucht sind.

Vorteilhafte Eigenschaften werden verstärkt, indem diese besser angepassten Lebewesen sich vermehren und in größerer Zahl überleben als die benachteiligten Lebewesen. Beim Gattungsverhalten ziehen erfolgreiche Individuen andere erfolgreiche Individuen stärker an, so dass sich die am besten entwickelten Organismen paaren und ihr Nachwuchs noch bessere Eigenschaften haben wird. Dieses Prinzip ist übrigens nicht nur im Reich der Tiere zu beobachten – nicht umsonst heißt es auch »Erfolg macht sexy«. Alles in allem eine sehr einleuchtende Theorie, die zu erstaunlichen Ergebnissen geführt hat.

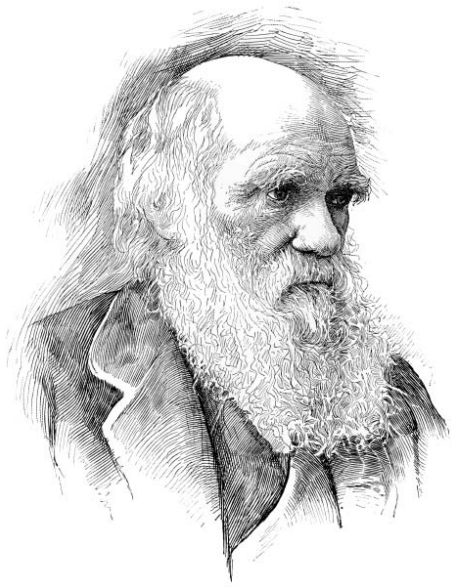


Abbildung 8.2 Charles Darwin – der Begründer der Evolutionstheorie
(Quelle: Steven Wynn, iStockphoto.com)

Allerdings dauert das alles auch sehr lange und kann sich nur über viele Tausende von Generationen entwickeln. Im Computer sieht die Sache aber ganz anders aus, denn hier sind ein paar Tausend Generationen durchaus in wenigen Sekunden durchlaufen; der Unterschied ist ungefähr so, als ob Sie selbst kochen oder etwas in der Mikrowelle aufwärmen – nur zeitlich noch viel krasser verkürzt.

Da dem Ganzen aber auch ein Hauch von Unzuverlässigkeit anhängt, werden evolutionäre Verfahren nicht überall eingesetzt. Das Problem bei diesen Verfahren ist, dass niemand genau sagen kann, wie das Ergebnis zustande kam. Die Ergebnisgüte lässt sich jedoch recht gut abschätzen. So kann man bereits vorher sagen, ob das Ergebnis eine gewisse Nähe zum Optimum erreichen wird. Wenn es zum Beispiel um eine Routenoptimierung ginge, so könnte man abschätzen, dass das Verfahren eine Route ermitteln wird, die maximal x Prozent vom Optimum entfernt ist.

Tatsächlich ist der Bereich der Logistik ein sehr gutes Beispiel, denn wenn man das Problem der Routenberechnung um ein Zeitfenster erweitert, so wird es zu komplex, um es in angebrachter Zeit direkt über voll deterministische Verfahren – also direkt und exakt – zu lösen. In diesem Zusammenhang spricht man auch von *nicht NP-vollständigen Problemen* – also Problemen, die nicht in Polynomialzeit gelöst werden können, und genau hier sind evolutionäre Algorithmen eine sehr gute Möglichkeit zur Annäherung an eine Lösung, die zwar nicht perfekt, aber allemal gut genug ist.

[+] Was ist Polynomialzeit?

Probleme, deren Rechenzeit mit der Problemgröße nicht stärker als mit einer Polynomfunktion wachsen, gelten als in Polynomialzeit lösbare Probleme. Hierbei wird davon ausgegangen, dass jene Probleme dieses Laufzeitverhalten auf einer deterministischen und sequentiellen Rechenmaschine zeigen.

Ein anderes Anwendungsszenario sind aktuelle Spiele, deren künstliche Intelligenz (KI) immer weitere Fortschritte macht. Vor dem Hype der evolutionären Algorithmen wurden für Spiele häufig sogenannte »Finite State Engines« – also rein zustandsbasierte Automaten, deren Verhalten nicht adaptiv ist – eingesetzt, so dass sie sich aufgrund fest definierter Zustände deterministisch verhielten. Durch dieses deterministische Verhalten war es jedoch notwendig, die entsprechenden Zustandsautomaten sehr genau auf Herz und Nieren zu überprüfen. Trotz ausgiebiger Tests kam es aber immer wieder zu bestimmten Abläufen, die der Spieler ausnutzen konnte, um den Computer zu besiegen, da der Computer aufgrund seiner deterministischen Steuerung permanent die gleichen Fehler beging. Mit diesem Problem haben die evolutionären Algorithmen gänzlich aufgeräumt, denn nun findet der Computer immer bessere Strategien, die sogar das Verhalten des menschlichen Spielers adaptieren. Gute Implementierungen sorgen für eine angemessenen Herausforderung des menschlichen Spielers – der Computer darf nicht zu stark, aber auch nicht zu schwach sein.

Solche Probleme lassen sich mit evolutionären Algorithmen auch in Verbindung mit neuronalen Netzen leicht vermeiden. Es geht heute nicht mehr darum, die Spiele-KI stark genug zu bekommen, sondern dank der genannten Verfahren besteht die Kunst darin, dass der Computergegner nicht schneller lernen darf als der Mensch und wir somit überhaupt noch eine Chance haben.

Die Kunst ist also, dass der Computer genau so viel stärker spielt, dass es für die Spieler einen guten Anreiz und die maximale Motivation bietet. Aber genau dieses Problem lässt sich wieder mit den genannten Verfahren lösen – im Prinzip lässt sich fast alles damit lösen. Die Evolution hat nicht umsonst funktioniert – das Grundprinzip ergibt einfach Sinn.

Ganz so einfach, wie das jetzt klingt, ist es jedoch nicht wirklich, denn für evolutionäre Algorithmen müssen Parameter bestimmt werden, die nicht immer sofort leicht ersichtlich sind. Hier helfen dann nur ausgiebiges Testen und ein sehr gutes Verständnis der Problemdomäne sowie der entscheidenden Einflussgrößen. Die exakten Möglichkeiten bei der Umsetzung der Mondlandung werden wir uns in Kürze noch detaillierter anschauen, jedoch bedarf es etwas Erfahrung, um ein Gefühl dafür zu entwickeln, welche Parameter und welche Verfahren für welche Problemdomänen die besten Ergebnisse liefern.

Für die Mondlandung benötigen wir neben der Biologie allerdings auch ein wenig Physik, welcher wir uns im folgenden Abschnitt widmen.

8.2 Ein bisschen Physik

Für eine Simulation einer Mondlandung benötigen wir ein bisschen Physik, wobei wir das Ganze natürlich stark vereinfacht simulieren. Ansonsten wäre die Landung nämlich wirklich keine triviale Aufgabe für einen Computer. Das Modell für die Mondlandung soll dementsprechend nur den freien Fall und die Schubwirkung der Raketen berücksichtigen, wobei wir beide Einflüsse ebenfalls stark vereinfacht betrachtet werden. Da der Mond keine Atmosphäre besitzt, ist auch nicht wirklich mit Reibungsverlusten zu rechnen. Die Vereinfachungen der Simulation sind aber dennoch relativ nah an der Realität – nur etwas ungenauer als die Berücksichtigung aller erdenklichen Faktoren.

Um die Fallgeschwindigkeit der Raumfähre zu jedem Zeitpunkt zu ermitteln, verwenden wir folgende Formel:

$$v(t) = -g * t + v_0$$

Dass sich die Gravitationskraft minimal verändert, je weiter die beteiligten Körper voneinander entfernt sind, vernachlässigen wir hier, da die resultierende Ungenauigkeit verschwindend gering ausfällt. Die Formel kennen Sie sicher, es wird lediglich die Anfangsgeschwindigkeit `v_0` zur sich über die Zeit höheren Fallgeschwindigkeit addiert. Die Fallgeschwindigkeit ergibt sich aus dem Produkt der vergangenen Zeit `t` und der Gravitationskonstante des Mondes `g`.

Um die aktuelle Höhe der Raumfähre zu ermitteln, nehmen wir folgende Formel zu Hilfe:

$$h(t) = -\frac{1}{2} * g * t^2 + v_0 * t + h_0$$

Die Schubkraft der Raketen lässt sich durch Reduzierung der Gravitationskonstante simulieren. Alternativ könnten wir sie auch separat berechnen und dann in das Endergebnis einfließen lassen. Die zuerst genannte Möglichkeit ist jedoch deutlich einfacher, weshalb wir sie vorziehen werden. Das erspart die nähere Betrachtung der Impulsgesetze, die die Kraftwirkung der Raketen in Abhängigkeit von der Masse und Geschwindigkeit des ausgestoßenen Gases betrachten.

8.3 Visualisierung des Landeanflugs

Kommen wir nun zur Darstellung unserer Raumfähre und der Landefläche auf dem Mond. Hierbei werden wir die Mondfläche nicht gekrümmt darstellen, was auch der Realität bezüglich der Größenverhältnisse zwischen Raumfähre und Mond am nächsten kommt. Da die Darstellung in diesem Kapitel eher nebensächlich ist, werden wir die Raumfähre als Rechteck zeichnen. Der Fokus soll hier auf den Algorithmen liegen – eine schickere Grafik wird später kein großes Hindernis darstellen.

Aus meiner Erfahrung heraus kann ich nur empfehlen, bei größeren Problemen nicht zu viel Wert auf unwichtige Details zu legen, denn das lenkt Sie vom Wesentlichen ab. Zwei passende Sprüche eines Kollegen dazu waren einmal: »Ich habe das Gefühl, es wird gerade an den goldenen Wasserhähnen gebaut, während der Rumpf des Schiffes noch nicht fertig ist«, oder kurz: »Wir brauchen keinen barocken Balkon.« Auch heute bleiben viele Softwareprojekt unvollendet auf der Strecke, da aufgrund zu vieler Wünsche die Komplexität oft nicht mehr kontrollierbar ist – solche Fehlentwicklungen werden oft auch sarkastisch mit »in Schönheit sterben« kommentiert. Also belassen wir es vorerst bei einer spartanischen Darstellung – um goldene Wasserhähne können Sie sich immer noch nach geglückter Landung kümmern.

[●] Den Quelltext zum Beispiel finden Sie wie üblich auf der beiliegenden CD unter *Kapitel08/LunarLander_v1.py*.

```
def resize((width, height)):
    if height == 0:
        height = 1
    glViewport(0, 0, width, height)
    glMatrixMode(GL_PROJECTION)
    glLoadIdentity()
    glOrtho(-10.0, fieldWidth, -10.0, fieldHeight, -6.0, 0.0)
    glMatrixMode(GL_MODELVIEW)
    glLoadIdentity()
```

Listing 8.1 Initialisierung des OpenGL-Fensters

Die Funktion `resize()` setzt den Viewport entsprechend der vorgesehenen Größe. Der Aufruf von `glOrtho()` sorgt für eine Parallelprojektion, wobei ausnahmsweise die y-Achse so definiert ist, dass die Null am unteren Ende des Fensters steht; also entgegen der für Computer üblichen Art, bei der der Punkt mit den Koordinaten (0, 0) in der Regel oben links im Fenster zu finden ist. Weitergehende Informationen zu OpenGL finden Sie im Anhang A, »OpenGL«.

Den Mond stellen wir der Einfachheit halber ebenfalls als schwebendes Rechteck dar. Hier wäre es natürlich genauso leicht möglich, dieses Rechteck bis zum Fensterrand auszudehnen. Stellen Sie sich einfach vor, dass der Mond eine Scheibe wäre und die Mondrakete doch sehr viel größer ist als üblich. Falls die Vorstellung schwer fällt, dann zeichnen Sie einfach das den Mond repräsentierende Rechteck entsprechend größer, so dass es bis zum Fensterrand reicht. Mit diesen Grafikprimitiven können Sie übrigens auch sehr leicht das klassische Ping-Pong-Spiel programmieren; hierfür fehlt nur noch ein zweiter Balken am oberen Fensterrand.

Für das Zeichnen von Mond und Raumschiff definieren wir eine Funktion, die wir `drawRect()` nennen:

```
def drawRect(x, y, width, height):
    glColor3f(1.0, 0.0, 0.0)
    glBegin(GL_QUADS)
    glVertex3f(x, y, 0.0)
    glVertex3f(x + width, y, 0.0)
    glVertex3f(x + width, y + height, 0.0)
    glVertex3f(x, y + height, 0.0)
    glEnd()
```

Listing 8.2 Funktion zum Zeichnen von Rechtecken

Innerhalb der Funktion `drawRect()` wird zunächst Rot als Zeichenfarbe festgelegt. Für das Zeichnen des Rechtecks verwenden wir das entsprechende Grafikprimitiv, das für Rechtecke `GL_QUADS` heißt. Innerhalb des Blocks verbinden wir daraufhin die vier definierten Punkte miteinander und füllen die resultierende Fläche mit der aktuellen Zeichenfarbe – also Rot.

```
def drawGround(x, y):
    width   = 600.0
    height  = 30.0
    drawRect(x, y, width, height)

def drawRocket(x, y):
    width   = 20.0
    height  = 30.0
    drawRect(x, y, width, height)
```

Listing 8.3 Das Zeichnen der Raktet und der Landeplattform

Für den Mond ändern sich die Parameter der Position nicht. Für das Raumschiff jedoch werden diese kontinuierlich anhand der Rechenergebnisse aktualisiert.

Im folgenden Abschnitt betrachten wir zunächst die einfachste Situation – den freien Fall des Raumschiffs. Diese Art der Landung ist auf lange Sicht natürlich nicht das Ziel, aber so können Sie die Berechnungen zunächst isoliert betrachten.

8.4 Der freie Fall

Für den freien Fall startet das Raumschiff in einer vorgegeben Höhe und in absoluter Ruhe in Bezug zum Mond. Diese absolute Ruhe ist natürlich nicht ansatzweise realistisch, wenn es um eine echte Mondlandung geht. Aus dieser Ausgangslage beschleunigen wir das Raumschiff nun aufgrund der Gravitationskraft des Mondes. Dabei ist die Gravitationskonstante des Mondes mit 1,635 m/s^2 definiert; dies entspricht in etwa einem Sechstel der Erdanziehungskraft.

Für die notwendigen Berechnungen werden wir die Geschwindigkeit und aktuelle Position des Raumschiffs zwischenspeichern, worauf sich alle weiteren Daten aus den Berechnungen ermitteln lassen. Die Berechnung der Positions- und Geschwindigkeitsveränderung erfolgt in der Funktion `calculateDelta()`:

```
def calculateDelta(gravity, velocity, height):
    t = 1.0/fps
    h = -0.5*gravity*t**2 + velocity*t + height
    v = -gravity + velocity

    return v, h
```

Listing 8.4 Berechnung der neuen Geschwindigkeit und Höhe

Die Position muss jeweils vor dem Zeichnen berechnet werden, weshalb wir für die verstrichene Zeit entsprechend eine Sekunde durch die Anzahl der Frames per Second teilen – abgekürzt durch `fps`. Die mit den Formeln berechneten Werte werden zurückgegeben und fließen so in die darauffolgenden Berechnungen ein. Die Formel für die Berechnung der Höhe `h` können Sie sich leicht durch Umstellen der bekannten Bewegungsgleichungen herleiten. Eine Kollisionsbehandlung bei Aufschlag auf der Mondoberfläche führen wir noch nicht durch, so dass das Raumschiff einfach durch die Mondoberfläche hindurchfällt.

Das Ganze sieht noch sehr einfach aus – man könnte fast meinen, die Abbildung entstamme dem Suprematismus. Wobei zwei geometrische Figuren auf einmal vermutlich schon zu viel des Guten wären – zumindest für Kenner dieser Kunstrichtung.

Abbildung 8.3 Suprematismus – »zwei rote Rechtecke auf schwarzem Grund«

Falls Ihnen Suprematismus nichts sagen sollte, dann googeln Sie einfach mal nach dem »schwarzen Quadrat auf weißem Grund« und eventuell auch nach dem »weißen Quadrat auf weißem Grund«. Eines der beiden Gemälde hat einen höheren Wert – mir fällt aber gerade nicht ein, welches. Momentan interessiert mich eigentlich auch viel eher, wie viel meine Schöpfung wert ist – oder ist die etwa wertlos?

Doch kommen wir von der Kunst zurück zum freien Fall. Die komplette Darstellung erfolgt nun in der Hauptroutine und sieht wie folgt aus:

```
def main():
    pygame.init()
    video_flags = OPENGL | HWSURFACE | DOUBLEBUF

    screenSize = (fieldWidth, fieldHeight)
    pygame.display.set_mode(screenSize, video_flags)
    resize(screenSize)

    init()
    gravity = 1.635
    velocity = 0
    height = 400
    while True:
        if not handleEvent(pygame.event.poll()):
```

```
            break

            velocity, height = calculateDelta(gravity,
             velocity, height)

            clearScreen()
            drawRocket(200, height)
            drawGround(100, 50)

            pygame.display.flip()

        pygame.time.delay(int(1000.0/fps))
```

Listing 8.5 Der Hauptablauf für die Mondlandung

Zunächst initialisieren wir `pygame` und setzen die entsprechenden Flags für hardwarebeschleunigtes OpenGL. Danach definieren wir die Gravitationskonstante `gravity` sowie die Anfangsgeschwindigkeit `velocity` und die Fallhöhe `height`. Wobei» Fallhöhe« hier nicht ganz korrekt ist, da wir die Mondoberfläche nicht am unteren Rand des Fensters darstellen, sondern einige Pixel weiter oben. Insofern stellt `height` die y-Koordinate des Raumschiffs innerhalb des Fensters dar.

In der `while`-Endlosschleife werden dann zunächst durch den Aufruf von `handleEvent()` die Benutzereingaben verarbeitet, wobei `handleEvent()` wie folgt definiert ist:

```
def handleEvent(event):
    if event.type == QUIT or
     (event.type == KEYDOWN and event.key == K_ESCAPE):
        return False
```

Listing 8.6 Ereignisbehandlung ohne Steuerung des Raumschiffs

Bis jetzt berücksichtigt die Eingabebehandlung nur das Schließen des Fensters, aber noch keine Steuerbefehle für das Raumschiff.

Zurück zur Hauptablaufschleife in der Funktion `main()`, in der wir wie bereits besprochen alle physikalischen Größen entsprechend initialisiert und dann innerhalb der Ereignisschleife jeweils neu berechnet haben. Die Gravitationskraft bleibt vorerst konstant – was bis auf die minimale Veränderung in Abhängigkeit von der Entfernung zwischen den beteiligten Körpern auch stimmt. Wie bereits erwähnt, werden wir als einfachste Lösung für das Einbringen der Schubkräfte der Raketen die Gravitationskonstante anpassen. Damit geben wir dem Begriff »Konstante« eine ganz neue Bedeutung. Falls Ihnen diese Art der Lösung aber zu

verwirrend ist, so können Sie auch gerne die Impulsgesetze anwenden, um die Kraft der Raketenstöße einzukalkulieren.

8.5 »Houston, bitte kommen!« – die Steuerung

Nachdem unsere Raumfähre im letzten Abschnitt noch hilflos gefallen ist, werden wir nun die nötigen Gegenmaßnahmen ergreifen. Zunächst reicht hierfür die Rakete, die das Raumschiff im Fall bremst. Für diese Rakete werden wir die (Pfeil nach unten)-Taste als Benutzereingabe abfangen und in einen Raketenstoß umsetzen. Die Behandlung dieses Ereignisses ist denkbar einfach, da hierfür nur bei gedrückter Taste die Gravitationskonstante entsprechend verändert wird. Sobald die Taste wieder losgelassen wird, wirkt wieder die ursprüngliche Gravitationskraft auf das Raumschiff ein. Dies ist die bereits mehrmals angesprochene Vereinfachung unseres Modells, wobei in der Realität natürlich immer beide Kräfte wirken und etwas aufwendiger zu verrechnen wären.

```
velocity = 0
height = 400
while True:
    gravity = 1.635
    if not handleEvent(pygame.event.poll()):
        break

    global command
    if command != (0, 0):
        if command == (0, +1):
            gravity = -2.0

    velocity, height = calculateDelta(gravity, velocity, height)
```

Listing 8.7 Die Hauptablauf-Schleife

Im Quelltext-Ausschnitt ist die Ereignisbehandlung umgesetzt, wobei Sie dies erst im folgenden Listing bezüglich der Funktion `handleEvent()` sehen werden, in welchem wir die Funktion `handleEvent()` näher betrachten werden. Die Gravitationskonstante wird nun bei jedem Schleifendurchlauf auf `1.635` gesetzt und bei entsprechenden Benutzereingaben auf den Wert `-2.0` angepasst, der die Raketenstöße einbezieht.

Wenn der Spieler die (Pfeil nach unten)-Taste drückt, wird die Bremsrakete durch Anpassung der Gravitationskonstante in die Berechnungen einbezogen. Die Stärke der Raketenstöße können Sie selbst noch feinjustieren – zu schwache

Wirkung führt zu sehr trägen Reaktionen des Raumschiffs, während eine zu starke Wirkung auch nicht leicht steuerbar ist, da dann das Landen sehr schwer wird.

```
def handleEvent(event):
    if event.type == QUIT or
     (event.type == KEYDOWN and event.key == K_ESCAPE):
        return False

    global command
    if event.type == KEYDOWN:
        if event.key == K_RIGHT:
            command = ( +1,  0)
        if event.key == K_LEFT:
            command = ( -1,  0)
        if event.key == K_UP:
            command = (  0, -1)
        if event.key == K_DOWN:
            command = (  0, +1)

    if event.type == KEYUP:
        command = (0, 0)

    return True
```

Listing 8.8 Die Ereignisbehandlung

Die Methode `handleEvent()` speichert nur den jeweils letzten Befehl des Benutzers, wobei jedoch die `Pfeil nach oben`-Taste nicht verarbeitet wird, da die Gravitationskraft für die Beschleunigung nach unten ausreichen sollte. Es gibt entsprechend auch keine Steuerrakete, die noch stärker nach unten beschleunigt. Wenn die Taste gedrückt bleibt, so gilt der Befehl weiterhin als aktiv, bis die Taste losgelassen wird. Es gilt immer der zuletzt erteilte Befehl als aktiv. Wenn irgendeine Taste losgelassen wird, dann wird der zuletzt aktive Befehl aufgehoben. Probieren Sie ruhig ein bisschen mit dieser Version herum, um ein Gefühl für die Steuerung zu bekommen, und variieren Sie auch die Schubkraft der Rakete, indem Sie den Wert `-2.0` entsprechend erhöhen oder verringern.

Nun fehlt noch die seitliche Aussteuerung des Anflugs. Bis jetzt passiert diesbezüglich nichts, denn das Raumschiff hat zu Beginn keine seitliche Geschwindigkeitskomponente. Deshalb werden wir zunächst eine weitere Geschwindigkeitsvariable zur Erfassung der seitlichen Geschwindigkeit einführen, die unabhängig von der Fallbeschleunigung anhand der nur seitlich wirkenden Steuerraketenschübe berechnet wird.

Es gibt an dieser Stelle zwei Möglichkeiten, das Raumschiff zu beeinflussen: Die Steuerraketen könnten das Raumfahrzeug auf der Stelle drehen und so die Wirkung der Bremsrakete in eine andere Richtung weisen lassen, oder die Steuerraketen wirken nur in waagerechter Richtung. Die zweite Variante ist einfacher zu steuern, weshalb wir sie zunächst vorziehen werden. Später können Sie gerne selbst versuchen, die andere Variante der Steuerung zu implementieren, die etwas schwerer umzusetzen ist.

Die zweite Geschwindigkeitskomponente führen wir folgendermaßen ein:

```
def calculateDelta(acceleration, velocity, position):
    t = 1.0/fps
    h = 0.5*acceleration*t**2 + velocity*t + position
    v = acceleration + velocity

    return v, position
```

Listing 8.9 Berechnung der Geschwindigkeit und Position

Zunächst passen wir die Funktion `calculateDelta()` an. Es geht hier nur um eine Namensänderung, um Verwirrung vorzubeugen: Aus dem Parameter `height` wurde nun `position`, da die Funktion auch für die Veränderung in der Horizontalen verwendet werden soll. Dabei wollen wir die Steuerraketen deutlich schwächer auslegen, so dass hier eine Beschleunigungskomponente von einem halben Meter je Quadratsekunde genügen sollte. Entsprechend benennen wir den Parameter `gravity` in `acceleration` um. Die Vorzeichen verschwinden jetzt auch, indem wir sie einfach in die entsprechenden Variablendefinitionen aufnehmen.

Die Anpassungen in der Hauptroutine sind im folgenden Listing **fett** dargestellt:

```
velocity = 0
sideVelocity = randint(5, 20)
height = 400
xPos = 200
while True:
    gravity = -1.635
    sideAcceleration    = +0.0
    if not handleEvent(pygame.event.poll()):
        break

    global command
    if command != (0, 0):
        if command == ( 0, +1):
            gravity = 2.0
        if command == ( 1,  0):
            sideAcceleration = +0.5
```

```
        if command == (-1,  0):
            sideAcceleration = -0.5

    velocity, height = calculateDelta(gravity, velocity, height)
    sideVelocity, xPos = calculateDelta(sideAcceleration,
     sideVelocity, xPos)

    clearScreen()
    drawRocket(xPos, height)
    drawGround(100, 50)
```

Listing 8.10 Anpassungen in der Hauptroutine

Der Code enthält einige neue Variablen, die sich um die Seitenbeschleunigung sowie -geschwindigkeit kümmern. Die seitliche wirkende Geschwindigkeitskomponente wird in der Variablen `sideVelocity` erfasst, die dazugehörige Beschleunigungskomponente in der Variablen `sideAcceleration`. Die seitliche Beschleunigung wird entsprechend aktiviert, wenn der Befehl dazu über ein Tastaturereignis gegeben wurde. Die Berechnung der neuen Position und der angepassten Geschwindigkeit erfolgt ebenfalls mit der Funktion `calculateDelta()`, in der wir die Bezeichner bereits angepasst haben.

Im Zusammenhang mit diesem sehr einfachen Beispiel ist der Bezeichner `calculateDelta` leicht verständlich. In der Praxis führen solche zunächst noch gut einzuordnenden Namen im späteren Verlauf schnell zu Missverständnissen, sobald das Projekt immer größer wird, denn solche nichtssagenden Bezeichner werden schnell verwechselt, wenn es ähnliche Namen für sehr unterschiedliche Funktionen in verschiedenen Bereichen der Software gibt, die nach und nach durch Integration zusammenfließen.

8.6 »Houston, wir haben ein Problem!« – die Kollisionserkennung

Da die Steuerung nun komplett ist, widmen wir uns der Kollision. Dabei müssen wir für einen erfolgreichen Landeanflug noch Rahmenbedingungen vorgeben. Diese lassen sich leicht in Form von Geschwindigkeitsgrenzen je Richtungskomponente festlegen. So wäre eine seitliche Geschwindigkeit unter einem Kilometer pro Stunde sicher ausreichend langsam für die Landung, während die Fallgeschwindigkeit nicht mehr als zwei Kilometer pro Stunde betragen sollte. Natürlich können Sie aber gerne andere Werte verwenden.

Für die Kollision vergleichen wir einfach die Positionen der beiden beteiligten Körper, wobei wir beachten müssen, dass die Unterkante des Raumschiffs die

Oberkante des Mondes berühren soll. Um die Kollision korrekt zu überprüfen, ist ein Vergleich aller Punkte der beiden Körper notwendig. Prinzipiell würde in diesem speziellen Fall zwar auch der Vergleich der unteren Punkte des Raumschiffs mit den oberen Punkten des Mondes ausreichen, aber dies würde schon fehlschlagen, wenn das Raumschiff von unten an die »Mondplatte« heranflöge. Deshalb implementieren wir gleich die korrekte vollständige Kollisionsprüfung, die alle Punkte einbezieht.

Um diesen Vergleich zu ermöglichen, werden wir die beiden Körper als Ansammlung von vier Punkten repräsentieren. In diesem Zusammenhang können wir die beiden Zeichenfunktionen löschen, weil wir nur noch eine Funktion zum Zeichnen von Körpern, die durch ein Rechteck dargestellt werden, verwenden. Diese Funktion existiert bereits unter dem Namen `drawRect()`, wobei wir sie ein bisschen anpassen, so dass als Parameter der zu zeichnende Körper als Liste übergeben wird, die die vier Punkte des zu zeichnenden Rechtecks repräsentiert. Falls sich die Körper so schnell bewegen, dass sie zwischen zwei Frames bereits kollidiert sind, so wäre eine Überprüfung aller dazwischenliegenden Punkte notwendig. In diesem Fall verzichten wir jedoch auf diese Überprüfung, da sich die verwendeten Körper in der Regel nicht so schnell bewegen.

Auch diesen Quelltext zum Beispiel finden Sie auf der beiliegenden CD unter *Kapitel08/LunarLander_v2.py*. **[◉]**

```
def drawRect(dimension):
    x, y, width, height = dimension
    glColor3f(1.0, 0.0, 0.0)
    glBegin(GL_QUADS)
    glVertex3f(x, y, 0.0)
    glVertex3f(x + width, y, 0.0)
    glVertex3f(x + width, y + height, 0.0)
    glVertex3f(x, y + height, 0.0)
    glEnd()
```

Listing 8.11 Anpassung der Funktion »drawRect«

Die folgende Funktion `collisionDetection()` übernimmt die Kollisionserkennung und wird je Frame einmal aufgerufen.

```
def collisionDetection(rocket, ground):
    rx1, ry1, width, height = rocket
    rx2, ry2                = rx1 + width, ry1 + height
    gx1, gy1, width, height = ground
    gx2, gy2                = gx1 + width, gy1 + height

    if (rx1 >= gx1 and rx1 <= gx2) or
```

```
    (rx2 >= gx1 and rx2 <= gx2):
      # wir befinden uns in Bodennaehe
      if (ry1 >= gy1 and ry1 <= gy2) or
        (ry2 >= gy1 and ry2 <= gy2):
          # die Koerper ueberdecken sich
          return True
  return False
```

Listing 8.12 Kollisionserkennung

Für die Kollisionsüberprüfung vergleichen wir zunächst die horizontale Position und dann bei Überschneidung die vertikale Position der beteiligten Körper. Die Kollisionsüberprüfung ist in diesem Beispiel sehr leicht, da nur zwei Körper beteiligt sind. Je mehr Körper sich bewegen und in die Kollisionsüberprüfung einbezogen werden müssen, desto aufwendiger werden die notwendigen Vergleiche, denn mit jedem weiteren Körper steigt der Aufwand überproportional, da alle Körper mit allen anderen Körpern auf Kollisionen untersucht werden müssen. Wenn es sich bei den Körpern zudem um Polygone mit sehr vielen Punkten handelt, so steigt der Aufwand noch weiter an.

Eine Möglichkeit zur Vereinfachung besteht darin, nur nahe Körper zu betrachten. Dazu wäre es denkbar, die Größe der Körper zu klassifizieren und entsprechend nur zu überprüfen, wenn die Körper im Verhältnis zu ihrer Größe sehr nah aneinanderliegen. Die Größe wäre hier vereinfachend als Radius des den Körper umgebenden Kreises denkbar, so dass die Position des Körpers dann nur noch über den Mittelpunkt definiert würde. So ließe sich eine Vorabprüfung durchführen, gefolgt von der eigentlichen Punkt-für-Punkt-Überprüfung. Wenn die Körper durch Gleichungen definiert werden, so ist natürlich auch das Gleichsetzen der beiden Gleichungen möglich, um festzustellen, ob es Schnittpunkte gibt.

Uns fehlt jetzt noch die Überprüfung der Randbedingungen, die zu erfüllen sind, um erfolgreich zu landen. Wie bereits geschrieben, beschränken wir uns hier auf die Geschwindigkeitsparameter, wobei wir zusätzlich überprüfen könnten, dass die Landung nicht zu nahe am Rand unserer Mondplatte stattfindet. Auf den zweiten Punkt verzichten wir zunächst, da eine echte Mondlandung auch keine Ränder berücksichtigen müsste. Später bietet dieser Punkt aber die Möglichkeit, den Schwierigkeitsgrad zu erhöhen; es wäre zum Beispiel denkbar, die Landung nur auf einem sehr kleinen Teilbereich zuzulassen, wobei Sie auch die Treibstoffmenge begrenzen könnten.

Die Überprüfung der Randbedingungen wird die Funktion `checkRules()` übernehmen:

```
def checkRules(velocity, sideVelocity):
    velocity = abs(velocity)
    sideVelocity = abs(sideVelocity)
    print velocity, sideVelocity
    if velocity < 12.0 and sideVelocity <= 2.0:
        return True
    else:
        return False
```

Listing 8.13 Überprüfung der Randbedingungen

Zunächst wird der absolute Betrag der beiden Geschwindigkeitskomponenten ermittelt, da die Richtung keine Rolle spielt. Danach wird überprüft, ob die beiden Geschwindigkeitsgrenzen eingehalten werden. Hier wird einfach mit `12.0` und `2.0` verglichen, was nicht unbedingt 12 km/h bzw. 2 km/h sein müssen; da die gesamten Berechnungen nicht skaliert wurden, können Sie bei den Grenzwerten einfach nach Gefühl vorgehen. Die beiden Geschwindigkeitsgrenzen sollten in größeren Anwendungen natürlich nicht einfach als »Magic Numbers« auftauchen, sondern einen sprechenden Bezeichner haben, so dass sofort klar ist, um welche Art von Konstante es sich hierbei handelt.

Um eine erfolgreiche oder missglückte Landung auch entsprechend zu honorieren beziehungsweise zu sanktionieren, benötigen wir nach der Überprüfung noch ein Feedback an den Benutzer und müssen das Spiel anhalten. Dies geschieht aus Gründen der Einfachheit durch eine Art ASCII-Art, für die wir die »Grafik« einfach in einer Textdatei definieren. Die Textdatei lesen wir dann einfach ein und stellen die relevanten Inhalte als Kästchen auf dem Bildschirm dar.

```
def drawElement(element, color):
    r, g, b = color
    glColor3f(r, g, b)
    glBegin(GL_QUADS)
    for part in element:
        x, y    = part
        x       = x * 10.0
        y       = y * 10.0
        glVertex3f(x, y, 0.0)
        glVertex3f(9.0 + x, y, 0.0)
        glVertex3f(9.0 + x, 9.0 + y, 0.0)
        glVertex3f(x, 9.0 + y, 0.0)
    glEnd()
```

Listing 8.14 Zeichnen von Elementen

Die Funktion `drawElement()` wird für die folgende Funktion `showASCIIArt` benötigt und zeichnet die Klötzchen, die am Ende den Text ergeben. Da wir in diesem Kapitel die Skala so definiert haben, dass Null unten liegt, müssen wir die Koordinaten für die Darstellung auf dem Bildschirm anpassen. Dies geschieht in der folgenden Funktion, indem `y` beim Wert `30` startet und reduziert wird, statt erhöht zu werden. Der Wert `30` reicht in diesem Fall vollkommen aus, da wir die Höhe der verwendeten Texte auf jeden Fall darauf beschränken – das sollten Sie sich einfach gut merken, denn es gibt keine Überprüfung dieser Vereinbarung.

```
def showASCIIArt(fileName, delay):
    datei = open(fileName, "r")
    asciiArt = []
    y = 30
    for zeile in datei:
        x = -1
        y -= 1
        for zeichen in zeile:
            if zeichen != '\n':
                x += 1
                if zeichen == 'X':
                    asciiArt.append((x, y))

    clearScreen()
    drawElement(asciiArt, (1.0, 0.0, 0.0))
    pygame.display.flip()
    pygame.time.delay(delay)
```

Listing 8.15 Darstellung von ASCII-Art

Die für das umgekehrte Koordinatensystem angepasste Funktion `showASCIIArt()` funktioniert jetzt auch einwandfrei für die Ergebnisdarstellung. Die darzustellenden Inhalte werden wie auch in Kapitel 3, »Snake« (Listing 3.25) aus Dateien geladen.

```
def showGameOver():
    showASCIIArt("GameOver.txt", 3000)
    restartGame()

def showYouWon():
    showASCIIArt("YouWon.txt", 5000)
    restartGame()
```

Listing 8.16 Das Ergebnis ausgeben

```
def restartGame():
    # reinitialize the game (restart)
    global direction, command, gameCycle,
      velocity, sideVelocity, height, xPos
    gameCycle           = 0
    direction           = (0, 0)
    command             = (0, 0)
    velocity            = 0
    sideVelocity        = randint(5, 20)
    height              = 400
    xPos                = 200
    pygame.event.clear()
```

Listing 8.17 Einen neuen Versuch starten

In der Praxis würden Sie hier natürlich nicht den Quelltext aus Kapitel 3 hineinkopieren, sondern eine entsprechende Bibliothek für ASCII-Art anlegen, die Sie nun einfach wieder importieren würden. Die Zeiten für das Ergebnisfeedback unterscheiden sich, weil ich mir dachte, dass es netter ist, ein positives Feedback länger zu geben als ein negatives.

So weit zum Spielablauf – jetzt kommen wir zu dem eigentlichen Schmankerl, denn wir wollen ja gar nicht selbst steuern, sondern den Computer landen lassen. Dabei wollen wir uns auch keine Gedanken um die Strategie machen, wie wir uns der Oberfläche langsam genug nähern können. Wir möchten lediglich, dass der Computer auf der Oberfläche landet und dabei gewisse Rahmenbedingungen einhält. Um das zu erreichen, werden wir jetzt mit genetischen Algorithmen arbeiten.

8.7 Das Steuer aus der Hand geben

Das Raumschiff selbst zu steuern, ist ja ganz lustig, aber das ist auf Dauer bei einem so einfachen Spiel doch etwas langweilig. Da muss dann schon die andere Variante her, bei der wir das Raumschiff um die eigene Achse drehen lassen und somit eine etwas komplexere Steuerung verlangen. Ist jedoch der Computer am Zug, um zu landen, dann führt auch eine so einfache Aufgabe sehr schnell zu interessanten Varianten. Außerdem können Sie den Schwierigkeitsgrad der Landung immer weiter steigern, um zu schauen, wie lange der Computer mithalten kann.

Wie Sie sich sicher schon vorstellen können, wird diese Aufgabe allerdings nicht die geringsten Probleme bereiten, denn Computer sind einfach extrem schnell. Aber wie finden Sie die passenden Parameter für die Lösung dieses Problems – also der Mondlandung? Welche Parameter gäbe es denn überhaupt? Das sind

die beiden Fragen, die wir als Nächstes beantworten müssen. Hier kommen wir wieder nun wieder auf evolutionäre Techniken zu sprechen, welche uns helfen werden die Landung erfolgreich zu absolvieren.

8.7.1 Parametrisierung genetischer Algorithmen

Genetische Algorithmen lassen sich über die Mutationsrate, die Populationsgröße sowie die Kreuzung der Lösungsvorschläge parametrisieren.

[+] Mutation und Kreuzung – was ist damit gemeint?

Schauen wir uns als Einstieg zunächst einmal ein Beispiel zur natürlichen Auslese an, worunter man das Überleben der am besten an den Lebensraum angepassten Individuen versteht. Vielleicht kennen Sie den John-Wayne-Western »Das Gesetz des Stärkeren« – das geht ungefähr in die gleiche Richtung.

Angenommen, es gab vor sehr langer Zeit eine Rasse, die große Ähnlichkeit mit Pferden hatte, sich aber nur von speziellen, an Bäumen wachsenden Früchten ernährte. Im Laufe der Zeit wurden die Bäume aber immer größer und trugen die Früchte immer höher, weil sie somit weniger zum »Opfer« der pferdeähnlichen Rasse wurden. Parallel musste sich unsere Rasse dann ebenfalls an die neuen Umstände anpassen, so dass diejenigen Individuen, die größer waren oder einen längeren Hals hatten, auch bessere Überlebenschancen hatten, da sie an mehr Futterquellen heranreichten. Entsprechend hatten die benachteiligten Individuen bald keine Nahrung mehr und starben so langsam aus, während sich die größeren Individuen fortpflanzten und dabei ihr Erbgut austauschten.

Diesen Austausch des Erbgutes wollen wir im Folgenden als die *Kreuzung* – biologisch gesehen bei zwei verschiedenen Rassen – beziehungsweise Paarung von zwei Individuen bezeichnen, die somit ihre Eigenschaften vermischen. Die Unterscheidung in Paarung und Kreuzung ist hierbei nicht notwendig, da wir hierfür zunächsteinmal Rassen definieren müssten, was aber für die Evolution keinerlei Bedeutung hat. Darüber hinaus wird auch nicht zwischen dominanten und nicht dominanten Eigenschaften unterschieden.

Diese Art der Entwicklung beziehungsweise Evolution ist aber nicht ganz ausreichend, denn ein multiplikativer Prozess der Verbesserung anhand der Kreuzung verschiedener Individuen ist nur möglich, wenn es überhaupt eine signifikante Verbesserung bei einem Teil der Population gibt. Da aber nicht bekannt ist, was in Zukunft ein Vorteil sein wird, spielt für diesen Prozess der Zufall die entscheidende Rolle. Durch an sich ungewollte Mutationen, die in den günstigen Fällen zu einer vorteilhaften Entwicklung neuer Eigenschaften führen, werden somit neue Individuen erzeugt, die besser an die veränderten Lebensbedingungen angepasst sind. Die Mutationen, die nicht vorteilhaft ausfallen, führen zu vereinzelten Individuen, die über die Zeit sehr schnell wieder aussterben.

Der entscheidende Punkt hierbei ist eine ausgewogene Mutationsrate, die dafür sorgt, dass die Diversität der Population hoch genug bleibt, aber eine Evolution durch Kreuzung dennoch zustande kommt. Ohne Mutation würde die Vielfalt total abflachen – bei zu hoher Mutation fiele das Kreuzen nicht mehr ins Gewicht, und es würde kein Optimum angestrebt, sondern es käme zufällig ohne Ziel zu schlecht angepassten Individuen.

Darüber hinaus gibt es verschiedene Erweiterungen, wie zum Beispiel das Sichern einer Elite, um die besten Individuen immer zu erhalten. Die einzelnen Verfahren für die Mutation, Kreuzung und Erhaltung der besten Individuen lassen sich dabei sehr unterschiedlich umsetzen. Je nachdem, um welche Art von Problem es sich handelt, sind manche Ansätze vorteilhafter als andere. Wobei hiermit gemeint ist, dass die besser zum Problem passenden Vorgehensweisen bezüglich der Kreuzung und Mutation in weniger Generationen das Problem lösen, weil sie sich der Lösung schneller nähern.

Abbildung 8.4 Die Evolution hat versagt, der lange Hals stört enorm.

8.7.2 Problemdarstellung und Kodierung

Um jetzt aber überhaupt beginnen zu können, müssen wir das Problem sinnvoll kodieren; dazu sind alle am Problem beteiligten Größen in eine für die Verfahren geeignete Problemdarstellung zu bringen.

Für die Landung auf dem Mond wären die zu kodierenden Daten unter anderem die Geschwindigkeitskomponenten, die Kraftwirkung der Raketenstöße und deren zeitliche Dauer. Es werden also vor allem eine Aktion und deren Dauer zu kodieren sein. Als mögliche Aktionen sind die Raketenstöße nach links, rechts und unten zu kodieren. Außerdem müssen wir die Aktion »tue nichts« kodieren, da ansonsten eine Annäherung an den Mond durch freien Fall nicht möglich wäre. Alternativ wäre es natürlich denkbar, die Raketenstöße in ihrer Intensität steuerbar zu machen. Dies führt jedoch nur eine weitere Indirektion ein, die keine Vorteile bei der Lösung des Problems böte. Für die Annäherung genügt es völlig, wenn wir zwischen freiem Fall und Raketenbeschleunigung nach oben unterscheiden.

Um nun mit der Suche nach einer Lösung zur Landung des Raumschiffs zu beginnen, hängen wir beliebige Aktionen aneinander und probieren sie aus. Dabei kommt am Anfang natürlich noch nichts wirklich Sinnvolles zustande, da es sehr unwahrscheinlich ist, dass zufällig erzeugte Aktionen zu einer erfolgreichen Landung führen. Sie können sich das Ganze in etwa als Schrotschuss-Variante vorstellen: Ein paar Kugeln finden immer das Ziel, vorausgesetzt, der Schütze hat zumindest halbwegs in die korrekte Richtung geschossen. Die zufälligen Aktionslisten können Sie sich wie Individuen vorstellen, die mehr oder weniger gut an ihren Lebensraum angepasst sind. Für jedes dieser Individuen lässt sich ein »Fitnesswert« berechnen, der die Annäherung an das eigentliche Ziel darstellt. Falls Ihnen die Schrotschuss-Analogie mehr zusagt, dann stellen Sie sich jedes Schrotkorn als ein Individuum vor, wobei die Individuen, die näher am Ziel einschlagen, entsprechend eine höhere Fitness besitzen.

Jetzt stellt sich die Frage, wie wir die Bewertung der Fitness am besten durchführen. Da es um eine erfolgreiche Landung geht, wird der Abstand des Raumschiffs vom Mond als ein Kriterium in die Berechnung eingehen. Darüber hinaus kalkulieren wir das Endresultat der Aktionen ein, so dass erfolgreiche Landungen sehr positiv in die Bewertung einfließen. Um tendenziell bessere Bruchlandungen von katastrophalen Bruchlandungen zu unterscheiden, können wir zusätzlich die Geschwindigkeit beim Aufprall für die Bewertung der Fitness heranziehen. Dabei werden wir wieder zwischen der seitlichen und der vertikalen Geschwindigkeitskomponente unterscheiden.

Wir müssen also Individuen kodieren, die wir testen können, um einen Fitnesswert zurückzuliefern. Die besten dieser Individuen kreuzen wir dann, um die nächste Generation von Individuen zu erzeugen. Dabei gibt es noch ein Problem, denn bei Optimierungsproblemen wird unter Umständen vor der eigentlichen Lösung ein lokales Maximum oder Minimum erreicht, das entsprechend noch nicht das Optimum darstellt, aber dennoch nicht mehr verlassen wird. Ob es sich um ein Maximum oder ein Minimum handelt, hängt dabei davon ab, ob die Optimierung eine Minimierung oder eine Maximierung anstrebt. Allgemeingültig kann man also sagen, dass lokale Extrema durchschritten werden müssen, um das globale Extremum der Funktion zu erreichen.

Stellen Sie sich ein Optimierungsproblem einfach wie eine Funktion vor, die auch lokale Extrema besitzen kann. Als Beispiel stellt Abbildung 8.5 eine Funktion mit zwei Freiheitsgraden dar, also eine im zweidimensionalen Koordinatensystem darstellbare Funktion. In der Realität haben die komplexen Optimierungsprobleme aber wesentlich mehr Freiheitsgrade, so dass das Ganze zum Beispiel bei drei Freiheitsgraden wie eine dreidimensionale Landschaft mit Bergen und Tälern aussähe. Bei noch mehr Freiheitsgraden versagt in der Regel die Vorstellungskraft, aber die Prinzipien bleiben gleich, nur dass es mehr als drei Dimensionen gibt,

was für den Menschen schwer vorstellbar ist. Allenfalls die vierte Dimension, die Zeit, ließe sich noch vorstellbar machen.

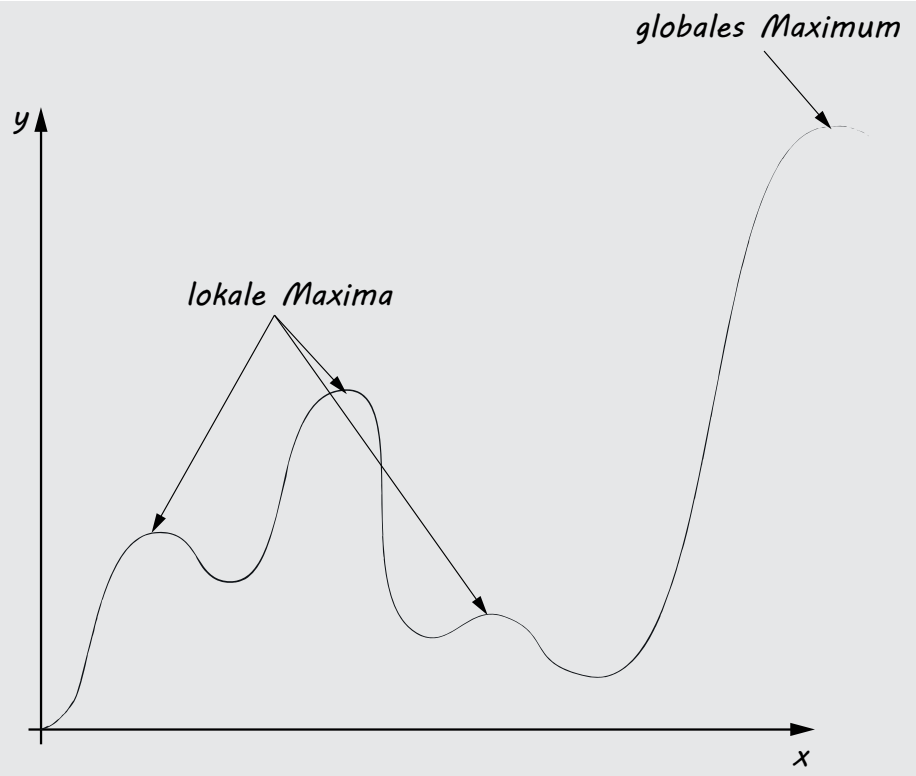


Abbildung 8.5 Lokale Extrema bei einer einfachen Funktion

Um bei der Optimierung nicht in lokalen Extrema zu verharren, führen wir zusätzlich zur normalen Kreuzung der Gene die Mutation ein, die dafür sorgt, dass lokale Extrema durchschritten werden, denn am Ende wollen wir das globale Optimum für die zu lösende Problemstellung erreichen.

Bevor wir jetzt aber mit der Optimierung beginnen, räumen wir erst einmal ein bisschen auf. Alle Aktionen bezüglich der Raumfähre kapseln wir in der Klasse `lander`, die wie folgt aussieht.

Den Quelltext zu diesem Beispiel finden Sie wieder auf der beiliegenden der CD-Rom im folgenden Pfad *Kapitel08/version3/lander.py*. **[◉]**

```
import random

class lander(object):
    def __init__(self, xPos, yPos, xVelocity, yVelocity):
        self.xVelocity      = xVelocity
        self.yVelocity      = yVelocity
        self.xPos           = xPos
        self.yPos           = yPos
```

```
    def calculateDelta(self, fps, xAcceleration, yAcceleration):
        dt = 1.0/fps
        self.xVelocity  += xAcceleration * dt
        self.xPos       += self.xVelocity * dt
        self.yVelocity  += yAcceleration * dt
        self.yPos       += self.yVelocity * dt

        return self.xPos, self.yPos

    def checkRules(self):
        yVelocity = abs(self.yVelocity)
        xVelocity = abs(self.xVelocity)
        if yVelocity < 40 and xVelocity < 30:
            return True
        else:
            return False
```

Listing 8.18 Die Klasse »lander«

Die Namensgebung habe ich etwas vereinheitlicht, so dass die Geschwindigkeits- und Beschleunigungsanteile, die jeweils in einen horizontalen und vertikalen Anteil aufgeteilt sind, klar erkennbar sind. Die Positionsveränderung wird komplett für beide Anteile innerhalb von `calculateDelta()` berechnet und zurückgegeben. Die Berechnung gestaltet sich zwar etwas anders, aber macht im Prinzip das Gleiche – es werden die Differenzen addiert, statt zum Beispiel den bis zum aktuellen Zeitpunkt komplett zurückgelegten Weg zu berechnen.

Da nun alle landerelevanten Daten innerhalb der Klasse `lander` gekapselt sind, wird auch die Überprüfung der einzuhaltenden Geschwindigkeitsgrenzen innerhalb der Klasse durch `checkRules()` übernommen. Die Toleranzen sind hier höher, da die Konstanten für die Beschleunigung um Faktor 10 höher eingestellt sind. Die Kollisionsüberprüfung bleibt innerhalb des Hauptmoduls, da hier auch die Positionsinformation für die Mondplatte benötigt wird, die vollkommen unabhängig von der Position der Raumfähre ist. Die neue Klasse wird entsprechend wie in Listing 8.19 im Hauptmodul verwendet:

[◉] Dieser Quelltext befindet sich ebenfalls auf der beiliegenden CD-Rom im Pfad *Kapitel08/version3/AutomatedLunarLander_v1.py*.

```
fieldWidth  = 800
fieldHeight = 600
fps         = 80

command             = [0, 0]
commandList         = findBestCommands()
```

```
lunarLander         = lander(200, 400, 0, 0)
def main():
    pygame.init()
    video_flags = OPENGL | HWSURFACE | DOUBLEBUF

    screenSize = (fieldWidth, fieldHeight)
    pygame.display.set_mode(screenSize, video_flags)
    resize(screenSize)

    init()
    ground = 350, 50, 100, 30

    while True:
        yAcceleration   = -16.35
        xAcceleration   = +0.0

        if not handleEvent(pygame.event.poll()):
            break

        commandAvailable = loadNextCommand()
        if commandAvailable:
            if command == ( 0, +1):
                yAcceleration = 20.0
            if command == ( 1,  0):
                xAcceleration = -10.0
            if command == (-1,  0):
                xAcceleration = +10.0

        xPos, yPos = lunarLander.calculateDelta(fps, xAcceleration,
         yAcceleration)

        clearScreen()
        rocket = xPos, yPos, 20, 30
        drawRect(rocket)
        drawRect(ground)

        #if collisionDetection(rocket, ground):
        if  yPos < ground[1]+ground[3]:
            print " Distance:", calculateDistance(rocket, ground)
            print "xVelocity:", lunarLander.xVelocity
            print "yVelocity:", lunarLander.yVelocity
            if lunarLander.checkRules():
                showYouWon()
            else:
                showGameOver()
```

```
        pygame.display.flip()

        pygame.time.delay(int(1000.0/fps))
```

Listing 8.19 Automatisierte Landung

Damit haben wir genug aufgeräumt. Jetzt fehlt uns nur noch die Möglichkeit, die Aktionen für die Landung im Vorhinein vorzugeben. Denn die verschiedenen Individuen, die unterschiedliche Permutationen von Aktionen repräsentieren, sollen sich an einer Landung versuchen. Um nicht jede noch so katastrophale Ladung mit ansehen zu müssen, werden wir nur die besten Individuen auch zum Landeanflug zulassen.

Für die automatisierte Landung wird nun ein neues Hauptmodul entstehen, das sich nur geringfügig von dem bisher erstellten unterscheidet. Anstelle des Event-Pollings wird jetzt einfach eine Liste mit Befehlen der Reihe nach abgearbeitet. Auf Benutzerereignisse reagieren wir trotzdem noch, denn ansonsten ließe sich das Fenster nicht schließen. Die Simulation ohne grafische Darstellung erfolgt in der Klasse `landerSimulation`.

[●] Den Quelltext zum Beispiel finden Sie wieder auf der beiliegenden CD unter *Kapitel08/version3/landerSimulation.py*.

```
class landerSimulation(object):
    def __init__(self):
        self.lunarLander = lander(200, 400, 0, 0)
```

Listing 8.20 Initalisierung der Simulation des Landeanflugs

Die Klasse `landerSimulation()` erzeugt für die Simulation des Landeanflugs zunächst eine Instanz der Klasse `lunarLander`, die mit der gleichen Position initialisiert werden muss wie im Modul `AutomatedLunarLander_v1.py`. Deshalb wäre es hier auch günstiger, wenn die Positionsdaten zentral einmalig definiert wären, statt sie hier als »Magic Numbers« zu übergeben. Für das Beispiel soll der Einfachheit halber aber dieses Vorgehen vorläufig genügen.

```
    def start(self, commandList):
        # commandList wird kopiert um
        # die Ausgangsliste zu erhalten
        self.commandList = commandList[:]
        while True:
            self.yAcceleration   = -16.35
            self.xAcceleration   = +0.0

            commandAvailable = self.loadNextCommand()
            if commandAvailable:
```

```
            if self.command == ( 0, +1):
                self.yAcceleration = 20.0
            if self.command == ( 1,  0):
                self.xAcceleration = -10
            if self.command == (-1,  0):
                self.xAcceleration = +10

        xPos, yPos = self.lunarLander.
         calculateDelta(80, self.xAcceleration,
          self.yAcceleration)

        rocket = xPos, yPos, 20, 30
        ground = 350, 50, 100, 30
```

Listing 8.21 Simulationshauptroutine

Zunächst wird die Befehlsliste `commandList` kopiert, damit die ursprüngliche Befehlsliste nicht verändert wird. Dies ist notwendig, da wir beim Abarbeiten der Befehle den jeweils zu bearbeitenden Befehl aus der Liste entfernen. Die Verarbeitung der Befehle geschieht analog zur bereits bekannten Vorgehensweise bei der manuellen Landung. Die neue Position des Raumschiffs wird ebenfalls wie bereits bekannt durch den Aufruf der Methode `calculateDelta()` berechnet, wobei `calculateDelta()` wie oben bereits beschrieben etwas anders arbeitet. Die Veränderung hat jedoch nichts mit der Automatisierung zu tun, sondern erfolgte einfach im Zuge der Optimierung. Die Position des Bodens beziehungsweise der Landeplattform auf dem Mond wird wieder durch »Magic Numbers« direkt im Quelltext definiert, was nur der Einfachheit halber passiert ist – auch hier böte sich wieder eine zentrale Datenhaltung mit Bezeichner an.

```
# um trotz der Subtraktionen sicherzustellen,
# dass die Fitness positiv bleibt
            self.fitness     = 100000
            if commandAvailable == False:
                # auf Boden warten
                while yPos >= ground[1]+ground[3]:
                    xPos, yPos = self.lunarLander.
                     calculateDelta(80, self.xAcceleration,
                      self.yAcceleration)
                rocket = xPos, yPos, 20, 30
                dist = self.calculateDistance(rocket, ground)
                self.fitness -= dist * 10
                self.fitness -= abs(self.lunarLander.xVelocity) * 4
                self.fitness -= abs(self.lunarLander.yVelocity) * 8

                if self.collisionDetection(rocket, ground):
```

```
                    if self.lunarLander.checkRules():
                        self.fitness += 100000

                break

        if (self.fitness < 0):
            raise Exception("negative fitness scores are evil")
        return self.fitness
```

Listing 8.22 Die Fitnessberechnung

Hier wird nun endlich die Fitness berechnet, wobei einfach die Entfernung zur Landeplattform mit der größten Gewichtung eingeht, gefolgt von der Fallgeschwindigkeit und seitlichen Geschwindigkeit. Als Bonus wird zum Schluss noch eine kräftige Summe addiert, wenn es sich um eine geglückte Landung handelt. Wie Sie sehen, stellt die Fitness also viele verschiedene Einflussgrößen in einer Zahl gebündelt dar. Hierbei wird der Fitnesswert mit `100000` initialisiert und danach um für die Fitness negative Faktoren reduziert. Eine erfolgreiche Landung wird mit weiteren `100000` als Bonus honoriert.

Die Zahlen und ihre individuelle Gewichtung sind frei zu definieren – hier müssen Sie einfach ein sinnvolles Gleichgewicht finden. Falls Ihnen das jetzt noch ein bisschen schleierhaft vorkommt, so stellen Sie sich die Fitness einfach als eine Art Notendurchschnitt vor; je größer die Zahl, desto besser. Wobei das »je größer, desto besser« natürlich für die Fitness gilt, denn beim Notendurchschnitt verhält es sich ja umgekehrt.

Aber zurück zum Quelltext und zur Verarbeitung der Befehle für die Landung.

```
def loadNextCommand():
    global command
    if commandList:
        command = commandList.pop(0)
    else:
        command = (0, 0)
```

Listing 8.23 Laden der Befehle

Die Methode `loadNextCommand()` liest so lange die Befehlsliste aus, bis diese leer ist. Danach werden keine Befehle mehr ausgeführt, so dass das Raumschiff einfach entsprechend der Mondbeschleunigungskonstante fällt. Für die automatische Landung werden wir zunächst rein zufällige Befehlssequenzen generieren, wofür die folgenden Klassen zum Zuge kommen.

Die einfachste Form der Kodierung soll dabei durch die Klasse `Gen` zum Ausdruck kommen, die einen einzelnen Befehl kodiert. Bei der Initialisierung von `Gen`

mittels des Standardkonstruktors wird ein zufällig ausgewählter Befehl für eine bestimmte Zeit als Initialzustand des Gens gesetzt. Die Klasse `Gen`, die also im Folgenden die kleinste Einheit darstellt, wird von der Klasse `Genome` verwendet.

Beim Lösen von Optimierungsproblemen mittels evolutionärer Algorithmen ist es wichtig, am Anfang wirklich alle Problemgrößen in sinnvoller Weise zu erfassen und zu kodieren. In der Klasse `Gen` werden die Aktion an sich und ihre Laufzeit kodiert. Das heißt genau genommen, welche Steuerdüse für welchen Zeitraum aktiviert werden soll. In der Klasse `Genome` werden daraufhin mehrere Instanzen der Klasse `Gen` in einer Liste gehalten, die somit eine Sequenz von Steuerbefehlen beschreibt.

Schauen wir uns zunächst an, wie sich die Klasse `Gen` aufbaut:

```
import random
from landerSimulation import landerSimulation
import copy

class Gen(object):
    __maxActionDuration     = 80
    __maxMutationDuration   = 20

    leftRocket    = (-1,  0)
    rightRocket   = (+1,  0)
    centralRocket = ( 0, +1)
    doNothing     = ( 0, -1)

    actions       = (leftRocket, rightRocket,
      centralRocket, doNothing)

    def __init__(self):
        self.action     = random.choice(Gen.actions)
        self.duration   = random.randint(1, Gen.__maxActionDuration)

    def mutateAction(self):
        self.action = random.choice(Gen.actions)

    def mutateDuration(self):
        delta = Gen.__maxMutationDuration -
         random.randint(1, 2*Gen.__maxMutationDuration)
        self.duration += delta
        if self.duration > self.__maxActionDuration:
            self.duration = self.__maxActionDuration

    def __eq__(self, other):
        return (self.action == other.action) and
```

```
            (self.duration == other.duration)
```

Listing 8.24 Die Klasse »Gen« – Kodierung einzelner Befehle

Im Body der Klasse `Gen` (vgl. Listing 8.24) werden zuerst die statischen Variablen `__maxActionDuration` und `__maxMutationDuration` deklariert, die die maximale Dauer eines Befehls und die maximale Verlängerung eines mutierten Befehls enthalten. Die Verlängerung der Befehlsdauer entspricht dabei der bei Mutationen vorkommenden Verstärkung der Eigenschaft. Daraufhin werden die verschiedene Aktionen als Tupel definiert, die identisch mit den Befehlstupeln sind, die wir in der manuellen Version der Simulation verwendet haben. Es handelt sich also um die Kodierung der zu zündenden Rakete. Bei der Initialisierung eines Gens wird wie erwähnt eine zufällige Aktion ausgewählt, die für eine zufällige Dauer ausgeführt werden soll. Es folgen zwei Methoden für die Mutation einzelner Gene, wobei der Befehl und die Dauer des Befehls unabhängig voneinander mutieren.

Die Mutation führt bei der Länge der Aktion mit verringerter Wahrscheinlichkeit noch zu einer Verstärkung, wobei aber die maximale Ausführungszeit nicht überschritten wird. Wie gerade erwähnt wurden die Befehle der Einfachheit halber identisch mit den Kommandos, die bei Benutzereingaben verarbeitet werden, umgesetzt. Natürlich ist auch eine beliebige andere Kodierung für die Befehle problemlos möglich. Zusätzlich können einzelne Gene mutieren, was von den Methoden `mutateAction()` und `mutateDuration()` berücksichtigt wird. Die Mutation dient dem Durchschreiten eventuell vorhandener lokaler Minima oder Maxima, je nachdem, ob es sich um ein Minimierungs- oder ein Maximierungsproblem handelt – unter anderem wird so auch eine sinnvolle Vielfalt erhalten. Dabei sind diese beiden Problemarten durch Negierung leicht in die jeweils andere zu überführen. Zudem wird der Vergleichsoperator überschrieben, wodurch später der exakte Vergleich zweier Gene möglich wird. Eine komplette Gensequenz wird wie bereits erwähnt in der Klasse `Genome` repräsentiert, die sich die Klasse `Gen` zunutze macht.

```
class Genome(object):
    def __init__(self, genes=None):
        if genes:
         # es ist wichtig hier eine Kopie zu erzeugen!
         # ansonsten werden die gleichen Gene in
         # verschiedenen Individuen referenziert
         # und manipuliert
            self.__genes     = copy.deepcopy(genes)
        else:
            self.__genes    = []
            for i in xrange(Evolution.numberOfGenes):
                self.__genes.append(Gen())
```

```
def getGenes(self):
    return self.__genes

def setGenes(self, genes):
    self.__genes = genes

genes = property(getGenes, setGenes)
```

Listing 8.25 Die Klasse »Genome« – Kodierung von Befehlssequenzen

Bei der Initialisierung der Klasse `Genome` wird zunächst eine zufällige Gensequenz erstellt, die somit das Individuum repräsentiert. Diese Gensequenz lässt sich nun anhand einer Simulation bewerten, wobei das Ergebnis der Bewertung als Fitness gespeichert wird.

```
def getCommandList(self):
    commandList = []
    for gen in self.__genes:
        for i in xrange(gen.duration):
            commandList.append(gen.action)

    return commandList
```

Listing 8.26 Klasse »Gen«: Umwandlung der Gensequenz in eine Befehlsliste

Sowohl für die Simulation als auch für die eigentliche Landung des fittesten Individuums müssen wir die Gensequenz in eine Befehlsliste überführen, die ohne Umwege interpretiert werden kann. Diese Umwandlung erfolgt durch die Methode `getCommandList()`, die einfach die einzelnen Befehle entsprechend der Befehlsdauer mehrfach in eine Liste kopiert. Hier wäre es auch möglich gewesen, die Verarbeitung der Befehle in der Simulation entsprechend anzupassen, damit auch die Gensequenz interpretiert werden kann. Einfacher und meiner Meinung nach der direktere Weg ist aber die hier verwendete Umwandlung in eine Befehlsliste, die den bereits bekannten Kriterien der schon verwendeten Befehlsliste entspricht.

```
def updateFitness(self):
 # hier wird der Fitnesswert berechnet
    simulation      = landerSimulation()
    commands        = self.getCommandList()
    self.fitness    = simulation.start(commands)
    return self.fitness
```

Listing 8.27 Klasse »Gen«: Berechnung der Fitness durch Simulation

Für jedes Individuum, das als Instanz der Klasse `Genome` existiert, muss es möglich sein, die Fitness zu berechnen. Diese Möglichkeit bietet uns die Methode `updateFitness()`, die die Landung anhand der Gensequenz des Individuums simuliert. Hierfür wird zunächst die Befehlsliste aus der Gensequenz erzeugt, was mit der gerade besprochenen Methode `getCommandList()` möglich ist.

```
def mutate(self):
    for gen in self.genes:
        if random.random() < Evolution.mutationRate:
            gen.mutateAction()
        if random.random() < Evolution.mutationRate/2.0:
            gen.mutateDuration()
```

Listing 8.28 Klasse »Gen«: Mutation

In der inneren Klasse `Genome` werden einige wenige Gene entsprechend der vorgegebenen Mutationsrate durch den Aufruf von `mutate()` angepasst. Die Mutationsrate stellt dabei einen nicht sehr einfachen Parameter bei der Optimierung dar: Ist sie zu hoch angesetzt, dann ist die Streuung der Individuen zu groß, so dass das Optimum nie wirklich stark genug angenähert werden kann. Ist sie hingegen zu niedrig, so besteht die Gefahr, in einem lokalen Minimum beziehungsweise Maximum zu landen. Der Vergleichsoperator »kleiner gleich« wird überschrieben, um die Fitness der verschiedenen Gensequenzen, die die einzelnen Individuen repräsentieren, zu vergleichen und zu sortieren.

Für die Lösung eines Problems ist in erster Linie die korrekte Kodierung desselben notwendig. Wenn die Kodierung das Problem gut abbildet, dann führen die folgenden Schritte automatisch zur Annäherung an die Lösung des Problems.

Zunächst wird eine zufällige Generation von Individuen erzeugt, in der die Ergebnisse zur Lösung des Problems entsprechend schlecht ausfallen. Die Kodierung der Individuen geschieht für diese erste Generation rein zufallsbasiert und erfolgt beim Aufruf von `Gen()` bei der Initialisierung des neuen Objekts der Klasse `Gen`.

Jedem Individuum teilen wir entsprechend der Güte der Lösungsannäherung einen Fitnesswert zu. Die Zuteilung des Fitnesswert erfordert eine Testlandung, die für den Benutzer unsichtbar simuliert wird. Die Testlandung bewerten wir dann anhand einiger Kriterien, wie Entfernung vom Landeziel sowie Geschwindigkeit beim Aufprall und Landeerfolg oder Fehlschlag.

Danach werden immer zwei Individuen ausgewählt, wobei die Wahrscheinlichkeit für die Auswahl direkt proportional zur Fitness ist. Hierfür verwenden wir ein Verfahren, das allgemein unter dem Namen *Roulette-Wheel-Selection* (vgl. Abbildung 8.6) bekannt ist. Der Name leitet sich daraus ab, dass die einzelnen Individuen im Prinzip einem Kreisstück zugeordnet werden, dessen Größe pro-

portional zur Fitness ist. Wenn Sie sich das Ganze jetzt als Roulette-Rad vorstellen, dann ist die Wahrscheinlichkeit, mit der die Kugel das entsprechende Individuum auswählt, direkt proportional zur jeweiligen Fitness.

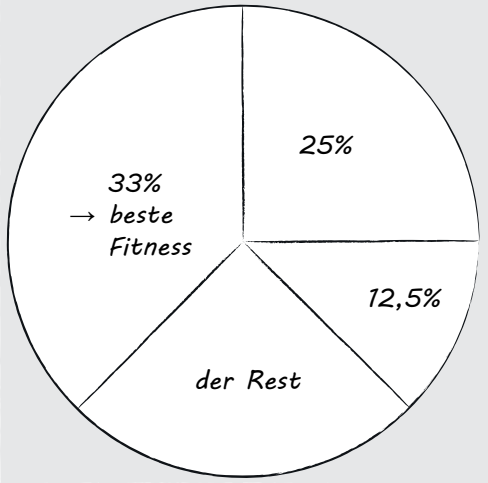


Abbildung 8.6 Roulette-Wheel-Selection

```
def rouletteWheelSelection(self, population):
    totalFitness = 0
    for x in population:
        totalFitness += x.fitness

    randomSlice     = random.random() * totalFitness;
    fitnessSum      = 0
    for x in population:
        fitnessSum += x.fitness
        if fitnessSum >= randomSlice:
            return x

    return population[0]
```

Listing 8.29 Implementierung der Roulette-Wheel-Selection

Um die Auswahl zu treffen, bilden wir zunächst die Summe aus allen Fitnesswerten und speichern sie in `totalFitness`. In der darauffolgenden Schleife wird dann für jedes Individuum der Fitnesswert in der Variablen `fitnessSum` summiert. Sobald diese Summe dann den zufällig ermittelten Wert von `totalFitness` erreicht hat, wird das Individuum an der aktuellen Iterationsstelle ausgewählt – also das zu dem Zeitpunkt in `x` referenzierte Individuum.

Bevor diese Art der Auswahl möglich ist, müssen wir die Population entsprechend der jeweiligen Fitness sortieren. Dies realisieren wir durch die Funktion `sortByFitness()`, die für jedes Individuum eine Landung simuliert und je nach

Ergebnisgüte eine entsprechende Fitness zuteilt. Durch die Sortierung entsprechend der jeweiligen Fitness stellen wir sicher, dass die erfolgreichen Individuen mit entsprechend hoher Fitness zuerst summiert werden.

```
def sortByFitness(self, population):
    for individuum in population:
        individuum.updateFitness()
    population.sort(lambda x, y: cmp(x.fitness, y.fitness),
     reverse=True)
    currentBest = Genome(population[0].genes)
    currentBest.fitness = population[0].fitness

    return currentBest
```

Listing 8.30 Sortierung der Individuen nach Fitness

Die Simulation wird in der Methode `updateFitness()` des jeweiligen Individuums gestartet und ist genauso aufgebaut wie die eigentlich Hauptroutine für die Mondlandung. Der einzige Unterschied ist, dass wir nicht auf den Bildschirm zeichnen und die einzelnen Befehle entsprechend sehr schnell durchlaufen werden. Um insgesamt eine bestmögliche Performance zu erreichen, laden wir zu Beginn im Modul `AutomatedLunarLander_v1.py` noch `psyco` und führen es aus. `psyco` ist eine Bibliothek, die Python-Programme vorkompiliert, ähnlich wie der *Just in Time*-Compiler(JIT) für Java.

```
import psyco
psyco.full()
```

Listing 8.31 Zur Geschwindigkeitssteigerung verwenden wir »psyco«.

Nachdem zwei Individuen ausgewählt wurden, rufen wir anschließend die Funktion `multiCrossOver()` auf, die die Nachkommen der beiden Individuen erzeugt. Ein kleines Problem bei der Qualität ist jedoch, dass hier viel Code kopiert wurde – dieses Problem sollten Sie im Zuge weiterer Optimierungsmaßnahmen noch in Angriff nehmen. Wichtig ist hierbei, dass alle die Simulation betreffenden Werte wirklich identisch sind, da sich die Auswertung der Simulation ansonsten für die Landung nicht verwenden lässt. Hiermit sind vor allem die Ausgangsposition der Raumfähre sowie der Landeplattform und die Ausgangsgeschwindigkeiten der Raumfähre gemeint. Wenn also die Fitness anhand der Simulation optimiert wird, aber die Simulation mit den später für die Durchführung relevanten Zuständen nichts zu tun hat, dann ist das in der Simulation am besten entwickelte Individuum für die spätere Landung nicht zu gebrauchen, denn es ist sozusagen einen anderen »Lebensraum« gewohnt.

Dieser Sachverhalt ist im Prinzip trivial: Wenn Sie jahrelang Skat spielen und dort Ihre Fähigkeiten perfektionieren, dann wird Ihnen das nahezu nutzlos sein, wenn Sie später eine Partie Schach spielen wollen. Das Beispiel ist bewusst etwas übertrieben gewählt und soll lediglich zum Ausdruck bringen, dass die angelernten Individuen sich nicht zur Laufzeit an die Umgebung anpassen. Natürlich wäre es auch denkbar, nicht die gesamte Landung zu simulieren, sondern nur Teilschritte auf dem Weg zum Mond.

Bei einer sehr performanten Umsetzung könnten Sie auch zur Laufzeit parallel weiter nach noch besseren Lösungen suchen, so dass der Lebensraum tatsächlich kontinuierlich adaptiert würde. Diese Art der Verwendung ist besonders für komplexe Strategiespiele sehr interessant, wobei hier häufig eine Kombination aus genetischen Algorithmen und neuronalen Netzen zum Einsatz kommt. Wie bereits erwähnt, wird für solche Fälle das Problem in der Regel in viele kleine Einzelschritte zerlegt, die individuell optimiert werden können. Etwa wie eine schrittweise Wegsuche, bei der der Gesamtweg in Zwischenziele unterteilt ist, die jeweils im nächsten Schritt zu erreichen sind.

Die Methode `multiCrossOver()`, die die beiden zur Paarung auserwählten Individuen kreuzt, sehen Sie in Listing 8.32.

```
def multiCrossOver(self, dad, mum):
    if (random.random() > Evolution.crossOverRate):
        return dad, mum # keine Kreuzung

    swapRate = random.random() * Evolution.numberOfGenes

    baby1, baby2    = [], []
    dadGenes        = dad.genes
    mumGenes        = mum.genes

    for x in range(Evolution.numberOfGenes):
        if random.random() < swapRate:
            baby1.append(dadGenes[x])
            baby2.append(mumGenes[x])
        else:
            baby1.append(mumGenes[x])
            baby2.append(dadGenes[x])

    return Genome(baby1), Genome(baby2)
```

Listing 8.32 Kreuzung von Individuen

In der Variablen `swapRate` wird ein Zufallswert gespeichert, der die Wahrscheinlichkeit der Wechsel bei der Kreuzung bestimmt. Für die Kreuzung gibt es sehr

viele Varianten, wobei für unterschiedliche Probleme auch unterschiedliche Formen der Kreuzung sinnvoll sind. Die hier verwendete Art der Kreuzung (ein Multi-Crossover, vgl. auch Abbildung 8.7) führt zu einer sehr starken Mischung der beiden Individuen und erfolgt außerdem über die gesamte Länge der Genkette relativ gleichmäßig.

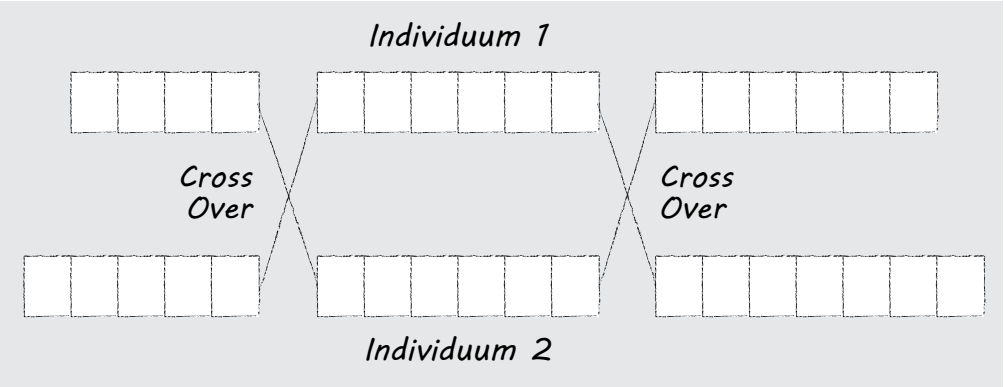


Abbildung 8.7 Multi-Crossover

Ein denkbare Alternative wäre eine einfache Kreuzung, bei der genau ein Kreuzungspunkt bestimmt wird, an dem das Erbmaterial entsprechend vertauscht wird. Aus der aktuellen Generation werden so lange neue Individuen für die Fortpflanzung ausgewählt, bis die neue Generation die Zielgröße erreicht hat. Die vier besten Individuen werden direkt in die neue Generation kopiert und somit gesichert. Dadurch wird verhindert, dass eine bereits sehr gute Lösung verlorengeht. Das Kopieren können Sie als elitäre Bevorzugung ansehen, die die Ergebnisse bei der Suche nach der Lösung verbessert und die Suche beschleunigt. Durch dieses Sichern der Besten wird ein gewisser Qualitätsstandard gewahrt. Ohne das Kopieren der besten Individuen ist es auch möglich, dass ein bereits erreichtes Level wieder verlassen wird. Ein Nachteil dieser Bevorzugung der bisher besten Individuen ist, dass sie möglicherweise die Vielfalt schneller abflacht. Im Extremfall würde dies das Festhängen an lokalen Optima bedeuten.

Nachdem die Nachfahren erzeugt wurden, werden sie noch entsprechend ihrer Mutationsrate zum Teil wahllos verändert. Die Mutationsrate ist jedoch sehr gering, so dass nur wenige Individuen mutieren. Sinn der Mutation einiger weniger Individuen ist das Überwinden von lokalen Optima bei der Suche des globalen Optimums. Insofern erhalten wir uns durch die geringfügige Mutation einiger weniger Individuen eine größere Vielfalt der Population. Die Mutation erfolgt jeweils durch Aufruf der Methode `mutate()` der inneren Klasse `Genome`.

```
def mutate(self):
    for gen in self.genes:
        if random.random() < Evolution.mutationRate:
            gen.mutateAction()
        if random.random() < Evolution.mutationRate/2.0:
            gen.mutateDuration()
```

Listing 8.33 Mutation im Computer

Es wird sowohl die Aktion als auch mit halber Wahrscheinlichkeit die Länge der Aktion verändert, wobei die Veränderung der Aktionslänge sich auf eine Verlängerung beschränkt. Dies ist gleichzusetzen mit der bei Mutation zum Teil auftretenden Verstärkung von Eigenschaften.

Alle Funktionen in Summe werden in der für die Evolution entscheidenden Hauptfunktion `evolve()` verwendet:

```
def evolve(self):
    population = self.createNewGeneration(Evolution.populationSize)
    nextGeneration = []
    generationCounter = 0
    bestIndividuum = None
```

Listing 8.34 Hauptablauf der Evolution

Die Hauptfunktion durchläuft alle Phasen der Evolution. Zunächst wird eine vollkommen zufällig erzeugte Startpopulation geschaffen, die dann schrittweise anhand der Evolutionsgesetzmäßigkeiten fortschreitet. Dabei wird immer das beste Individuum gesichert, so dass es über die Zeit nicht verlorengeht. Sobald eine Lösung des Problems durch eines der Individuen gefunden wurde, wird die Evolution abgebrochen und das entsprechende Individuum, das in der Variablen `bestIndividuum` gespeichert ist, zurückgegeben.

Wie bereits erwähnt liegt die Schwierigkeit bei genetischen Algorithmen in der Wahl der korrekten Parameter für die Evolution sowie in der Art der Kodierung des Problems und der Fitnessberechnung. Je besser die Parameter zur Kodierung und zur Fitnessberechnung passen, desto schneller wird im Laufe der Evolution eine Lösung gefunden. Bei sehr schlechter Stimmigkeit eines Kriteriums wird möglicherweise nie eine Lösung gefunden. Wie die einzelnen Parameter zu bestimmen sind, hängt sehr stark vom Problem ab und variiert entsprechend von Fall zu Fall. Mit ein wenig Übung und Tüfteln werden Sie aber mit der Zeit ein gutes Gefühl dafür bekommen.

Neben den Parametern sind auch die Verfahren bezüglich der Kreuzung, Mutation und Eliteförderung von Problem zu Problem unterschiedlich anzupassen. Allein

für die Kreuzung gibt es viele verschiedene sinnvolle Möglichkeiten, die sich jeweils für unterschiedliche Problemdomänen besser oder schlechter eignen.

Schauen wir weiter, wie innerhalb von `evolve()` alle notwendigen Schritte zusammenlaufen:

```
while (generationCounter < Evolution.maxGenerations):
    print generationCounter, " ",
    if generationCounter % 20 == 0:
        print
    currentBest = self.sortByFitness(population)
    if (bestIndividuum == None) or
      (currentBest.fitness > bestIndividuum.fitness):
        bestIndividuum  = currentBest
        print bestIndividuum.fitness
        if bestIndividuum.fitness > 100000:
            # wir haben eine Lösung gefunden
            return bestIndividuum
    # erhalte die besten Individuen für die nächste
    # Generation dabei benötigen wir eine gerade Anzahl
    # von Individuen, um immer jeweils einen Vater und
    # eine Mutter für die Kreuzung zur Verfügung zu haben,
    # da sonst die Roulette-Wheel-Selection nicht funktioniert
    nextGeneration  = []
    nextGeneration += population[0:Evolution.saveBestCount]
```

Listing 8.35 Fortschreiten der Evolution

In der `while`-Schleife wird zunächst das aktuell beste Individuum in der Variablen `currentBest` gesichert und überprüft, ob bereits eine Lösung gefunden wurde. Dann werden entsprechend der in der Variablen `Evolution.saveBestCount` vorgegeben Zahl die besten Individuen der aktuellen Population für die nächste Generation gesichert. Dieses Vorgehen ist natürlich nicht von der Natur abgeschaut – hierbei handelt es sich eher um so etwas wie Klone der besten Individuen der aktuellen Population. Probieren Sie selbst ruhig ein bisschen herum – das Ganze funktioniert natürlich auch ohne das Sichern der besten Lebewesen. Dieses Sichern ist die bereits angesprochene Eliteförderung beziehungsweise elitäre Bevorzugung. Aber weiter im Text beziehungsweise Quelltext.

```
    babyCount   = 0
    while babyCount < (Evolution.populationSize):
        dad             = self.rouletteWheelSelection(population)
        mum             = self.rouletteWheelSelection(population)
        baby1, baby2    = self.multiCrossOver(dad, mum)

        baby1.mutate()
```

```
        baby2.mutate()

        nextGeneration += baby1, baby2
        babyCount      += 2

    population           = nextGeneration
    generationCounter   += 1

return bestIndividuum
```

Listing 8.36 Generationenwechsel

Hier folgt die Auswahl eines Vaters und einer Mutter für die Kreuzung, aus denen zwei Kinder hervorgehen. Für die Kreuzung wurde das bereits angesprochene Multi-Crossover-Verfahren verwendet, das mehrere Kreuzungspunkte über den gesamten Genstrang der beiden Elternteile festlegt und somit zu einer relativ gleichmäßigen Vermischung der Eigenschaften der beiden Individuen führt. Daraufhin wird jedes der Kinder noch einer kleinen Mutation unterzogen – keine Angst, es wird ihnen nicht wehtun, es ist sogar gut für sie. Die Kinder werden dann in die nächste Generation übernommen, auf der die übernächste Generation aufgebaut wird. Das alles wiederholt sich so lange, bis eine Lösung gefunden oder die Anzahl der maximal zu durchlaufenden Generationen erreicht wurde. Was hier noch nicht berücksichtigt wurde, aber durchaus sinnvoll wäre, ist der Abbruch für den Fall, dass die Populationsvielfalt zu stark abgeflacht ist, denn dann ist es sehr unwahrscheinlich, dass sich noch etwas weltbewegend Neues ergibt.

Eine Möglichkeit, dieser Abflachung noch stärker als nur mit Mutation entgegenzuwirken, ist die Einmischung von zusätzlichen neuen Individuen, die wie ganz zu Beginn rein zufällig kodiert wurden. Auf diese Möglichkeit verzichte ich aber im Zuge dieses Beispiels – Sie können gerne für Ihre eigenen Versuche auch auf diese Möglichkeit zurückgreifen, die eine sehr viel stärkere Einbringung neuer Impulse erlaubt.

Beim Kreuzen gibt es wie bereits erwähnt natürlich noch verschiedenste andere Möglichkeiten, die Sie ebenfalls austesten können. Ebenso ist auch die elitäre Bevorzugung nicht immer sinnvoll und sollte immer an das jeweilige Problem angepasst werden.

Mit der Anzahl der am Problem beteiligten Parameter wird auch die korrekte Kodierung und Parameterwahl für die Evolution sehr viel schwieriger. Bei zwei zu optimierenden Parametern lässt sich das Problem auch als Funktion mit einem globalen Optimum und eventuell einigen vorhandenen lokalen Optima darstellen. Bei drei Parametern ergibt sich bereits eine 3D-Welt mit Tälern und Hügeln.

Ab vier Parametern ist die Darstellung schon nicht mehr leicht möglich, wobei Sie hier noch die Zeit als vierten Parameter einführen könnten, so dass das Problem wie die Wellentäler und -hügel eines bewegten Sees oder Meeres aussieht.

Kommen wir nun zur Berechnung der Fitness – die bereits als Fitnesswert erwähnt wurde – der einzelnen Individuen, die häufig ein großes Problem darstellt, denn es ist eine möglichst gute Heuristik zu wählen, die bei deutlich geringerem Rechenaufwand eine aussagekräftige Ergebnisprognose liefert. In diesem Beispiel simulieren wir einfach die komplette Landung, statt eine Heuristik für die Bewertung zu verwenden.

[+] Heuristik – was ist das?

Unter *Heuristik* versteht man eine Abschätzung, die deutlich leichter zu treffen ist als die korrekte Ermittlung des Resultats eines Vorgangs. Heuristiken finden insofern in den verschiedensten Bereichen ihre Anwendung, so zum Beispiel auch beim Schach, wenn es darum geht, die aktuelle Stellung abzuschätzen, da man nicht beliebig weit vorausrechnen kann.

Natürlich sind solche Heuristiken – je nachdem, wie fortschrittlich sie sind – auch fehlerbehaftet, denn ansonsten wäre jede exakte Berechnung gar nicht sinnvoll, wenn man auch sehr leicht eine absolut zuverlässige Heuristik heranziehen könnte. Dementsprechend geht es bei Heuristiken in der Regel um einen Kompromiss zwischen Zeiteinsatz und Aussagekraft beziehungsweise Genauigkeit. Gerade im Schach ist die Heuristik die entscheidende Komponente für die Spielstärke der Programme. Die Suchverfahren zum Durchschreiten des Spielbaums haben schon sehr viele Optimierungen erfahren und sind entsprechend ausgereift und fortschrittlich. Was noch stark hinterherhinkt, ist die Stellungsbewertung, also die Heuristik. Im Falle von Schach erfolgt häufig eine statische Bewertung der Stellung anhand der Materialverteilung, der Königssicherheit sowie der kontrollierten Felder und Aktivität der Figuren.

Wie oben schon erwähnt, unternehmen wir in diesem Beispiel einfach einen Versuch und beurteilen das Ergebnis, aber es gibt auch Probleme, in denen ein Versuch zu viel Zeit in Anspruch nähme, so dass man nicht um die Verwendung einer guten Heuristik herumkommt. Denkbar ist natürlich auch ein Kompromiss, der zunächst eine heuristische Bewertung vornimmt und danach für die vielversprechenden Individuen eine Simulation durchführt. Somit wird nicht für jedes Individuum die komplette Simulation notwendig, aber es ist immer noch gewährleistet, dass nur wirklich gute Individuen für die Kreuzung herangezogen werden. Hiervon unberührt bleibt natürlich die geringe Wahrscheinlichkeit der Wahl eines schlechten Individuums bei der Roulette-Wheel-Selection erhalten.

Für unsere Mondlandung ermitteln wir die Fitness wie folgt:

```
# stellt sicher, dass die Fitness positiv bleibt
self.fitness     = 100000
if commandAvailable == False:
    self.fitness -= self.calculateDistance(rocket, ground)*10
    break

if self.collisionDetection(rocket, ground):
    if self.lunarLander.checkRules():
        self.fitness += 1000000
```

Listing 8.37 Fitnessberechnung

Sobald alle vom Individuum kodierten Befehle abgearbeitet wurden, wird die Entfernung zur Landeplattform beziehungsweise zum Mond ermittelt und von dem vorher zugewiesenen Wert abgezogen. Somit erhalten weiter entfernte Individuen eine geringere Bewertung als jene, die sich dem Ziel stärker nähern. Zusätzlich wird zum Schluss überprüft, ob die Landung erfolgreich war. In diesem Fall wird ein Bonus zur Bewertung addiert.

Die Berechnung der Entfernung erfolgt in der Methode `calculateDistance()`:

```
    def calculateDistance(self, rocket, ground):
        rx1, ry1, rx2, ry2      = rocket
        gx1, gy1, gx2, gy2      = ground
        groundCenterUpside      = (gx1 + gx2/2.0, gy1 + gy2)
        rocketCenterDownside    = (rx1 + rx2/2.0, ry1)

        return math.sqrt((rocketCenterDownside[0] -
         groundCenterUpside[0])**2 +
          (rocketCenterDownside[1] -
           groundCenterUpside[1])**2)
```

Listing 8.38 Berechnung des Abstandes der Raumfähre von der Landeplattform

Es wird der Abstand von der Mitte der Raumschiffunterseite zur Mitte der Mondoberfläche ermittelt. Dabei kommt einfach der Satz des Pythagoras zur Anwendung, da es sich um ein schlichtes rechtwinkliges Dreieck handelt, bei dem die beiden Katheten bekannt sind. Die Kollisionsüberprüfung funktioniert genauso wie auch schon bei der manuell gesteuerten Version. Soweit sogut – schauen Sie sich alles nochmal in Ruhe an und spielen Sie ruhig auch mit den Parametern für die Evolution, um ein Gefühl für die Problematik der Parameterwahl zu bekommen.

8.8 War das schon alles?

Mit dem in diesem Kapitel angerissenen Verfahren lassen sich sehr viele Probleme angehen. Unter anderem werden evolutionäre Algorithmen – wie bereits erwähnt – in der Logistik für die Routenplanung mit Zeitfenster eingesetzt. Ein anderes bereits erwähntes Gebiet bietet die Spieleentwicklung, wobei es in erster Linie um die Schaffung adäquater Gegner geht. Etwas allgemeiner lässt sich sagen, dass Sie evolutionäre Algorithmen für nahezu jedes Optimierungsproblem einsetzen können. Sinnvoll sind sie jedoch nur bei Problemen, welche nicht in kurzer Zeit auch deterministisch gelöst werden können. Die Nachteile genetischer Algorithmen liegen vor allem in der nicht Erklärbarkeit der Ergebnisfindung. Einen gewissen Ausgleich schafft aber die Möglichkeit, die Ergebnisgüte zumindest als prozentuale Abweichung vom Optimum abschätzen zu können.

»Aus großer Kraft folgt große Verantwortung.«

Aus dem Film »Spiderman« – gilt aber auch für Google

9 Spiderman

9.1 Webcrawler – folge den Zeichen

In diesem Kapitel entwickeln wir eine kleine Suchmaschine, die selbständig anhand einer Vorgabe von Internetadressen weitere Seiten sucht und diese entsprechend der dort auftretenden Wörter indiziert. Für die Umsetzung bedienen wir uns »Beautiful Soup«, einer bereits vorhandenen, frei verfügbaren Bibliothek zum Parsen von *HTML*, der ***H**yper**t**ext **M**arkup **L**anguage*.

HTML ist die Sprache, in der Webseiten beschrieben sind. Bei dieser Art der Beschreibung geht es hauptsächlich um die inhaltliche Strukturierung der Webseiten. Für die Layout-Beschreibung hat sich neben HTML *CSS* etabliert, was für *Cascading Style Sheets* steht. Vor der Entstehung von CSS wurde das Layout mit der strukturellen Beschreibung in HTML vermischt.

Aufgrund dieser noch existierenden Mischformen von HTML ist das Parsen nicht immer ganz einfach. HTML an sich ist auch nicht wohlgeformt, was die Sache nicht vereinfacht. »Wohlgeformt« würde diesbezüglich unter anderem heißen, dass jeder Tag auch geschlossen werden muss, also zum Beispiel »<html>...</html>«. Diesem Umstand ist zu verdanken, dass ein neuer Standard etabliert wurde, der besser zu parsen ist. Suchmaschinen müssen aber in der Lage sein, alles zu parsen, wenn möglich auch fehlerhafte Seiten, die sehr häufig vorzufinden sind. Im günstigsten Fall trifft man auf eine *XHTML*-Seite, welche entsprechend wohlgeformt ist.

Ein *Webcrawler* sucht auf den ihm bekannten Seiten einfach nach Links und nach Wörtern und ist im Grunde genommen sehr einfach aufgebaut. Er verfolgt die gefundenen Links weiter, so dass er die entsprechenden Seiten auch indiziert. Beim Indizieren berücksichtigt er die vorkommenden Wörter in ihrer Häufigkeit. Die Daten legt er in einer Datenbank ab, um sie später bei Suchanfragen zurückliefern zu können. Dabei ordnet er den Webseiten die auf den Seiten vorkommenden Wörter inklusive der Auftretenshäufigkeit sowie der Position des ersten Auftre-

tens zu. Darüber hinaus wird auch die Verlinkung der einzelnen Seiten in der Datenbank gespeichert.

Die Hauptaufgabe des Parsens verlegen wir einfach in die bereits etablierte und erwähnte Bibliothek »Beautiful Soup«. Mit dieser »schönen Suppe« beschäftigen wir uns im nächsten Abschnitt.

9.2 Beautiful Soup – Suppe?!

Beautiful Soup ist eine Bibliothek zum Parsen von HTML, die Sie auf der Begleit-CD im Ordner zum aktuellen Kapitel finden oder auch im Internet unter *http://www.crummy.com/software/BeautifulSoup/* in der neuesten Version herunterladen können.

Die Verwendung der Bibliothek ist denkbar einfach und sieht wie folgt aus:

```
import urllib2
from BeautifulSoup import BeautifulSoup

c=urllib2.urlopen('http://www.google.de')
webContent = c.read()

soup=BeautifulSoup(webContent)
text = getTextOnly(soup)
```

Listing 9.1 Beispiel zur Verwendung von »Beautiful Soup«

[o] Den Quelltext zum Beispiel finden Sie wie üblich auf der beiliegenden CD unter *Kapitel09/spiderman.py*.

Nach dem Laden der benötigten Bibliotheken wird eine URL geöffnet, in diesem Fall `http://www.google.de`, und eingelesen. Der komplette HTML-Source-Code wird dann einer neuen Instanz von `BeautifulSoup` übergeben, die den Zugriff auf die HTML-Elemente erlaubt. Wie dieser Zugriff aussieht, wurde wiederum in der Funktion `getTextOnly()` verborgen, die ebenfalls auf zwei Hilfsfunktionen aufsetzt:

```
def getTextOnly(soup):
    # Reduzierung auf den <title>-
    # und <body>-Tag
    v=soup.string
    if v==None:
        c=soup.contents
        c=filter(noscript, c)
        c=filter(nostyle, c)
```

```
        resulttext=''
        for t in c:
            subtext=getTextOnly(t)
            resulttext+=subtext+'\n'
        return resulttext
    else:
        return v.strip()
```

Listing 9.2 Die Reduzierung auf das Wesentliche

Grob dargestellt gibt die Funktion `getTextOnly()` nur die Inhalte des `Body`- und `Title`-Tags zurück; der `Title`-Tag enthält den Titel der Seite, der `Body`-Tag die eigentlichen Inhalte. Dabei blendet die Funktion zusätzlich eingebettete Java-Skripte und Style-Sheets aus, die eben nicht zum Inhalt einer Seite zählen, sondern lediglich Layout- und Interaktionsaufgaben übernehmen. Die Skripte und Style-Sheets werden über `filter()` aussortiert, und zwar über ein sogenanntes Funktionsmapping auf alle Elemente des übergebenen Containers. Die übergebene Funktion wird also auf alle Elemente der Liste angewendet, wobei das jeweilige Element als Parameter übergeben und das Ergebnis in eine neue Liste eingetragen wird. Die dahinterstehenden Funktionen sind wie folgt aufgebaut und sind innerhalb der Methode `getTextOnly()` definiert:

```
def noscript(s):
    s = str(s).lower()
    if s==None or s.startswith('<script'):
        return False
    else:
        return True

def nostyle(s):
    s = str(s).lower()
    if s==None or s.startswith('<style'):
        return False
    else:
        return True
```

Listing 9.3 JavaScript und CSS löschen

Die Funktion `noscript()` wandelt zunächst alles in Kleinbuchstaben um und sucht dann nach Inhalten, die mit `<script` beginnen. Alle enthaltenen Elemente werden entsprechend beim Aufruf von `filter()` aus dem übergebenen Container entfernt. Die Funktion `nostyle()` geht analog dazu für alle Elemente vor, die Style-Sheets repräsentieren.

Mit Beautiful Soup ist es jedoch auch möglich, auf alle Tags direkt zuzugreifen. Die Funktionen, die gerade als Beispiel dienten, greifen in diesem Zusammenhang nur auf den Content zu. Außer HTML kann Beautiful Soup auch XML – was für Extensible Markup Language steht – parsen. Der Zugriff auf einzelne Tags erfolgt dabei denkbar einfach über den Tag-Namen.

```
c=urllib2.urlopen('http://www.google.de')
webContent = c.read()

soup=BeautifulSoup(webContent)

print str(soup.title)
```

Listing 9.4 Zugriff auf den Titel der Seite

In diesem Beispiel wird das `Title`-Tag ausgelesen und ausgegeben. Dabei werden die einschließenden Tag-Klammern mit ausgegeben. Solange nicht in einen String umgewandelt wurde, können Sie auch auf die gleiche Art auf die in einem Tag enthaltenen Tags zugreifen. Solche verschachtelten Tags treten zum Beispiel bei Tabellen auf. Aber das ist noch nicht alles – die Möglichkeiten sind noch viel weitreichender, wie folgendes Beispiel zeigt.

```
soup=BeautifulSoup(webContent)
links = soup('a')
print links
```

Listing 9.5 Zugriff auf alle enthaltenen Links

Links werden in HTML mit dem Tag <a> angegeben. Der Aufruf von `soup('a')` gibt dementsprechend eine Liste aller Links innerhalb des HTML-Quelltextes wieder. Analog dazu lassen sich auch alle anderen Tags als Liste zurückgeben.

Weitere Informationen zur Verwendung von Beautiful Soup finden Sie auf der Webseite zur Bibliothek unter *http://www.crummy.com/software/BeautifulSoup/*.

9.3 SQL – die Daten im Griff

In diesem Abschnitt kümmern wir uns um die vom Webcrawler in der Datenbank hinterlegten Daten. Zuerst beschäftigen wir uns kurz mit *SQL*, der *Structured Query Language*, die zum Zugriff auf die mit Python mitgelieferte Datenbank SQLite dient. In SQLite werden alle vom Crawler ermittelten Daten in Tabellen abgelegt und stehen für Abfragen zur Verfügung.

Der Zugriff über SQL erfolgt durch Öffnen einer Verbindung zur Datenbank und Erzeugung einer Cursor-Instanz. Über den Cursor lassen sich daraufhin Daten in die Datenbank schreiben und auslesen.

Zunächst erstellen Sie die erforderlichen Tabellenstrukturen über die folgenden Aufrufe, die in der Methode `createTables()` der Klasse `Crawler` hinterlegt sind:

```
import sqlite3
connection  = sqlite3.connect("crawler.db")
cursor      = connection.cursor()

cursor.execute("""CREATE TABLE websites
                  (id INTEGER PRIMARY KEY, url TEXT)""")

cursor.execute("""CREATE TABLE words
                  (id INTEGER PRIMARY KEY, word TEXT)""")

cursor.execute("""CREATE TABLE webIndex
                  (  id INTEGER PRIMARY KEY, website INTEGER,
                   word INTEGER, count INTEGER,
                   position INTEGER)""")
```

Listing 9.6 Erzeugen von Tabellen mittels SQL

Die SQL-Befehle lesen sich in der Regel schon relativ leicht, so dass deren Bedeutung auch für nicht Eingeweihte normalerweise erkennbar ist. Für die einzelnen Tabellen wird jeweils ein Bezeichner mit Datentyp angegeben. Die einzelnen Bezeichner definieren hierbei die in der Tabelle vorzuhaltenden Spalten. Um alle Einträge eindeutig identifizieren zu können, wird für jede Tablle in der ersten Spalte ein Primärschlüssel mit dem Bezeichner `id` angelegt.

Die SQL-Aufrufe erzeugen also drei Tabellen, wobei jede Tabelle als erste Spalte einen von SQLite vergebenen Primärschlüssel `PRIMARY KEY` erhält. Diese Schlüssel sind über die gesamte Tabelle eindeutig, von den restlichen Daten unabhängig und dienen somit als zuverlässige Referenz auf die einzelnen Zeilen der Tabellen.

Die Tabelle `websites` enthält alle Webseiten, die vom Crawler besucht wurden. In der Tabelle `words` liegen Wörter, die der Crawler jemals geparst hat. Bei den Wörtern wird die Großkleinschreibung nicht berücksichtigt – in der Tabelle werden alle Wörter nur in Kleinbuchstaben gespeichert.

Die Tabelle `webIndex` speichert die Relation zwischen den Wörtern und Webseiten ab – hier wird also hinterlegt, welche Wörter auf welcher Seite auftauchen. Zusätzlich speichert die Tabelle die Häufigkeit des Auftretens der Wörter sowie die Position des ersten Erscheinens auf der Seite. Die Position kann dann zum Beispiel für ein Ranking der Seiten herangezogen werden. Die Häufigkeit des Auftretens

werten wir ebenfalls für eine Relevanzbeurteilung der Webseiten bezüglich der Suchanfrage aus.

Jetzt ist noch zu klären, wie die Daten aus den Tabellen wieder abgefragt werden können und wie sie überhaupt in die Tabellen geschrieben werden. Kommen wir zunächst zum Schreiben von Daten.

```
werte = ( (None, "http://www.google.de"),
          (None, "http://www.microsoft.com"),
          (None, "www.chessbase.de"))

sql = "INSERT INTO websites (id, url) VALUES (?, ?)"
cursor.executemany(sql, werte)
connection.commit()
```

Listing 9.7 Schreibender Zugriff auf die Datenbank

Die Anweisungen im Quelltext tragen drei Webseiten in die Tabelle `websites` ein. Die Daten werden in einem verschachtelten Tupel angegeben, wobei für den Primärschlüssel jeweils `None` übergeben wird, da dieser automatisch vergeben wird. Analog zum Eintragen von Webseiten werden auch andere Daten in die Tabellen übertragen. Für das Eintragen der Daten in die Tabelle `webIndex` sind jedoch vorher noch die Positionen und die jeweilige Anzahl des Auftretens der auf der Seite enthaltenen Wörter zu ermitteln.

Die Daten werden erst nach dem Befehl `connection.commit()` tatsächlich in die Datenbank geschrieben. Durch dieses Vorgehen wahren Datenbank-Management-Systeme die Transaktionssicherheit. Unter *Transaktionssicherheit* versteht man die Sicherheit, dass entweder alle gewünschten Daten übernommen oder geändert werden oder gar keine Daten verändert werden. Ein Szenario, das anschaulich den Sinn von Transaktionssicherheit darstellt, ist die Überweisung von Geld von einem Konto auf ein anderes. Gäbe es keine Transaktionssicherheit, so würde ein möglicher Fehler des Systems den Verlust von Geld im Nirvana erlauben. Dies wäre der Fall, wenn nach dem Abziehen des Geldbetrages vom Ausgangskonto die restlichen zur Überweisung notwendigen Datenänderungen ausblieben und das Geld somit nie auf dem Empfängerkonto ankäme. Um die Transaktionssicherheit zu wahren, lassen Entwickler erst zum Abschluss aller in sich konsistenten Datenänderungen die Transaktion durchführen. Bei entsprechender Vorgehensweise ist auch das Zurücknehmen – man spricht in diesem Fall auch von Rollback – der gesamten Transaktion als ein Schritt möglich.

Wir haben unsere Daten jetzt versammelt, nun müssen wir sie auch wieder auslesen. Dazu nutzen wir diese ebenfalls sehr einfachen Befehle:

```
cursor.execute("SELECT id, url FROM websites")
print cursor.fetchall()
```

Listing 9.8 Abfragen von Daten

`cursor.fetchall()` liefert dabei alle Datensätze entsprechend der Abfrage zurück.

Falls die Abfrage jedoch sehr viele Datensätze liefern wird, so ist es gegebenenfalls günstiger, diese einzeln auszulesen:

```
cursor.execute("SELECT * FROM websites")
row = cursor.fetchone()
while row:
    print row
    row = cursor.fetchone()
```

Listing 9.9 Zeilenweise Ausgabe von Daten einer Tabelle

Diese Art, die Zeilen einer Tabelle einzeln auszulesen, eignet sich besonders dann, wenn jede Zeile individuell verarbeitet werden soll. Außerdem wird dabei nicht so viel Speicher am Stück benötigt, da jeweils nur der Speicher für die gerade zu verarbeitenden Daten reserviert werden muss.

Eine noch einfachere Möglichkeit, auf die Zeilen einzeln zuzugreifen, bietet die Iterator-Schnittstelle der `Cursor-Klasse`:

```
for row in cursor:
     print row
```

Listing 9.10 Alternativer Zugriff auf einzelne Zeilen

9.4 Indizieren – das war mir jetzt zu viel Text!

Nun haben wir einen Stapel von Webseiten und Wörtern, der aber noch völlig ungeordnet und somit für uns nutzlos ist. Also klassifizieren wir diese Webseiten jetzt. Was genau sollte dabei die Beurteilung und Einordnung einer Seite beeinflussen? Das ist sicher keine einfache Frage und lässt sich auch nicht direkt und allgemeingültig beantworten. Üblich bei der Klassifizierung sind jedoch die Auftretenshäufigkeit der Suchbegriffe und auch deren Position im Text. Je häufiger und früher ein Begriff auftaucht, desto höher ist in der Regel seine Bedeutung. Zusätzlich werten einige Suchmaschinen die Entfernung verschiedener Suchbegriffe voneinander aus, um die Trefferliste weiter zu sortieren. Google verwendet darüber hinaus die Anzahl der auf die Seite verweisenden Links, die wiederum verschiedene Wertigkeit haben. Dabei sind Links von sehr häufig verlinkten

Seiten höherwertiger als Links von Seiten, die kaum verlinkt wurden. Der von Google für das Ergebnis dieser Linkanalyse eingeführte Name dieser Bewertung lautet »Pagerank«.

Das Auswerten von Seiten wird auch als *Indizieren* bezeichnet und erzeugt enorme Datenmengen, die in der Regel in Datenbanken hinterlegt werden. Wie im letzten Abschnitt beschrieben, verwenden wir für unsere Testzwecke die Datenbank SQLite und greifen mit SQL auf die Daten zu. Für die produktive Nutzung einer Suchmaschine wäre eine so kleine, dateibasierte Datenbank natürlich nicht adäquat, aber für Tests ist sie eine sehr gute Alternative zu den großen Datenbank-Management-Systemen.

Im Folgenden sehen Sie die Initialisierung unseres Crawlers, wobei je nachdem, ob `createDatabase` auf True oder False steht, die Tabellen der Datenbank angelegt werden. Für das erste Starten sollten die Tabellen entsprechend angelegt werden. Alternativ zur Verwendung einer boolschen Variablen für die Erzeugung der Datenbank könnten Sie auch überprüfen, ob die angegebene Datei »crawler.db« bereits existiert oder nicht.

```
class spiderman(object):
    def __init__(self, createDatabase=False):
        self.connection  = sqlite3.connect("crawler.db")
        self.cursor      = self.connection.cursor()

        if createDatabase:
            self.cursor.execute("""CREATE TABLE websites
                                   (id INTEGER PRIMARY KEY,
                                    url TEXT)""")
            self.cursor.execute("""CREATE TABLE words
                                   (id INTEGER PRIMARY KEY,
                                    word TEXT)""")
            self.cursor.execute("""CREATE TABLE webIndex
                                   (  id INTEGER PRIMARY KEY,
                                      website INTEGER,
                                    word INTEGER, count INTEGER,
                                    position INTEGER)""")
            self.cursor.execute("""CREATE TABLE webLinks
                                    ( id INTEGER PRIMARY KEY,
                                      link INTEGER,
                                      outgoingLink INTEGER,
                                      description TEXT )""")
            self.connection.commit()
```

Listing 9.11 Initialisierung des Crawlers

Bei Erzeugung der Klasse wird lediglich eine Datenbankverbindung aufgebaut und falls nötig – wie oben erwähnt – die Struktur der Datenbank angelegt. Für die Datenhaltung benötigen wir drei Tabellen, in denen die Indizierung der Webseiten gespeichert wird. Dabei werden nur die Wörter mit ihrer Position und der Auftretenshäufigkeit sowie die Links abgelegt.

Die Nutzung der Klasse `spiderman` gestaltet sich wie folgt:

```
if __name__ == "__main__":
#  crawler = spiderman(createDatabase=True)
    crawler = spiderman()
    links   = ['http://www.chessbase.de']
    crawler.crawl(links)
    links   = crawler.find('Kasparov')
    for link in links:
        print "Adresse:   ", link[2]
        print "Vorkommen: ", link[0]
        print "Position:  ", link[1]
        print "--------------------------------"
```

Listing 9.12 Nutzung der Klasse »Spiderman«

Die auskommentierte Zeile wird nur für den ersten Aufruf benötigt und erzeugt die in diesem Fall noch nicht vorhandene Datenbankstruktur. Diese Art der Erzeugung ließe sich auch – wie oben bereits erwähnt – durch eine Suche nach der Datenbankdatei automatisieren, so dass beim Erzeugen der Klasse die Datenbankstruktur nur angelegt wird, wenn die Datenbankdatei nicht gefunden werden konnte.

Danach wird eine Linkliste erzeugt, deren Inhalte die Ausgangspunkte für die Indizierung des Webs darstellen. Die Instanz der Klasse `spiderman` wird dann über die Methode `crawl()` aufgerufen, um die Indizierung der in der Liste genannten Webseiten zu starten. Nach erfolgreicher Indizierung ist die Suche nach Wörtern über die Methode `find()` der Klasse `spiderman` möglich. Dabei werden alle Webseiten, die den gesuchten Begriff enthalten, in Listenform zurückgegeben. Die Methode `find()` sieht wie folgt aus:

```
def find(self, text):
# zunächst wird der Text in
# Kleinbuchstaben umgewandelt
    text = text.lower()
    # der übergebene Text wird
    # hier in der Datenbank der
    # indizierten Seiten gesucht
    sql = """ SELECT wi.count, wi.position, ws.url
              FROM webIndex wi
```

```
            JOIN websites ws ON ws.id=wi.website
            WHERE wi.word='""" + str(text) + "'"
    self.cursor.execute(sql)

    return self.cursor.fetchall()
```

Listing 9.13 Seiten anhand von Suchbegriffen finden

Die Daten der Indizierung wurden normalisiert, weshalb sie über verschiedene Tabellen verstreut liegen. Deshalb werden sie zunächst mittels `JOIN` wieder zusammengefügt und dann nach dem Suchkriterium gefiltert.

[+] Normalisierung

Normalisierung bezeichnet die schrittweise Zerlegung von Relation innerhalb eines Datenschemas. Sinn der Zerlegung ist die Reduzierung beziehungsweise Vermeidung von unnötiger Redundanz.

Die Methode `fetchall()` gibt daraufhin das Ergebnis der Abfrage als Liste zurück. Sobald Sie die Klasse `spiderman` produktiv einsetzen wollen, werden Sie hierbei jedoch ein Problem bekommen, denn die Ergebnislisten können sehr groß sein. In diesem Fall lässt sich die Ergebnismenge aber auch Schritt für Schritt zurückgeben, wobei nicht so viel Hauptspeicher reserviert werden muss. Für das schrittweise Auslesen der Ergebnisliste verwenden Sie einfach die Methode `fetchone()` der Klasse `cursor`.

Das Indizieren der Webinhalte erfolgt über die Methode `crawl()`:

```
def crawl(self, links, depth=2):
    if depth == 0:
        return

    newpages    = []
```

Listing 9.14 Start der Indizierung

Die Methode `crawl()` arbeitet rekursiv und ruft sich entsprechend selbst auf, weshalb zunächst die Abbruchbedingung für die Rekursion überprüft wird, da eine Rekursion ohne Abbruchbedingung in einer Endlosschleife enden würde. Einen Fehler bei der Definition der Abbruchbedingung einer rekursiven Funktion erkennen Sie leicht daran, dass ein Stackoverflow zustande kommt, der unter Python zu folgendem Fehler führt: »RuntimeError: maximum recursion depth exceeded«. Der dabei berücksichtigte Parameter `depth()` gibt die Tiefe der Indizierung an, also die Linktiefe. Eine Linktiefe von `2` heißt in diesem Fall, dass alle Links auf den übergebenen Seiten ebenfalls indiziert werden, genauso wie die

Links, die jeweils auf den Seiten der verlinkten Seiten stehen. Die Verlinkungen im Internet stellen im Prinzip so etwas wie einen Baum dar, wobei es jedoch zu Zyklen kommen kann; es handelt sich dementsprechend also nicht wirklich um einen Baum, sondern um einen Graphen. Zyklen im Graphen stellen allerdings kein Problem dar, da bereits indizierte Seiten nicht erneut indiziert werden.

Aber jetzt zum nächsten Abschnitt der Methode `crawl()`:

```
for link in links:
    if self.isNotIndexed(link):
        try:
            c = urllib2.urlopen(link)
        except:
            print "Link nicht erreichbar: %s" % link
            continue
        try:
            text = c.read()
            soup = BeautifulSoup(text)
        except Exception, arg:
            print "Fehler beim Parsen %s" % link
            print Exception, arg
            continue
```

Listing 9.15 Das Laden neuer Webseiten

Zunächst überprüfen wir, ob der übergebene Link bereits indiziert wurde. Falls der Link noch nicht indiziert ist, so wird versucht die Seite zu laden. Wenn die Seite korrekt geladen werden konnte, dann wird sie mit Hilfe von Beautiful Soup für die weitere Auswertung geparst.

Für alle übergebenen Links wird – wie gerade beschrieben – zunächst durch Aufruf der Methode `isNotIndexed()` überprüft, ob die Seite bereits indiziert wurde. Die Methode stellt sich wie folgt dar:

```
def isNotIndexed(self, link):
    self.cursor.execute("SELECT * FROM websites WHERE url='" +
     str(link) + "'")
    row = self.cursor.fetchone()
    if row:
        id  = row[0]
        self.cursor.
         execute("SELECT * FROM webIndex WHERE website=" +
         str(id))
        row = self.cursor.fetchone()
        if row:
            return False
```

```
        else:
            return True
    else:
        return True
```

Listing 9.16 Überprüfung, ob eine Webseite bereits indiziert wurde

Zunächst wird überprüft, ob bereits ein Eintrag für die Seite in die Tabelle `websites` erfolgte. Falls ja, so wird zusätzlich überprüft, ob es bereits Einträge für die auf der Seite auftretenden Wörter gibt. Die Bedingungen arbeiten ohne Vergleichsoperatoren, da Python die Rückgabe von `None` als `False` auswertet. Falls Sie solche Konstrukte verwirren, so können Sie die Bedingung auch in `if not (row == None)` umschreiben, was gleichbedeutend ist. Wenn noch keine Indizierung der Seite erfolgt ist, dann wird der Seiteninhalt eingelesen und mittels der Klasse `BeautifulSoup` geparst.

Sollte beim Parsen oder beim Einlesen der Seite ein Fehler auftreten, so wird dieser ausgegeben und die Seite übersprungen. Entsprechend geht die Indizierung dann einfach mit der nächsten Seite weiter. Es kann also vorkommen, dass Sie bestimmte Ihnen bekannte Inhalte nicht finden können, da die entsprechende Seite sich nicht laden oder parsen ließ. Im Allgemeinen ist der Parser aber sehr tolerant in Bezug auf fehlerhafte Webseiten – es mag aber hin und wieder dennoch zu Problemen kommen.

Aber weiter im Listing – wir sind immer noch nicht am Ende der Methode `crawl()` angekommen. Diejenigen unter Ihnen, die sich gerne mit Refactoring auseinandersetzen, finden hier wieder einen guten Ansatzpunkt. Die Methode ist eindeutig zu lang – es gibt immer etwas zu tun.

```
text        = self.getTextOnly(soup)
words       = self.separateWords
wordList    = self.countWords(words)
self.associateWordsWithLink(link, wordList)
```

Listing 9.17 Vorbereitung der Indizierung

Jetzt passiert eine Menge: Zunächst wird der Text aus dem HTML-Gerüst getrennt, was die Methode `getTextOnly()` übernimmt, deren Aufbau ich bereits im Zusammenhang mit `BeautifulSoup` erläutert habe. Aus dem resultierenden Text wird mit `separateWords()` eine Liste der enthaltenen Wörter erstellt:

```
def separateWords(self, text):
   # Sonderzeichen funktionieren so nicht,
   # aber um die Sonderfälle können Sie
   # sich später noch kümmern
```

```
    splitter=re.compile('\\W*')
    return [s.lower() for s in splitter.split(text) if s!='']
```

Listing 9.18 Den Text in Wörter unterteilen

Die Wortliste wird auf relativ einfache Weise erstellt, wobei viele Sonderfälle, die auch sinnvoll wären, unter den Tisch fallen. Was die Sonderfälle angeht, so werden zum Beispiel Worttrennungen nicht erkannt, aber auch interessante Inhalte wie »C++« werden nicht berücksichtigt. Falls Sie also die Indizierung besser gestalten wollen, so wäre hier der erste Ansatzpunkt. Auch die Performance lässt sich hier stark verbessern, da die Methode momentan Regular Expressions und eine dynamisch erweiterte Liste verwendet. Hier ging es mir zunächst nur um eine sehr einfache Möglichkeit, die funktioniert und nicht zu kompliziert zu verstehen ist. Der Einfachheit halber werden auch alle Wörter in Kleinbuchstaben umgewandelt, da die Groß-/Kleinschreibung in der Regel keine Rolle spielen sollte.

Mit der so erstellten Liste geht es dann zur Methode `countWords()`, die die Wörter allerdings nicht nur zählt, sondern auch deren Position vermerkt. Die Position eines Wortes auf einer Seite können wir so später auch als Relevanzkriterium heranziehen. Die Methode `countWords()` gestaltet sich wie folgt:

```
def countWords(self, words):
    mydict  =
    position = 0
    for word in words:
        if len(word) > 5:
            if mydict.has_key(word) == 1:
                mydict[word][0] = mydict.get(word)[0] + 1
            else:
                mydict[word] = [1, position]

            position += 1

    wordList = mydict.items()
    wordList = sorted(wordList, key=operator.itemgetter(1),
     reverse=True)

    return wordList
```

Listing 9.19 Zählen aller Wörter mit mehr als fünf Buchstaben

Zunächst wird ein Dictionary angelegt, das die einzelnen Wörter als Schlüssel, zusammen mit der Anzahl ihres Auftretens, hinterlegt. Beim Durchschreiten der Wortliste wird dann zunächst überprüft, ob es zum aktuellen Wort bereits einen

Eintrag im Dictionary gibt oder ob dieser erzeugt werden muss. Sollte das Wort vorher noch nicht aufgetreten sein, so wird der entsprechende Eintrag erzeugt und die Position des ersten Auftretens mit abgespeichert. Bei jedem weiteren Vorkommen wird dann entsprechend innerhalb des erzeugten Eintrags nur ein Zähler erhöht, der die Auftretenshäufigkeit wiedergibt.

Das Ergebnis des Methodenaufrufs ist die Wortliste, die die Wörter mit der Position des ersten Auftretens sowie der Auftretenshäufigkeit enthält. Diese Liste dient nun zur Indizierung der Seite, wofür wir die in Listing 9.20 definierte Methode `associateWordsWithLink()` einsetzen. Genau genommen übernimmt die Methode `associateWordsWithLink()` nur den Schritt der Speicherung der Daten in die dafür vorgesehenen Tabellen der Datenbank.

```
def associateWordsWithLink(self, link, wordList):
    website = self.getWebsiteId(link)
    if website == None:
        website = self.saveWebsite(link)
    for entry in wordList:
        word, params    = entry
        count, position = params
        print "word = ", word, "count = ", count, "pos = ", position
        sql = "INSERT INTO webIndex " +
         "(website, word, count, position)" +
          "VALUES (?, ?, ?, ?)"
        werte = website, word, count, position
        self.cursor.execute(sql, werte)
```

Listing 9.20 Speicherung der Indizierungsdaten einer Webseite

Aufgrund der bereits angesprochenen Normalisierung der Datenstruktur wird zunächst die passende ID zur aktuell zu indizierenden Seite ermittelt. Die Methode hierfür ist sehr einfach aufgebaut und bedient sich einer weiteren Methode für den Fall, dass es noch keine ID für die angeforderte Seite gibt:

```
def getWebsiteId(self, url):
    sql = "SELECT id FROM websites WHERE url='" + str(url) + "'"
    self.cursor.execute(sql)
    row = self.cursor.fetchone()

    if row == None:
        return self.saveWebsite(url)
    else:
        return row[0]
```

Listing 9.21 Abfrage des Primärschlüssels einer Webseite

Zugegebenermaßen nicht ganz konsistent wird innerhalb der Methode diesmal nicht die abgekürzte Version der Bedingung verwendet, sondern direkt auf `None` verglichen. Falls Ihnen die Kurzschreibweise gut gefallen hat, so ist diese auch möglich; dafür müssen Sie lediglich den `if`- und `else`-Zweig vertauschen.

Die Methode `saveWebsite()` legt einen neuen Eintrag in der Tabelle `websites` an und gibt die für den neuen Eintrag erzeugte ID zurück:

```
def saveWebsite(self, link):
    sql = "INSERT INTO websites (url) VALUES (?)"
    self.cursor.execute(sql, [str(link)])
    sql = "SELECT id FROM websites WHERE url='" +
     str(link) + "'"
    self.cursor.execute(sql)

    return self.cursor.fetchone()[0]
```

Listing 9.22 Speichern einer Webseite

Die gesonderte Abfrage des gerade erzeugten Eintrages ist eigentlich nicht notwendig, da es in der Regel möglich ist, die zuletzt vergebene ID abzufragen. Da es hier jedoch momentan nicht um darum geht die effizienteste aller Lösungen zu finden, soll uns die gesonderte Abfrage hier genügen. Die direkte Dereferenzierung nach dem Aufruf von `fetchone()` ist in diesem Fall auch in Ordnung, da aufgrund des vorherigen Aufrufs eine Ergebnismenge ungleich `None` sichergestellt ist. Falls Sie ein ausgesprochener Verfechter des defensiven Programmierens sind, so schadet natürlich auch eine zusätzliche Überprüfung dieser Bedingung nichts.

Kommen wir jetzt aber wieder zur Methode `crawl()`:

```
linksOnPage = soup('a')
for outgoingLink in linksOnPage:
    if ('href' in dict(outgoingLink.attrs)):
        url = outgoingLink['href']
        url = url.split('#')[0] # remove location portion (anker)
        if url[0:4]=='http' and self.isNotIndexed(url):
            newpages.append(url)
            description = self.getTextOnly(outgoingLink)
            self.saveLink(link, url, description)
```

Listing 9.23 Das Verfolgen der Links auf einer Webseite

Nach Abschluss der Indizierung der Seite werden alle auf der Seite auftretenden Links ermittelt und in der Liste `linksOnPage` hinterlegt, die danach für alle Elemente durchlaufen wird. Die Links werden dann einzeln auf ihren relevanten Anteil – also nur die URL ohne Beiwerk wie Anker oder Daten, die per GET

übergeben wurden – gekürzt. Anker sind Sprungpunkte innerhalb einer Seite, so dass ein Link nicht zwangsläufig auf den Start einer Seite zeigen muss. Diese Sprungpunkte sind aber irrelevant, da immer die gesamte Seite indiziert wird und nicht nur der Abschnitt, der angesprungen wird.

Nach dem eventuell notwendigen Entfernen des Ankers wird überprüft, ob es sich um eine absolute Adressangabe handelt; »absolut« heißt in diesem Fall vollständig, also inklusive des Domainnamens. Diese Überprüfung stellt sicher, dass relative Links – also Links ohne Angabe der Domain – nicht indiziert werden. Diesbezüglich wäre eine Anpassung eventuell sinnvoll, da aufgrund der häufig anzutreffenden relativen Links vermutlich viele Unterseiten einer Domain nicht indiziert werden. Als Alternative können Sie an dieser Stelle natürlich auch den absoluten Link aus den vorhandenen Informationen ableiten und genauso verwenden. Diese einfachere Vorgehensweise des Verwerfens erspart im Grunde nicht viel, sondern nur das Zusammenfügen der aktuellen Domain mit dem relativen Pfad. Wobei Sie hier noch Sonderfälle wie JavaScript-Links berücksichtigen sollten, die aber unter normalen Umständen auch keine großen Probleme bereiten. Zu den absoluten Links wird dann noch der den Link beschreibende Text mit der Methode `getTextOnly()` ermittelt. Alles zusammen wird schließlich durch Aufruf von `saveLink()` in der Tabelle `webLinks` abgespeichert.

```
def saveLink(self, link, outgoingLink, description):
    sql     = "INSERT INTO webLinks" +
     " (link, outgoingLink, description)" +
     " VALUES (?, ?, ?)"
    werte   = self.getWebsiteId(link),
     self.getWebsiteId(outgoingLink),
     description
    self.cursor.execute(sql, werte)
```

Listing 9.24 Speichern eines Links

Die Methode `saveLink()` speichert zu jeder Webseite die darauf vorhandenen Links zu anderen Seiten. Diese Informationen verwenden Sie momentan zwar noch nicht, aber später ließe sich hiermit zum Beispiel ein vereinfachtes Pageranking anhand der verlinkenden Seiten durchführen. Das Zählen der Links zu einer Seite ist in dieser einfachen Form relativ leicht: Alles, was Sie benötigen, ist eine Abfrage, die für die Spalte `outgoingLink` die entsprechende Webseite als Bedingung vorschreibt und dann die Links ausgibt, die übrig bleiben.

9.5 Was denn noch?

Gerade in diesem Kapitel ist die Frage, wie es weitergehen könnte, sehr leicht zu beantworten, denn es gibt noch viele Möglichkeiten zu entdecken. Unter anderem könnten Sie das eben erwähnte Pageranking einführen, das die Ergebnisse bei Suchanfragen sortiert. Oder sortieren Sie erst einmal die Ergebnisse anhand verschiedener Kriterien, beispielsweise nach der Häufigkeit des Auftretens oder nach dem ersten Auftreten

Ein weiteres Problem, dem Sie sich annehmen könnten, ist die Gewichtung verschiedener in die Sortierung einfließender Kriterien. Wenn eine vernünftige Gewichtung möglich ist, lassen sich so auch verschiedene Relevanzkriterien miteinander kombinieren. Bis jetzt hätten Sie hierfür drei Kriterien, aber die Möglichkeiten sind nahezu unbegrenzt. Falls Ihnen das noch nicht reicht, so beziehen Sie bei der Suche nach mehreren Wörtern auch den Abstand zwischen den verschiedenen Suchbegriffen ein. Dies ist auch mit den bis jetzt gespeicherten Daten möglich, da für den Abstand zwischen zwei Suchbegriffen die Position des ersten Auftretens ausreicht. Eventuell tritt das Wort später zwar noch einmal in viel geringerem Abstand zu dem anderen Begriff auf, aber diesen Spezialfall könnten Sie fast vernachlässigen.

Bevor Sie neue Ideen umsetzen, könnten Sie auch ein bisschen aufräumen, denn es gibt so manche Stelle, die schon etwas unübersichtlich ist oder einfach nicht so performant umgesetzt wurde, wie es möglich wäre. Beim Aufräumen des Codes werden Sie ihn auch noch besser verstehen als nur beim Lesen dieses Kapitels. Idealerweise testen Sie jede kleine Änderung sofort; dann fällt Ihnen gleich auf, was Sie verbessert haben. Eine Möglichkeit für Performancegewinne bieten Ihnen die List Comprehensions, welche Sie durch Verwendung funktionaler Eigenschaften von Python ersetzen können. Probieren Sie das doch einfach mal aus ...

Ideen gibt es genug – viel Spaß beim eigenen Experimentieren!

»Raffiniert ist der Herrgott, aber boshaft ist er nicht.«

Albert Einstein

10 Pendelkette – klack, klack, klack

In diesem Kapitel programmieren wir eine Pendelkette. Falls Sie sich gerade fragen, was das sein könnte: Es handelt sich dabei um mehrere hintereinander aufgehängte Kugeln, die aneinanderstoßen und sich dabei in Bewegung halten. Abbildung 10.1 zeigt das Prinzip. Wobei dies die einfache Version des Pendels ist – im Laufe des Textes werde ich natürlich noch Verbesserungsmöglichkeiten ansprechen.

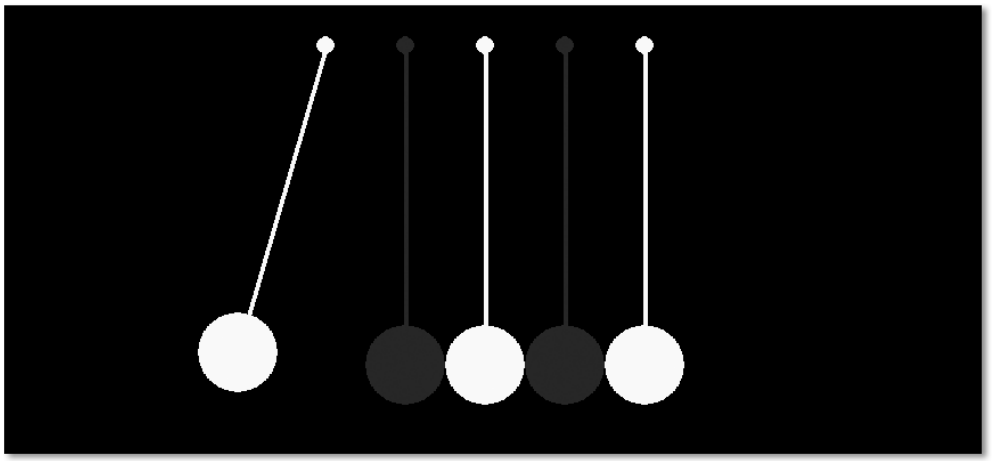


Abbildung 10.1 Pendelkette

10.1 Ein einzelnes Pendel

Wir beginnen wie üblich unsere Implementierung so einfach wie möglich. Deshalb erstellen wir zunächst ein Einzelpendel ohne die anderen für die Kette noch notwendigen Pendel. Dazu stellen wir erst einmal ein paar Grundüberlegungen an: Wie sieht die Bahn der Kugel aus? Ganz offensichtlich handelt es sich um eine Kreisbahn, so dass die Berechnungen, die wir bereits im ersten Kapitel, »Es schlägt 12«, für die Uhrzeiger angestellt haben, hier ebenfalls zur Anwendung kommen. Darüber hinaus soll das Pendel einer Dämpfung unterliegen, die sich in der realen Welt aus dem Luftwiderstand und den Reibungsverlusten ergibt. Es

geht hier aber nicht um die exakt reale Wiedergabe eines Pendels, denn das ist nahezu unmöglich.

Wir starten zunächst mit einer Näherung für die Berechnungen der Pendelpositionen in Abhängigkeit von der Zeit. Im Fachjargon wird ein solches Pendel auch als *mathematisches Pendel* bezeichnet, da es nur ein mathematisches Modell ist, das nicht mit der Realität übereinstimmt. Die Darstellung soll zunächst nur mit der vereinfachten 2D-API von `pygame` erfolgen.

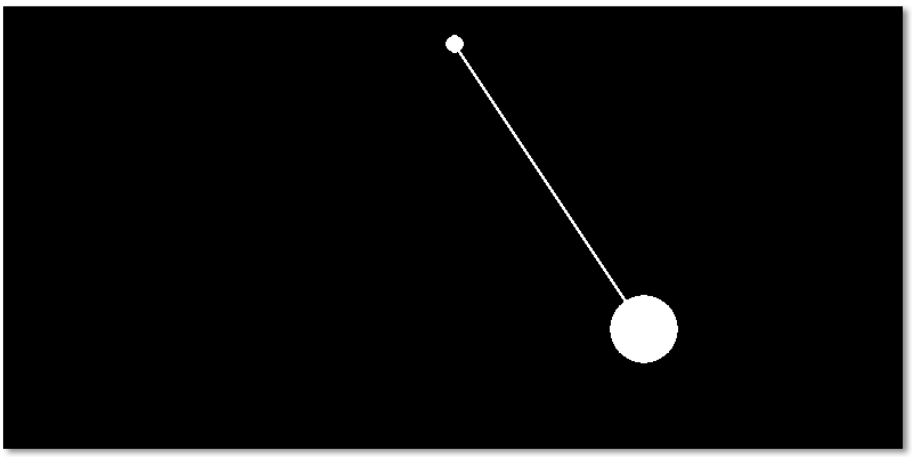


Abbildung 10.2 Einzelnes Pendel

```
def main():
   # Initialise screen
    global screen, alpha
    pygame.init()
    pygame.display.set_caption('Pendel-Simulation')
    screen = pygame.display.set_mode(
     (windowWidth, windowHeight),
      pygame.HWSURFACE | pygame.DOUBLEBUF)

   # Event loop
    delay   = 20
    time    = 0.0
    alpha   = 45.0
    while True:
        handleEvents()
        screen.fill(backgroundColor)

        drawPendulum(white, 3, 300, alpha, 360)
        alpha    = calculateDelta(time)
        time    += 0.02
```

```
        pygame.display.flip()
        pygame.time.delay(delay)
```

Listing 10.1 Der Hauptablauf für die Pendel-Simulation

Die Hauptablaufroutine erzeugt einen neuen `screen`, der für die Darstellung des Pendels zuständig ist. Das eigentliche Zeichnen des Pendels erfolgt innerhalb der Endlosschleife. Die Hardwarebeschleunigung der Grafikkarte wird durch die Option `pygame.HWSURFACE` aktiviert. Zusätzlich wird mittels `pygame.DOUBLEBUF` das Double Buffering aktiviert, um eine flüssige Darstellung zu sichern. Der Auslenkungswinkel des Pendels wird durch die Variable `alpha` bestimmt und mit 45 Grad initialisiert. In der Variablen `time` wird die vergangene Zeit hinterlegt und beim Aufruf von `calculateDelta()` übergeben. Daraufhin ermittelt die Funktion `calculateDelta()` den aktuellen Auslenkungswinkel des Pendels. Der Aufruf `pygame.display.flip()` sorgt für die Darstellung der neuen Pendelposition, welche zuvor durch Aufruf von `drawPendulum()` in den Hintergrundpuffer gezeichnet worden ist. Das Programm lässt sich über jedes beliebige Tastatur- oder Mausereignis wieder schließen. Die Ereignisbehandlung übernimmt wie üblich die Funktion `handleEvents()`, die sich folgendermaßen darstellt:

```
def handleEvents():
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit(0)
        elif event.type == pygame.KEYDOWN:
            sys.exit(0)
        elif event.type == pygame.MOUSEBUTTONDOWN:
            sys.exit(0)
```

Listing 10.2 Die Ereignisbehandlung

Innerhalb der `while`-Schleife wird bei jedem Durchgang zunächst der Bildschirm gelöscht und dann mit der neuen Position des Pendels bemalt. Dabei berechnet die Funktion `calculateDelta()` die Position des Pendels je Iteration neu:

```
def calculateDelta(time, alpha=45.0, gravity=9.81, wireLength=5.0):
    # dampingConstant = 1.0/20.0
    # e^(-dampingConstant*time) = 1.0/e^(dampingConstant*time)
    damping = 1.0/math.exp(time/20.0)
    alpha = alpha*math.cos(
     math.sqrt(gravity/wireLength)*time)*damping

    return alpha
```

Listing 10.3 Berechnung des aktuellen Auslenkungswinkels

Was genau hinter der Berechnung steckt, können Sie im nächsten Abschnitt lesen. Falls Sie die mathematischen Hintergründe nicht so sehr interessieren sollten, so können Sie den Abschnitt auch problemlos überspringen.

10.2 Das mathematische Pendel

Die Berechnung des aktuellen Auslenkungswinkels nutzt das Modell des mathematischen Pendels, das nur eine Näherung darstellt, welche entsprechend nicht mit der Realität übereinstimmt. Die Dämpfung wird exponentiell eingerechnet, wobei die `dampingConstant` die Stärke der Dämpfung bestimmt. Die Herleitung der Näherung ergibt sich aus den folgenden Sachverhalten: Entsprechend des 2. Newtonschen Gesetzes, das besagt, dass die Kraft proportional zur Beschleunigung ist – also $F = m * a$, lässt sich die Herleitung der Näherung für die Pendelschwingung starten. Da das Pendel in einem Punkt fixiert ist und ansonsten nur die Schwerkraft auf das Pendel wirkt, ergibt sich über ein einfaches Kräfteparallelogramm die resultierende Kraft, die auch über die folgende Formel berechnet werden kann:

$$F(t) = -m * g * \sin(\phi(t))$$

Das Kräfteparallelogramm ist aufgrund des 3. Newtonschen Axioms – »actio et reactio« – eine gleichseitige Raute. Die Formel ergibt sich aus den Verhältnissen, die in Abbildung 10.3 dargestellt sind.

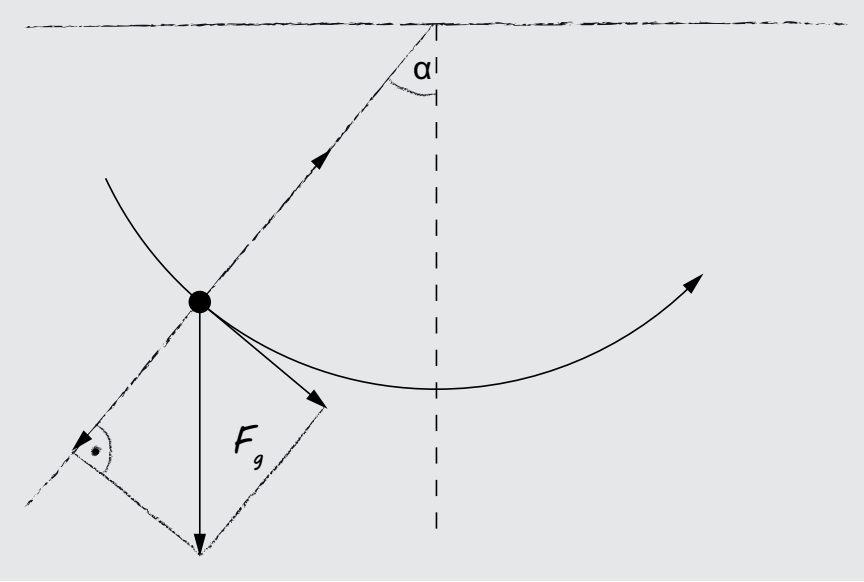


Abbildung 10.3 Kräfteparallelogramm beim Fadenpendel

Die zweite Ableitung der zurückgelegten Distanz $l\Phi(t)$ des Pendels in Abhängigkeit von der Zeit ergibt die Beschleunigung:

$$a(t) = l * \ddot{\phi}(t)$$

Da zusätzlich $F = m * a$ gilt, lässt sich folgende Beziehung aufstellen:

$$m * l * \ddot{\phi}(t) = -m * g * \sin(\phi(t))$$

Nach entsprechenden Umformungsschritten erhalten wir schließlich die folgende nichtlineare Differentialgleichung zweiter Ordnung, die den Winkel in Beziehung zur Zeit setzt:

$$\ddot{\phi}(t) = -\frac{g}{l} * \sin(\phi(t))$$

Für kleine Auslenkungswinkel lässt sich die Kleinwinkelnäherung $\sin(\phi) \approx \phi$ nutzen und linearisieren, wobei »klein« in diesem Fall für Winkel kleiner gleich 5 Grad steht:

$$\ddot{\phi}(t) \approx -\frac{g}{l} * \phi(t)$$

Die durch die Annäherung entstandene Gleichung ist die Differentialgleichung der harmonischen Schwingung. Die allgemeine Lösung dieser Gleichung sieht so aus:

$$\phi(t) = a * \cos(\sqrt{\frac{g}{l}} * t) + b * \sin(\sqrt{\frac{g}{l}} * t)$$

Äquivalent zu dieser Gleichung ist auch die folgende vereinfachte Gleichung, die sich durch ein paar Umformungen ermitteln lässt:

$$\phi(t) = \phi_{max} * \cos(\sqrt{g}l * t + \alpha)$$

Die Bezeichner a, b, Φ_{max} und α sind dabei Konstanten, die die Anfangsbedingungen beschreiben. Wie bereits erwähnt ist diese Näherung allerdings nicht sehr nahe an der Realität, da wir in der Regel Auslenkungen simulieren, die mit Winkeln größer 5 Grad starten. Die korrekte Berechnung der Pendelbewegung ist unter Zuhilfenahme von Kenntnissen über elliptische Integrale möglich. Die allgemeine Lösung lässt sich damit über folgende Reihe ermitteln:

$$T\phi = 2 * \pi\sqrt{\frac{l}{g}} * (1 + (\frac{1}{2})^2 * sin^2(\frac{\phi}{2}) + (\frac{1*3}{2*4})^2 * sin^4(\frac{\phi}{2}) + ...)$$

Die Dämpfung durch den Luftwiderstand sowie durch Reibungsverluste ist ungefähr exponentiell und wird in der Implementierung entsprechend berücksichtigt. Die zeitlich abhängige Dämpfung ermittelt sich dann folgendermaßen:

$$D(t) = e^{-k*t}$$

k ist dabei die Dämpfungskonstante, über die die Stärke der Dämpfung festgelegt ist.

Soviel zur Mathematik – im nächsten Abschnitt werden wir das Pendel darstellen.

10.3 Grafische Darstellung des Pendels

Im letzten Abschnitt wurde es ein bisschen mathematisch, aber jetzt kommen wir wieder zu einfacheren Aufgaben, wie zum Beispiel dem Zeichnen des Pendels:

```
def drawPendulum(color, width, length, position, scale):
    end = getCirclePoint(position, scale, length);
    pygame.draw.line(screen, color, windowCenter, end, width)
    pygame.draw.circle(screen, white, end, 30)
    pygame.draw.circle(screen, white, windowCenter, 8)
```

Listing 10.4 Zeichnen des Pendels

Die Funktion `drawPendulum()` zeichnet das Pendel entsprechend dem in `position` angegebenen Winkel. Falls Sie das erste Kapitel, »Es schlägt 12«, gelesen haben, so wird Ihnen die Funktion `getCirclePoint()` bereits bekannt vorkommen, denn sie ist Ihnen im ersten Kapitel bereits begegnet:

```
def getCirclePoint(position, scale, radius):
    degrees = getWirePositionDegrees(position, scale)
    xPos = int(round(math.cos(degrees/180.0*math.pi)*
      radius+windowCenter[0]))
    yPos = int(round(math.sin(degrees/180.0*math.pi)*
      radius+windowCenter[1]))
    return (xPos, yPos)
```

Listing 10.5 Bestimmung der Endposition des Pendels zu einem vorgegebenen Winkel

Wie der Name schon sagt, wird ein Punkt auf dem Kreis ermittelt, der entsprechend der Variablen `radius` vom Mittelpunkt des Kreises entfernt liegt. Die Position wird durch `getWirePostionDegrees()` anteilig zur Skala ermittelt:

```
def getWirePositionDegrees(position, scale):
    offset = -270
    degrees = 360 / scale * position + offset
    return degrees
```

Listing 10.6 Ermittlung des Auslenkungswinkels

Damit wären wir fertig mit dem Zeichnen des Pendels und hätten alle das Einzel-Pendel betreffenden Probleme gelöst, wenn unsere Implementierung auch, wie gesagt, für Auslenkungen mit Winkeln größer fünf Grad nicht sehr genau ist. Eine genauere Lösung wäre über die Reihenbildung wie oben beschrieben möglich oder über ein numerisches Verfahren wie zum Beispiel das Runge-Kutter-Verfahren. Weiterführenden Informationen finden Sie im Literaturverzeichnis unter dem Verweis [Press 2007].

10.4 Abstraktion des Pendels

Wir haben also die grundsätzliche Funktionalität und schreiben nun eine eigene Klasse für die Repräsentation des Pendels, die sowohl die Darstellung als auch die Berechnungen für das Pendel übernehmen wird. Diese Aufgaben auszulagern, ist insofern sinnvoll, als wir daraufhin mehrere Pendel leicht mit der neuen Klasse darstellen lassen können.

```
import math, pygame
# dies ist lediglich ein mathematische Pendel -
# ein physikalisches Pendel benötigt genauere Berechnungen
class Pendulum(object):
    def __init__(self, alphaMax, wireLength, wireSickness,
      ballSize, gravity, dampingConstant,
      color, fixPoint):
        self.__alphaMax         = alphaMax
        self.__wireLength       = wireLength
        self.__wireThickness    = wireThickness
        self.__ballSize         = ballSize
        self.__gravity          = gravity
        self.__dampingConstant  = dampingConstant
        self.__color            = color
        self.__fixPoint         = fixPoint
```

Listing 10.7 Initialisierung der Klasse »Pendulum«

Initialisiert wird die Klasse `Pendulum` mit dem maximalen Auslenkungswinkel `alphaMax` des Pendels, der Länge des Pendelfadens `wireLength`, der Dicke des Pendelfadens `wireThickness`, der Größe des Pendels `ballSize`, der Gravitationskonstanten `gravity`, der Dämpfungskonstanten `dampingConstant`, der Farbe `color` des Pendels sowie dem Punkt `fixPoint`, an dem das Pendel aufgehängt ist.

```
def calculateAlpha(self, time):
    # dampingConstant = 1.0/20.0
    # e^(-dampingConstant*time) = 1.0/e^(dampingConstant*time)
    damping = 1.0/math.exp(self.__dampingConstant*time)
    alpha   = self.__alphaMax*math.cos(
     math.sqrt(self.__gravity/self.__wireLength)*
     time)*damping

    return alpha
```

Listing 10.8 Berechnung des Auslenkungswinkels

Die Methode `calculateAlpha` berechnet den Auslenkungswinkel in Abhängigkeit von der Zeit. Dabei wird eine Dämpfung entsprechend der bei der Initialisierung übergebenen Dämpfungskonstante `dampingConstant` berücksichtigt. Die Dämpfung erfolgt exponentiell abhängig von der Zeit. Der Winkel `alpha` wird entsprechend der mathematischen Näherung berechnet, die allerdings nur für kleine Auslenkungswinkel realistisch erscheint. Dennoch wird das Pendel auch bei größeren Auslenkungswinkeln an sich ganz gut ausschauen, wenn es sich auch nicht genau wie ein physikalisches Pendel bewegt.

```
def draw(self, screen, position, scale):
    end = self.__getCirclePoint(position,
      scale, self.__wireLength);
    pygame.draw.line(screen, self.__color,
      self.__fixPoint, end,
      self.__wireSickness)
    pygame.draw.circle(screen, self.__color,
      end, self.__ballSize)
    pygame.draw.circle(screen, self.__color,
      self.__fixPoint,
      self.__wireSickness*2)
```

Listing 10.9 Zeichnen des Pendels

Das Zeichnen erfolgt analog zu den Uhrzeigern aus dem ersten Kapitel (vgl. Listing 1.8), weshalb ich die Parameter `position` und `scale` so beibehalten habe. Hier können Sie aber auch eine Version der Methode `draw()` schreiben, der Sie nur den Winkel `alpha` übergeben.

```
def __getCirclePoint(self, position, scale, radius):
    degrees = self.__getWirePositionDegrees(position, scale)
    xPos = int(round(math.cos(degrees/180.0*math.pi)*
      radius+self.__fixPoint[0]))
    yPos = int(round(math.sin(degrees/180.0*math.pi)*
      radius+self.__fixPoint[1]))
    return (xPos, yPos)
```

Listing 10.10 Den zum Auslenkungswinkel passenden Punkt ermitteln

Auch die Methode `getCirclePoint()` ist wie die vergleichbare Methode aus dem ersten Kapitel (vgl. Listing 1.4) aufgebaut und bestimmt einen Punkt auf der Kreisbahn abhängig vom Radius und vom Winkel. Wegen der Übernahme der Funktion aus dem Uhrenkapitel wird hier mit `position` und `scale` gearbeitet. Natürlich ließe sich diese Methode auf die zwei Parameter `alpha` und `radius` reduzieren, da hier die Skala immer 360 Grad beträgt.

```
def __getWirePositionDegrees(self, position, scale):
    offset = -270
    degrees = 360 / scale * position + offset
    return degrees
```

Listing 10.11 Ermittlung des Auslenkungswinkels

Die Methode `getWirePositionDegrees()` ermittelt anhand des gewünschten Auslenkungswinkels – der hier wieder über die zwei Parameter `position` und `scale` dargestellt wird – den für die Darstellung korrekten Winkel. Im Uhrenkapitel haben wir damit unterschiedliche Skalen realisiert; hier dient die Methode nur der Verrechnung der Konstanten `offset`, die festlegt, dass der Winkel 0 Grad entsprechend dem Lot des Pendels definiert ist, so dass 0 Grad einem gerade nach unten hängendem Pendel entsprechen.

10.5 Die Wirkung der Parameter

In diesem Abschnitt betrachten wir die Auswirkungen der Parameter näher. Dazu bedienen wir uns zweier Pendel, die verschiedene Fadenlängen haben. Für die Umsetzung kommt uns die neue Klasse `pendulum` zur Hilfe, die mittels zweier Instanzen die beiden unterschiedlichen Pendel repräsentieren wird.

```
import pygame, sys
from pendulum import Pendulum

windowMargin    = 30
windowWidth     = 800
```

```
windowHeight    = 400
fixPoint1       = windowWidth/2-120, 40
fixPoint2       = windowWidth/2+200, 40
backgroundColor = (0, 0, 0)
color1          = (250, 0,   0)
color2          = (  0, 0, 250)
```

Listing 10.12 Parameterdefinitionen

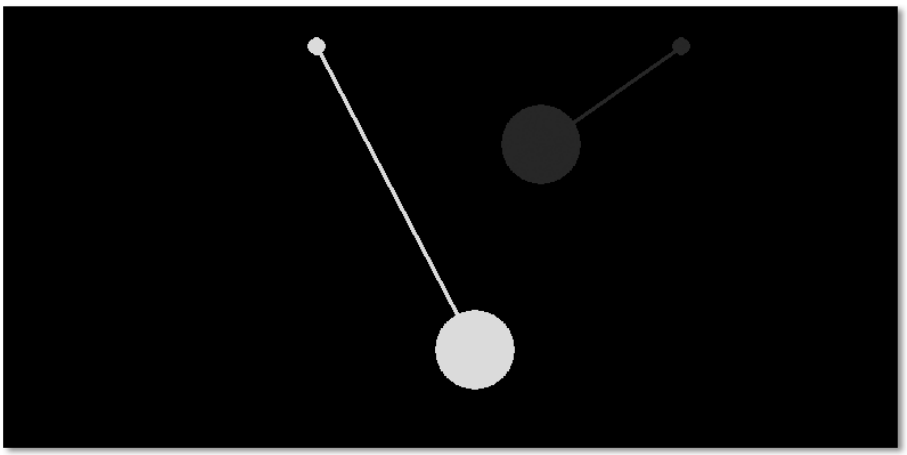


Abbildung 10.4 Vergleich zweier Pendel

Wir importieren die neue Klasse und definieren die notwendigen Parameter, die in der darauffolgenden Hauptroutine zum Einsatz kommen. Die beiden Befestigungspunkte der Pendel legen wir als Abstand vom horizontalen Fenstermittelpunkt fest. Den vertikalen Abstand vom oberen Fensterrand setzen wir auf 40 Pixel. Die Pendel werden mit `color1` und `color2` unterschiedlich eingefärbt, so dass Sie ein rotes und ein blaues Pendel sehen werden.

```
def main():
    # Initialise screen
    global screen
    pygame.init()
    pygame.display.set_caption('Pendel-Simulation')
    screen = pygame.display.set_mode(
     (windowWidth, windowHeight),
      pygame.HWSURFACE | pygame.DOUBLEBUF)

    # Event loop
    delay           = 20
    time            = 0.0
    wireThickness   = 4
    ballSize        = 35
    gravity         = 9.81
    dampingConstant = 1.0/500.0
```

```
alpha1          = 45.0
alpha2          = 70.0
wireLength1     = 300.0
wireLength2     = 150.0
pendulum1       = Pendulum(alpha1, wireLength1, wireThickness,
  ballSize, gravity, dampingConstant,
  color1, fixPoint1)
pendulum2       = Pendulum(alpha2, wireLength2, wireThickness,
  ballSize, gravity, dampingConstant,
  color2, fixPoint2)
```

Listing 10.13 Bildschirmvorbereitung und Initialisierung der Pendel

In der Hauptroutine werden weitere Parameter festgelegt, die die zwei Pendel beschreiben. Die Pendel starten mit unterschiedlichen Auslenkungswinkeln. Wie Sie sehen werden, schwingt das Pendel mit dem kürzeren Faden schneller als das mit dem längeren Faden. Spielen Sie ruhig ein bisschen mit den Einstellungen, und schauen Sie sich an, wie sich die verschiedenen Parameter auf das Pendel auswirken.

```
while True:
    handleEvents()
    screen.fill(backgroundColor)

    pendulum1.draw(screen, alpha1, 360)
    pendulum2.draw(screen, alpha2, 360)
    alpha1 = pendulum1.calculateAlpha(time)
    alpha2 = pendulum2.calculateAlpha(time)
    time    += 0.1

    pygame.display.flip()
    pygame.time.delay(delay)
```

Listing 10.14 Die Hauptablaufschleife für die Pendelaktualisierung

Die Hauptroutine zeichnet die beiden Pendel je Durchlauf und berechnet ihre Position neu. Dabei gibt es für jedes Pendel eine Variable `alpha1` und `alpha2` für den Auslenkungswinkel, die unabhängig aktualisiert werden kann. Dies ist schon aufgrund der unterschiedlichen Schwingungsgeschwindigkeit nicht anders möglich. Momentan ist die Zeit noch an die Framerate gekoppelt, so dass die Pendel auf unterschiedlich schnellen Rechnern unterschiedlich schnell schwingen, wenn die Framerate ins Stocken geraten sollte.

10.6 Eine Pendelkette für den Tisch

Mit den bis hierher erstellten Klassen und Hintergrundinformationen werden wir nun eine Pendelkette programmieren, die aus insgesamt fünf Kugeln bestehen wird.

Die Zusammenstöße der Kugeln werden wir dabei etwas vereinfacht auswerten. Statt hier den Impulserhaltungssatz oder weiterreichende physikalische Simulationen anzustreben, missbrauchen wir einfach den Auslenkungswinkel für einen kleinen Trick.

```
windowMargin    = 30
windowWidth     = 870
windowHeight    = 400
fixPoint1       = 290, 40
fixPoint2       = 360, 40
fixPoint3       = 430, 40
fixPoint4       = 500, 40
fixPoint5       = 570, 40
backgroundColor = (0, 0, 0)
color1          = (250, 0,   0)
color2          = (  0, 0, 250)
```

Listing 10.15 Parameter für die Pendelkette

Zunächst definieren wir alle Fixpunkte, die für die einzelnen Pendel gelten sollen. An den Definitionen für die Fenstergröße hat sich nichts geändert, und auch die Farben bleiben bei Grün und Blau; im Prinzip ließen sich auch alle Pendel in einer Farbe darstellen.

```
alpha1          = 45.0
alpha2          = 0.0
alpha3          = 0.0
pendulum1       = Pendulum(alpha1, wireLength, wireThickness,
 ballSize, gravity, dampingConstant,
 color1, fixPoint1)
pendulum2       = Pendulum(alpha2, wireLength, wireThickness,
 ballSize, gravity, dampingConstant,
 color2, fixPoint2)
pendulum3       = Pendulum(alpha2, wireLength, wireThickness,
 ballSize, gravity, dampingConstant,
 color1, fixPoint3)
pendulum4       = Pendulum(alpha2, wireLength, wireThickness,
 ballSize, gravity, dampingConstant,
 color2, fixPoint4)
pendulum5       = Pendulum(alpha2, wireLength, wireThickness,
```

```
ballSize, gravity, dampingConstant,
color1, fixPoint5)
```

Listing 10.16 Initialisierung der Pendel

Für den bereits angesprochenen kleinen »Trick« definieren wir drei Winkel. Es handelt sich dabei um die Winkel, die die maximale Auslenkung für die jeweiligen Pendel definieren. Den ersten Winkel ordnen wir dem ersten Pendel zu, also dem Pendel ganz links. Den zweiten Winkel weisen wir allen mittleren Pendeln zu, also den drei Pendeln in der Mitte. Der dritte Winkel gehört zum letzten Pendel ganz rechts.

Die Vereinfachung der Abläufe ergibt sich nun dadurch, dass, wenn `alpha1` Null Grad erreicht, `alpha1` mit `alpha2` vertauscht wird. Dadurch wird dem letzten Pendel nun der gleiche Freiheitsgrad eingeräumt wie vorher nur dem ersten Pendel. Zugleich wird der Freiheitsgrad des ersten Pendels auf 0 Grad reduziert, weshalb es in der Ruhestellung verharrt. Dies ist zugegebenermaßen nicht gerade sehr physikalisch, aber das dabei sichtbare Verhalten ist nicht stärker verfälscht, als es dies nicht ohnehin schon durch die Näherung ist. In der Realität ginge natürlich durch den Zusammenstoß noch mehr Energie verloren, die in Reibungs- und Verformungsenergie umgesetzt würde. Diesen Effekt könnten Sie aber leicht zusätzlich bei der Dämpfung berücksichtigen; dazu müssten Sie lediglich die Anzahl der Zusammenstöße bei der Dämpfungsberechnung zusätzlich berücksichtigen.

```
while True:
    handleEvents()
    screen.fill(backgroundColor)

    pendulum1.draw(screen, alpha1, 360)
    pendulum2.draw(screen, alpha2, 360)
    pendulum3.draw(screen, alpha2, 360)
    pendulum4.draw(screen, alpha2, 360)
    pendulum5.draw(screen, alpha3, 360)
    alpha1 = pendulum1.calculateAlpha(time)
    alpha3 = pendulum2.calculateAlpha(time)
    time   += 0.2

    if alpha1 < 0:
        alpha1, alpha3 = alpha3, alpha1

    pygame.display.flip()
    pygame.time.delay(delay)
```

Listing 10.17 Hauptablauf für die Pendelaktualisierung

Die einzelnen Pendel werden gezeichnet, wobei die Neuberechnung der Winkel nur für `alpha1` und `alpha3` notwendig ist. Der Trick wird durch die `if`-Bedingung realisiert, die wie bereits beschrieben die Winkel vertauscht und damit die Freiheitsgrade der jeweiligen Pendel anpasst. Falls mehrere Pendel mitschwingen sollen, müssen Sie lediglich den zusätzlich zu beteiligenden Pendeln den Winkel `alpha1` beziehungsweise `alpha3` zuweisen. So wäre es zum Beispiel leicht möglich, auch zwei Pendel ausschwingen zu lassen.

```
pendulum1.draw(screen, alpha1, 360)
pendulum2.draw(screen, alpha1, 360)
pendulum3.draw(screen, alpha2, 360)
pendulum4.draw(screen, alpha3, 360)
pendulum5.draw(screen, alpha3, 360)
```

Listing 10.18 Zeichnen der Pendel

In diesem Fall schwingen zwei Pendel aus, da den ersten beiden Pendeln der Winkel `alpha1` und den letzten beiden Pendeln der Winkel `alpha3` zugewiesen wurde.

10.7 Pendeln Sie doch weiter!

Momentan ist die grafische Darstellung der Pendel noch recht dürftig, aber das lässt sich leicht ändern. Sie können die Pendel natürlich auch im 3D-Raum schwingen lassen. Hierfür ist nicht viel notwendig – Sie müssen lediglich einen OpenGL-Kontext anfordern und benutzen. Für das Zeichnen einer Kugel lassen sich vorgefertigte Funktionen aus der GLUT-Bibliothek verwenden. Zudem können Sie auch Soundeffekte ins Spiel bringen, welche die Zusammenstöße der Kugeln verdeutlichen und entsprechend bei kleineren Auslenkungswinkeln leiser werden. Die Kugeln selbst lassen sich auch mit Materialeigenschaften belegen, so dass sie metallisch darstellbar sind. Die schwierigste, aber gleichzeitig interessanteste Erweiterung wäre die Benutzerinteraktion, so dass der Benutzer selbst die Startauslenkung und Anzahl der ausgelenkten Pendel bestimmen kann.

»Auch ein perfektes Chaos ist etwas Vollkommenes.«

Jean Genet

11 SOS – Save our Screens

In diesem Kapitel kümmern wir uns um den erholsamen Schlaf für unsere Bildschirme – es geht also um schöne Bildschirmschoner. Obwohl Bildschirmschoner in der heutigen Zeit, wo LCD-Bildschirme gang und gäbe sind, ihren eigentlichen Nutzen verloren haben, sind sie dennoch sehr beliebt, auch wenn es dabei nur um schöne Animationen am Bildschirm geht. Die Vielfalt an Bildschirmschonern ist inzwischen immens – auf dem einen Schreibtisch steht der Bildschirm in Flammen, auf dem anderen hat er sich gerade in ein Aquarium verwandelt; es ist grafisch in der Regel sehr ansprechend, was sich heutzutage dem Benutzer an Bildschirmschonern anbietet.

Sehr beliebt und praktisch sind auch jene Bildschirmschoner, die das Fotoalbum auf den Bildschirm zaubern – dabei erspart man sich gleich die ansonsten nutzlosen LCD-Bilderrahmen. Die verbrauchen zwar weniger Strom, aber so häufig schaut man in der Regel doch nicht darauf, so dass ein entsprechender Bildschirmschoner nicht das Gleiche in der Mittagspause bieten könnte.

Zum Schluss des Kapitels wird es noch ein bisschen philosophisch: Wir beschäftigen uns mit der Frage, ob alles eher vorhersagbar oder rein zufällig geschieht. Dabei werden Sie unter anderem Fraktale kennenlernen, die vollkommen chaotisch wirken, aber dennoch wiederkehrende Strukturen erkennen lassen. Fraktale stellen damit ein gutes Beispiel für das aus einfachen Regeln möglicherweise herausbrechende Chaos dar. Darüber hinaus bieten sie natürlich phantastische Grafiken als Bildschirmschoner.

11.1 Ein Klassiker

Wir beginnen zunächst mit einer schon sehr lange bekannten Animation von drei Farbbalken, die über den Bildschirm sausen. Das ist als Einstiegsübung sehr leicht umsetzbar und hat einen sehr schönen ausgeglichenen Schoneffekt – okay, der ist heute natürlich nicht mehr von Bedeutung.

Unser Ziel ist, drei Balken in den Farben Rot, Grün und Blau über den Bildschirm rollen zu lassen. Die Balken sind durch einen Verlauf gekennzeichnet, der sie dreidimensional aussehen lässt. Abbildung 11.1 gibt einen Anhaltspunkt, wie das Ganze am Ende aussehen soll; natürlich wirkt es in Farbe etwas schöner.

Abbildung 11.1 RGB Balken – hier natürlich nur in Schwarz-Weiß zu sehen

Zum Zeichnen der Balken dient uns die folgende Funktion, welche mit verschiedenen Farben als Parameter aufgerufen werden kann.

```
# berechnet einen schoenen Farbverlauf fuer die
# angegebenen Farbwerte jede Farbkomponente wird
# hierbei proportional zum Farbwert eingerechnet
def drawBar(yStartPos, barHeight, barWidth, colorChoice):
    factor = int(math.floor(255.0 / barHeight))
    for i in range(0, barHeight):
        if i < barHeight/2.0:
            # hier wird die Helligkeit erhoeht
            brightness  = i*factor
        else:
            # hier wird die Helligkeit reduziert
            brightness  = 255-i*factor

        color = map(lambda x: x*brightness, colorChoice)
        pygame.draw.line(screen, color,
          (0, yStartPos+i),
          (barWidth, yStartPos+i))
```

Listing 11.1 Berechnung eines Farbverlaufs

Die ganz am Anfang ermittelte Variable `factor` stellt die Schrittweite für den Helligkeitsverlauf dar. Für die Darstellung wird der gesamte zur Verfügung stehende Helligkeitsbereich gleichmäßig ausgenutzt. Die Höhe des Balkens gibt somit den verfügbaren Raum an, der mit einem gleichmäßigen Verlauf zu füllen ist.

Bis zur Mitte des Balkens wird innerhalb der Schleife der Helligkeitswert für die Darstellung erhöht. Für die untere Hälfte des Balkens wird er daraufhin wieder reduziert. Für jeden Schleifendurchlauf wird die Helligkeit entsprechend mit der als Parameter `colorChoice` übergebenen Farbe verrechnet. Hierbei bedienen wir uns der Möglichkeit, eine anonyme Funktion auf alle Komponenten des Parameters zu »mappen«.

Nachdem die aktuelle Farbe ermittelt wurde, wird sie als Linie auf den Bildschirm gezeichnet. Statt für jeden Aufruf die Farbabstufung neu zu berechnen, könnte man die einzelnen Farbwerte natürlich auch in einer Liste speichern und einfach über den Index darauf zugreifen. Die Berechnung der Werte ist jedoch so leicht, dass es hierbei nicht zu signifikanten Geschwindigkeitseinbußen kommt.

Die Position und die Bewegungsrichtung der drei unterschiedlich gefärbten Balken werden in den folgenden Variablen festgehalten:

```
# die Positionen und Bewegungsrichtungen für die einzelnen Balken
y1          = 0
dir1        = 1
y2          = 200
dir2        = 1
y3          = 400
dir3        = 1

running     = True
barHeight   = 50
winHeight   = 600
winWidth    = 800
screen      = pygame.display.set_mode((winWidth, winHeight),
 pygame.FULLSCREEN);
clock       = pygame.time.Clock()
maxfps      = 100
```

Listing 11.2 Intitialisierung

Da sich alle Balken über die gesamte Fensterbreite erstrecken, ist für die Position die Angabe der y-Komponente ausreichend. Die Variablen, die mit `dir` beginnen, stehen für die Richtung in die sich der Balken gerade bewegt. Dabei kennzeichnet eine positive Eins die Bewegung nach unten und eine negative Eins die Bewegung nach oben. Sobald ein Balken den Fensterrand erreicht wird die Richtung

entsprechend invertiert. Da alle Balken ihren Zustand durch jeweils zwei Variablen festhalten, ist es möglich, hier später auch leicht eine Klasse einzuführen, die alle die Balken betreffenden Daten und Funktionen kapselt.

Die Bewegung der Balken über den Bildschirm implementieren Sie mit Hilfe der Funktion `moveAndCheckDirection`:

```
# hier wird ueberprueft, ob ein Rand erreicht
# worden ist und entsprechend die Richtung angepasst
def moveAndCheckDirection(direction, yPosition,
barHeight, windowHeight):
    yPosition += direction
    if yPosition + barHeight > windowHeight or
      yPosition < 0:
        direction *= -1

    return direction, yPosition
```

Listing 11.3 Richtungswechsel an den Bildschirmrändern

Der Hauptablauf des Programms stellt sich unter Zuhilfenahme der bereits erläuterten Funktionen nun dar wie in Listing 11.4; die einzige noch nicht erläuterte Funktion ist hierbei die Funktion `handleEvents()`:

```
while running:
    clock.tick(maxfps)
    running = handleEvents(running)

    screen.fill((0, 0, 0))
    drawBar(y1, barHeight, winWidth, (0, 0, 1))
    drawBar(y2, barHeight, winWidth, (0, 1, 0))
    drawBar(y3, barHeight, winWidth, (1, 0, 0))

    dir1, y1 = moveAndCheckDirection(dir1, y1,
     barHeight, winHeight)
    dir2, y2 = moveAndCheckDirection(dir2, y2,
     barHeight, winHeight)
    dir3, y3 = moveAndCheckDirection(dir3, y3,
     barHeight, winHeight)

    pygame.display.flip()
```

Listing 11.4 Hauptablauf

Die Funktion `handleEvents()` verarbeitet die Benutzereingaben und reagiert entsprechend darauf. Da es sich hier nur um die Überprüfung einer Abbruchbedin-

gung handelt, stellt sich das Ganze sehr einfach dar. Es gibt lediglich einen kleinen Trick bezüglich der zeitlichen Verzögerung vor dem eigentlichen Eventhandling:

```
def handleEvents(running):
    event   = pygame.event.poll()

    if pygame.time.get_ticks() - start > 1000:
        print start
        if event.type == pygame.QUIT:
            running = False
        elif event.type == pygame.KEYDOWN:
            running = False
        elif event.type == pygame.MOUSEMOTION:
            running = False
        elif event.type == pygame.MOUSEBUTTONDOWN:
            running = False

    return running
```

Listing 11.5 Ereignisbehandlung

Der Trick stellt sicher, dass Ereignisse erst eine Sekunde nach dem Start des Programms ausgewertet werden. Jetzt stellt sich Ihnen sicher die Frage, warum die Ereignisse innerhalb der ersten Sekunde ignoriert werden sollen. Die Erklärung hierfür ist sehr einfach, denn dieses Programm lässt sich leicht auch als Bildschirmschoner gestalten und sollte nicht direkt nach dem Start wieder beendet werden. Ohne die Verzögerung würden kleinste Mausbewegungen, die beim Start ausgeführt wurden, noch im Puffer der Ereignisbehandlungsroutine liegen und direkt zum Beenden des Programms führen. Ein Leeren der Event-Queue zu Beginn des Programms würde hier keine Abhilfe leisten. Jeder Start als Bildschirmschoner aufgrund von Inaktivität wäre natürlich nicht von diesem Problem betroffen, aber hin und wieder möchten Sie sicher auch einfach mal den Bildschirmschoner von Hand starten, beispielsweise um etwas zu testen.

11.2 Ein bissl aufräumen schadet nie

Die erste Implementierung aus dem letzten Abschnitt ist zwar vollkommen okay und lauffähig, aber es geht doch ein wenig schöner. Die Daten und Funktionen bezüglich der Balken lassen sich gut zu einer Klasse zusammenfassen, die sich dann wie folgt gestaltet:

```
class Bar(object):
    def __init__(self, yStartPos, barHeight,
      barWidth, windowHeight, colorChoice):
        self.__yPos         = yStartPos
        self.__height       = barHeight
        self.__width        = barWidth
        self.__windowHeight = windowHeight
        self.__color        = colorChoice
        self.__direction    = 1

    # berechnet einen schoenen Farbverlauf fuer die
    # angegebenen Farbwerte jede Farbkomponente wird
    # hierbei proportional zum Farbwert eingerechnet
    def __drawBar(self):
        factor = int(math.floor(255.0 / self.__height))
        for i in range(0, self.__height):
            if i < self.__height/2.0:
                # hier wird die Helligkeit erhoeht
                brightness  = i*factor
            else:
                # hier wird die Helligkeit reduziert
                brightness  = 255-i*factor

            color = map(lambda x: x*brightness,
             self.__color)
            pygame.draw.line(screen, color,
              (0, self.__yPos+i),
              (self.__width, self.__yPos+i))

    # prueft, ob ein Rand erreicht worden ist,
    # und passt die Bewegungsrichtung an
    def move(self):
        self.__yPos += self.__direction
        if self.__yPos + barHeight > self.__windowHeight or
          self.__yPos < 0:
            self.__direction *= -1

        self.__drawBar()
```

Listing 11.6 Die Klasse »Bar«

Die Klasse `Bar` enthält nur eine öffentlich zugängliche Methode namens `move()`, die sowohl die Verschiebung der Balken als auch das Zeichnen übernimmt. Das Zeichnen wird hierbei an die nicht öffentlich sichtbare Methode `__drawBar()`

delegiert. Unter Verwendung der neuen Klasse vereinfacht sich die Hauptroutine etwas:

```
while running:
    clock.tick(maxfps)
    running = handleEvents(running)

    screen.fill((0, 0, 0))
    bar1.move()
    bar2.move()
    bar3.move()
    pygame.display.flip()
```

Listing 11.7 Hauptablauf bei Verwendung der Klasse »Bar«

Viele vorher noch sichtbaren Schritte werden jetzt innerhalb der neuen Klasse `Bar` gekapselt und irritieren nicht mehr beim Betrachten der Hauptroutine. Was hier im Kleinen ganz nett vereinfacht, ist für größere Projekte unerlässlich, um eine gewisse Wartbarkeit und Softwarequalität zu erhalten. Die Initialisierung der Zustandsvariablen hat sich entsprechend auch etwas vereinfacht, da einige Dinge in die Initialisierung unserer neuen Klasse verschoben werden konnten. Zudem wurden dabei die zusammengehörenden Daten entsprechend zusammen in der neuen Klasse gruppiert.

```
# hier sind die Farbbalken sowie die Fenstergroesse definiert
barHeight   = 50
winHeight   = 600
winWidth    = 800

bar1 = Bar(  0, barHeight, winWidth, winHeight, (0, 0, 1))
bar2 = Bar(200, barHeight, winWidth, winHeight, (0, 1, 0))
bar3 = Bar(400, barHeight, winWidth, winHeight, (1, 0, 0))
```

Listing 11.8 Initialisierung bei Verwendung der Klasse »Bar«

Soviel zu einem sehr einfachen Bildschirmschoner – im nächsten Abschnitt wird es jetzt ein wenig philosophisch.

11.3 Reduktionismus – alles eine Frage von Zuständen

Der Reduktionismus ist eine philosophische Lehre, die behauptet, dass alle natürlichen Vorgänge vollständig deterministisch erklärbar sind. Diese Idee erscheint

mir persönlich auch sehr einleuchtend, wenn man zunächst davon ausgeht, dass zum Beispiel Menschen bei der Geburt beziehungsweise zum Zeitpunkt der Befruchtung der Eizelle einen vorgegebenen Zustand haben. Dieser Zustand kann sich im Grunde genommen nur durch äußere Einflüsse ändern. Wenn Sie sich vorstellen, dass der Mensch eine Art komplexer Zustandsautomat wäre, dann wären wir momentan genau bei der Vorstellung, die ich hier kurz als gegeben voraussetzen möchte.

Der Grund für diese Annahme aus meiner Sicht wäre einfach, dass innerhalb des menschlichen Gehirns keine Materie aus dem Nichts entstehen kann. Entsprechend ist alles, was im Gehirn ausgebrütet wird, ein Resultat der bereits dort vorhandenen Information in Kombination mit den von außen einwirkenden zusätzlichen Einflüssen. Aber selbst wenn diese Annahme korrekt wäre, so ließe sich dennoch aufgrund der extrem komplexen Zustandsübergänge und der unendlich großen Datenmenge natürlich nicht auf das Verhalten eines Menschen schließen. Es würde allein schon daran scheitern, den aktuellen Zustand auszulesen, da in den Größenordnungen, die für das Auslesen interessant wären, bereits die Heisenbergsche Unschärferelation zum Tragen käme. Diese besagt im Groben, dass der Aufenthaltsort sehr kleiner Teilchen nicht ausgelesen werden kann. Dieses Problem wäre jedoch noch das geringere, denn so wie auch die Wetterprognosen nicht exakt sein können, so sind auch Prognosen von Einflüssen auf andere sehr komplexe Systeme nicht möglich.

Doch wo liegt eigentlich das Problem mit der Wettervorhersage? Es liegt darin, dass sehr kleine Ungenauigkeiten in der Beschreibung des Ausgangszustandes bei den Berechnungen extrem verstärkt werden, so dass am Ende ein kumulierter Fehler entsteht, der so enorme Ausmaße annimmt, dass nichts mehr so ist, wie es sein sollte. Diese Beschreibung ist vermutlich nicht so leicht nachzuvollziehen, weshalb Sie das Ganze am besten anhand eines kleinen Beispiels probieren. Dafür soll eine sehr einfache Regel verwendet werden, um eine Zahlenreihe zu produzieren. Ausgehend von einer beliebigen Gleitkommazahl wird in jedem Schritt die Zahl verdoppelt, wobei der Vorkommaanteil verworfen werden soll. Sie werden durchaus überrascht sein, wie chaotisch diese Reihe erscheint, obwohl die Regel sehr klar definiert wurde. Zudem wird es anhand der beschriebenen Vorgehensweise leicht zu zeigen sein, dass kleine Änderungen bei der Ausgangszahl schon nach wenigen Schritten zu komplett anderen Zahlenreihen führen. Dabei ist noch zu berücksichtigen, dass diese einfache Regel natürlich kein Vergleich zu den komplexen Regeln für die Berechnung von Wettermodellen ist.

```
import math

c = 0.8404
print c, "->",
for x in range(12):
    c *= 2
    f, i = math.modf(c)
    c = f
    print c,
```

Listing 11.9 Eine chaotische Zahlenreihe

Die Variable `c` wird mit dem Wert `0.8404` initialisiert, der dann innerhalb der Schleife verdoppelt wird. Das Ergebnis der Verdopplung wird durch den Aufruf von `math.modf()` in den ganzzahligen und den Fließkommaanteil aufgeteilt, wobei uns nur der Fließkommaanteil interessiert. Dieses einfache Programm repräsentiert die oben beschriebene Regel, die schon für beachtlich chaotisch wirkende Zahlenreihen sorgt. Sie sehen daran sehr gut, wie stark sich auch kleine Ungenauigkeiten schnell aufschaukeln, wenn Sie den Startwert leicht verändern.

```
0.8404 -> 0.6808 0.3616 0.7232 0.4464 0.8928 0.7856
0.5712 0.1424 0.2848 0.5696 0.1392 0.2784
0.8403 -> 0.6806 0.3612 0.7224 0.4448 0.8896 0.7792
0.5584 0.1168 0.2336 0.4672 0.9344 0.8688
```

Listing 11.10 Chaos in seiner einfachsten Form

Eine winzige Änderung bei der Initialisierung führt nach zehn Durchläufen der Regel bereits zu komplett unterschiedlichen Zahlen. Die Verkettungen allerdings, die in Wettermodellen und bei anderen natürlichen Vorgängen bestehen, sind um vieles komplexer als die einfache Regel dieses Beispiels.

Ein grundlegendes Problem stellt in dem Zusammenhang auch die Ungenauigkeit der Computer dar, die Gleitkommazahlen nur näherungsweise repräsentieren, statt sie exakt darzustellen. In Python bietet die Klasse `Decimal` zwar eine exakte Darstellung von Fließkommazahlen an, aber die Verwendung würde nur Sicherheit vortäuschen, denn es reichen bereits die Fehler beim Messen der Daten, um zu gravierenden Fehlprognosen zu gelangen. Die tatsächlichen Modelle sind so stark rückkoppelnd, dass eine exakte Prognose nicht möglich ist. Dies ist der Grund, warum der Wetterbericht für viele Tage im Voraus oft nicht hält, was er verspricht.

Kommen wir aber wieder zum Zustandsmodell für Menschen. Der Gedanke, dass jeder Mensch nur von seinem Ausgangszustand und den äußeren Einflüssen »gelenkt« wird, ist schon ein bisschen gewöhnungsbedürftig.

Im ersten Moment könnte man fast meinen, dass alles egal sei, weil man ja sowieso keinen direkten Einfluss nehmen kann beziehungsweise kein freier Wille vorhanden sein kann. Die Grundlage dieser Behauptung wäre, dass im Gehirn ja keine neue Materie entstehen kann, sondern nur bereits vorhandene Atome miteinander agieren können. Im Prinzip ist das aber egal, denn die Vorgänge und Zustände sind in jedem Fall so komplex, dass jegliche Voraussage vergeblich wäre.

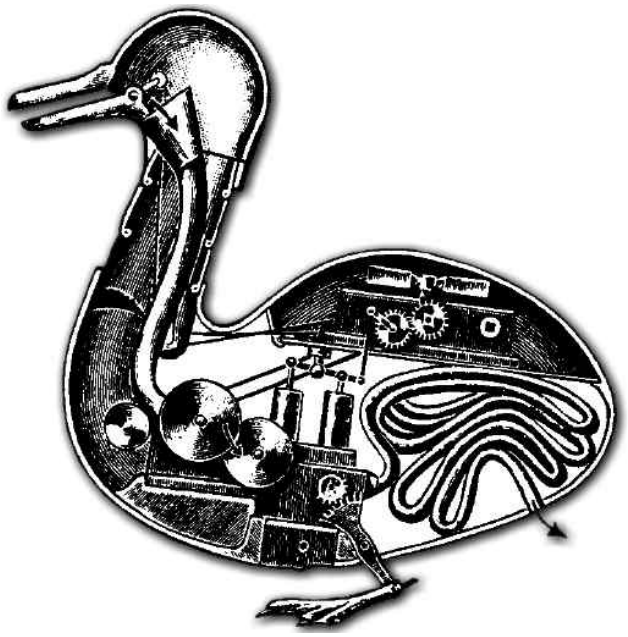


Abbildung 11.2 René Descartes – De homine (1622)

So viel zum Reduktionismus – der Lehre der vollkommen deterministischen Welt, in der alles nur eine Frage von Zuständen ist, die durch äußere Einflüsse verändert werden. Persönlich glaube ich ja an die vollkommen deterministische Sicht der Dinge – mir ist zumindest noch nicht klar, woher ansonsten der »freie Wille« kommen soll, wenn nicht aufgrund des Ausgangszustands, der über viele kleine Einflüsse von außen angepasst wurde. Gerade in der heutigen durch die Werbung und anderen Medien sehr stark bestimmten Welt wird diese Ansicht glaubhafter denn je – ist es wirklich die eigene Entscheidung oder nur eine Reaktion aufgrund der äußeren Einflüsse? Sehr interessante Literatur zum Thema Manipulation durch Werbung und Medien finden Sie, wenn Sie nach »Neuromarketing« suchen.

11.4 Was ist Chaos?

Irgendwann haben Physiker und auch andere Wissenschaftler festgestellt, dass es mit dem Determinismus nicht so weit her ist. Zumindest erkannten sie, dass die

verwendeten Modelle nicht genau genug sind, um alle Vorgänge in der natürlichen Art und Weise zu beschreiben. Es kam bei vielen Modellen einfach zu nicht erklärbar großen Veränderungen, die auf einmal aus dem Nichts zu entstehen schienen. So gerieten ab einem bestimmten Grad die ansonsten so schön prognostizierbaren Bedingungen total aus dem Ruder. Luft- oder Wasserströmungen etwa, die durch Hindernisse beeinflusst werden, lassen ab einer bestimmten Strömungsgeschwindigkeit seltsame Turbulenzen entstehen. Die klassische Physik konnte diese Phänomene nicht erklären, was zur Geburtsstunde der Chaostheorie wurde.

Von nun an wurde versucht, diese im Einzelnen nicht fassbaren starken Schwankungen der im System beteiligten Faktoren im Ganzen zu betrachten. Dabei erkannte man auch, dass innerhalb der an sich recht chaotisch wirkenden Schwankungen immer wieder Ähnlichkeiten auftraten. Die Selbstähnlichkeit ist in der Natur an sehr vielen Stellen zu finden, so zum Beispiel beim Farn, dessen Struktur sich in immer kleineren Einheiten wiederholt. Aber auch so einfache Dinge wie Bäume sind sich in einer Grundstruktur sehr ähnlich, wenn man sich den Stamm mit seinen Verästelungen anschaut, die immer wieder ähnliche Muster bilden.

Eine interessante Entdeckung zu jener Zeit machte Benoît Mandelbrot, der erkannte, dass die Länge der Küstenlinien sehr unterschiedlich angegeben wurde. Grund für die Unterschiede bei der Längenangabe waren die zur Längenermittelung verwendeten Maßstäbe der Karten. Je kleiner der Maßstab einer Küste wird, desto mehr Details werden sichtbar, da die Glättung der Küstenlinie entsprechend milder ausfällt. Wenn man den Prozess unendlich oft durchführt, so wäre eine schier grandiose Küstenlänge erzielbar, die die noch so kleinsten Details in die Längenberechnung mit eingehen lässt.

Ähnlich verhalten sich *Fraktale*, eine besondere Art von Mengen, die sich selbst ähnelnde Strukturen in unendlicher Menge enthalten. Der Begriff »Fraktal« wurde von Benoît Mandelbrot, dem Entdecker dieser Mengen, geprägt und geht auf das lateinische »frangere« zurück, was so viel wie »brechen« bedeutet. Damit stellte Mandelbrot den Bezug zu den gebrochenen Zahlen sowie zu der Unregelmäßigkeit von Fragmenten her. Das Erstaunliche an der Vielfalt der Fraktale ist die Einfachheit der zugrundeliegenden Berechnungen, die iterativ immer wieder ausgeführt werden. Eines der bekanntesten Fraktale – von dem Sie sehr wahrscheinlich schon gehört haben – ist die nach Benoît Mandelbrot benannte Mandelbrotmenge.

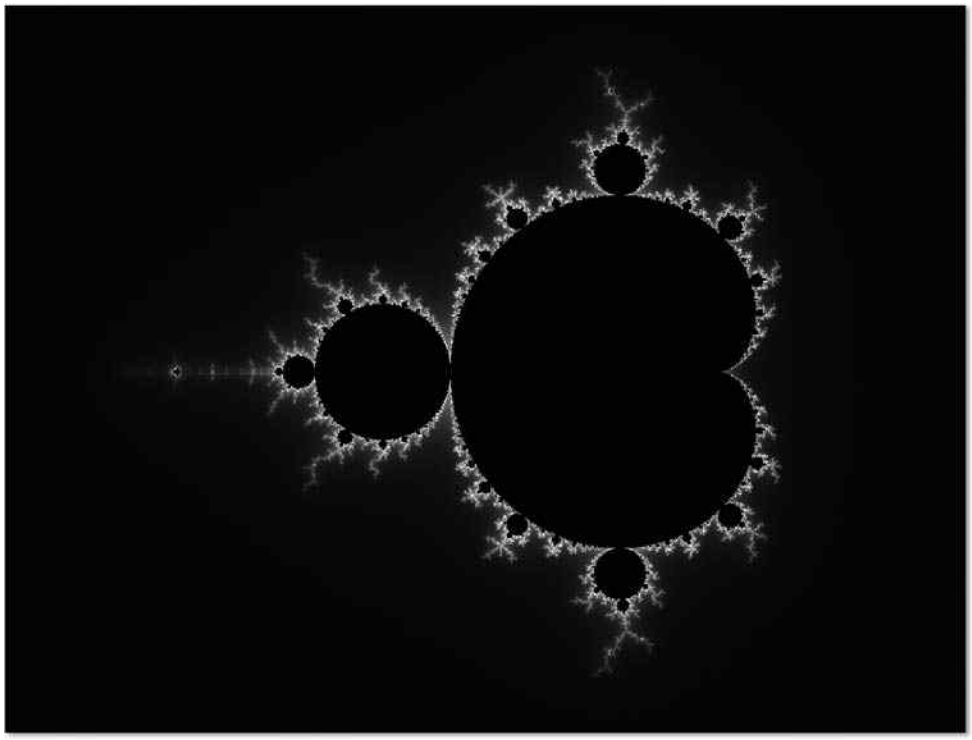


Abbildung 11.3 Mandelbrot-Fraktal

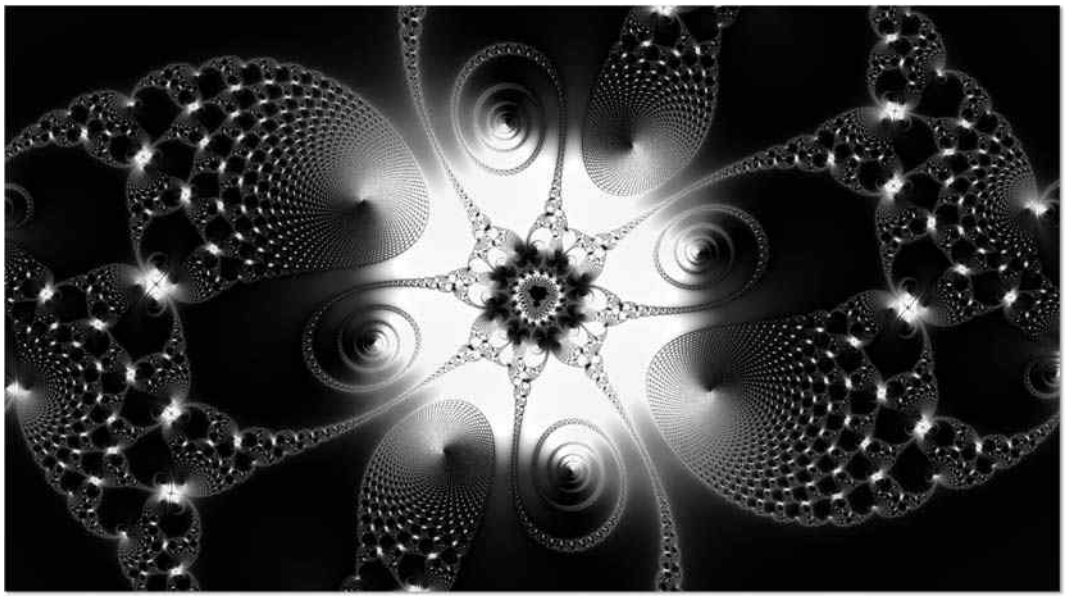


Abbildung 11.4 Fraktal

Wie Sie sich sicher jetzt schon denken können, lassen sich in Python auch sehr einfach Fraktale berechnen und darstellen. Als erstes kleines Beispiel dient uns der folgende Quelltext, der einen Ausschnitt aus der Julia-Menge berechnet und als Grafik speichert:

```
import Image
import ImageDraw

SIZE = 2048
image = Image.new("L", (SIZE, SIZE))
d = ImageDraw.Draw(image)

c = -0.767793511962608 + 0.0952946560332372j
for x in range(SIZE):
    for y in range(SIZE):
        re = (x * 2.0 / SIZE) - 1.0
        im = (y * 2.0 / SIZE) - 1.0

        z=re+im*1j
        for i in range(128):
            if abs(z) > 2.0: break
            z = z * z + c
        d.point((x, y), i * 2)

image.save(r"julia2.png", "PNG")
```

Listing 11.11 Einen Ausschnitt der Julia-Menge darstellen

Die Julia-Menge **[+]**

Die Julia-Mengen sind Teilmengen der komplexen Zahlenebene, welche von Pierre Fatou und Gaston Maurice Julia beschrieben worden. Eine Julia-Menge kann zu einer holomorphen oder meromorphen Funktion gehören. Wir betrachten hier den Sonderfall, bei welchem die Julia-Menge eine fraktale Menge darstellt. Das Bildungsprinzip folgt – wie für Fraktale üblich – dem rekursiven Anwenden der gleichen Funktion, welche für alle komplexen Zahlen definiert sein muss.

Die Bibliotheken `Image` und `ImageDraw` werden für die Erzeugung der Grafik verwendet. Für alle Punkte wird deren Wachstumsverhalten untersucht und entsprechend in der Grafik als zugehöriger beziehungsweise nicht mehr dazugehöriger Punkt zur Menge dargestellt. Für die Darstellung wurde auf der x-Achse der Wertebereich von -1 bis +1 gewählt: die Berechnung `(x * 2.0 / SIZE) - 1.0` stellt das sicher. Für die y-Achse wird ebenfalls nur der Bereich von -1 bis +1 betrachtet, was durch die Berechnung `(x * 2.0 / SIZE) - 1.0` sichergestellt wird. Der betrachtete Ausschnitt der Julia-Menge wird dabei durch den Parameter `c` festgelegt.

11.5 Ein etwas schönerer Schongang

Mit dem Wissen über die Fraktale werden wir nun auch einen schönen Bildschirmschoner entwickeln, der Fraktale auf den Bildschirm zaubert. Hierfür verwenden wir wieder Pygame, um die Darstellungen auf den Bildschirm zu zeichnen. Der Einfachheit halber verwenden wir den gleichen Code wie bereits oben, nur dass wir ein paar Anpassungen vornehmen, so dass Pygame für die grafische Ausgabe verwendet wird. Beginnen wir zunächst mit der Initialisierung von Pygame:

```
import sys, pygame
from pygame.locals import *

SCREEN_WIDTH=800; SCREEN_HEIGHT=600

max_iteration = 70
scale = 3.0/(SCREEN_HEIGHT*500.0)

def init():
    global screen
    pygame.init()
    screen = pygame.display.set_mode((SCREEN_WIDTH,
     SCREEN_HEIGHT), pygame.FULLSCREEN)
    # die folgende Zeile lässt den Mauszeiger
    # verschwinden, der Zeiger wird im Prinzip
    # ausmaskiert bzw. ist danach durchsichtig
    pygame.mouse.set_cursor((8,8), (0,0),
     (0,)*(64/8), (0,)*(64/8))
```

Listing 11.12 Initialisierung von »Pygame«

Für dieses Beispiel benötigen wir nur die Bibliothek `pygame`, wobei wir, wie bereits erwähnt, `pygame` für die graphische Ausgabe verwenden. Vor der Verwendung von `pygame` ist immer ein Aufruf von `pygame.init()` notwendig, der die Bibliothek für die Verwendung initialisiert. Im darauf folgenden Aufruf definieren wir einen `screen`, auf den wir zeichnen wollen. Der Parameter `pygame.FULLSCREEN` sorgt dabei, wie der Name schon vermuten lässt, für die Darstellung im Vollbildmodus. Die letzte Zeile des Listings sorgt dafür, dass der Mauszeiger unsichtbar wird.

Kommen wir jetzt gleich zur Hauptschleife des Programms:

```
def main():
    # die Hauptschleife des Programmes, welche sich
    # um die Ereignisbehandlung kümmert und die
    # Zeichenroutine aufruft
    global scale
    update_screen()
    while True:
        event = pygame.event.poll()
        if event.type == QUIT or
         (event.type == KEYDOWN and
          event.key == K_ESCAPE):
            break
        pygame.time.delay(50)
```

Listing 11.13 Der Hauptablauf mit Ereignisbehandlung

Zunächst wird durch Aufruf der Funktion `update_screen()` der Bildschirm gelöscht und die Funktion zum Zeichnen des Fraktals aufgerufen (vgl. Listing 11.14). Die `while`-Schleife dient hier nur als Ereignisbehandlungsschleife, die darauf wartet, dass der Benutzer das Programm per Tastendruck auf (Esc) beendet. Die Funktion `update_screen()` sieht wie folgt aus:

```
def update_screen():
    # hier wird der Bildschirm gelöscht,
    # so dass dieser zunächst komplett schwarz ist,
    # danach wird das Fraktal durch Aufruf von
    # draw_field gezeichnet
    global screen, mouse_pos
    screen.fill((255, 255, 255))
    draw_field(screen)
    pygame.display.flip()
```

Listing 11.14 Neuzeichnen des Bildschirmes

Die nach dem Löschen des Bildschirms aufgerufene Funktion `draw_field()` zum Zeichnen des Fraktals gestaltet sich dabei wie folgt:

```
def draw_field(scr):
    # hier steht der bereits bekannte Code aus
    # dem vorherigen Beispiel mit PIL = Python Imaging Library
    c = -0.767793511962608 + 0.0952946560332372j
    for x in range(SCREEN_WIDTH):
        for y in range(SCREEN_HEIGHT):
            re = (x * 2.0 / SCREEN_WIDTH) - 1.0
            im = (y * 2.0 / SCREEN_HEIGHT) - 1.0
```

```
            z=re+im*1j
            for i in range(max_iteration):
                if abs(z) > 2.0: break
                z = z * z + c

            pygame.draw.line(scr, set_color(i), (x, y), (x, y))
```

Listing 11.15 Zeichnen eines Ausschnitts des Fraktals

Die Funktion `draw_field()` ist fast genauso aufgebaut wie die bereits bekannte Funktion, die ein Fraktal in eine Grafikdatei schreibt. Es gibt lediglich zwei Anpassungen: die Verwendung von Pygame für die Grafikausgabe und das Setzen der Farbe durch Aufruf der Funktion `set_color()`. Die Verwendung von Pygame bedarf sicherlich keines weiteren Kommentars, weshalb wir uns direkt dem Aufruf von `set_color()` zuwenden können.

```
def set_color(i):
    # hier wird eine hübsche Farbkombination gewählt,
    # so dass die Fraktale nicht nur in einer Farbe mit
    # Farbabstufung erscheinen - hier gibt es viel Spielraum
    # für eigene Versuche, schöne Farben zu erhalten
    if i == max_iteration:
        col = (0,0,0)
        return col
    else:
        c = (i*15)
        g=255
        b=255
        r = 255 - c
        if r < 0:
            r=0
            g = 255*2 - c
            if g < 0:
                g=0
                b = 255*3 - c
                if b < 0:
                    r=g=b=0
        return (r, g, b)
```

Listing 11.16 Farbwahl je nach Iterationsschritt

Die Bestimmung der einzelnen Farbanteile erfolgt hier eher nach Gefühl als nach einer festen Regel – deshalb bietet sich hier auch die Gelegenheit, ein wenig mit der Implementierung von `set_color()` zu spielen. Um das Ganze nun auch zu starten, fehlen nur noch die letzten Zeilen des Scripts:

```
if __name__ == '__main__':
    init()
    main()
```

Listing 11.17 Das Modul ausführbar machen

Die Bedingung in der ersten Zeile sorgt dafür, dass das Programm nur dann gestartet wird, wenn das Modul direkt aufgerufen wird. Wird das Modul jedoch nur importiert, so wird das Programm nicht direkt gestartet, was dann in der Regel auch nicht gewünscht ist.

11.6 Schon ganz nett, was fehlt denn noch?

In diesem Kapitel haben Sie noch am meisten Spielraum für eigene Erweiterungen, denn hier wurde lediglich der Grundstein für Bildschirmschoner gelegt. Was als Nächstes auf dem Programm stehen könnte, sind zum Beispiel wechselnde Parameter für das darzustellende Fraktal, denn immer das gleiche Fraktal gibt einen schlechten Bildschirmschoner im eigentlichen Sinne ab. Diese Anpassung ist jedoch sehr leicht, da Sie hierfür lediglich die Parameter für das Fraktal in gewissen Abständen verändern müssen. Zusätzlich müsste der Aufruf für die Darstellung des Fraktals dann natürlich innerhalb der Hauptschleife erfolgen, in der momentan nur die Ereignisbehandlung stattfindet. Die Zeitspanne für das Warten in der Hauptschleife sollten Sie hierfür dann auch erhöhen.

Etwas schwieriger, aber keinesfalls weniger interessant ist die Verwendung von Threads für die Berechnung der Fraktale. Hierdurch ließe sich zum einen die für die Berechnung notwendige Zeit stark reduzieren, und zum anderen könnte das Programm während der Berechnung immer noch auf Benutzereingaben reagieren.

Diese beiden Punkte wären aber noch lange nicht alles, es fehlt auch noch die Umwandlung in eine *exe*-Datei, falls Sie unter Windows programmieren. Die unter Windows üblichen Bildschirmschoner sind normale Executables, die nur die Endung *scr* tragen, aber ansonsten nichts Besonderes sind. Bei Bildschirmschonern werden zusätzlich Startparameter übergeben, die darauf hinweisen, ob der Benutzer Einstellungen vornehmen möchte oder der Bildschirmschoner gerade gestartet werden soll.

Falls Sie gar keinen Bildschirmschoner bauen möchten, sondern lieber in das Fraktal zoomen wollen, so stellt das natürlich auch eine noch mögliche Ausbaustufe dar.

Anhang

A OpenGL-Kurzreferenz

A.1 Konventionen

Grundsätzlich gilt für alle Funktionsaufrufe der OpenGL-API folgende Namenskonvention:

<Bibliothekspräfix><Befehl><Parameteranzahl><Parametertyp>

A.2 Culling und Winding

Culling und Winding werden nicht immer benötigt, aber es kann einen entscheidenden Unterschied machen, wenn Sie zum Beispiel mit Culling arbeiten, was enorme Performancegewinne ermöglicht.

Culling ist eine Möglichkeit, nicht sichtbare Flächen beim Zeichnen zu ignorieren, indem man zum Beispiel bei Körpern die »Rückseite« der Polygone konsequent nach innen richtet und per `glEnable(GL_CULL_FACE)` nicht mehr mitzeichnen lässt. Was OpenGL als Vorderseite interpretiert, lässt sich auch explizit über `glFrontFace(GL_CW)` ändern. Als Parameter sind `GL_CW` und `GL_CCW` zulässig; sie stehen für »im Uhrzeigersinn (clockwise)« und »gegen den Uhrzeigersinn (counter-clockwise)«.

Winding bezeichnet in diesem Zusammenhang die Richtung, in der die Punkte gesetzt wurden, also ob die Punkte des Polygons im Uhrzeigersinn oder gegen den Uhrzeigersinn gezeichnet wurden. Dabei wird je nach Richtung der Punktreihenfolge die Ober- und Unterseite der Fläche definiert.

A.3 Was genau ist »die Matrix«?

OpenGL arbeitet sehr viel mit Matrizen. Für jede Manipulation der Ansicht wird eine entsprechende Matrix manipuliert. Welche Matrix jeweils gerade aktiv ist, wird mittels `glMatrixMode(Modus)` vorher festgelegt. Es gibt entsprechend zum Beispiel eine Matrix, die alle Punkte der aktuellen Szene enthält. Somit können Sie die Ansicht der Szene drehen, indem Sie die entsprechende Matrix drehen.

`glMatrixMode(Modus)` legt fest, welcher Matrixmodus gerade aktiv ist. Der Matrixmodus wirkt sich dabei auf die Aufrufe von `glTranslate()`, `glRotate`, `glScale()`, `glLoadIdentity()`, `glPushMatrix()`, `glPopMatrix()`, `glLoadMatrix()`, `glMultMatrix()` aus. `glTranslate(x, y, z)` verschiebt die Matrix um die als Pa-

rameter angegebenen Koordinaten. `glRotate(Winkel, x, y, z)` rotiert die Matrix um die angegebene Achse entsprechend der angegebenen Gradzahl. Möchten Sie zum Beispiel die Matrix 90 Grad um die x-Achse rotieren lassen, dann ist der Aufruf `glRotate(90, 1.0, 0.0, 0.0)` notwendig.

A.4 Die Matrix manipulieren

Durch Translation und Rotation lassen sich alle in OpenGL verwendeten Matrizen entsprechend manipulieren. Das Schöne daran ist, dass Sie leicht alle dargestellten Objekte aus einer anderen Perspektive betrachten können. Die Objekte müssen sich dafür in keiner Weise ändern, obwohl die Darstellung entsprechend der neuen Perspektive komplett anders aussehen kann.

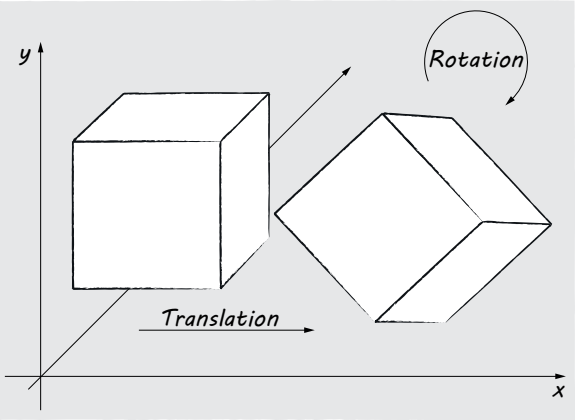


Abbildung A.1 Translation und Rotation

`glScale(x, y, z)` skaliert die Matrix entsprechend der Skalierungsfaktoren für die einzelnen Dimensionen. So führt zum Beispiel der Aufruf `glScale(3.0, 1.0, 1.0)` zu einer dreifachen Streckung der x-Achse bei gleichbleibendem Verhältnis für die y- und z-Achse. Auf Deutsch: Das Bild wird in die Breite gezogen.

`glLoadIdentity()` lädt die sogenannte Identitätsmatrix und macht somit alle Transformationen rückgängig, stellt also den Ursprungszustand wieder her.

Mit den Aufrufen von `glPopMatrix()` und `glPushMatrix()` können Sie den Zustand auf dem Stack zwischenspeichern und wieder laden. Diese Funktionalität ist sinnvoll, wenn viele Transformationen in Folge erfolgen und ein vorheriger Zwischenstand später wiederverwendet werden muss. Für alle Fälle, wo es genügt, direkt den Ausgangszustand wiederherzustellen, bietet sich der Aufruf von `glLoadIdentity()` an.

`glLoadMatrix()` dient dem Laden einer beliebigen Matrix, die die aktuelle Matrix ersetzt.

`glMulMatrix()` multipliziert die aktuelle Matrix mit der als Parameter übergebenen Matrix. Stellen Sie sich das Ganze einfach so vor: Alles passiert in der Matrix und wird entsprechend der durch den Viewport aktuell eingenommenen Perspektive sichtbar.

Der Aufruf von `glOrtho(links, rechts, unten, oben, nah, fern)` erstellt eine Matrix für Parallelprojektion mit den als Parameter angegebenen Maßen. Die Maße beschreiben einen 3D-Raum, in dem dank Parallelprojektion die Entfernung keinen Einfluss auf die Größe der dargestellten Matrixinhalte hat.

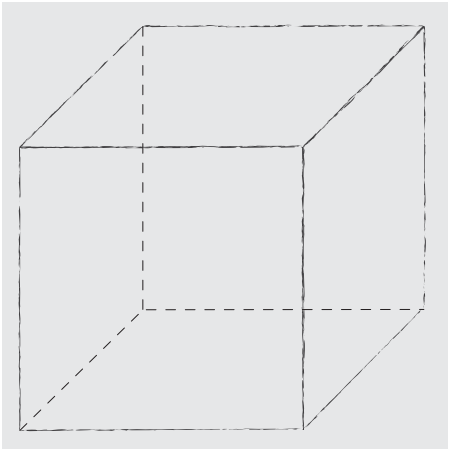


Abbildung A.2 Die Parallelprojektion

Möchten Sie eine perspektivische Darstellung der Inhalte erreichen, so greifen Sie auf `gluPerspective(BetrachtungswinkelY, AspektratioX, nah, fern)` oder auf `glFrustrum(links, rechts, unten, oben, nah, fern)` zurück. Die Parameter von `glFrustrum()` entsprechen zwar den Parametern von `glOrtho()`, werden aber zur Multiplikation mit der aktuellen Matrix verwendet, so dass eine perspektivische Darstellung erfolgt.

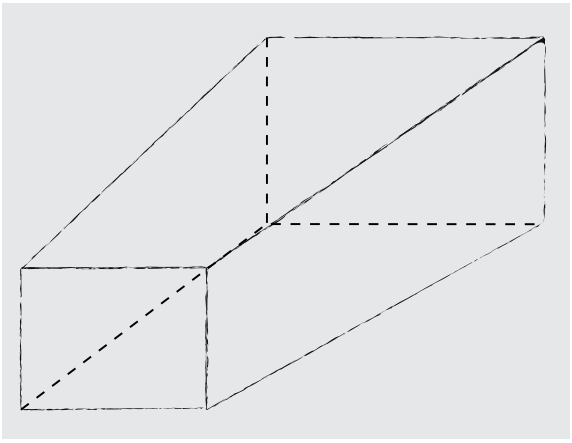


Abbildung A.3 Die perspektivische Projektion

Die `AspektratioX` bei Verwendung von `gluPerspective()` stellt das Verhältnis zwischen Höhe und Breite dar, also genauer gesagt Höhe geteilt durch Breite. Der Aufruf von `glOrtho()` liefert eine Parallelprojektionsmatrix.

A.5 Grafikprimitive

Im Folgenden als Ergänzung noch alle von OpenGL unterstützten Grafikprimitive:

`GL_POINTS` dient zur Darstellung einzelner Punkte, die allerdings keine perpektivische Darstellung unterstützen und dementsprechend unabhängig von der Entfernung des Betrachters immer die gleiche Größe aufweisen. Die Größe der Punkte lässt sich über den Aufruf von `glPointSize(Größe)` setzen, wobei nicht jede Größe unterstützt wird. Den unterstützten Größenbereich ermitteln Sie über `glGetFloatv(GL_POINT_SIZE_RANGE)` und die Schrittweite über `GL_POINT_SIZE_GRANULARITY`. Falls eine perspektivische Darstellung erwünscht ist, so ist es möglich, über Parameter der Punkte auch darauf einzuwirken.

`GL_LINES` stellt gerade Linien dar, die durch einen Startpunkt und einen Endpunkt definiert werden. Die Stärke der Linien ist wie auch die Größe der Punkte nicht beliebig setzbar, sondern nur innerhalb vorgegebener Schrittweiten eines Bereiches, die Sie prinzipiell über die Aufrufe `glGetFloatv(GL_LINE_WIDTH_RANGE)` und `glGetFloatv(GL_LINE_WIDTH_GRANULARITY)` einsehen können. Gesetzt wird die Linienstärke dann durch `glLineWidth(Stärke)`.

`GL_LINE_STRIP` erzeugt viele zusammenhängende gerade Linien, die jeweils durch zwei Punkte repräsentiert werden, wobei der Startpunkt der nächsten Gerade dem Endpunkt der vorherigen entspricht. Die Linienstärke ist dieselbe wie für `GL_LINES`.

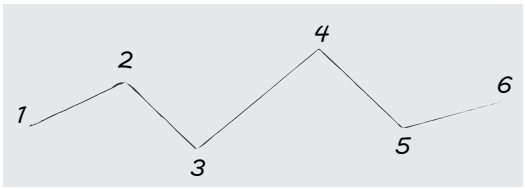


Abbildung A.4 Ein Line-Strip

`GL_LINE_LOOP` dient zur Darstellung eines durch Linien eingeschlossenen Bereichs. Der nächste Punkt wird jeweils durch eine Gerade mit dem vorherigen verbunden, wobei der letzte Punkt zusätzlich mit dem ersten Punkt verbunden wird.

`GL_TRIANGLES` stellt Dreiecke dar, die jeweils durch drei Punkte spezifiziert werden.

GL_TRIANGLE_STRIP generiert Streifen, die im ersten Kapitel (vgl. Listing 1.16) sinnvoll eingesetzt werden. Wir haben zum Beispiel einen Ring mittels eines GL_TRIANGLE_STRIPs dargestellt.

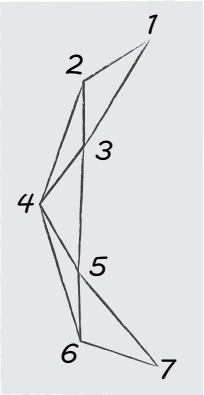


Abbildung A.5 Ein Triangle-Strip

GL_TRIANGLE_FAN dient zur Darstellung von mehreren Dreiecken, die alle einen Punkt gemeinsam haben, wobei sich jeweils ein Paar von Dreiecken einen weiteren Punkt teilt, der sich von dem erstgenannten unterscheidet. Dem Ziffernblatt unserer Uhr aus Kapitel 1, »Es schlägt 12«, haben wir mit GL_TRIANGLE_FAN einen ausgefüllten Kreis als Hintergrund spendiert.

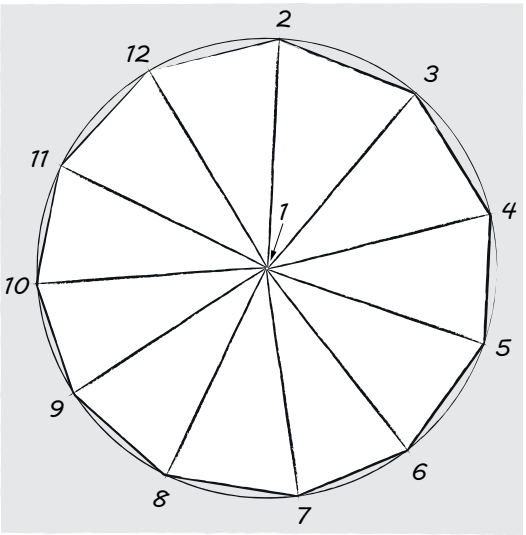


Abbildung A.6 Ein Triangle-Fan

GL_QUADS stellt Polygone mit vier Eckpunkten dar, wobei zu beachten ist, dass alle vier Punkte in einer Ebene liegen müssen.

GL_QUAD_STRIP zeigt verbundene Polygone an, die für sich genommen vier Eckpunkte besitzen.

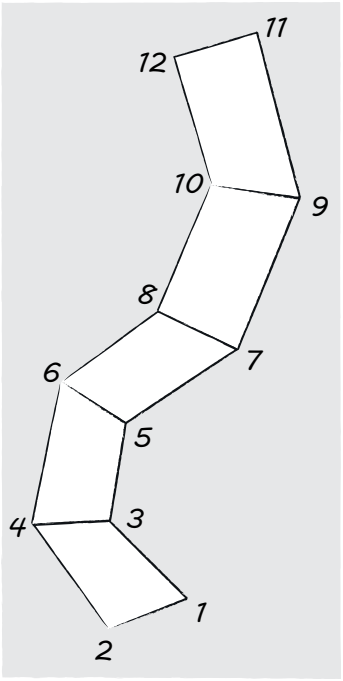


Abbildung A.7 Ein Quad-Strip

`GL_POLYGON` dient zur Darstellung von Polygonen mit beliebig vielen Eckpunkten, wobei alle Punkte in einer Ebene liegen müssen und das Polygon in Summe konvex sein muss. Ein Polygon ist nur dann konvex, wenn es nicht möglich ist, eine Gerade durch das Polygon zu zeichnen, die mehr als einen Eintrittspunkt hat. Das heißt, die Gerade darf das Polygon nur an einer Stelle »betreten« und auch nur an einer Stelle »verlassen«.

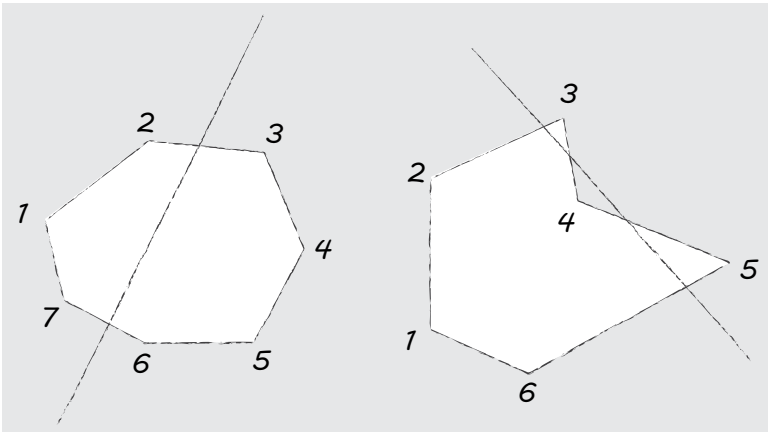


Abbildung A.8 Konvex versus konkav

B Erforderliche Software installieren

Sicher kribbeln Ihnen schon die Finger und Sie möchten direkt mit dem Programmieren des ein oder anderen Programms aus diesem Buch beginnen. Auf der beiligenden CD finden Sie deshalb alles, was Sie benötigen, um sofort loszulegen. Hier eine Übersicht über die enthaltenen Programme, mit denen die ebenfalls auf der CD enthaltenen Beispiele lauffähig sind:

- Java Development Kit 6 (Update 21)
- Eclipse JDT 3.6 (Helios)
- Python 2.6
- PyDev 1.6.1
- NumPy und SciPy
- PIL (Python Imaging Library)
- Primo 3.0.9

B.1 Die Entwicklungsumgebung installieren

In diesem Abschnitt beschreibe ich Ihnen die Installation von Eclipse, der von mir präferierten Entwicklungsumgebung für Python. Eclipse stellt eine in Java entwickelte Plattform dar, die unter anderem auch zur Entwicklung der gleichnamigen IDE (= Integrated Development Environment) verwendet wurde. Ursprünglich war Eclipse ein von IBM initiiertes Projekt, das später kostenlos der Open-Source-Community zur Verfügung gestellt wurde. Inzwischen hat sich Eclipse in vielen Bereichen als Entwicklungsumgebung durchgesetzt. Python wird jedoch nicht von Haus aus durch Eclipse unterstützt, sondern durch ein zusätzlich zu installierendes Plugin namens »PyDev«.

Die Installation von Eclipse ist sehr einfach, denn hierfür müssen Sie lediglich das ZIP-Archiv auspacken. Der enthaltene Ordner *eclipse* stellt den Programmordner dar und ist entsprechend unter *Programme* oder einen anderen aus Ihrer Sicht geeigneten Ort zu kopieren. Im Ordner *eclipse* befindet sich eine ausführbare Datei namens *eclipse.exe*, für die Sie auf dem Desktop eine Verknüpfung anlegen sollten.

Um Eclipse zu starten, benötigen Sie zusätzlich eine Java-Umgebung. Führen Sie also gegebenenfalls vor dem ersten Start von Eclipse die Java-Installation aus. Auf Macs ist Java bereits vorinstalliert, es sei denn, Sie haben Windows installiert. Da

die Versionen auf der CD zum Buch ihre Aktualität irgendwann verlieren, können Sie die aktuellen Versionen auch auf den entsprechenden Internetseiten laden. Eclipse finden Sie unter *http://www.eclipse.org*, Java unter *http://java.sun.com*.

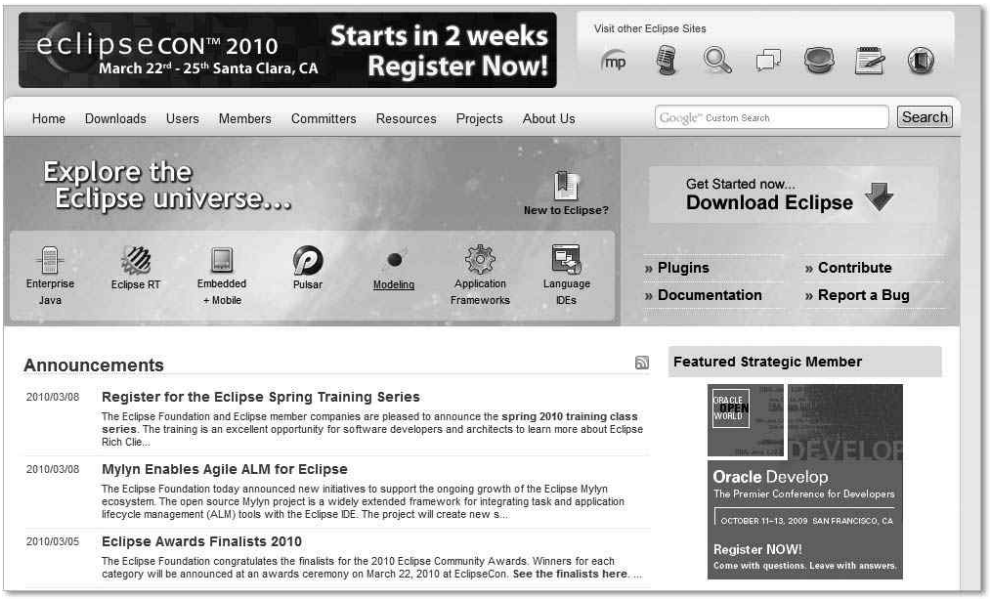


Abbildung B.1 Die Eclipse-Homepage

Auf der Einstiegsseite klicken Sie einfach oben auf Downloads, um auf die folgende Seite zu gelangen.

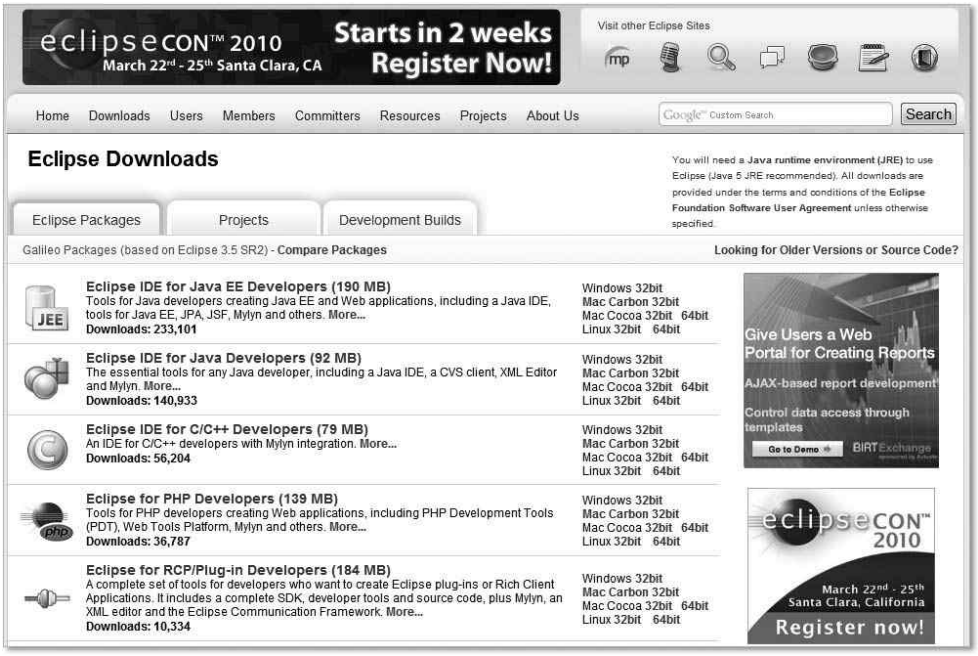


Abbildung B.2 Eclipse-Downloads

Unter den Downloads wählen Sie bitte den für Java-Entwickler. Prinzipiell liefe es zwar auch mit einem abgespeckten Eclipse, aber mit der Version für Java-Entwickler können Sie bei Bedarf auch Jython verwenden. *Jython* ist eine Python-Implementierung, die in der Java Virtual Machine läuft. Mit Jython haben Sie den vollen Zugriff auf alle unter Java zur Verfügung stehenden Bibliotheken und können entsprechend auch sehr gut mit bestehenden Java-Entwicklungen arbeiten oder darauf aufbauen.

Java können Sie unter *http://java.sun.com* herunterladen. Auf der Startseite finden Sie rechts Popular Downloads mit einem Eintrag Java SE – wobei »SE« für »Standard Edition« steht. Mit etwas Glück funktioniert auch direkt der folgende Link: *http://java.sun.com/javase/downloads/*, der die Seite aus Abbildung B.3 erscheinen lassen müsste.

Abbildung B.3 Java-Downloads

Auf der Seite klicken Sie einfach auf JDK Download – »JDK« steht für »Java Development Kit«. Bei Bedarf können Sie natürlich auch eines der angebotenen Bundles downloaden, aber für die Zwecke in diesem Buch ist das nicht notwendig.

B.2 Python unter Windows installieren

[●] Python finden Sie auf der beiliegenden CD im Verzeichnis *Software/<IhrSystem>*.

Alle Python-Beispiele aus diesem Buch laufen mit den Python-Versionen 2.5 und 2.6 – Python 3.x wurde nicht verwendet, da für diese Version viele Bibliotheken noch nicht verfügbar sind. Den Python-Installer erkennen Sie ganz einfach daran, dass im Namen »Python« auftaucht. Nach dem Starten des Setups sehen Sie den Screen aus Abbildung B.4.

Abbildung B.4 Python-Setup

Falls Sie Ihren Rechner mit anderen Benutzern teilen, ist es unter Umständen sinnvoll, nicht für alle Benutzer zu installieren. In dem Fall ist Python für die anderen Benutzer nicht im START-Menü verlinkt. Falls Sie allein an Ihrem Rechner arbeiten, dann ist die Auswahl im Prinzip egal, da Sie vermutlich nur ein Benutzerkonto eingerichtet haben. Ich ziehe es immer vor, die Software für alle Benutzer des Rechners zu installieren.

Im darauffolgenden Screen wählen Sie das Zielverzeichnis für die Installation – es empfiehlt sich, sich hier mit der Vorgabe zu begnügen, da lange Verzeichnisse mit eventuellen Leerzeichen im Pfad zu Problemen führen können.

Abbildung B.5 Auswahl des Zielverzeichnisses

Als Nächstes müssen Sie die zu installierenden Pakete auswählen. Die Vorauswahl installiert alle Pakete, was in der Regel auch angebracht ist, weshalb Sie den Dialog ohne Änderungen einfach mit NEXT bestätigen können.

Abbildung B.6 Auswahl der Pakete

Nach der Auswahl zeigt Ihnen das darauffolgende Fenster den Fortschritt der Installation an. Wenn alles problemlos funktioniert hat, wird die erfolgreiche Installation durch den in Abbildung B.7 dargestellten Screen bestätigt.

Abbildung B.7 Nach erfolgreicher Installation

Die einzelnen Screens können sich von denen, die unter Mac OS angezeigt werden, unterscheiden – der Inhalt ist jedoch weitestgehend identisch.

Falls Sie Eclipse noch nicht installiert haben, können Sie trotzdem schon ein bisschen mit Python spielen, denn es wird IDLE als ein einfacher Editor zum Probieren mitgeliefert.

B.3 Python unter Mac OS installieren

Unter Mac OS müssen Sie ebenfalls Python separat installieren, da die bereits vorinstallierte Version von Python nicht mit Pygame kompatibel ist. Zudem gestaltet sich die Installation unter Mac OS nicht ganz so trivial wie unter Windows. Starten Sie zunächst ein Terminal, um den folgenden Befehl einzugeben:

```
$ sudo sh setuptools-0.6c9-py2.5.egg
```

Listing B.1 Install »setuptools«

Jetzt entpacken Sie *PyOpenGL-3.0.0b8.tar.gz* und fahren mit den folgenden Befehlen fort:

```
$ cd PyOpenGL-3.0.0b8/
$ python setup.py build
$ sudo python setup.py install
```

Listing B.2 2. Install PyOpenGL

Ab jetzt sollte OpenGL innerhalb von Python importierbar sein. Sie können das gerne innerhalb von Eclipse überprüfen. Hierfür sollten Sie den Python-Interpreter zunächst noch einmal entfernen und neu hinzufügen, damit alle Abhängigkeiten neu untersucht werden. Hierauf sollte sich PyGame ganz normal installieren lassen, so dass keine Eingaben im Terminal mehr erforderlich sind.

Falls Sie direkt versucht haben, PyGame zu installieren, so hat der Installer Ihnen eine Fehlermeldung präsentiert, die besagt, dass PyGame eine Python-Installation voraussetzt. Tatsächlich hieß das aber, dass die vorinstallierte Version von Python nicht reicht und Sie eine andere aufspielen müssen.

B.4 Java unter Windows installieren

Die Installation von Java stellt auch keine größere Hürde dar. Der Download von Java ist im Abschnitt B.1 beschrieben. Im ersten Screen müssen Sie die Lizenzbedingungen akzeptieren. Danach wählen Sie wie gewohnt das Zielverzeichnis und die zu installierenden Komponenten aus.

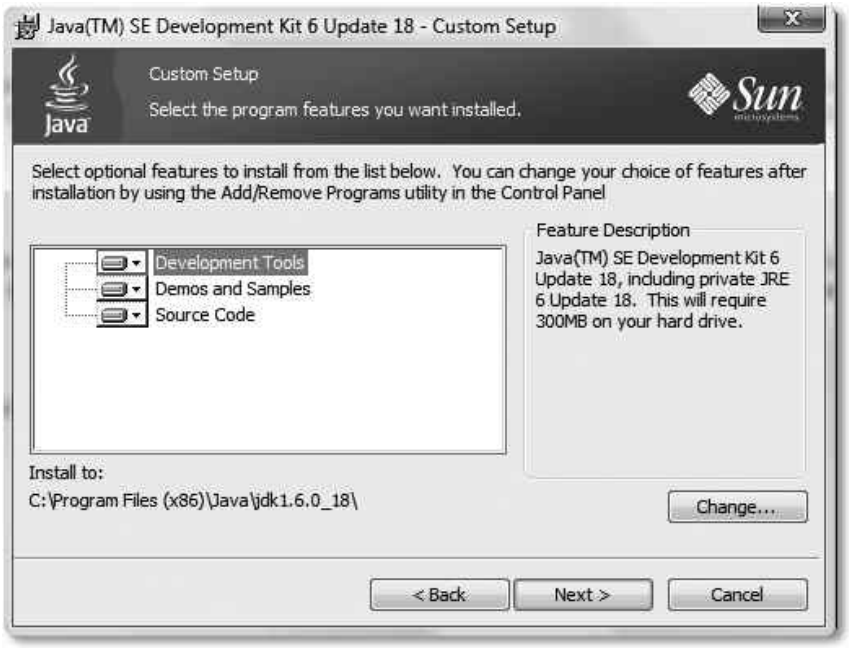


Abbildung B.8 Java-Setup

Die Vorauswahl ist auch in diesem Fall absolut in Ordnung, so dass Sie dieses Fenster einfach nur bestätigen müssen. Im darauffolgenden Screen wird der Installationsfortschritt angezeigt. Eine erfolgreiche Installation wird zum Schluss mit der Anzeige aus Abbildung B.9 bestätigt.

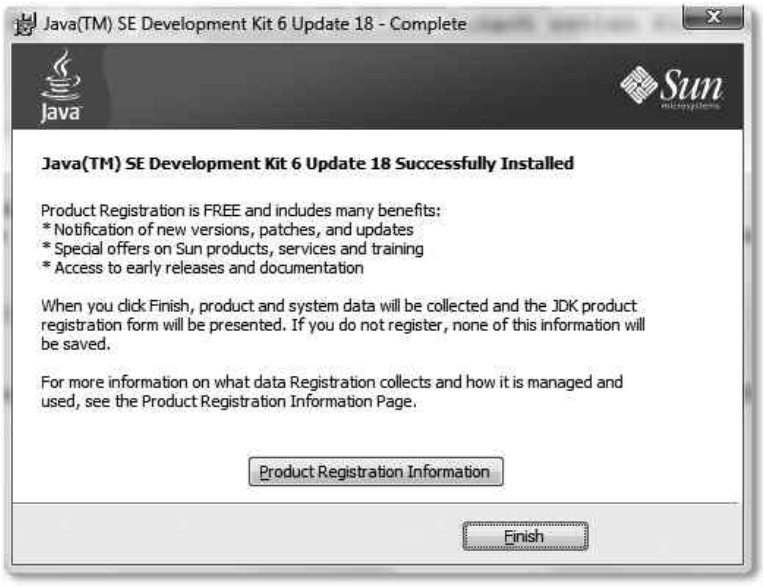


Abbildung B.9 Nach erfolgreicher Installation

Auf die optional angebotene Registrierung würde ich an Ihrer Stelle einfach verzichten; außer Werbemails an Ihre E-Mail-Adresse wird die Registrierung Ihnen kaum Vorteile bieten.

B.5 Pygame unter Windows installieren

Um Pygame zu installieren, verwenden Sie einfach die mitgelieferte Installationsdatei auf der beiliegenden CD unter *tools/pygame*, oder laden Sie sich eine aktuelle Version von *http://www.pygame.org/download.shtml* herunter. Unter Mac OS kann es notwendig sein, vor der Installation von Pygame eine neuere Version von Python zu installieren.

B.6 PyDev installieren und konfigurieren

Um unter Eclipse komfortabel Python-Programme entwickeln zu können, empfiehlt es sich, ein passendes Plugin für die Python-Entwicklung zu installieren. Mein persönlicher Favorit ist hierbei PyDev, das Sie auf der beiliegenden CD unter *Tools/IDE/PyDev* finden. Alternativ können Sie die Software aber auch unter *http://pydev.org/download.html* herunterladen.

Wie für Eclipse üblich gibt es zwei verschiedene Varianten der Installation. Die erste wäre die Nutzung des Update-Managers von Eclipse, wofür Sie lediglich die Update-Site von PyDev wissen müssen, die folgendermaßen lautet: *http://pydev.org/updates*. Die zweite Variante wäre das Laden des kompletten Pa-

ketes als ZIP-File von SourceForge, woraufhin Sie die im ZIP-File enthaltenen Ordner *Feature* und *Plugins* in Ihren Eclipse-Ordner kopieren müssten und Eclipse einmal neu starten sollten.

Schauen wir uns zunächst die Installation über den Update-Manager von Eclipse an. Zum Starten des Update-Managers gehen Sie einfach in das Menü HELP, und wählen Sie INSTALL NEW SOFTWARE... aus.

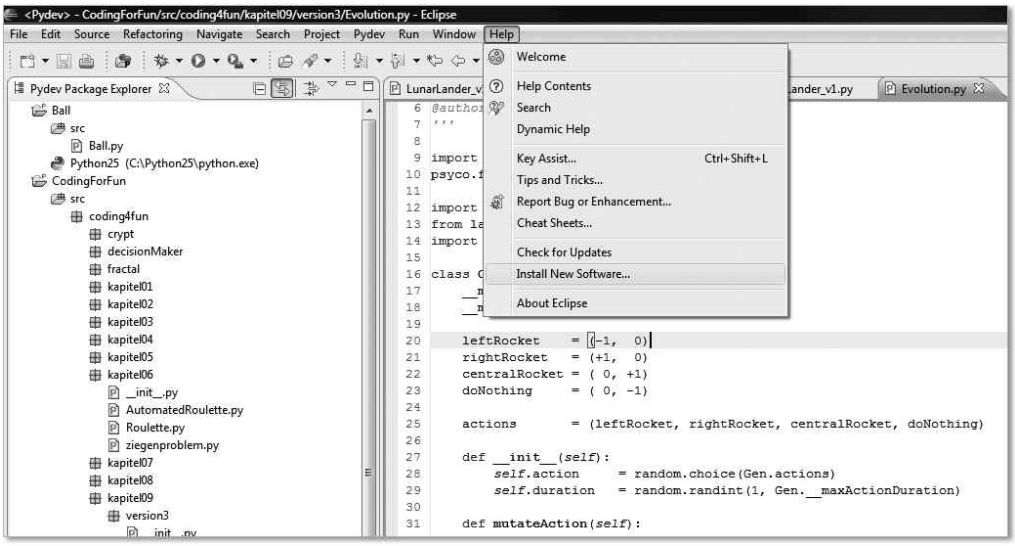


Abbildung B.10 Plugin für Eclipse installieren

Jetzt sollten Sie ein Dialogfeld sehen, in dem die bereits bekannten Update-Sites gelistet sind. Ganz oben in diesem Dialogfeld finden Sie rechts die Möglichkeit, eine neue Update-Site anzulegen. Klicken Sie auf ADD.

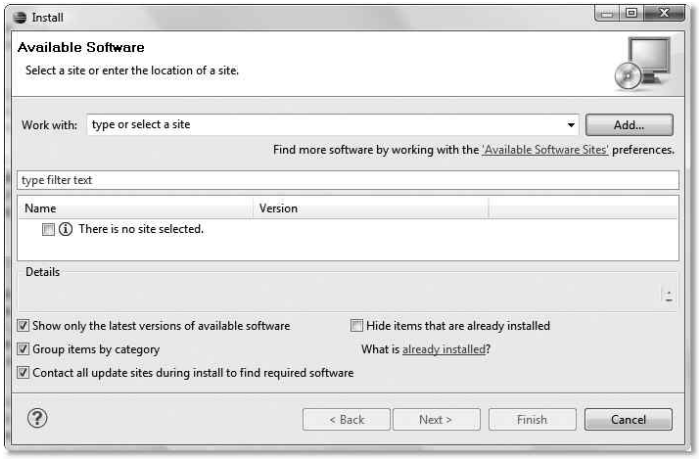


Abbildung B.11 Update-Manager

Im darauffolgenden Dialog fügen Sie die Update-Site *http://pydev.org/updates* ein und vergeben einen passenden Namen für die Quelle. Sie können einfach zweimal die URL einfügen – das sollte aussagekräftig genug sein.

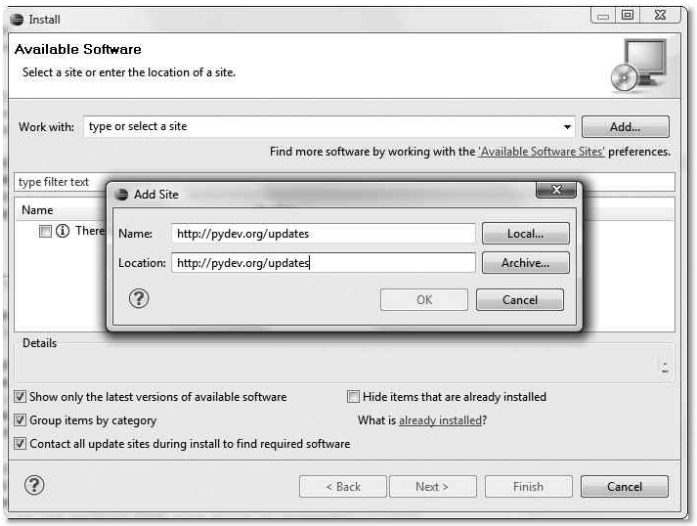


Abbildung B.12 Neue Update-Site hinzufügen

Danach erscheint im ursprünglichen Dialog eine Liste mit Checkboxen für die nähere Auswahl der zu installierenden Pakete. Wählen Sie einfach die Checkbox, die die größte Versionsnummer repräsentiert, also die aktuellste Version von PyDev. Falls Sie auch mit »Mylyn« arbeiten, so können Sie hierfür eine Schnittstellenintegration laden. Falls Sie Mylyn nicht kennen, dann brauchen Sie die Checkbox auch nicht auszuwählen.

Was ist Mylyn?

Mylyn ist ein PlugIn für Eclipse, dass eine aufgabenfokussierte Benutzeroberfläche anbietet. Den einzelnen Entwicklungsdateien sind dabei bestimmten Aufgaben zugeordnet, so dass Mylyn für die verschiedenen Aufgaben entsprechende Sichten bereitstellen kann, in denen nur die für die Aufgabe benötigten Dateien und Menüs angezeigt werden. Dadurch soll der Entwickler so wenig ablenkende Informationen wie möglich bekommen, so dass er sich optimal auf die gerade zu erledigende Aufgabe konzentrieren kann.

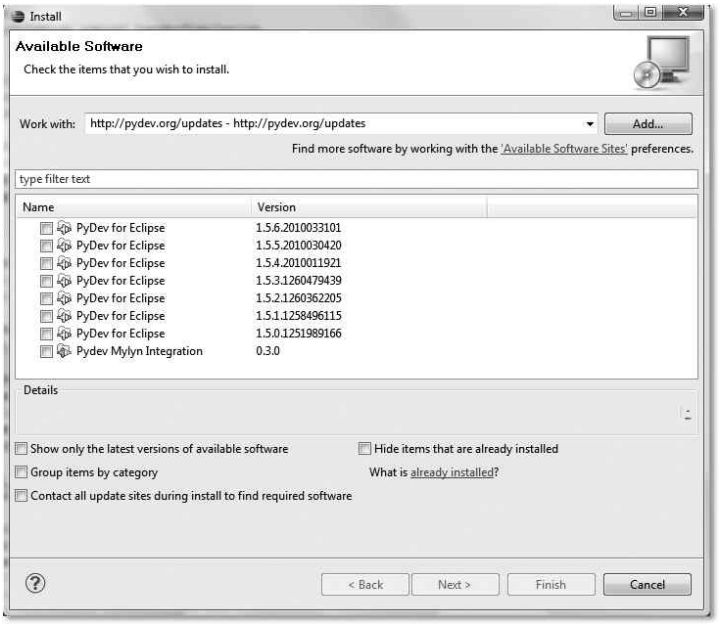


Abbildung B.13 Paketauswahl

Nach der Auswahl des entsprechenden Paketes durchschreiten Sie über den NEXT-Button verschiedene Dialoge, die Sie durch die Installation sowie den vorherigen Download von PyDev führen.

Ganz zum Schluss sollten Sie Eclipse neu starten, worauf Sie auch automatisch hingewiesen werden. Nach dem Neustart müssen Sie noch den zu verwendenden Python-Interpreter festlegen. Klicken Sie dafür im Menü WINDOW auf PREFERENCES, was den Dialog aus Abbildung B.14 zutage fördert.

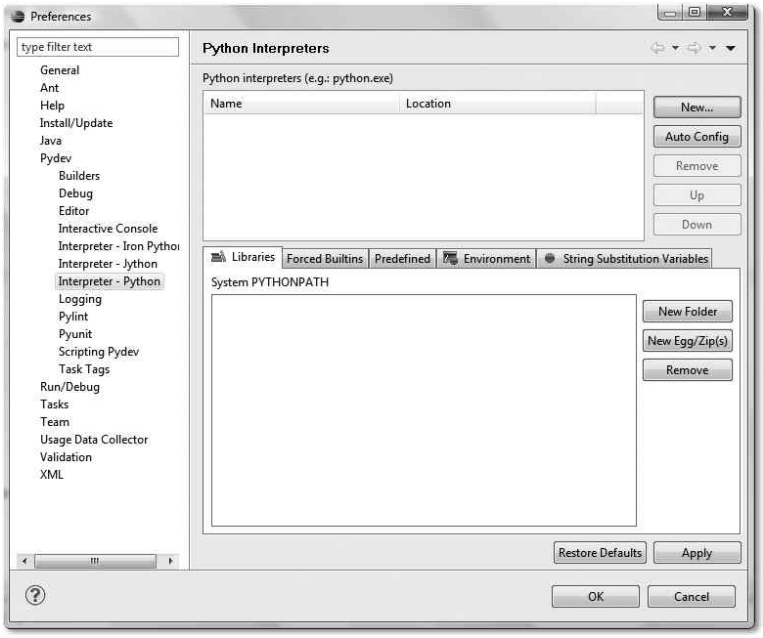


Abbildung B.14 PyDev-Interpreter festlegen

Im Bild ist schon alles korrekt aufgeklappt, bei Ihnen wird dies noch nicht der Fall sein, so dass Sie links zunächst PyDev auswählen müssen und darauf Interpreter – Python. Dann sollte Ihr Dialog so wie in der Darstellung aussehen. Nun klicken Sie rechts auf New, um den zu verwendenden Interpreter auszuwählen. Sie sollten hierfür vorher bereits Python sowie Pygame installiert haben. Unter Mac OS ist Python bereits vorinstalliert und befindet sich im Ordner *usr/bin/*. Den Interpreter finden Sie im Installationsordner Ihrer Python-Installation. Die Auswahl des Python-Interpreters erfolgt über den Dialog aus Abbildung B.15.

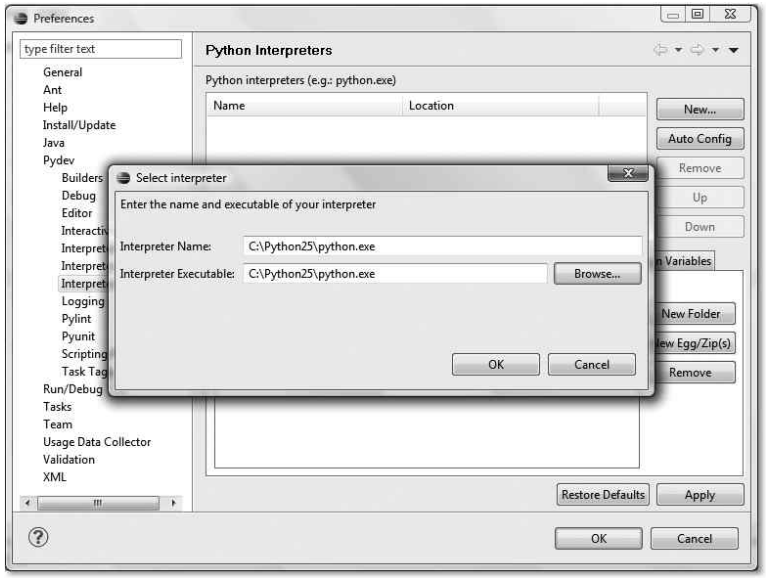


Abbildung B.15 Build-Umgebung initialisieren

Nach der Auswahl sehen Sie das Fenster aus Abbildung B.16.

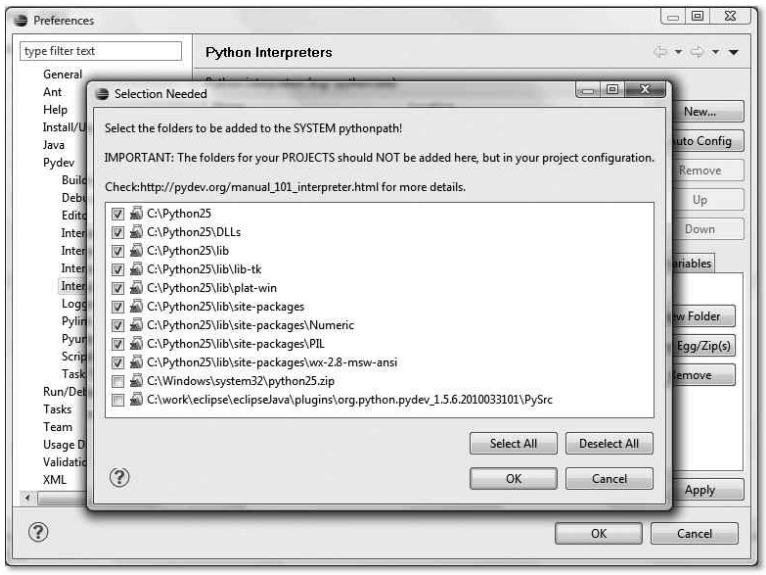


Abbildung B.16 Build-Umgebung initialisieren

Die Vorauswahl passt bereits perfekt, so dass Sie den Dialog unverändert bestätigen können. Nachdem die Build-Umgebung einmal komplett durchkompiliert wurde, sollte der Screen aus Abbildung B.17 erscheinen.

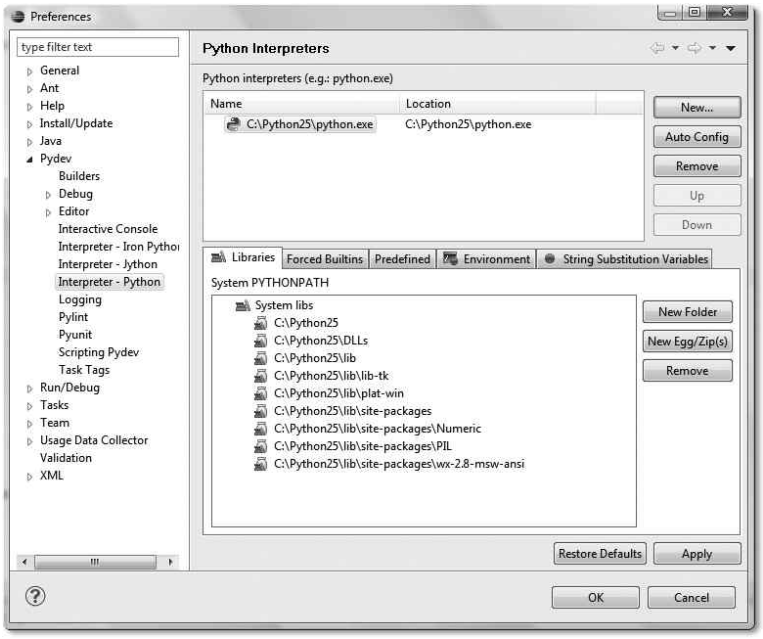


Abbildung B.17 Build-Umgebung initialisieren

Falls Sie nachträglich noch andere Bibliotheken installieren, so ist es ratsam, den Python-Interpreter noch einmal zu entfernen und neu hinzuzufügen, um die Build-Umgebung erneut zu kompilieren. Ansonsten kann es sein, dass eine neu installierte Bibliothek als nicht vorhanden dargestellt wird, da die Build-Umgebung nach der Installation nicht aktualisiert wurde.

C Literaturverzeichnis

Knuth 1999:
Donald Ervin Knuth.
The Art of Computer Programming 1–3.
Addison-Wesley Longman, Amsterdam, 1999.

McConnell 2004:
Steve McConnell.
Code Complete 2nd Edition.
Microsoft Press, 2004.

Oram 2007:
Andy Oram and Greg Wilson.
Beautiful Code.
O'Reilly, 2007.

Fowler 2005:
Martin Fowler.
Refactoring – Improving the Design of Existing Code.
Addison-Wesley, 2005.

Gamma 2005:
Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides.
Design Patterns – Elements of Reusable Object-Oriented Software.
Addison-Wesley, 2005.

Martin 2009:
Robert C. Martin.
Clean Code.
Prentice Hall, 2009.

Leffingwell 2001:
Dean Leffingwell, Don Widrig.
Managing Software Requirements – A Unified Approach.
Addison-Wesley, 2001.

Kerievsky 2005:
Joshua Kerievsky.
Refactoring to Patterns.
Addison-Wesley, 2005.

Siedersleben 2004:
Johannes Siedersleben.
Moderne Software-Architektur – Umsichtig planen, robust bauen mit Quasar.
dpunkt.verlag, 2004.

Bentley 2000:
Jon Bentley.
Programming Pearls 2nd Edition.
Addison-Wesley, 2000.

Graham 1998:
Ronald L. Graham, Donald E. Knuth, Oren Patashnik.
Concrete Mathematics 2nd Edition.
Addison-Wesley, 1998.

Aho 2008:
Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullmann
Compiler – Prinzipien, Techniken und Werkzeuge 2. Auflage
Pearson Studium, 2008.

Graham 1998:
Ronald L. Graham, Donald E. Knuth, Oren Patashnik.
Concrete Mathematics 2nd Edition.
Addison-Wesley, 1998.

Härder 1999:
Theo Härder, Erhard Rahm.
Datenbanksysteme – Konzepte und Techniken der Implementierung.
Springer, 1999.

Saake 2004:
Gunter Saake, Kai-Uwe Sattler.
Algorithmen und Datenstrukturen – Eine Einführung mit Java 2. Auflage.
dpunkt.lehrbuch, 2004.

Buckland 2002:
Mat Buckland.
AI Technics for Game Programming.
Premier Press, 2002.

Wise 2004:
Edwin Wise.
Hands-On AI with Java – Smart Gaming, Robotics, and More.
McGrawHill, 2004.

Segaran 2007:
Toby Segaran.
Programming Collective Intelligence.
O'Reilly, 2007.

Millington 2007:
Ian Millington.
Game Physics Engine Development.
Morgan Kaufmann, 2007.

Davison 2005:
Andrew Davison.
Killer Game Programming.
O'Reilly, 2005.

Scharlau 1980:
W. Scharlau, H. Opolka.
Von Fermat bis Minkowski – Eine Vorlesung über Zahlentheorie und ihre Entwicklung.
Springer, 1980.

Bundschuh 2008:
Prof. Dr. Peter Bundschuh.
Einführung in die Zahlentheorie 6. Auflage.
Springer, 2008.

Polya 1988:
G. Polya.
How to Solve It – A new Aspect of Mathematical Method.
Princeton Science library, 1988.

Adams 2001:
James L. Adams.
Conceptual Blockbusting – A Guide to Better Ideas.
Basic Books, 2001.

Beutelspacher 2007:
Albrecht Beutelspacher.
Kryptologie 8. Auflage
Vieweg, 2007

Elliott 1995:
Robert James Elliott.
Hidden Markov Models, Estimation and Control: 029 (Applications of Mathematics)
Springer, 1995.

Press 2007: William H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery.
Numerical Recipes 3rd Edition: The Art of Scientific Computing
Cambridge University Press, 2007.

Index

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Z