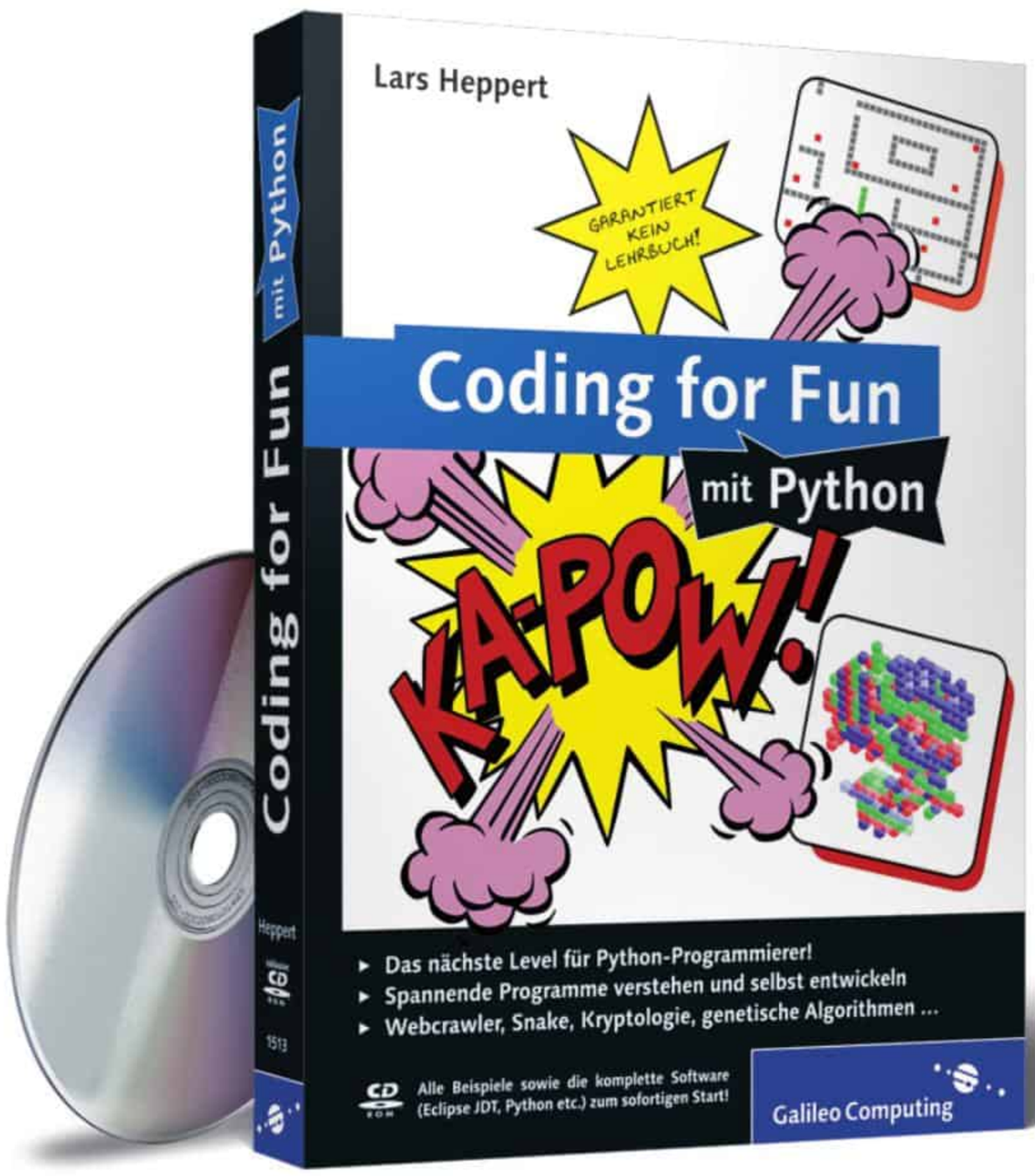
Lars Heppert

Coding for Fun mit Python



Galileo Press

Auf einen Blick

Inhalt

Lars Heppert, Coding for Fun mit Python, Version 1.02 vom 12. August 2010 Galileo Press, ISBN: ohne ISBN
Layout: gp.cls, Version 3.3.002 (11th July 2010), (C) Dr. Daniel Lauer, Bonn Mit TEX bearbeitet am 9. September 2010, 0:00 Uhr Umlaut-Check: äöüÄÖÜ.
S. 5 Einstellungen: ohne Marginalien – mit Seitenrand – mit Taben – mit Kapitel-Intro – Einsteiger – Zoom auf 100.0%

Lars Heppert, Coding for Fun mit Python, Version 1.02 vom 12. August 2010 Galileo Press, ISBN: ohne ISBN
Layout: gp.cls, Version 3.3.002 (11th July 2010), (C) Dr. Daniel Lauer, Bonn Mit TEX bearbeitet am 9. September 2010, 0:00 Uhr Umlaut-Check: äöüÄÖÜ.
S. 6 Einstellungen: ohne Marginalien – mit Seitenrand – mit Taben – mit Kapitel-Intro – Einsteiger – Zoom auf 100.0%

Lars Heppert, Coding for Fun mit Python, Version 1.02 vom 12. August 2010 Galileo Press, ISBN: ohne ISBN
Layout: gp.cls, Version 3.3.002 (11th July 2010), (C) Dr. Daniel Lauer, Bonn Mit TEX bearbeitet am 9. September 2010, 0:00 Uhr Umlaut-Check: äöüÄÖÜ.
S. 7 Einstellungen: ohne Marginalien – mit Seitenrand – mit Taben – mit Kapitel-Intro – Einsteiger – Zoom auf 100.0%

Lars Heppert, Coding for Fun mit Python, Version 1.02 vom 12. August 2010 Galileo Press, ISBN: ohne ISBN
Layout: gp.cls, Version 3.3.002 (11th July 2010), (C) Dr. Daniel Lauer, Bonn Mit TEX bearbeitet am 9. September 2010, 0:00 Uhr Umlaut-Check: äöüÄÖÜ.
S. 8 Einstellungen: ohne Marginalien – mit Seitenrand – mit Taben – mit Kapitel-Intro – Einsteiger – Zoom auf 100.0%

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.*

Lars Heppert, Coding for Fun mit Python, Version 1.02 vom 12. August 2010 Galileo Press, ISBN: ohne ISBN
Layout: gp.cls, Version 3.3.002 (11th July 2010), (C) Dr. Daniel Lauer, Bonn Mit TEX bearbeitet am 9. September 2010, 0:00 Uhr Umlaut-Check: äöüÄÖÜ.
S. 9 Einstellungen: ohne Marginalien – mit Seitenrand – mit Taben – mit Kapitel-Intro – Einsteiger – Zoom auf 100.0%

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

Lars Heppert, Coding for Fun mit Python, Version 1.02 vom 12. August 2010 Galileo Press, ISBN: ohne ISBN
Layout: gp.cls, Version 3.3.002 (11th July 2010), (C) Dr. Daniel Lauer, Bonn Mit TEX bearbeitet am 9. September 2010, 0:00 Uhr Umlaut-Check: äöüÄÖÜ.
S. 10 Einstellungen: ohne Marginalien – mit Seitenrand – mit Taben – mit Kapitel-Intro – Einsteiger – Zoom auf 100.0%

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

Lars Heppert, Coding for Fun mit Python, Version 1.02 vom 12. August 2010 Galileo Press, ISBN: ohne ISBN
Layout: gp.cls, Version 3.3.002 (11th July 2010), (C) Dr. Daniel Lauer, Bonn Mit TEX bearbeitet am 9. September 2010, 0:00 Uhr Umlaut-Check: äöüÄÖÜ.
S. 11 Einstellungen: ohne Marginalien – mit Seitenrand – mit Taben – mit Kapitel-Intro – Einsteiger – Zoom auf 100.0%

»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.

Lars Heppert, Coding for Fun mit Python, Version 1.02 vom 12. August 2010 Galileo Press, ISBN: ohne ISBN
Layout: gp.cls, Version 3.3.002 (11th July 2010), (C) Dr. Daniel Lauer, Bonn Mit TEX bearbeitet am 9. September 2010, 0:00 Uhr Umlaut-Check: äöüÄÖÜ.
S. 45 Einstellungen: ohne Marginalien – mit Seitenrand – mit Taben – mit Kapitel-Intro – Einsteiger – Zoom auf 100.0%

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.

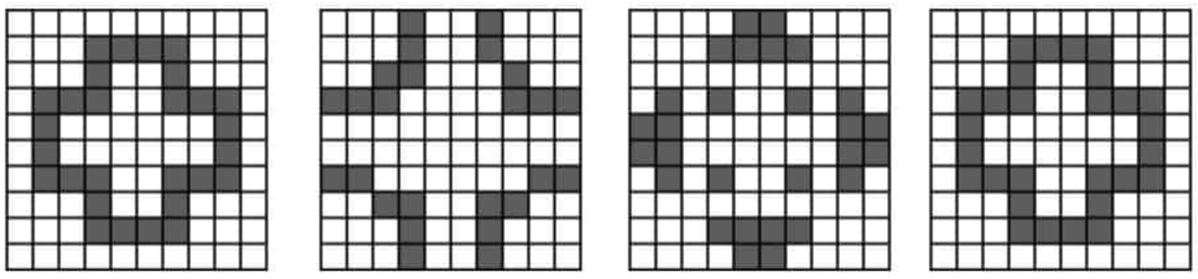


Abbildung 2.1 Periode-3-Oszillator

Lars Heppert, Coding for Fun mit Python, Version 1.02 vom 12. August 2010 Galileo Press, ISBN: ohne ISBN
Layout: gp.cls, Version 3.3.002 (11th July 2010), (C) Dr. Daniel Lauer, Bonn Mit TEX bearbeitet am 9. September 2010, 0:00 Uhr Umlaut-Check: äöüÄÖÜ.
S. 46 Einstellungen: ohne Marginalien – mit Seitenrand – mit Taben – mit Kapitel-Intro – Einsteiger – Zoom auf 100.0%

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.

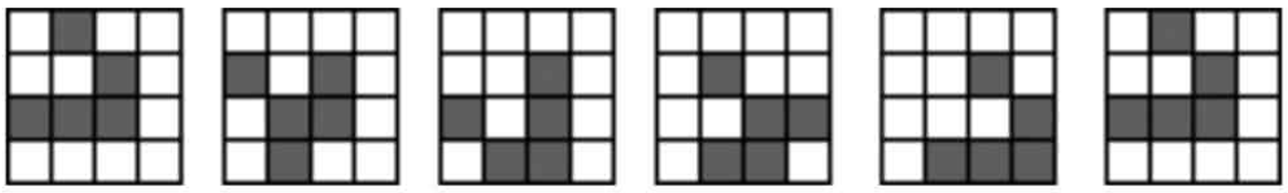


Abbildung 2.2 Gleiter

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.

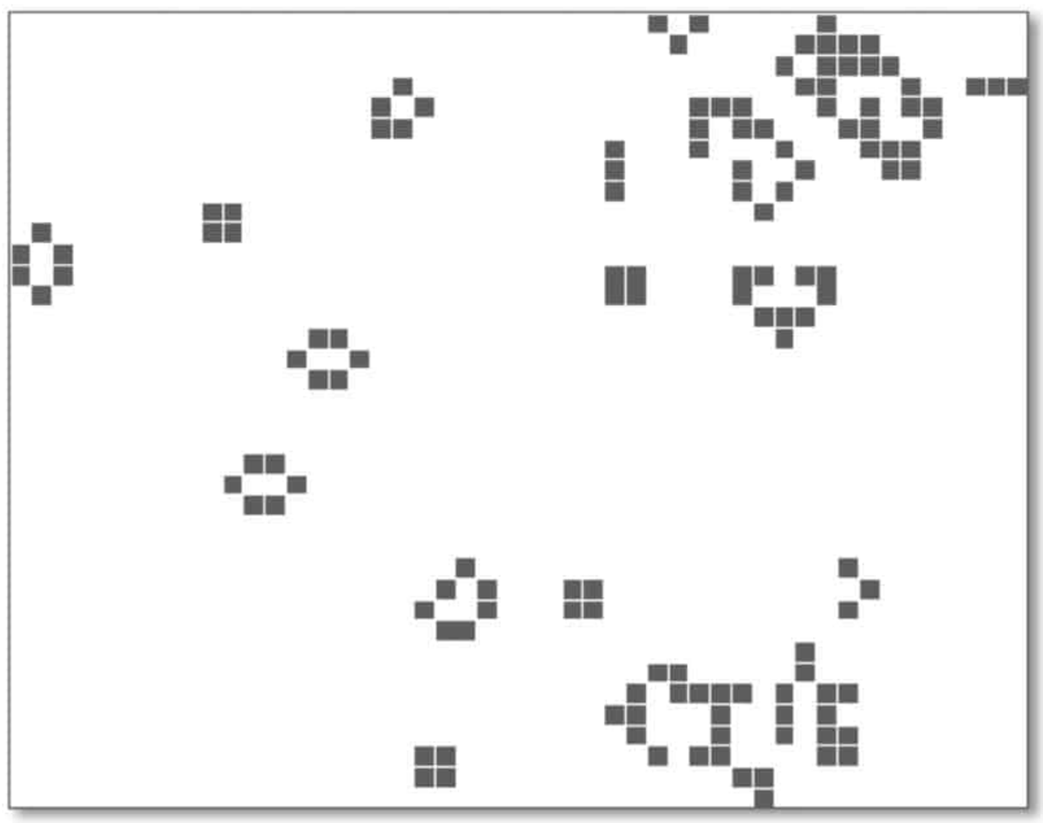


Abbildung 2.3 »Game of Life« in 2D

Lars Heppert, Coding for Fun mit Python, Version 1.02 vom 12. August 2010 Galileo Press, ISBN: ohne ISBN
Layout: gp.cls, Version 3.3.002 (11th July 2010), (C) Dr. Daniel Lauer, Bonn Mit TEX bearbeitet am 9. September 2010, 0:00 Uhr Umlaut-Check: äöüÄÖÜ.
S. 47 Einstellungen: ohne Marginalien – mit Seitenrand – mit Taben – mit Kapitel-Intro – Einsteiger – Zoom auf 100.0%

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«.

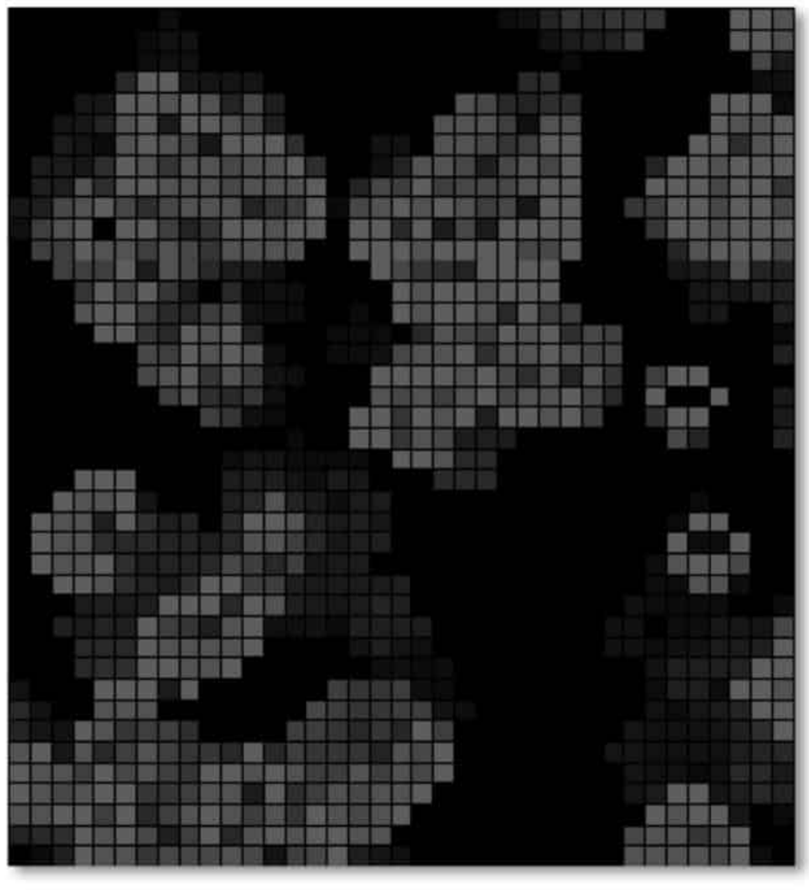


Abbildung 2.4 »Game of Life« in 2D mit »langsam sterbenden« Lebewesen

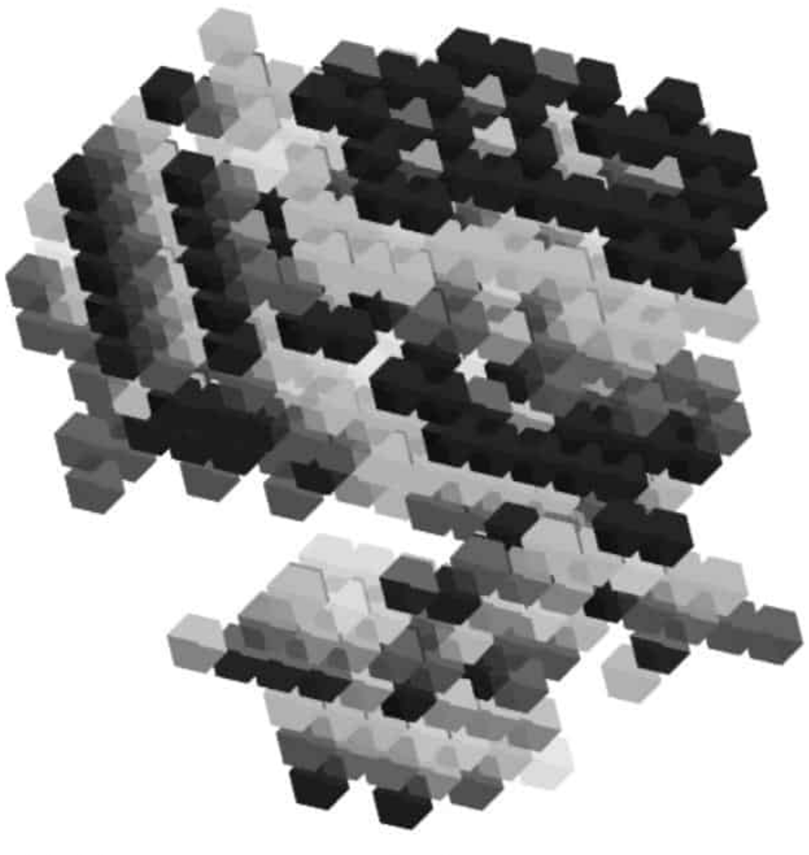


Abbildung 2.5 »Game of Life« in 3D

Lars Heppert, Coding for Fun mit Python, Version 1.02 vom 12. August 2010 Galileo Press, ISBN: ohne ISBN
Layout: gp.cls, Version 3.3.002 (11th July 2010), (C) Dr. Daniel Lauer, Bonn Mit TEX bearbeitet am 9. September 2010, 0:00 Uhr Umlaut-Check: äöüÄÖÜ.
S. 48 Einstellungen: ohne Marginalien – mit Seitenrand – mit Taben – mit Kapitel-Intro – Einsteiger – Zoom auf 100.0%

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

Lars Heppert, Coding for Fun mit Python, Version 1.02 vom 12. August 2010 Galileo Press, ISBN: ohne ISBN
Layout: gp.cls, Version 3.3.002 (11th July 2010), (C) Dr. Daniel Lauer, Bonn Mit TEX bearbeitet am 9. September 2010, 0:00 Uhr Umlaut-Check: äöüÄÖÜ.
S. 49 Einstellungen: ohne Marginalien – mit Seitenrand – mit Taben – mit Kapitel-Intro – Einsteiger – Zoom auf 100.0%

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.

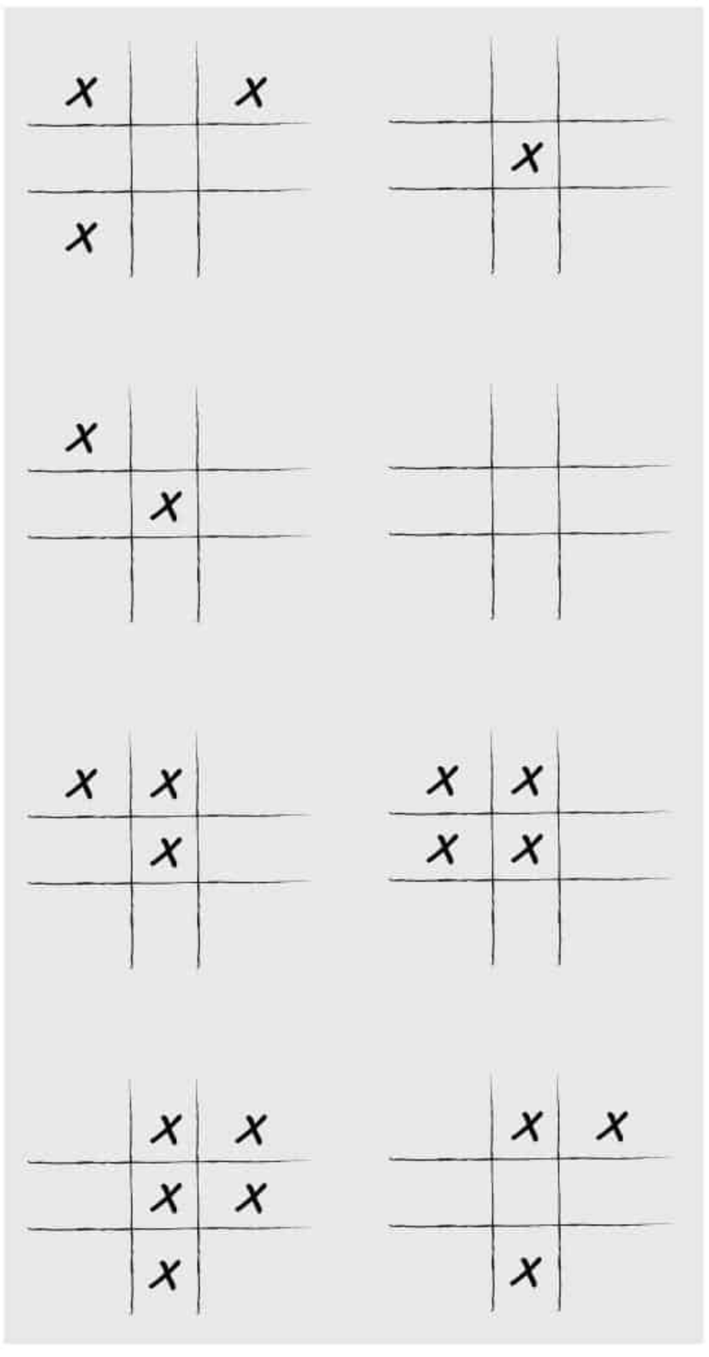


Abbildung 2.6 Die Regeln des »Game of Life«

Lars Heppert, Coding for Fun mit Python, Version 1.02 vom 12. August 2010 Galileo Press, ISBN: ohne ISBN
Layout: gp.cls, Version 3.3.002 (11th July 2010), (C) Dr. Daniel Lauer, Bonn Mit TEX bearbeitet am 9. September 2010, 0:00 Uhr Umlaut-Check: äöüÄÖÜ.
S. 50 Einstellungen: ohne Marginalien – mit Seitenrand – mit Taben – mit Kapitel-Intro – Einsteiger – Zoom auf 100.0%

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
```


Lars Heppert, Coding for Fun mit Python, Version 1.02 vom 12. August 2010 Galileo Press, ISBN: ohne ISBN
Layout: gp.cls, Version 3.3.002 (11th July 2010), (C) Dr. Daniel Lauer, Bonn Mit TEX bearbeitet am 9. September 2010, 0:00 Uhr Umlaut-Check: äöüÄÖÜ.
S. 51 Einstellungen: ohne Marginalien – mit Seitenrand – mit Taben – mit Kapitel-Intro – Einsteiger – Zoom auf 100.0%

```
            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

Lars Heppert, Coding for Fun mit Python, Version 1.02 vom 12. August 2010 Galileo Press, ISBN: ohne ISBN
Layout: gp.cls, Version 3.3.002 (11th July 2010), (C) Dr. Daniel Lauer, Bonn Mit TEX bearbeitet am 9. September 2010, 0:00 Uhr Umlaut-Check: äöüÄÖÜ.
S. 52 Einstellungen: ohne Marginalien – mit Seitenrand – mit Taben – mit Kapitel-Intro – Einsteiger – Zoom auf 100.0%

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:

Lars Heppert, Coding for Fun mit Python, Version 1.02 vom 12. August 2010 Galileo Press, ISBN: ohne ISBN
Layout: gp.cls, Version 3.3.002 (11th July 2010), (C) Dr. Daniel Lauer, Bonn Mit TEX bearbeitet am 9. September 2010, 0:00 Uhr Umlaut-Check: äöüÄÖÜ.
S. 53 Einstellungen: ohne Marginalien – mit Seitenrand – mit Taben – mit Kapitel-Intro – Einsteiger – Zoom auf 100.0%

```
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.

Lars Heppert, Coding for Fun mit Python, Version 1.02 vom 12. August 2010 Galileo Press, ISBN: ohne ISBN
Layout: gp.cls, Version 3.3.002 (11th July 2010), (C) Dr. Daniel Lauer, Bonn Mit TEX bearbeitet am 9. September 2010, 0:00 Uhr Umlaut-Check: äöüÄÖÜ.
S. 54 Einstellungen: ohne Marginalien – mit Seitenrand – mit Taben – mit Kapitel-Intro – Einsteiger – Zoom auf 100.0%

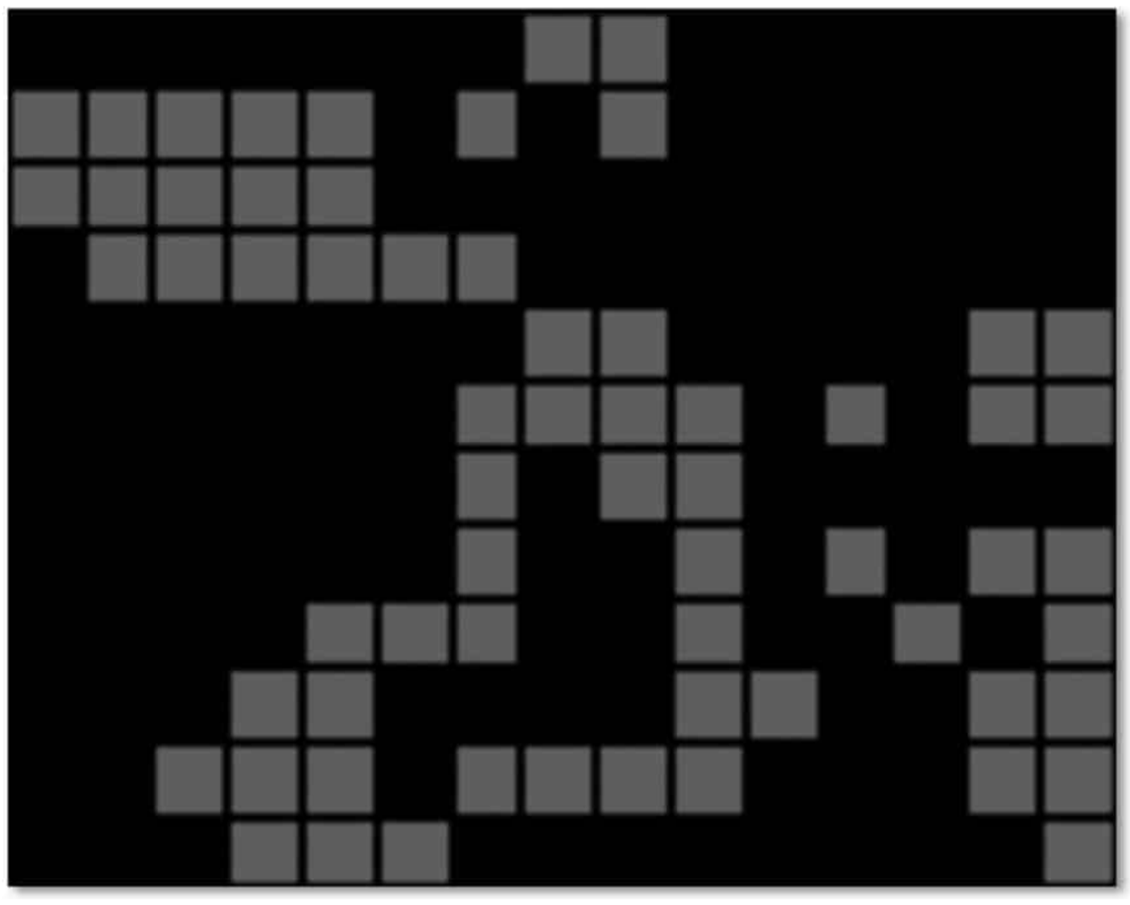


Abbildung 2.7 Das Gitternetz definiert sich wie ein Loch durch seine Umgebung – der Hintergrund scheint einfach durch

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

Lars Heppert, Coding for Fun mit Python, Version 1.02 vom 12. August 2010 Galileo Press, ISBN: ohne ISBN
Layout: gp.cls, Version 3.3.002 (11th July 2010), (C) Dr. Daniel Lauer, Bonn Mit TEX bearbeitet am 9. September 2010, 0:00 Uhr Umlaut-Check: äöüÄÖÜ.
S. 55 Einstellungen: ohne Marginalien – mit Seitenrand – mit Taben – mit Kapitel-Intro – Einsteiger – Zoom auf 100.0%

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.

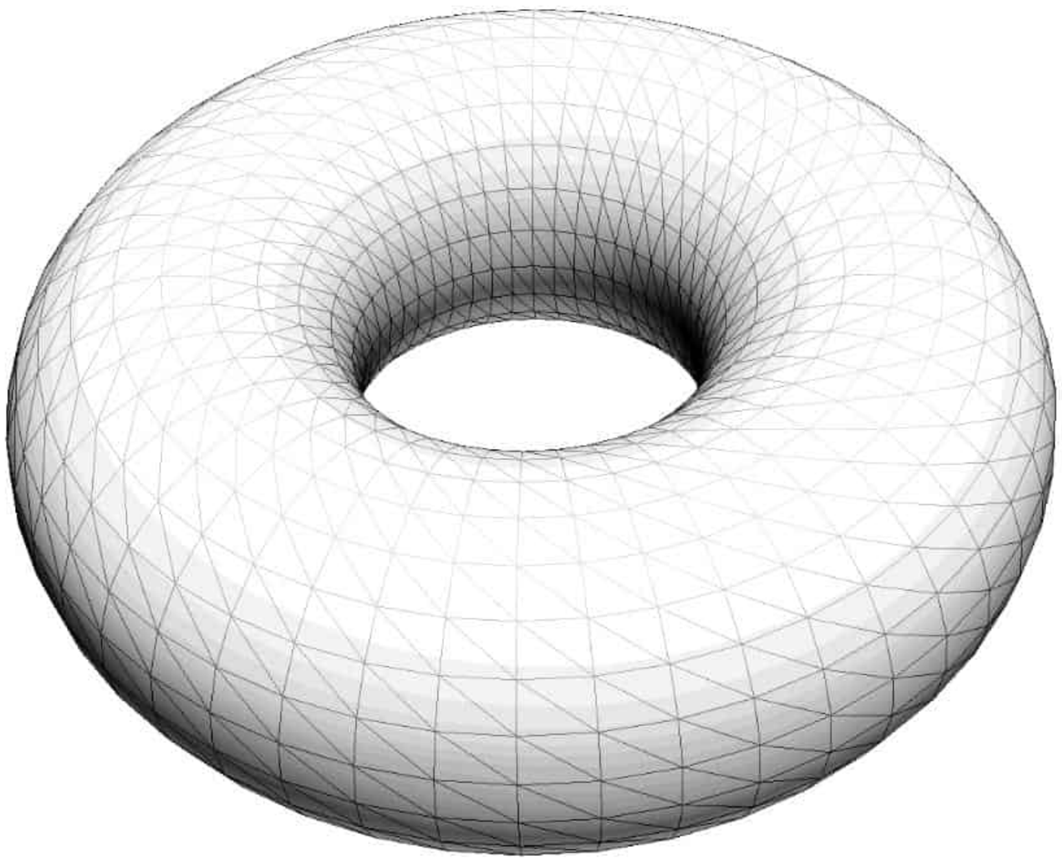


Abbildung 2.8 Eine Donut-Welt (in der Mathematik ein »Torus«)

Lars Heppert, Coding for Fun mit Python, Version 1.02 vom 12. August 2010 Galileo Press, ISBN: ohne ISBN
Layout: gp.cls, Version 3.3.002 (11th July 2010), (C) Dr. Daniel Lauer, Bonn Mit TEX bearbeitet am 9. September 2010, 0:00 Uhr Umlaut-Check: äöüÄÖÜ.
S. 56 Einstellungen: ohne Marginalien – mit Seitenrand – mit Taben – mit Kapitel-Intro – Einsteiger – Zoom auf 100.0%

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].
```


Lars Heppert, Coding for Fun mit Python, Version 1.02 vom 12. August 2010 Galileo Press, ISBN: ohne ISBN
Layout: gp.cls, Version 3.3.002 (11th July 2010), (C) Dr. Daniel Lauer, Bonn Mit TEX bearbeitet am 9. September 2010, 0:00 Uhr Umlaut-Check: äöüÄÖÜ.
S. 57 Einstellungen: ohne Marginalien – mit Seitenrand – mit Taben – mit Kapitel-Intro – Einsteiger – Zoom auf 100.0%

```
                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

Lars Heppert, Coding for Fun mit Python, Version 1.02 vom 12. August 2010 Galileo Press, ISBN: ohne ISBN
Layout: gp.cls, Version 3.3.002 (11th July 2010), (C) Dr. Daniel Lauer, Bonn Mit TEX bearbeitet am 9. September 2010, 0:00 Uhr Umlaut-Check: äöüÄÖÜ.
S. 58 Einstellungen: ohne Marginalien – mit Seitenrand – mit Taben – mit Kapitel-Intro – Einsteiger – Zoom auf 100.0%

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*. [o]

```
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)
```


Lars Heppert, Coding for Fun mit Python, Version 1.02 vom 12. August 2010 Galileo Press, ISBN: ohne ISBN
Layout: gp.cls, Version 3.3.002 (11th July 2010), (C) Dr. Daniel Lauer, Bonn Mit TEX bearbeitet am 9. September 2010, 0:00 Uhr Umlaut-Check: äöüÄÖÜ.
S. 59 Einstellungen: ohne Marginalien – mit Seitenrand – mit Taben – mit Kapitel-Intro – Einsteiger – Zoom auf 100.0%

```
            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

Lars Heppert, Coding for Fun mit Python, Version 1.02 vom 12. August 2010 Galileo Press, ISBN: ohne ISBN
Layout: gp.cls, Version 3.3.002 (11th July 2010), (C) Dr. Daniel Lauer, Bonn Mit TEX bearbeitet am 9. September 2010, 0:00 Uhr Umlaut-Check: äöüÄÖÜ.
S. 60 Einstellungen: ohne Marginalien – mit Seitenrand – mit Taben – mit Kapitel-Intro – Einsteiger – Zoom auf 100.0%

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

Lars Heppert, Coding for Fun mit Python, Version 1.02 vom 12. August 2010 Galileo Press, ISBN: ohne ISBN
Layout: gp.cls, Version 3.3.002 (11th July 2010), (C) Dr. Daniel Lauer, Bonn Mit TEX bearbeitet am 9. September 2010, 0:00 Uhr Umlaut-Check: äöüÄÖÜ.
S. 61 Einstellungen: ohne Marginalien – mit Seitenrand – mit Taben – mit Kapitel-Intro – Einsteiger – Zoom auf 100.0%

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

Lars Heppert, Coding for Fun mit Python, Version 1.02 vom 12. August 2010 Galileo Press, ISBN: ohne ISBN
Layout: gp.cls, Version 3.3.002 (11th July 2010), (C) Dr. Daniel Lauer, Bonn Mit TEX bearbeitet am 9. September 2010, 0:00 Uhr Umlaut-Check: äöüÄÖÜ.
S. 62 Einstellungen: ohne Marginalien – mit Seitenrand – mit Taben – mit Kapitel-Intro – Einsteiger – Zoom auf 100.0%

Index

Lars Heppert, Coding for Fun mit Python, Version 1.02 vom 12. August 2010 Galileo Press, ISBN: ohne ISBN
Layout: gp.cls, Version 3.3.002 (11th July 2010), (C) Dr. Daniel Lauer, Bonn Mit TEX bearbeitet am 9. September 2010, 0:00 Uhr Umlaut-Check: äöüÄÖÜ.
S. 321 Einstellungen: ohne Marginalien – mit Seitenrand – mit Taben – mit Kapitel-Intro – Einsteiger – Zoom auf 100.0%

H

I

J

K

L

M

Lars Heppert, Coding for Fun mit Python, Version 1.02 vom 12. August 2010 Galileo Press, ISBN: ohne ISBN
Layout: gp.cls, Version 3.3.002 (11th July 2010), (C) Dr. Daniel Lauer, Bonn Mit TEX bearbeitet am 9. September 2010, 0:00 Uhr Umlaut-Check: äöüÄÖÜ.
S. 322 Einstellungen: ohne Marginalien – mit Seitenrand – mit Taben – mit Kapitel-Intro – Einsteiger – Zoom auf 100.0%

Lars Heppert, Coding for Fun mit Python, Version 1.02 vom 12. August 2010 Galileo Press, ISBN: ohne ISBN
Layout: gp.cls, Version 3.3.002 (11th July 2010), (C) Dr. Daniel Lauer, Bonn Mit TEX bearbeitet am 9. September 2010, 0:00 Uhr Umlaut-Check: äöüÄÖÜ.
S. 323 Einstellungen: ohne Marginalien – mit Seitenrand – mit Taben – mit Kapitel-Intro – Einsteiger – Zoom auf 100.0%

S

T

U

V

W

Lars Heppert, Coding for Fun mit Python, Version 1.02 vom 12. August 2010 Galileo Press, ISBN: ohne ISBN
Layout: gp.cls, Version 3.3.002 (11th July 2010), (C) Dr. Daniel Lauer, Bonn Mit TEX bearbeitet am 9. September 2010, 0:00 Uhr Umlaut-Check: ääüÄÖÜ.
S. 324 Einstellungen: ohne Marginalien – mit Seitenrand – mit Taben – mit Kapitel-Intro – Einsteiger – Zoom auf 100.0%

X

Z

Lars Heppert, Coding for Fun mit Python, Version 1.02 vom 12. August 2010 Galileo Press, ISBN: ohne ISBN
Layout: gp.cls, Version 3.3.002 (11th July 2010), (C) Dr. Daniel Lauer, Bonn Mit TEX bearbeitet am 9. September 2010, 0:00 Uhr Umlaut-Check: äöüÄÖÜ.
S. 325 Einstellungen: ohne Marginalien – mit Seitenrand – mit Taben – mit Kapitel-Intro – Einsteiger – Zoom auf 100.0%