

Processing.py in Beispielen

Visualisierungen und interaktive Anwendungen mit
Python und Processing programmieren

Jörg Kantel

16. September 2017

Inhaltsverzeichnis

1 Einleitung	11
2 Download und Installation	13
3 JEP: Just Enough Python (Gerade genug Python)	15
4 Start: Rotkäppchen und die drei Tanten	17
Der Quellcode	18
Literatur	20
5 Punkte und Pixel	21
Turmite	21
Die Turmite programmieren	22
Quellcode	23
Weitere mögliche Experimente	24
Literatur	25
Wir backen uns ein Mandelbrötchen	26
Das Programm	27
Der komplette Quellcode	27
Pixel-Array versus set()	29

Programm 1: Mandelbrot-Menge mit set()	30
Programm 2: Mandelbrot-Menge mit Pixel-Array	31
Julia-Menge	32
Julia-Menge interaktiv	33
Julia-Menge animiert	35
Schnelle Bildmanipulation: Das Pixel-Array	36
Fantastic Feather Fractal	37
Der Quellcode	38
Literatur	39
6 Linien	41
Anschauliche Mathematik: Die Schmetterlingskurve	41
Der Lorenz-Attraktor, eine Ikone der Chaos-Theorie	45
Der Quellcode	47
Links	48
Literatur	49
7 Shapes	51
For Your Eyes Only – Processing.py zieht Kreise	51
Credits	54
Spaß mit Kreisen: Konfetti	54
Syntaktischer Zucker: »with« in Processing.py	56
Spaß mit Kreisen in Processing.py: Cantor-Käse und mehr	59
Cantors Doppelkäse	61
Literatur	63
Weitere geometrische Grundformen	63
Rechtecke	64

Kreise und Ellipsen	65
Dreieck	66
Unregelmäßige Vierecke	66
Kreisbögen	67
Der Quelltext	67
Credits	69
Visualisierung: Die Sonntagsfrage	69
Der Quellcode	71
Quellen	72
Der Baum des Pythagoras	72
Die Funktion drawPythagoras	73
Die Farben	74
Der Quellcode	75
Erweiterungen und Änderungen	76
Credits	76
8 Text(verarbeitung) in Processing.py	77
Als die Pangramme laufen lernten	79
Font, Font, Font	81
UTF-8-Text aus Dateien lesen	83
Keine Emojis	85
Caveat	85
Spaß mit Processing.py: Rentenuhr	86
9 Bildmanipulation mit Processing.py	91
Jeder sein kleiner Warhol	91
Der Quellcode	92
Ressourcen	93

Filter für die Bildverarbeitung	93
Filter interaktiv	99
Pointillismus	100
Der Quellcode	102
Noch mehr Pointillismus	103
Der Quellcode	105
10 Animationen	107
Ein kleiner roter Luftballon	107
Viele, viele rote Luftballons	108
Es kann nicht nur einen geben	111
In Lorenzkirch ist Jahrmarkt	113
Credits	115
11 Spaß mit (SVG-) Shapes: Pinguine im Eismeer	117
Und nun das Eismeer	118
Wartet, da ist noch mehr	121
Credits	121
12 Objekte und Klassen mit Kitty	123
Hallo Hörnchen – Hallo Kitty revisited	123
Moving Kitty	126
Klasse Kitty!	128
»Cute Planet« mit Processing.py	130
Fluffy Planet	134
Quellcode	135

13 Zelluläre Automaten	139
Das Demokratie-Spiel	139
Der Code	141
Caveat	143
Literatur	144
Frösche und Schildkröten oder: Wie entsteht Segregation?	144
Schauen wir uns das doch einfach einmal an:	145
Der Quellcode	147
Literatur	149
Der Waldbrand-Simulator	150
Kein Spiel ohne Regeln	150
Die Realisierung	152
Der Quellcode (1)	155
Ein größerer Wald	157
Beispielsimulation	157
Der Quellcode (2)	161
Caveat	163
14 3D mit Processing.py	165
Kugeln und Kisten	165
Der Quellcode	167
Caveat	167
Und es geht doch: Kugeln und Texturen	167
Und noch eine Textur	169
Die Erde ist eine Kiste	170
Quellcode	171
Licht und Schatten	172
Licht aus – Spot an!	174

Quellcode	174
Credits	177
Literatur	177
Einen Globus basteln	177
Quellcode Globe 01	178
Quellcode Globe 02	179
Quellcode Globe 03	181
Quellcode Globe 04	183
Credits	185
Literatur	185
15 Einen eigenen Wetterbericht mit OpenWeather-Map	187
OpenWeatherMap	188
Die Wetterstation mit Processing.py	189
16 WordCram: Processing.py und eine Processing (Java) Bibliothek	197
17 Running Orc mit Processing.py	201
Running Orc in vier Richtungen	203
Ork mit Kollisionserkennung	208
Ein Ork im Labyrinth	218
Der autonome Ork	227
Caveat	237
Drei Orks und ein Held	237
Der Quellcode	240
Meditieren mit den Orks	245
Credits	246

18 Exkurs Rauhnächte: Spaß mit Processing.py	247
Das vollständige Programm	248
Maus versus Tastatur	250
Caveat	250
Weitere Credits	251
19 Exkurs: Walking Pingus	253
Der Quellcode	255
Pingus Links	258
20 Das Avoider Game	259
Game Stage 1	259
Die Spiel-Idee	260
Das Sprite-Modul	260
Das Hauptprogramm	263
Stage 2	266
Das Spiel	267
Das Hauptprogramm	269
Der Quellcode	271
Stage 3: Sternenhimmel	275
Die Sterne	276
Der Quellcode	278
Stage 4: PowerUp und PowerDown	284
Power Items	284
Easing	286
Das Hauptprogramm	287
Der Quellcode	289
Screenshots	296
Nachtrag: Avoider Game Stage 4a	297

21 Epilog	299
22 Anhang	301
Literaturverzeichnis	301
Index	301

Kapitel 1

Einleitung

Kapitel 2

Download und Installation

Kapitel 3

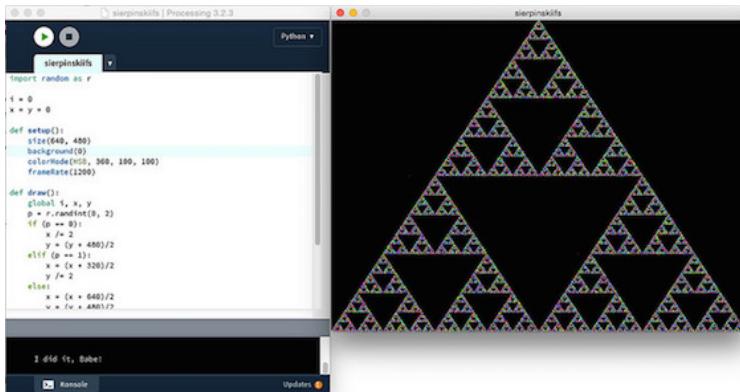
JEP: Just Enough
Python (Gerade genug
Python)

Kapitel 4

Start: Rotkäppchen und die drei Tanten

Rotkäppchen hat nicht nur eine Großmutter, sondern – was weniger bekannt ist – auch drei Tanten, Agathe, Beatrice und Cynthia. Diese wohnen in drei Häusern, die zusammen ein Dreieck bilden. Wenn Rotkäppchen nicht ihre Großmutter besucht, dann besucht sie eine der drei Tanten. Letzten Sonntag jedoch war sie sehr unschlüssig, welche sie besuchen sollte. Sie startete, um Agathe einen Besuch abzustatten. Jedoch genau auf dem halben Weg zu Agathe wurde sie unsicher und überlegte es sich noch einmal. Sie beschloß, eine ihrer drei Tanten aufzusuchen, es könnte auch wieder Agathe gewesen sein. Doch es war wie verhext: Jedesmal, wenn sie genau den halben Weg zurückgelegt hatte, wurde sie wieder unsicher und entschloß sich neu, einer ihrer drei Tanten aufzusuchen, möglicherweise die gleiche, möglicherweise eine andere. Und das wieder, und wieder, und wieder

...



William P. Beuamont [Beaum1996] nannte es das »Tantenspiel«. Ziel ist es nicht, herauszufinden, welche Tante gewinnt (es kann gar keine gewinnen), sondern welche Figur entsteht, wenn man Rotkäppchens Irrweg visualisiert. Ich habe das einmal mit [Processing.py](#) nachprogrammiert und herausgekommen ist obige Figur, in der Fachliteratur auch als **Sierpinski Dreieck** bekannt, benannt nach dem polnischen Mathematiker *Waclaw Sierpiński*, der das Fraktal schon 1915 als erster beschrieb.

Der Quellcode

Normalerweise wird dieses Fraktal mit einem rekursiven Algorithmus erzeugt, aber es geht eben auch mithilfe dieses »Chaos-Spiels« [Herrm1994]

```

import random as r

i = 0
x = y = 0

def setup():
    size(640, 480)
    background(0)
    colorMode(HSB, 360, 100, 100)
    frameRate(1200)

```

```
def draw():
    global i, x, y
    p = r.randint(0, 2)
    if (p == 0):
        x /= 2
        y = (y + 480)/2
    elif (p == 1):
        x = (x + 320)/2
        y /= 2
    else:
        x = (x + 640)/2
        y = (y + 480)/2
    stroke(i%360, 100, 100)
    point(x, y)
    i += 1
    if (i > 120000):
        print("I did it, Babe!")
        noLoop()
```

Die Schleife wird 120.000 mal durchlaufen, bevor sie stoppt. Damit ich nicht ewig auf das Ergebnis warten muß, habe ich die Framerate auf 1.200 FPS gesetzt. Das ist vermutlich etwas übertrieben, in diversen Foren habe ich Vermutung gefunden, daß Processing kaum eine Framerate von 1.000 FPS überschreiten kann. Das habe ich experimentell bestätigt, obiger Sketch lief auf meinem schnellsten Rechner, einem Mac Pro mit 3,5 GHz 6-Core Intel Xeon E5, 2 Minuten und 20 Sekunden. Wären genau 1.000 FPS erreicht worden, hätte er exakt 2 Minuten laufen müssen.

Aber man sieht sehr schön, wie sich das Dreieck zufällig, aber dennoch erkennbar, zusammensetzt. Je nach zufälligem Startwert liegen die ersten drei bis vier Punkte noch außerhalb des Fraktals, danach geht aber alles seinen geordneten Gang. Und an den Farben erkennt man, daß auch die Reihenfolge, in der die einzelnen Punkte des Fraktals von Rotkäppchen angelaufen werden, ebenfalls zufällig sind.

Literatur

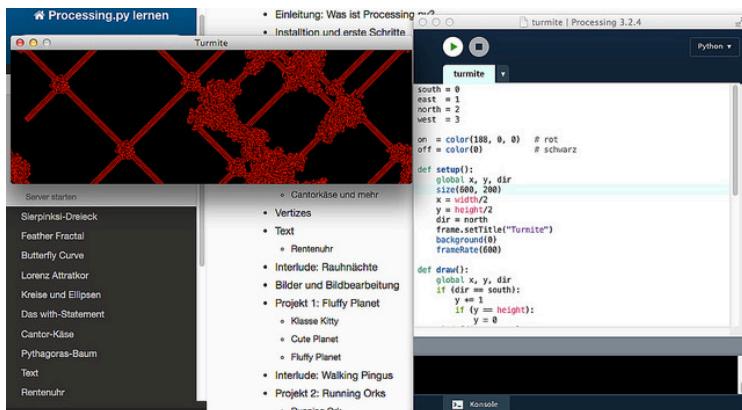
- [Beaum1996] William P. Beaumont: *Conquering the Math Bogeyman*, in Clifford A. Pickover (Ed.): *Fractal Horizons – The Future Use of Fractals*, New York (St. Martin's Press) 1996, Seiten 3 - 15
- [Herrm1994] Dietmar Herrmann: *Algorithmen für Chaos und Fraktale*, Bonn (Addison-Wesley) 1994, Seiten 132ff.

Kapitel 5

Punkte und Pixel

Turmite

Turmiten sind quadratische, 1x1 Pixel große, kybernetische Kreaturen mit einer höchst kümmerlichen Andeutung eines Gehirns. Sie können die Farben des Pixels oder der Zelle, auf der sie gerade stehen, erkennen und danach handeln. Ist die Zelle schwarz, färben sie sie rot und bewegen sich um ein Feld nach links. Ist die Farbe rot, färben sie die Zelle schwarz und bewegen sich um ein Feld nach rechts.



Wird solch eine Turmite auf eine schwarze, unendlichen Ebene gesetzt, erzeugt sie zuerst ein chaotisches Muster. Doch nach ungefähr 10.000 Schritten bildet sie auf einmal eine Turmiten-Autobahn, eine regelmäßige Struktur, die immer nach 104 Schritten in denselben Zustand zurückkehrt, nur jeweils um 2 Felder verschoben.

Die Turmite programmieren

Ich habe eine dieser Turmiten in einem Processing.py-Sketch zum Leben erweckt. Damit sie nicht in der Unendlichkeit der Ebene entfleucht, habe ich die Ecken des Fensters miteinander verklebt und sie so in eine [Torus](#)-Welt verwandelt. Wenn die Turmite am unteren Ende des Fensters verschwindet, taucht sie am oberen Ende wieder auf, verschwindet sie am rechten Rand erscheint sie wieder am linken Rand. Für beide Ränder gilt das natürlich auch umgekehrt, die Welt der Turmite ist also ein fett aufgeblasener Fahrradschlauch, auf dem sie sich entlang bewegt.

Den Farbsensor der Turmite habe ich mit dem Processing-Befehl `get(x, y)` simuliert. Er liest die Farbe des Pixels. Analog dazu gibt es die Funktion `set(x, y, color)`, die die Farbe `color` an die Stelle `x, y` schreibt. Die beiden Farben habe ich im Sketch `on` für schwarz und `off` für rot genannt. Ich bin von der Metapher ausgegangen, daß die Turmite auf der schwarzen Ebene ein Feld entweder einschaltet (also rot färbt) oder es wieder ausschaltet (es wird wieder schwarz).

Als ich damals auf meinem Atari-ST mein erstes Turmitenprogramm schrieb, dauerte es ewig, bis die Turmite mit ihrer Autobahn im Unendlichen verschwunden war (sie das Bildschirmfenster verlassen hatte). An eine Rückkehr via Torus wagte ich nicht zu denken, dafür reichte meine Geduld nicht aus. Nun in Processing.py habe ich die Framerate auf 600 gesetzt und so geht es doch recht schnell voran.

Interessant ist, daß die Turmite, wenn sie auf eine von ihr geschaffene Autobahn trifft, zwar erst einmal wieder ein chaotisches Verhalten an den Tag legt, aber über kurz oder lang wieder eine Autobahn baut. Diese Turmiten-Autobahnen kennen nur zwei

Orientierungen, sie verlaufen entweder parallel oder stehen senkrecht aufeinander.

Quellcode

Nach dem oben Beschriebenen dürfte der Quellcode leicht verständlich sein. In der `setup()`-Funktion wird die Hintergrundfarbe auf schwarz und die Turmite in die Mitte des Fensters mit der Ausrichtung nach Norden gesetzt.

Im ersten Abschnitt der `draw()`-Funktion wird die Turmite gemäß Ihrer aktuellen Orientierung bewegt und die Behandlung der Fensterränder berücksichtigt. Dann wird die Farbe der aktuellen Zelle gelesen (mit `get(x, y)`) und je nach Zustand eine neue Farbe gesetzt und die Orientierung der Turmite den Regeln entsprechend geändert. Das ist alles.

```
south = 0
east = 1
north = 2
west = 3

on = color(188, 0, 0)      # rot
off = color(0)             # schwarz

def setup():
    global x, y, dir
    size(600, 200)
    x = width/2
    y = height/2
    dir = north
    frame.setTitle("Turmite")
    background(0)
    frameRate(600)

def draw():
    global x, y, dir
    if (dir == south):
        y += 1
        if (y == height):
```

```

        y = 0
    elif (dir == east):
        x += 1
        if (x == width):
            x = 0
    elif (dir == north):
        if (y == 0):
            y = height - 1
        else:
            y -= 1
    elif (dir == west):
        if (x == 0):
            x = width - 1
        else:
            x -= 1

    if (get(x, y) == on):
        set(x, y, off)
        if (dir == south):
            dir = west
        else:
            dir -= 1
    else:
        set(x, y, on)
        if (dir == west):
            dir = south
        else:
            dir += 1

```

Weitere mögliche Experimente

Die Turmiten gehen auf *Greg Turk* zurück, der damals Doktorand an der Universität von North Carolina in Chapel Hill war. Er zeigte, daß sie eine zweidimensionale [Turingmaschine](#) sind. Später hat sie *Christopher Langton* weiterentwickelt und beschrieben – daher ist sie auch unter dem Namen »Langtons Ameise« (*Langton's Ant*) bekannt. Die hier vorgestellte ist die einfachste Form solch einer Ameise. Ein nächster Schritt wäre beispielsweise, die Welt mit zwei Turmiten zu bevölkern, die eine färbt die Ebene

rot, wenn sie auf ein schwarzes Feld trifft, die andere färbt sie blau. Natürlich müßten dann beide Ameisen auch Regeln implementiert bekommen, wie sie zu verfahren haben, wenn sie auf ein blaues respektive ein rotes Feld treffen.

Von Turk selber gibt es zum Beispiel eine Turmite mit zwei Zuständen, nennen wir diese A und B und mit folgendem Regelsatz:

Zustand	A	B
grün	schwarz, vorwärts, B	grün, rechts, A
schwarz	grün, links, A	grün, rechts, A

Sie erzeugt ein Spiralmuster, »immer größer werdende strukturierte Gebiete, die sich in regelmäßiger Anordnung um einen Startpunkt winden«.

Eine weitere Turmite, die mit vier Farben hantiert, braucht nur einen Zustand, um ebenfalls ein interessantes, symmetrisches Muster zu bilden. Hier ihr Regelsatz:

Zustand	A
blau	rot, rechts, A
rot	gelb, rechts, A
gelb	grün, links, A
grün	blau, links, A

Es gibt also noch viel zu entdecken in der Welt der Turmiten und Ameisen.

Literatur

- A.K. Dewdney: *Turmiten*, in: Immo Diener (Hg.): *Computer-Kurzweil 2, Spektrum Akademischer Verlag: Verständliche Forschung*, Heidelberg 1992, Seiten 156-160
- [Ameise \(Turingmaschine\)](#) in der Wikipedia.

Wir backen uns ein Mandelbrötchen

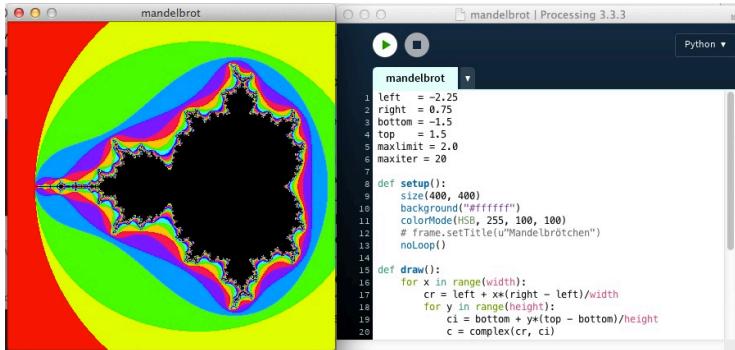


Abbildung 5.1: Screenshot

Die [Mandelbrot-Menge](#) ist die zentrale Ikone der Chaos-Theorie und das Urbild aller Fraktale. Sie ist die Menge aller komplexen Zahlen c , für welche die durch

$$z_0 = 0 \quad (5.1)$$

$$(5.2)$$

$$z_{n+1} = z_n^2 + c \quad (5.3)$$

$$(5.4)$$

$$(5.5)$$

rekursiv definierte Folge beschränkt ist. Bilder der Mandelbrot-Menge können erzeugt werden, indem für jeden Wert des Parameters c , der gemäß obiger Rekursion endlich bleibt, ein Farbwert in der komplexen Ebene zugeordnet wird.

Die komplexe Ebene wird in der Regel so dargestellt, daß in der Horizontalen (in der kartesischen Ebene die x -Achse) der Realteil der komplexen Zahl und in der Vertikalen (in der kartesischen Ebene die y -Achse) der imaginäre Teil aufgetragen wird. Jede komplexe Zahl entspricht also einen Punkt in der komplexen Ebene. Die zur Mandelbrotmenge gehörenden Zahlen werden im Allgemeinen schwarz dargestellt, die übrigen Farbwerte werden

der Anzahl von Iterationen (`maxiter`) zugeordnet, nach der der gewählte Punkt der Ebene einen Grenzwert (`maxlimit`) verläßt. Der theoretische Grenzwert ist `2.0`, doch können besonders bei Ausschnitten aus der Menge, um andere Farbkombinationen zu erreichen, auch höhere Grenzwerte verwendet werden. Bei Ausschnitten muß auch die Anzahl der Iterationen massiv erhöht werden, um eine hinreichende Genauigkeit der Darstellung zu erreichen.

Das Programm

Python kennt den Datentyp `complex` und kann mit komplexen Zahlen rechnen. Daher drängt sich die Sprache für Experimente mit komplexen Zahlen geradezu auf. Zuerst werden mit `cr` und `ci` Real- und Imaginärteil definiert und dann mit

```
c = complex(cr, ci)
```

die komplexe Zahl erzeugt. Für die eigentliche Iteration wird dann – nachdem der Startwert `z = 0.0` festgelegt wurde – nur eine Zeile benötigt:

```
z = (z**2) + c
```

Wie schon in anderen Beispielen habe ich für die Farbdarstellung den HSB-Raum verwendet und über den *Hue*-Wert iteriert. Das macht alles schön bunt, aber es gibt natürlich viele Möglichkeiten, ansprechendere Farben zu bekommen, beliebt sind zum Beispiel selbsterstellte Paletten mit 256 ausgesuchten Farbwerten, die entweder harmonisch ineinander übergehen oder bestimmte Kontraste betonen.

Der komplette Quellcode

```
left   = -2.25
right  = 0.75
bottom = -1.5
```

```
top      = 1.5
maxlimit = 2.0
maxiter = 20

def setup():
    size(400, 400)
    background("#ffffff")
    colorMode(HSB, 255, 100, 100)
    # frame.setTitle(u"Mandelbrötchen")
    noLoop()

def draw():
    for x in range(width):
        cr = left + x*(right - left)/width
        for y in range(height):
            ci = bottom + y*(top - bottom)/height
            c = complex(cr, ci)
            z = 0.0
            i = 0
            for i in range(maxiter):
                if abs(z) > maxlimit:
                    break
                z = (z**2) + c
            if i == (maxiter - 1):
                set(x, y, color(0, 0, 0))
            else:
                set(x, y, color((i*48)%255, 100, 100))
```

Um zu sehen, wie sich die Farben ändern, kann man durchaus mal mit den Werten von `maxlimit` spielen und diesen zum Beispiel auf 3.0 oder 4.0 setzen. Auch die Erhöhung der Anzahl der Iterationen `maxiter` verändert die Farbzuzuordnung, verlängert aber auch die Rechenzeit drastisch, so daß man speziell bei Ausschnitten aus der Mandelbrotmenge schon einige Zeit auf das Ergebnis warten muß.

Pixel-Array versus set()

Will man einzelne Pixel im Ausgabefenster oder in einem Bild manipulieren, bietet Processing(.py) grundsätzlich zwei Möglichkeiten: Zum einen kann man mit

```
set(x, y, color)
```

direkt einen Farbpunkt an eine bestimmte Position x, y setzen, oder aber man lädt mit

```
loadPixels()
```

das gesamte Ausgabe-Fenster in ein eindimensionales Pixel-Array, um dann mit

```
pixels[x + y*width] = color()
```

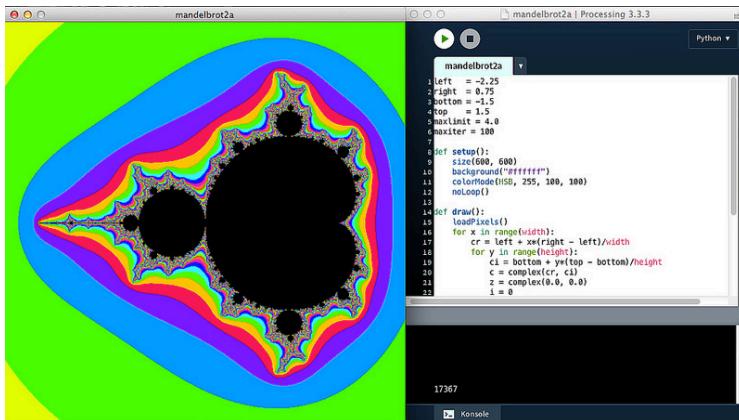
die Farbe an die gewünschte Stelle x, y zu setzen. Anschließend darf man nicht vergessen, mit

```
updatePixels()
```

Processing dazu zu bewegen, die geänderten Pixel auch anzuzeigen. Dadurch, daß das Pixel-Array eindimensional ist und so die gewünschte Position mit x + y*width angesprochen werden muß, ist die erste Version (für die es übrigens auch noch ein entsprechendes `get(x, y)` gibt, mit dem man die Farbe an der gewünschten Stelle abfragen kann) einfacher handzuhaben, aber die [Reference zu Processing](#) zu bedenken:

Setting the color of a single pixel with `set(x, y)` is easy, but not as fast as putting the data directly into `pixels[]`.

Das gilt aber nicht immer, mit dem im [letzten Abschnitt gebackenen Mandelbrötchen](#) habe ich die Probe aufs Exempel gemacht. Zwei nahezu identische Programme habe ich gegeneinander anstreiten lassen.



Programm 1: Mandelbrot-Menge mit set()

```

left      = -2.25
right     = 0.75
bottom   = -1.5
top       = 1.5
maxlimit = 4.0
maxiter  = 100

def setup():
    size(600, 600)
    background("#ffffff")
    colorMode(HSB, 255, 100, 100)
    noLoop()

def draw():
    for x in range(width):
        cr = left + x*(right - left)/width
        for y in range(height):
            ci = bottom + y*(top - bottom)/height
            c = complex(cr, ci)
            z = complex(0.0, 0.0)
            i = 0
            while abs(z) < maxlimit and i < maxiter:
                z = z*z + c
                i += 1
            if i == maxiter:
                pixels[x+y*width] = 0
            else:
                pixels[x+y*width] = map(i, 0, maxiter, 0, 255)
    updatePixels()

```

```
z = complex(0.0, 0.0)
i = 0
for i in range(maxiter):
    if abs(z) > maxlimit:
        break
    z = (z**2) + c
    if i == (maxiter - 1):
        set(x, y, color(0, 0, 0))
    else:
        set(x, y, color((i*48)%255, 100, 100))
println(millis())
```

Programm 2: Mandelbrot-Menge mit Pixel-Array

```
left    = -2.25
right   = 0.75
bottom  = -1.5
top     = 1.5
maxlimit = 4.0
maxiter = 100

def setup():
    size(600, 600)
    background("#ffffff")
    colorMode(HSB, 255, 100, 100)
    noLoop()

def draw():
    loadPixels()
    for x in range(width):
        cr = left + x*(right - left)/width
        for y in range(height):
            ci = bottom + y*(top - bottom)/height
            c = complex(cr, ci)
            z = complex(0.0, 0.0)
            i = 0
            for i in range(maxiter):
                if abs(z) > maxlimit:
```

```

        break
    z = (z**2) + c
    if i == (maxiter - 1):
        pixels[x + y*width] = color(0, 0, 0)
    else:
        pixels[x + y*width] = color((i*48)%255, 100, 100)
updatePixels()
println(millis())

```

Und – Überraschung! – das Programm mit `set()` war fast immer geringfügig schneller als das Programm mit den Pixel-Arrays. Auf meinem betagten MacBook Pro benötigte das erste Programm rund 15.000 bis 16.000 Millisekunden, während das zweite Programm um die 18.000 Millisekunden benötigte. Der Unterschied ist nicht groß, aber dennoch bemerkenswert. Es liegt zum einen sicher daran, daß die benötigte Zeit für die Berechnung des Apfelmännchens im Vergleich zu der benötigten Zeit, dieses zu zeichnen, riesig ist. Zum anderen wird die `draw()`-Schleife ja auch nur einmal durchlaufen und so kann das Pixel-Array seine Fähigkeit der schnellen Pixelmanipulation nicht richtig ausspielen.

Die Erkenntnis daraus: Es kann sich durchaus lohnen, auch mal das Handbuch zu hinterfragen.

Julia-Menge

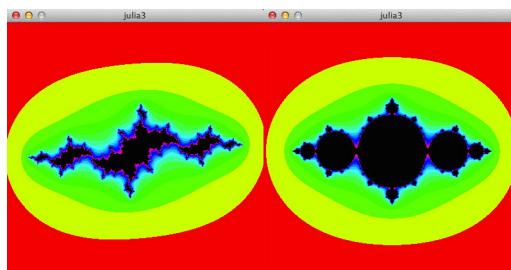


Abbildung 5.2: Screenshot

Die **Julia-Menge** wurde 1918 von den beiden französischen Mathematikern *Gaston Maurice Julia* (nachdem sie benannt wurde) und *Pierre Fatou* (dessen Zugang heute die meisten Lehrbücher folgen) unabhängig voneinander beschrieben. Sie steht im engen Zusammenhang zur im letzten Abschnitt beschriebenen **Mandelbrot-Menge**. Während die Mandelbrot-Menge, die Menge aller komplexen Zahlen c ist, die der iterierten Gleichung

$$z_0 = 0 \quad (5.6)$$

(5.7)

$$z_{n+1} = z_n^2 + c \quad (5.8)$$

(5.9)

(5.10)

folgen, ist bei der Julia-Menge c konstant:

$$z_n^2 + c \quad (5.11)$$

(5.12)

(5.13)

Die Mandelbrot-Menge ist also eine Beschreibungsmenge aller Julia-Mengen. Jedem Punkt c der komplexen Zahlenebene entspricht eine Julia-Menge. Eigenschaften der Julia-Menge lassen sich an der Lage von c relativ zur Mandelbrot-Menge beurteilen: Wenn der Punkt c Element der Mandelbrot-Menge ist, dann ist die Julia-Menge zusammenhängend. Andernfalls ist sie eine Cantormenge unzusammenhängender Punkte. Ist der Imaginärteil $ci = 0$, dann ist die Julia-Menge symmetrisch (vlg. Abbildung links oben), ansonsten kann sie alle möglichen Formen annehmen.

Julia-Menge interaktiv

Ich habe die obigen Bilder mit diesem Programm erzeugt, daß den Parameter c in Abhängigkeit von der Mausposition setzt:

```

left    = -2.0
right   = 2.0
bottom  = 2.0
top     = -2.0
maxlimit = 3.0
maxiter = 25

def setup():
    size(400, 400)
    background("#555ddd")
    colorMode(HSB, 1)

def draw():
    cr = map(mouseX, 0, width, left, right)
    ci = 0
    # ci = map(mouseY, 0, height, top, bottom)
    c = complex(cr, ci)
    for x in range(width):
        zr = left + x*(right - left)/width
        for y in range(height):
            zi = bottom + y*(top - bottom)/height
            z = complex(zr, zi)
            i = 0
            for i in range(maxiter):
                if abs(z) > maxlimit:
                    break
                z = (z**2) + c
            if i == (maxiter-1):
                set(x, y, color(0))
            else:
                set(x, y, color(sqrt(float(i)/maxiter), 100, 100))
    println("cr = " + str(cr))
    println("ci = " + str(ci))

```

Kommentiert man die Zeile `ci = 0` aus und aktiviert stattdessen die auskommentierte Zeile darunter, erhält man (theoretisch) alle Julia-Mengen, sonst erzeugt das Programm nur die symmetrischen. Richtig flüssig ist die Animation allerdings nicht, Processing.py gerät – zumindest auf meinem betagten MacBook Pro – schon ganz schön ins Stottern.

Julia-Menge animiert

Das gilt auch für das zweite Programm, das die Parameter der Julia-Menge anhand zweier Sinus- (wahlweise auch Cosinus-) Funktionen periodisch durchläuft:

```
left    = -2.0
right   = 2.0
bottom  = 2.0
top     = -2.0
maxlimit = 3.0
maxiter = 25

def setup():
    size(400, 400)
    background("#555ddd")
    colorMode(HSB, 1)

def draw():
    # cr = 0
    cr = 2*sin(frameCount)
    ci = 0
    # ci = 2*cos(frameCount)
    c = complex(cr, ci)
    for x in range(width):
        zr = left + x*(right - left)/width
        for y in range(height):
            zi = bottom + y*(top - bottom)/height
            z = complex(zr, zi)
            i = 0
            for i in range(maxiter):
                if abs(z) > maxlimit:
                    break
                z = (z**2) + c
            if i == (maxiter-1):
                set(x, y, color(0))
            else:
                set(x, y, color(sqrt(float(i)/maxiter), 100,
println("cr = " + str(cr))
println("ci = " + str(ci))
```

Auch hier kommt das Programm ganz schön ins Schwitzen. Das läßt allerdings dem Betrachter Zeit, die Schönheit der Julia-Menge zu bewundern.

Schnelle Bildmanipulation: Das Pixel-Array

In den letzten beiden Abschnitt habe ich gezeigt, daß Processing.py zwar relativ schnell ist, aber 120.000 Operationen in einem Bildfenster doch eine gewisse Zeit benötigen. Falls man jedoch auf die Animation verzichten kann (und damit auf `point()` oder `get()` und `set()`), geht es auch wesentlich schneller: Jedes Bild in Processing(.py) – und das schließt das Graphikfenster ein – wird intern als eine eindimensionale Liste der Farbwerte gespeichert. Die erste Position der Liste ist das erste Pixel links oben, die letzte Position folgerichtig das letzte Pixel rechts unten.

Ein `pixels[]`-Array in Processing speichert in dieser Form die Farbwerte für jedes Pixels des Ausgabefensters. Um es zu initialisieren, muß vor der ersten Nutzung die Funktion `loadPixels()` aufgerufen werden. Manipulationen im Pixel-Array werden erst sichtbar, wenn die Funktion `updatePixels()` aufgerufen wird. `loadPixels()` und `updatePixels()` bilden so ein ähnliches Geschwisterpaar von Funktionen, wie zum Beispiel `beginShape()` und `endShape()`. Doch einen Unterschied gibt es: Wird das Pixel-Array nur zum Lesen der Farbwerte genutzt, muß `updatePixels()` natürlich nicht aufgerufen werden. Da die Manipulationen eines Pixel-Arrays nur im Hauptspeicher des Rechners stattfinden, sind sie natürlich viel schneller als jede andere Processing-Funktion, die Bildinformationen manipuliert.

Da das Pixel-Array ein eindimensionales Array ist, muß auf die Zeilen und Spalten mit einem kleinen Trick zugegriffen werden. Jeder Punkt (`x, y`) steht im Pixelarray an der Position `x + (y*width)`. An die Farbwerte eines Pixels kommt man mit dem Aufruf

```
i = x + (y*width)
color(c) = pixels[i]
```

Die einzelnen Farbwerte im RGB-Raum kann man danach so auslesen:

```
r = red(c)
g = green(c)
b = blue(c)
```

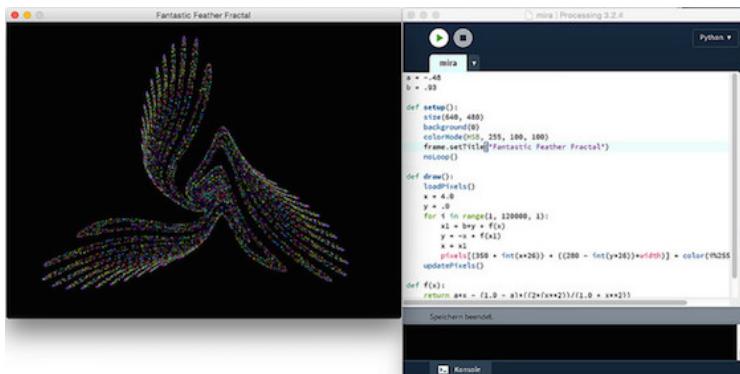
Das Setzen eines Pixels erfolgt genau umgekehrt:

```
pixel[i] = color(r, g, b)
```

Natürlich kann man auch jeden anderen Farbraum (Graustufen, HSV), den Processing kennt, nutzen.

Fantastic Feather Fractal

Um zu zeigen, wie schnell die Manipulationen eines Pixel-Arrays sind, möchte ich wieder eine Iteration über 120.000 Schritte durchführen. Als Demonstrationsobjekt habe ich das *Fantastic Feather Fractal* gewählt, das *Clifford A. Pickover* in seinem Buch »Mazes for the Mind« vorgestellt hat. Wenn Ihr untenstehenden Quellcode laufen läßt, werdet Ihr feststellen, daß das fertige Fraktal fast unmittelbar nach dem Aufruf im Graphikfenster erscheint.¹



¹Das ist keine Einschränkung, denn der Fensterrand ist ja im Grunde auch nichts anderes als eine undurchdringliche Mauer und im Zweifelsfall macht man das Spielfeld einfach um die Mauerdicke größer.

Das *Feather Fractal* ist ein »seltsamer Attraktor«, ein Attraktor eines dynamischen Systems, das sich zwar chaotisch verhält, aber dennoch eine *kompakte Menge* ist, die es nie verläßt. Die Parameter des Sketches entstammen der oben genannten Quelle von *Pickover*, die Faktoren um das Ergebnis dem Bildfenster anzupassen habe ich durch wildes Herumexperimentieren gefunden².

Der Quellcode

```
a = -.48
b = .93

def setup():
    size(640, 480)
    background(0)
    colorMode(HSB, 255, 100, 100)
    frame.setTitle("Fantastic Feather Fractal")
    noLoop()

def draw():
    loadPixels()
    x = 4.0
    y = .0
    for i in range(1, 120000, 1):
        x1 = b*y + f(x)
        y = -x + f(x1)
        x = x1
        pixels[(350 + int(x*26)) + ((280 - int(y*26))*width)] = color
    updatePixels()

def f(x):
    return a*x - (1.0 - a)*((2*(x**2))/(1.0 + x**2))
```

Wenn ich später noch auf Bildmanipulationen in Processing zu-

²Natürlich sollte man sicherstellen, daß man diese Fonts auch verwerten darf, aber im Netz findet man viele Fonts zur freien Verwendung. Gute Anlaufstellen dafür sind zum Beispiel [Google Fonts](#), die [\(Open\) Font Library](#) oder [The League of Moveable Type](#).

rückkomme, werden die Pixel-Arrays noch einmal ausführlich behandelt werden.

Literatur

- Clifford A. Pickover: *Mazes for the Mind. Computers and the Unexpected*, New York (St. Martin's Press) 1992. Das Buch gehört zu den Besten des umtriebigen Autors und da es aufgrund seines Alters antiquarisch für ein paar Cent zu bekommen ist, solltet Ihr zuschlagen. Das Feder-Fraktal ist auf den Seiten 33f. beschrieben, die über 400 anderen Seiten erfüllen fast jeden Traum eines an Computer-Experimenten interessierten Menschen.
- Florian Freistetter: *Best of Chaos: Der seltsame Attraktor*, Science Blogs (Astrodicticum Simplex) vom 4. Februar 2015 (Ich bin ein Fan von *Florian Freistetter*, er ist einer der wenigen guten deutschsprachigen Erklärbären für Naturwissenschaften)

Kapitel 6

Linien

Anschauliche Mathematik: Die Schmetterlingskurve



Abbildung 6.1: Schmetterling

Seit ich Ende der 1980er Jahre mit meinem damals hochmodernen [Atari Mega ST](#) erste Schritte mit einem graphikfähigen

Personalcomputer unternommen hatte, habe ich die Schmetterlingskurve immer wieder als Test für die Graphikfähigkeit und Schnelligkeit von Programmiersprachen und Rechnern benutzt. Sie wird in [Polarkoordinaten](#) beschrieben und ihre Formel ist

$$\rho = e^{\cos(\theta)} - 2 \cdot \cos(4 \cdot \theta) + \sin\left(\frac{\theta}{12}\right)^5$$

oder in Python-Code:

```
r = exp(cos(theta)) - 2*cos(4*theta) + (sin(theta/12))**5
```

Die Gleichung ist hinreichend kompliziert um selbst in C geschriebene Routinen auf meinen damals unglaubliche 8 MegaBit schnellen Atari alt aussehen zu lassen. Rechenzeiten von 10 - 20 Minuten waren keine Seltenheit. Heute dagegen muß man den Rechner schon künstlich verlangsamen, damit man sieht, wie sich die Kurve aufbaut. Denn sonst erscheint sofort die fertige Kurve, um die sinnliche Erfahrung, wie diese entsteht, wird man betrogen. Daher habe ich sie in *Processing.py* innerhalb der `draw()`-Schleife zeichnen lassen, wobei die Schleifenvariable `theta` bei jedem Durchlauf um 0.02 erhöht wurde.

Der Code ist – dank *Processing.py* – wieder von erfrischender Einfachheit und Kürze:

```
def setup():
    global theta, xOld, yOld
    theta = xOld = yOld = 0.0
    size(600, 600)
    background(100, 100, 100)
    colorMode(HSB, 100)

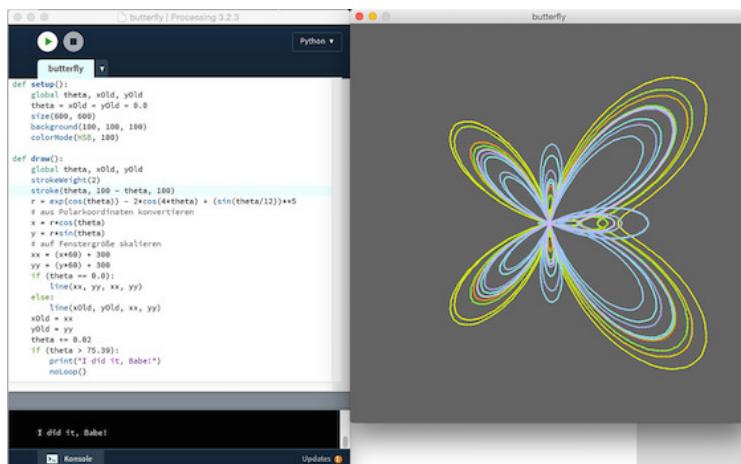
def draw():
    global theta, xOld, yOld
    strokeWeight(2)
    stroke(theta, 100 - theta, 100)
    r = exp(cos(theta)) - 2*cos(4*theta) + (sin(theta/12))**5
    # aus Polarkoordinaten konvertieren
    x = r*cos(theta)
```

```
y = r*sin(theta)
# auf Fenstergröße skalieren
xx = (x*60) + 300
yy = (y*60) + 300
if (theta == 0.0):
    point(xx, yy)
else:
    line(xOld, yOld, xx, yy)
xOld = xx
yOld = yy
theta += 0.02
if (theta > 75.39):
    print("I did it, Babe!")
    noLoop()
```

In `setup()` ist eigentlich nur bemerkenswert, daß ich nach der Festlegung des grauen Hintergrunds (noch als RGB), den `colorMode` auf HSB geändert habe. Damit lassen sich nämlich recht einfach diverse Farbeffekte erzielen. Ich habe dabei den *Hue*-Wert in Abhängigkeit von `theta` gesetzt, die Sättigung auf `100 - theta` und die *Brightness* konstant bei 100 belassen. Da `theta` nie größer als 75,39 wird, wird es also auch nie größer als 100 und damit sind diese Umrechnungen gefahrlos.

Damit erreicht man, daß zu Beginn, wo die Sättigung noch ziemlich voll ist, die Zeichnung mit einem satten rot beginnt, während im Laufe der Iteration die weiteren Farben immer blasser werden. Ich fand dies das ästhetisch anspruchsvollste Ergebnis, aber um das selber nachvollziehen zu können, solltet Ihr ruhig damit experimentieren, zum Beispiel mit `stroke(theta, 100, 100)` oder `stroke(100-theta, theta, 100)` oder was immer Ihr wollt.

Ihr bekommt so diesen wunderschönen Schmetterling auf den Monitor gezeichnet:



Um die Entstehung der Kurve zu verstehen, empfiehlt *Stan Wagon*¹, nacheinander folgende Formeln plotten zu lassen:

In Polarkoordinaten:

```

r = exp(cos(theta)) # ergibt eine Art Kreis
r = -2*cos(4*theta) # ergibt eine Art Blume
r = exp(cos(theta)) - 2*cos(4*theta) # ergibt einen sehr einfachen S

```

Dann in kartesischen Koordinaten:

```

x = -2*cos(4*theta)
y = -sin(theta/12)**5

```

Und dann ruhig auch noch einmal (wieder in Polarkoordinaten):

```
r = exp(cos(theta)) - 2*cos(4*theta) - (sin(theta/12))**5
```

Ihr seht dann, daß es eigentlich unerheblich ist, ob Ihr den Störungsanteil der Formel addiert oder subtrahiert: Der Schmetterling ist nahezu identisch, lediglich an der anderen Farbgebung erkennt Ihr, daß es zwei verschiedene Formeln sind.

¹Stan Wagon: *Mathematica® in Aktion*, Heidelberg (Spektrum Akademischer Verlag) 1993

Die Schmetterlingskurve und ähnliche Kurven wurden von *Temple Fay*² an der Universität von Southern Mississipi entwickelt. Sie eignen sich vorzüglich zum Experimentieren. So weist Pickover³ darauf hin, daß die Kurve

```
r = exp(cos(theta)) - 2.1*cos(6*theta) + sin(theta/30)**7
```

eine bedeutend größere Wiederholungsperiode besitzt. Ihr solltet Euch auch das ruhig einmal ansehen. Interessante Vergleiche mit der Originalschmetterlingskurve können Ihr auch ziehen, wenn Ihr mit

```
r = exp(cos(2*theta)) - 1.5*cos(4*theta)
```

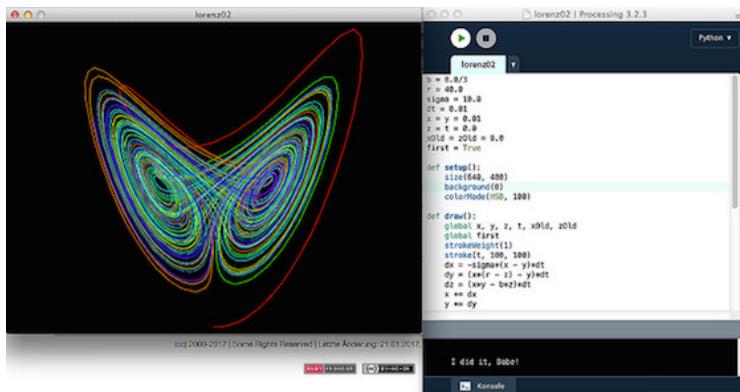
eine ganz simple Form des Schmetterlings zeichnen lasst. Denn die heutigen Rechner sind schließlich hinreichend schnell, daß Ihr nicht mehr Minuten- oder gar Stundenlang auf ein Ergebnis warten müßt und zum anderen lädt die Möglichkeit des schnellen Skizzierens mit der Processing-IDE geradezu zu eigenen Experimenten ein.

Der Lorenz-Attraktor, eine Ikone der Chaos-Theorie

Nachdem ich im letzten Abschnitt die Schmetterlingskurve mit Processing.py gezeichnet hatte, wollte ich nun darauf aufbauen und eine Ikone der Chaos-Forschung, den **Lorenz-Attraktor** damit zeichnen. Ich hatte das ja auch schon einmal [mit R getan](#) – dort findet Ihr auch weitere Hintergrundinformationen zu diesem Attraktor –, aber mit R wurde nur das fertige Ergebnis visualisiert. Hier kommt es mir aber wieder darauf an, die Entstehung der Kurve verfolgen zu können und dafür ist, wie schon bei der Schmetterlingskurve, Processing gut geeignet:

²Temple Fay: *The Butterfly Curve*, American Math. Monthly, 96(5); 442-443

³Clifford A. Pickover: *Mit den Augen des Computers. Phantastische Welten aus dem Geist der Maschine*, München (Markt&Technik) 1992, S. 41ff.



Als einer der ersten hatte 1961 [Edward N. Lorenz](#), ein Meteorologe am [Massachusetts Institute of Technology](#) (MIT), erkannt, daß Iteration Chaos erzeugt. Er benutzte dort einen Computer, um ein einfaches nichtlineares Gleichungssystem zu lösen, das ein simples Modell der Luftströmungen in der Erdatmosphäre simulieren sollte. Dazu benutzte er ein System von sieben Differentialgleichungen, das [Barry Saltzman](#) im gleichen Jahr aus den [Navier-Stokes-Gleichungen](#)⁴ hergeleitet hatte. Für dieses System existierte keine analytische Lösung, also mußte es numerisch (d.h. wie damals und auch heute noch vielfach üblich in FORTRAN) gelöst werden. Lorenz hatte entdeckt, daß bei nichtperiodischen Lösungen der Gleichungen vier der sieben Variablen gegen Null strebten. Daher konnte er das System auf drei Gleichungen reduzieren:

$$\frac{dx}{dt} = -\sigma(y - z) \quad (6.1)$$

(6.2)

$$\frac{dy}{dt} = (\rho - z)x - y \quad (6.3)$$

(6.4)

$$\frac{dz}{dt} = xy - \gamma z \quad (6.5)$$

⁴Das ist keine Einschränkung, denn der Fensterrand ist ja im Grunde auch nichts anderes als eine undurchdringliche Mauer und im Zweifelsfall macht man das Spielfeld einfach um die Mauerdicke größer.

Processing.py besitzt im Gegensatz zu R oder NumPy kein Modul zur numerischen Lösung von Differentialgleichungen und so habe ich das einfache Eulersche Poligonzugverfahren zur numerischen Berechnung benutzt

```
dx = -sigma*(x - y)*dt
dy = (x*(r - z) - y)*dt
dz = (x*y - b*z)*dt
x += dx
y += dy
z += dz
```

und dabei konstant `dt = 0.01` gesetzt. Das benötigt natürlich mehr Rechenkapazität, als sie Lorenz je zur Verfügung standen, aber trotz der größeren Genauigkeit ändert sich nichts am chaotischen Verhalten der Kurve. Für die Farbberechnung habe ich dieses mal nur den Farbwert (*Hue*) bei jeder Iteration geändert, Sättigung (*Saturation*) und Helligkeit (*Brightness*) bleiben konstant auf dem höchsten Wert. Das ergibt kräftige Farben, die von Rot über Orange nach Gelb und dann nach Grün, Blau und Violett wandern. So kann man schön erkennen, daß die beiden »Flügel« des Attraktors immer wieder, aber für uns unvorhersehbar, durchlaufen werden.

Der Quellcode

Hier nun der vollständige Quellcode des Skripts. Er ist kurz und selbsterklärend und folgt weitestgehend dem Pascal-Programm aus [Herm1994], Seiten 80ff.

```
b = 8.0/3
r = 40.0
sigma = 10.0
dt = 0.01
x = y = 0.01
z = t = 0.0
xOld = zOld = 0.0
first = True
```

```
def setup():
    size(640, 480)
    background(0)
    colorMode(HSB, 100)

def draw():
    global x, y, z, t, xOld, zOld
    global first
    strokeWeight(1)
    stroke(t, 100, 100)
    dx = -sigma*(x - y)*dt
    dy = (x*(r - z) - y)*dt
    dz = (x*y - b*z)*dt
    x += dx
    y += dy
    z += dz
    # auf Fenstergröße skalieren
    xx = (x*8) + 320
    zz = 470 - (z*5.5)
    if first:
        point(xx, zz)
    else:
        line(xOld, zOld, xx, zz)
    xOld = xx
    zOld = zz
    first = False
    t = t + dt
    if (t >= 75.0):
        print("I did it, Babe!")
        noLoop()
```

Links

- Der [Lorenz Attractor](#) auf Wolfram MathWorld

Literatur

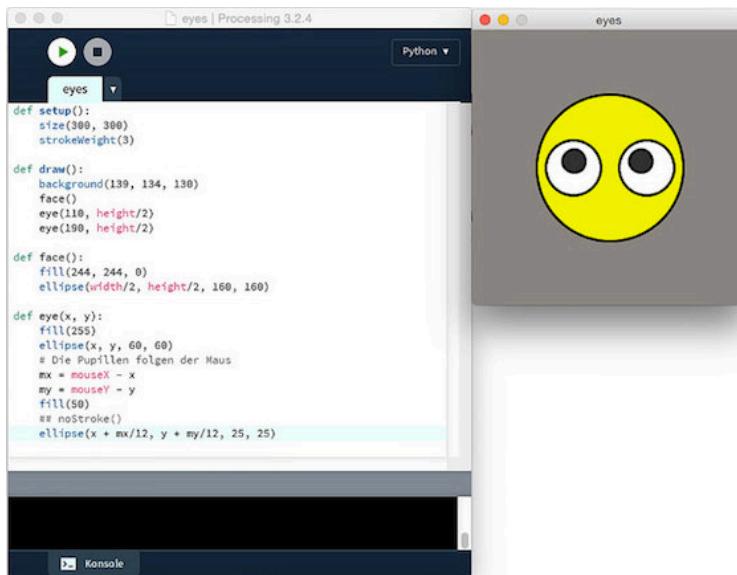
- [Herm1994] Dieter Hermann: *Algorithmen für Chaos und Fraktale*, Bonn (Addison-Wesley) 1994, S. 80ff.
- [Pief1991] Frank Piefke: *Simulationen mit dem Personalcomputer*, Heidelberg (Hüthig) 1991
- [Stew1993] Ian Stewart: *Spielt Gott Roulette?*, Frankfurt (Insel TB) 1993

Kapitel 7

Shapes

For Your Eyes Only – Processing.py zieht Kreise

Nachdem ich in den vorherigen Tutorials zu Processing.py, dem Python-Mode von Processing, schon mit Punkten und Linien hantiert habe, wird es nun Zeit, etwas mit Kreisen und Ellipsen anzustellen (sie werden in Processing mit dem gleichen Befehl erzeugt).



Ein einfacher Kreis ist schnell erzeugt. Mit diesem kleinen *Sketch* malt Ihr einen grellen roten Kreis auf schwarzem Grund:

```
def setup():
    size(500, 500)

def draw():
    background(0)
    fill(255, 0, 0)
    ellipse(width/2, height/2, 450, 450)
```

Die Funktion `ellipse()` besitzt vier Parameter, die ersten beiden sind die x- und y-Koordinaten, die per Default die Mitte des Kreises oder der Ellipse bezeichnen, die beiden anderen sind der Durchmesser des Kreises oder der Ellipse (auch wenn sie in der Literatur oft mit `r` bezeichnet werden, nicht der Radius). Bei einem Kreis müssen die letzten beiden Parameter immer den gleichen Wert besitzen. Wenn Ihr aber zum Beispiel die Funktion mit

```
ellipse(width/2, height/2, 350, 450)
```

oder

```
ellipse(width/2, height/2, 450, 350)
```

aufruft, dann seht Ihr, wie aus den Kreisen Ellipsen werden.

Nun steht Processing aber für Interaktivität. Daher möchte ich aus fünf Kreisen ein Gesicht zaubern, dessen Pupillen dem Mauszeiger folgen. Auch dieser *Sketch* ist hübsch kurz geraten:

```
def setup():
    size(300, 300)
    strokeWeight(3)

def draw():
    background(139, 134, 130)
    face()
    eye(110, height/2)
    eye(190, height/2)

def face():
    fill(244, 244, 0)
    ellipse(width/2, height/2, 160, 160)

def eye(x, y):
    fill(255)
    ellipse(x, y, 60, 60)
    # Die Pupillen folgen der Maus
    mx = mouseX - x
    my = mouseY - y
    fill(50)
    ellipse(x + mx/12, y + my/12, 25, 25)
```

Es wäre nicht wirklich notwendig gewesen, aber der Modularität willen habe ich das Zeichnen des Gesichtes in die Funktion `face()` und das Zeichnen der Augen in die Funktion `eye()` ausgelagert. Mit den Werten in dem `ellipse()`-Aufruf bei den Augen habe ich solange experimentiert, bis sie meinen Vorstellungen entsprachen. Nun sieht aber alles aus wie in dem obigen Screenshot.

Credits

Die Idee zu den Augen habe ich einem ([Java-\)](#) [Processing-Tutorial](#) von *Thomas Koberger* entnommen, das ich variiert und nach Processing.py übertragen habe. Auf [seinen Seiten](#) findet man übrigens noch viele weitere, interessante und lehrreiche Tutorials, so daß ich Euch einen Besuch dort empfehle.

Für die Farben habe ich mal wieder wild nach einer [Seite mit Farbpaletten](#) gegoogelt und fand die gefundene dann zwar nicht unbedingt schön, aber ungemein praktisch.

Spaß mit Kreisen: Konfetti

Der folgende kleine Sketch ist nicht mehr als eine Fingerübung. Er soll Euch zeigen, wie man schon mit wenigen Zeilen Code und Processings-Zufallsfunktion `random()` viele bunte Konfettischnipsel auf den Bildschirm zaubern kann:

Und hier der Quellcode des Sketches in Processing.py:

```
def setup():
    size(400, 400)
    frame.setTitle("Konfetti!")
    background(0)

def draw():
    x = random(width)
    y = random(height)
    dia = random(5, 25)
    r = random(255)
    g = random(255)
    b = random(255)
    fill(r, g, b)
    ellipse(x, y, dia, dia)
```

Für so wenige Programmzeilen ist das Ergebnis doch recht ansprechend, oder?

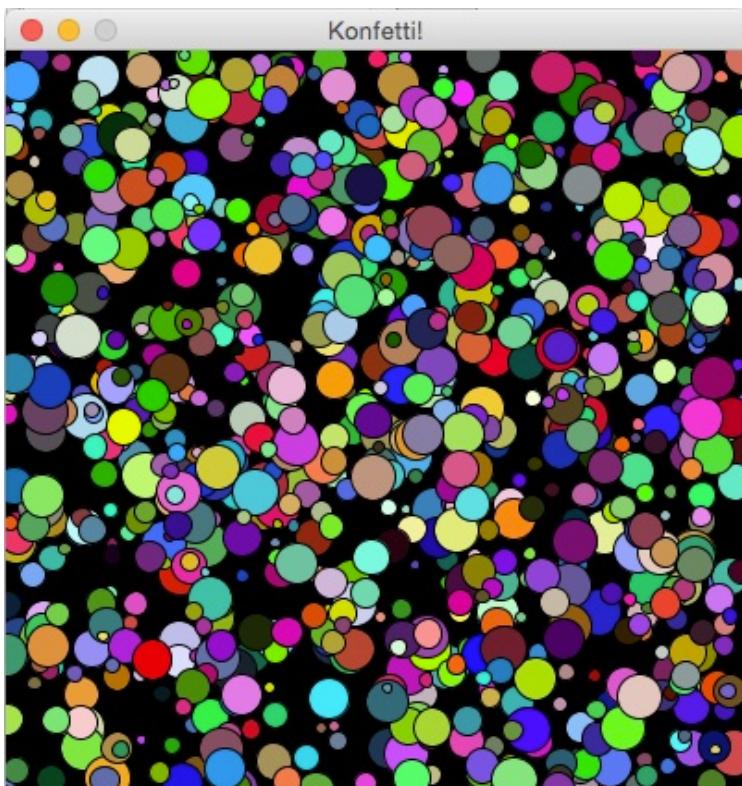
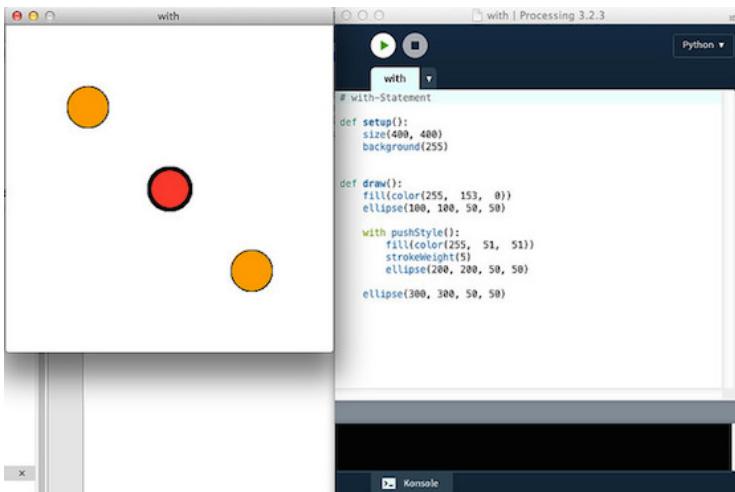


Abbildung 7.1: Screenshot

Syntaktischer Zucker: »with« in Processing.py

Wenn man in Processing.py irgendetwas zum Beispiel zwischen `beginShape()` und `endShape()` klammert, fühlt sich das nicht sehr »pythonisch« an. Ich denke dann die ganze Zeit: Das gehört doch eingerückt! In Processings Java-Mode kann man das auch machen, weil man in Java Leerzeichen einsetzen kann, wie man will – sie haben dort keine Bedeutung. Doch Python reagiert ja sehr sensibel auf Einrückungen, da hier Leerzeichen Teil der Syntax sind. Aber die Macher von Processing.py haben dies bedacht und uns einen Ausweg aus diesem Dilemma geboten: Das `with`-Statement.



In seiner einfachsten Form sieht das so aus. Statt zum Beispiel

```
def setup():
    size(400, 400)
    background(255)

def draw():
    fill(color(255, 153, 0))
    strokeWeight(1)
    ellipse(100, 100, 50, 50)
```

```
fill(color(255, 51, 51))
strokeWeight(5)
ellipse(200, 200, 50, 50)
fill(color(255, 153, 0))
strokeWeight(1)
ellipse(300, 300, 50, 50)
```

zu schreiben, schreibt man einfach:

```
def setup():
    size(400, 400)
    background(255)

def draw():
    fill(color(255, 153, 0))
    ellipse(100, 100, 50, 50)

    with pushStyle():
        fill(color(255, 51, 51))
        strokeWeight(5)
        ellipse(200, 200, 50, 50)
        ellipse(300, 300, 50, 50)
```

Die Ausgabe ist in beiden Fällen identisch, aber der zweite Sketch ist in meinen Augen bedeutend eleganter und fühlt sich viel pythonischer an. Außerdem erspart man sich viel Tipparbeit.

Da ich die Verwendung des `with`-Statements auch erst durch eines der mitgelieferten Beispielprogramme herausbekommen habe, hier eine (hoffentlich) komplette Liste der Möglichkeiten:

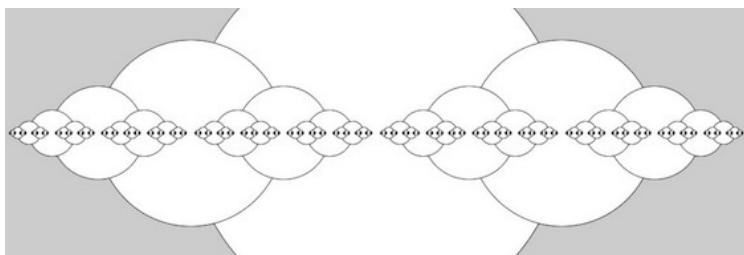
<code>with pushMatrix():</code>	<code>pushMatrix()</code>
<code>translate(10, 10)</code>	<code>translate(10, 10)</code>
<code>rotate(PI/3)</code>	<code>rotate(PI/3)</code>
<code>rect(0, 0, 10, 10)</code>	<code>rect(0, 0, 10, 10)</code>
<code>rect(0, 0, 10, 10)</code>	<code>popMatrix()</code>
	<code>rect(0, 0, 10, 10)</code>

<code>with beginContour():</code>	<code>beginContour()</code>
-----------------------------------	-----------------------------

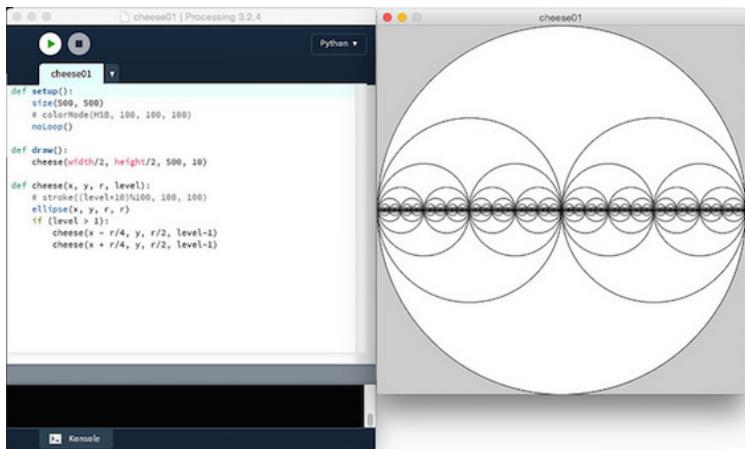
<code>doSomething()</code>	<code>doSomething()</code>
	<code>endContour()</code>
<code>with beginCamera():</code>	<code>beginCamera()</code>
<code>doSomething()</code>	<code>doSomething()</code>
	<code>endCamera()</code>
<code>with beginPGL():</code>	<code>beginPGL()</code>
<code>doSomething()</code>	<code>doSomething()</code>
	<code>endPGL()</code>
<code>with beginShape():</code>	<code>beginShape()</code>
<code>vertex(x, y)</code>	<code>vertex(x, y)</code>
<code>vertex(j, k)</code>	<code>vertex(j, k)</code>
	<code>endShape()</code>
<code>with beginShape(TRIANGLES):</code>	<code>beginShape(TRIANGLES)</code>
<code>vertex(x, y)</code>	<code>vertex(x, y)</code>
<code>vertex(j, k)</code>	<code>vertex(x, y)</code>
	<code>endShape()</code>
<code>with beginClosedShape():</code>	<code>beginShape()</code>
<code>vertex(x, y)</code>	<code>vertex(x, y)</code>
<code>vertex(j, k)</code>	<code>vertex(j, k)</code>
	<code>endShape(CLOSED)</code>

Links steht die Schreibweise mit dem `with()`-Statement, rechts die traditionelle Form. Abgesehen davon, daß die `with`-Schreibweise immer mindestens eine Zeile kürzer ist, sorgt sie durch die Einrückungen auch für eine bessere Übersicht und eine bessere Lesbarkeit.

Spaß mit Kreisen in Processing.py: Cantor-Käse und mehr



Wie im letzten Beitrag gezeigt, ist es in Processing (und damit auch in Processing.py, dem Python-Mode für Processing) recht einfach, einfache Kreise oder Ellipsen zu zeichnen. Aber das ist auf die Dauer natürlich ein wenig langweilig, daher wende ich mich nun einer rekursiven Figur zu, die zwar ebenfalls nur aus Kreisen besteht, aber dennoch einige interessante Eigenschaften aufweist, dem **Cantor-Käse**, einer Figur, die der **Cantor-Menge** topologisch ähnlich ist. Sie wird konstruiert, in dem aus einem Kreis bis auf zwei kleinere Kreise alles entfernt wird. Aus diesen zwei kleineren Kreisen wird wiederum bis auf zwei kleinere Kreise alles entfernt. Nun hat man schon vier Kreise, aus denen man jeweils bis auf zwei kleinere Kreise alles entfernt. Und so weiter und so fort ...



Das schreit natürlich nach einer rekursiven Funktion und die ist in Python (genauer: in Processings Python-Mode) recht schnell erstellt:

```
def setup():
    size(500, 500)
    # colorMode(HSB, 100, 100, 100)
    noLoop()

def draw():
    cheese(width/2, height/2, 500, 10)

def cheese(x, y, r, level):
    ellipse(x, y, r, r)
    if (level > 1):
        cheese(x - r/4, y, r/2, level-1)
        cheese(x + r/4, y, r/2, level-1)
```

Das Ergebnis könnt Ihr in obenstehenden Screenshot bewundern. Im Screenshot sieht man noch, daß ich auch versucht habe, mit Farbe zu experimentieren, aber ein wirklich befriedigendes Ergebnis war dabei nicht herausgekommen

Ich hatte diese Figur auch schon mal in [Shoes zeichnen lassen](#) und dabei Probleme mit der Rekursionstiefe festgestellt (ab einer Rekursionstiefe von 15 stürzte Shoes gnadenlos ab). Hier scheint Processing robuster zu sein, eine Rekursionstiefe von 15 nahm die Software gelassen hin, ließ sich dann natürlich Zeit mit der Ausgabe. Das muß schließlich alles berechnet werden.

Weil der Durchmesser der Kreise in der Literatur oft mit **r** bezeichnet wird, neige ich dazu, Radius und Durchmesser zu wechseln. Setzt man dann den Algorithmus 1:1 um, zum Beispiel wie in diesem Sketch

```
def setup():
    size(1000, 500)
    noLoop()

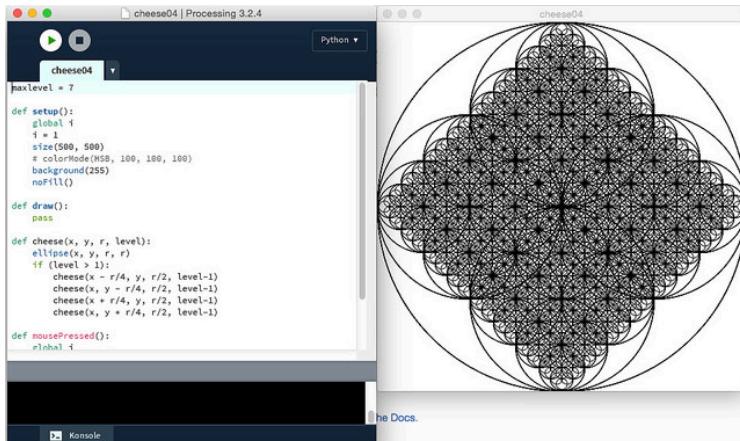
def draw():
    cheese(width/2, height/2, 500, 10)
```

```
def cheese(x, y, r, level):
    ellipse(x, y, r, r)
    if (level > 1):
        cheese(x - r/2, y, r/2, level-1)
        cheese(x + r/2, y, r/2, level-1)
```

kommt die Figur heraus, die den Kopf dieses Beitrages zierte. Das ist zwar streng genommen kein Cantor-Käse mehr, aber dennoch ein interessantes Ergebnis. Das macht den Vorteil des schnellen Skizzierens in Processing aus: Selbst Fehler können unerwartete und notierenswerte Ergebnisse liefern. Man hebt dann den Sketch einfach auf.

Cantors Doppelkäse

Schon bei meinen Experimenten mit Shoes [hatte ich mich gefragt](#), wie es denn aussähe, wenn man diese Figur sich nicht nur in der Horizontalen, sondern auch in der Vertikalen ausbreiten lässt?



Dabei habe ich auch gleich ein interaktives Element eingeführt: Startet man das Programm, zeigt es zuerst nur ein weißes Fenster, nach dem ersten Mausklick sieht man die erste Rekursionsstufe, einen einfachen Kreis, der nächste Mausklick zeigt vier darin eingeschriebene Kreise, der nächste Mausklick zeigt dann in

jedem der kleinen Kreise wiederum vier eingeschriebene Kreise und so weiter und so fort ...

```
maxlevel = 7

def setup():
    global i
    i = 1
    size(500, 500)
    # colorMode(HSB, 100, 100, 100)
    background(255)
    noFill()

def draw():
    pass

def cheese(x, y, r, level):
    ellipse(x, y, r, r)
    if (level > 1):
        cheese(x - r/4, y, r/2, level-1)
        cheese(x, y - r/4, r/2, level-1)
        cheese(x + r/4, y, r/2, level-1)
        cheese(x, y + r/4, r/2, level-1)

def mousePressed():
    global i
    cheese(width/2, height/2, 500, i)
    i += 1
    if (i >= maxlevel):
        noLoop()
```

Das Programm stoppt dann bei einer Rekursionstiefe von sieben. Auch hier ist Processing robuster als Shoes, höhere Rekursionsstufen waren kein Problem, nur man sah dann nicht viel mehr als ein auf der Spitze stehendes Quadrat mit ein paar Ausbuchtungen – die Auflösung des Bildschirms setzt hier neuem Erkenntnisgewinn Grenzen.

Interessant und neu für mich war, daß man – um überhaupt ein Zeichenfenster zu bekommen, in das man mit der Maus kli-

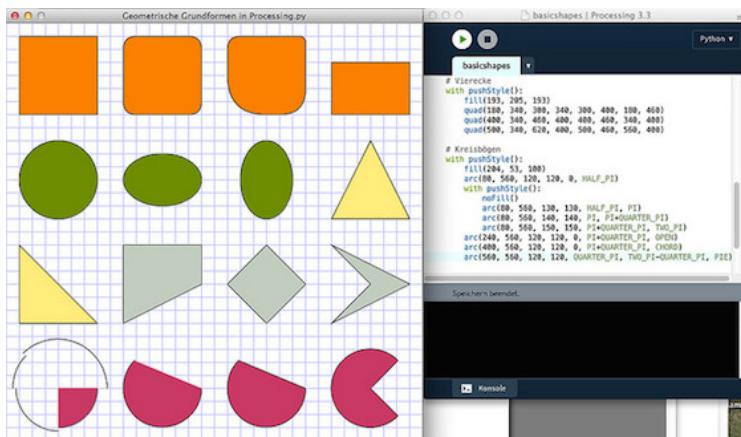
cken konnte – eine leere `draw()`-Funktion benötigte. Eigentlich logisch, aber ich hatte vorher nie darüber nachgedacht.

Literatur

- Clifford A. Pickover: *Mit den Augen des Computers. Phantastische Welten aus dem Geist der Maschine*, München (Markt und Technik) 1992. Diese deutsche Übersetzung von *Computers and the Imagination* ist eine geniale Fundgrube für alle, die Simulationen und mathematische Spielesereien mit dem Computer lieben. Es ist eines der besten Bücher [Pickovers](#). Dem Cantor-Käse ist auf den Seiten 171-181 ein eigenes Kapitel gewidmet.
- Chris Robart: *Programming Ideas: For Teaching High School Computer Programming*, (<%= imageref("pdf")%> 260 KB, 2nd Edition) 2001. Ebenfalls eine Fundgrube voller Ideen, deren Download sich in jedem Fall lohnt.

Weitere geometrische Grundformen

Processing besitzt ein kleines Set von geometrischen Primitiven in 2D (im Englischen *Shapes* genannt) mit denen sich so einiges anstellen lässt. Neben den schon bekannten Punkten und Kreisen und Ellipsen, gibt es noch einige andere, die ich der Reihe nach vorstellen möchte:



Rechtecke

Rechtecke (`rect()`) sind die einfachste Grundform. Dennoch besitzen auch sie einige Besonderheiten. Es gibt sie nämlich in der Form

```

rect(x, y, w, h)
rect(x, y, w, h, r)
rect(x, y, w, h, tl, tr, bl)

```

Bei vier Parametern sind die ersten beiden Parameter, die x- und y-Koordinate der linken, oberen Ecke des Rechtecks und die beiden anderen Parameter geben die Breite und Höhe des Rechtecks an. Gilt `w == h`, dann ist das Rechteck natürlich ein Quadrat.

Wird `rect()` mit fünf Parametern aufgerufen, dann ist der fünfte Parameter als Radius für die Abrundung der Ecken verantwortlich. Mit acht Parametern bekommt jede Ecke einen eigenen Radius für die abgerundeten Ecken einen eigenen Radius zugeschrieben. Dabei wird von *links oben* über *rechts oben* und *rechts unten* nach *links unten* vorgegangen.

Rechtecke besitzen per Default den `rectMode(CORNER)`. Wird ein anderer `rectMode()` eingegeben, dann ändert sich die Bedeutung

des dritten und vierten Parameters. Ist er **CORNERS**, dann benennen die ersten beiden Parameter weiterhin die linke, obere Ecke, der dritte und vierte Parameter aber die x- und y-Koordinaten der rechten, unteren Ecke.

Ist der **rectMode(CENTER)**, dann benennen die ersten beiden Parameter den Mittelpunkt des Rechteckes, der dritte und vierte Parameter gibt aber weiterhin die Breite und Höhe des Rechtecks an.

Dahingegen sind beim **rectMode(RADIUS)** die ersten beiden Parameter die x- und y-Koordinaten des Mittelpunkts des Rechtecks, während die dritte und vierte Koordinate jeweils die Hälfte der Breite und die Hälfte der Höhe angeben.

Der **rectMode(CENTER)** ist vor allen Dingen dann vom Vorteil, wenn Rechtecke mit Kreisen oder Ellipsen koordiniert werden, da bei diesen per Default **ellipseMode(CENTER)** gilt. Zu diesen kommen ich daher im Anschluß [noch einmal](#).

Kreise und Ellipsen

Ellipsen und Kreise (als Spezialform der Ellipse) werden in Processing mit dem Befehl

```
ellipse(x, y, w, h)
```

erzeugt. Dabei sind **x** und **y** der Mittelpunkt der Ellipse und **w** und **h** per Default die Breite und Höhe der Ellipse. Sind **w == h**, dann bildet die Ellipse einen Kreis.

Ändert man jedoch den Default-Mode **CENTER**, dann ergeben sich folgende Bedeutungsänderungen der vier Parameter.

Beim **ellipseMode(RADIUS)** bilden die ersten beiden Parameter weiterhin den Mittelpunkt der Ellipse oder des Kreises, der dritte und vierte Parameter gibt jedoch die Hälfte der Höhe und die Hälfte der Breite der Ellipse oder des Kreises an.

Ist der **ellipseMode(CORNER)**, dann benennen die x- und y-Koordinaten die linke, obere Ecke der Ellipse oder des Kreises,

die beiden anderen Parameter geben weiterhin die Breite und Höhe an.

Heißt es jedoch `ellipseMode(CORNERS)`, dann benennen die x- und y-Koordinaten die linke, obere Ecke des die Ellipse oder den Kreis umschließenden Rechtecks, der dritte und vierte Parameter die rechte untere Ecke dieses Rechtecks.

!!! tip “Achtung” Die Modes `CORNER`, `CORNERS`, `CENTER` und `RADIUS` müssen immer in Großbuchstaben eingegeben werden, da Processing und Python streng zwischen Groß- und Kleinschreibung unterscheiden.

Dreieck

Das Dreieck ist eines der einfachsten geometrischen Grundformen in Processing. Es existiert nur in der Form

```
triangle(x1, y1, x2, y2, x3, y3)
```

und hat auch keinen besonderen Mode. Die jeweiligen x- und y-Koordinaten sind die Koordinaten des ersten, zweiten und dritten Punktes. Bei der Reihenfolge wird – oben beginnend – immer im Uhrzeigersinn vorgegangen. Das ist alles.

Unregelmäßige Vierecke

Ähnlich einfach verhält es sich mit den unregelmäßigen Viercken. Sie werden mit

```
quad(x1, y1, x2, y2, x3, y3, x4, y4)
```

erzeugt und auch hier sind es absolute Koordinaten und das Geblde besitzt ebenfalls keinen besonderen Mode. Auch hier wird bei der Zählung links oben begonnen und dann werden die Ecken ebenfalls im Uhrzeigersinn abgearbeitet.

Kreisbögen

Kreisbögen sind mit der Ellipse (genauer: dem Kreis verwandt) und besitzen die gleichen Modi wie diese (mit dem gleichen Default CENTER). Sie werden wie folgt aufgerufen:

```
arc(x, y, w, h, start, stop)
arc(x, y, w, h, start, stop, mode)
```

Die x- und y-Koordinaten sind im Default-Mode der Mittelpunkt des Kreises, während w und h im Default-Mode die Breite und Höhe des Kreisen angeben. **start** und **stop** sind die Winkel (in *radians*) für die Länge des Kreisbogens.

Dann gibt es hier noch einen besonderen **mode**. Der kann OPEN (das ist der Default), CHORD oder PIE heißen. Im Default OPEN bleibt der Kreisbogen offen, falls es jedoch ein **fill()** gibt, wird er dennoch gefüllt. Bei CHORD wird der Kreisbogen geschlossen und bei PIE bildet er ein Kuchenstück, wie man es von Tortengraphiken kennt.

Der Quelltext

In diesem Beispielprogramm habe ich alle angesprochenen geometrischen Primitive in ihren diversen Erscheinungsformen zeichnen lassen. Mit dem oben geschriebenen dürfte es einfach nachzuvollziehen ein.

```
def setup():
    size(640, 640)
    frame.setTitle("Geometrische Grundformen in Processing.py")
    # noLoop()

def draw():
    background(255)
    drawGrid()
    stroke(0)

    # Rechtecke
```

```
with pushStyle():
    fill(255,127,36)
    rect(20, 20, 120, 120)
    rect(180, 20, 120, 120, 20)
    rect(340, 20, 120, 120, 20, 10, 40, 80)
    rect(500, 60, 120, 80)

# Kreise und Ellipsen
with pushStyle():
    fill(107, 142, 35)
    ellipse(80, 240, 120, 120)
    ellipse(240, 240, 120, 80)
    ellipse(400, 240, 80, 120)

# Dreiecke
with pushStyle():
    fill(255, 236, 139)
    triangle(560, 180, 620, 300, 500, 300)
    triangle(20, 340, 140, 460, 20, 460)

# Vierecke
with pushStyle():
    fill(193, 205, 193)
    quad(180, 340, 300, 340, 300, 400, 180, 460)
    quad(400, 340, 460, 400, 400, 460, 340, 400)
    quad(500, 340, 620, 400, 500, 460, 560, 400)

# Kreisbögen
with pushStyle():
    fill(204, 53, 100)
    arc(80, 560, 120, 120, 0, HALF_PI)
    with pushStyle():
        noFill()
        arc(80, 560, 130, 130, HALF_PI, PI)
        arc(80, 560, 140, 140, PI, PI+QUARTER_PI)
        arc(80, 560, 150, 150, PI+QUARTER_PI, TWO_PI)
    arc(240, 560, 120, 120, 0, PI+QUARTER_PI, OPEN)
    arc(400, 560, 120, 120, 0, PI+QUARTER_PI, CHORD)
    arc(560, 560, 120, 120, QUARTER_PI, TWO_PI-QUARTER_PI, PIE)
```

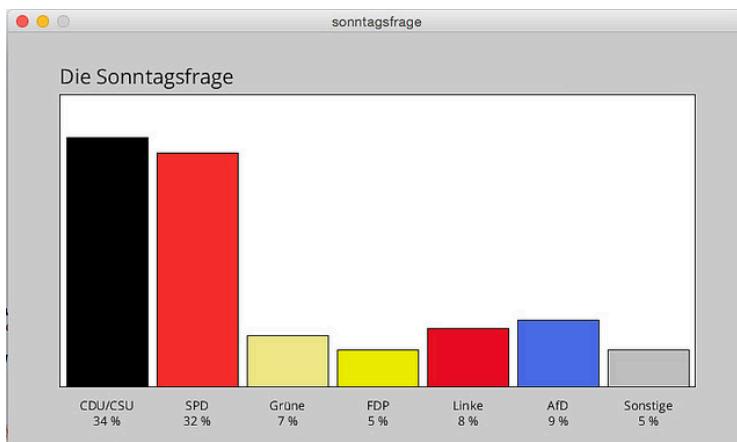
```
def drawGrid():
    stroke(200, 200, 255)
    for i in range(0, width, 20):
        line(i, 0, i, height)
    for i in range(0, height, 20):
        line(0, i, width, i)
```

Ich habe das Fenster mit einem 20 x 20 Pixel großen Raster wie auf kariertem Schulpapier versehen, damit Ihr die Eckpunkte der einzelnen Shapes auszählen könnt, falls Euch die Koordinaten nicht sofort klar werden.

Credits

Teilweise folgt dieser Sketch einer Idee von *Jan Vantomme* aus seinem Buch «[Processing 2: Creative Coding Programming Cook-book](#)» (Seiten 31 ff.). Ich habe sie abgewandelt, die Beispiele für die Kreisbögen hinzugefügt und vom Java-Mode in den Python-Mode übertragen.

Visualisierung: Die Sonntagsfrage



Man kann sich durchaus zu Recht fragen, ob es überhaupt sinnvoll ist, so etwas wie den obigen Barchart in Processing.py per Fuß zu erstellen. Schließlich gibt es (auch in Python) Bibliotheken wie etwa die [Matplotlib](#), die für diese Aufgabe spezialisiert sind und mit wenigen Zeilen Code wunderbarer Graphiken auf den Monitor zaubern und sie auch gleichzeitig publikationsreif in einer Datei ablegen können.

Aber auf der anderen Seite schadet es nichts, wenn man selber genau weiß, wie man so etwas anstellen kann. Denn zum einen hat man vielleicht Gründe, die Umgebung von Processing.py nicht verlassen zu wollen. Und zum anderen gibt es doch auch immer wieder Spezialfälle, die von den spezialisierten Bibliotheken nicht abgedeckt werden.

Daher habe ich hier einmal die Ergebnisse der Sonntagsfrage (»Wenn am nächsten Sonntag Bundestagswahl wäre ...«), die die *Forschungsgruppe Wahlen* am 10. März dieses Jahres veröffentlicht hat, nur mit den Hausmitteln von Processing.py in einem einfachen Barchart dargestellt.

Für die Daten, Namen und Farben habe ich drei Listen erstellt. Das hat den Vorteil, daß man bei einer neuen Umfrage nur die Ergebnisse in der Liste `prozente[]` ändern muß – an den anderen Listen ändert sich zumindest bis zur Bundestagswahl im nächsten Jahr nichts.

Normalerweise werden Rechtecke in Processing ja mit dem Befehl `rect(x, y, w, h)` erzeugt, setzt man jedoch `rectMode(CORNERS)`, dann werden die realen Eckpunkte als Parameter erwartet, also `rect(x1, y1, x2, y2)`.

Für die Anpassung der Balken an den Bildschirmausschnitt habe ich auf Processings `map(value, dataMin, dataMax, targetMin, targetMax)`-Funktion zurückgegriffen, die einen Datenwert von einem Bereich in einen anderen überträgt. Nun hat Python selber aber auch noch eine eingebaute `map(function, iterable, ...)`-Funktion, die mit der Processing-Funktion in Konflikt steht. Aber die Macher von Processing.py haben sich viel Mühe gegeben, diesen Konflikt aufzulösen. Erfüllt `map()` die Signatur der Python-Funktion, wird diese aufgerufen, ansonsten die Processing-Funktion.

Der Quellcode

```
parteien = ["CDU/CSU", "SPD", "Grüne", "FDP", "Linke", "AfD", "Die Linke"]
prozente = [34, 32, 7, 5, 8, 9, 5]
farben = [color(0, 0, 0), color(255, 48, 48), color(240, 230, 140),
          color(238, 238, 0), color(238, 18, 37),
          color(65, 105, 225), color(190, 190, 190)]

titel = "Die Sonntagsfrage"

def setup():
    global X1, X2, Y1, Y2
    size(720, 405)
    X1 = 50
    X2 = width - X1
    Y1 = 60
    Y2 = height - Y1
    font1 = createFont("OpenSans-Regular.ttf", 20)
    textAlign(font1)
    noLoop()

def draw():
    global X1, X2, Y1, Y2
    fill(255)
    rectMode(CORNERS)
    rect(X1, Y1, X2, Y2)
    fill(0)
    textSize(20)
    text(titel, X1, Y1 - 10)
    delta = (X2 - X1)/(len(prozente))
    w = delta*0.9
    x = w*1.22
    textSize(12)
    for i in range(len(prozente)):
        # Balken zeichnen
        h = map(prozente[i], 0, 40, Y2, Y1)
        fill(farben[i])
        rect(x - w/2, h, x + w/2, Y2)
        # Parteinamen und Prozente unter der X-Achse
        textAlign(CENTER, TOP)
```

```
fill(0)
text(parteien[i], x, Y2 + 10)
text(str(prozente[i]) + " %", x, Y2 + 25)
x += delta
```

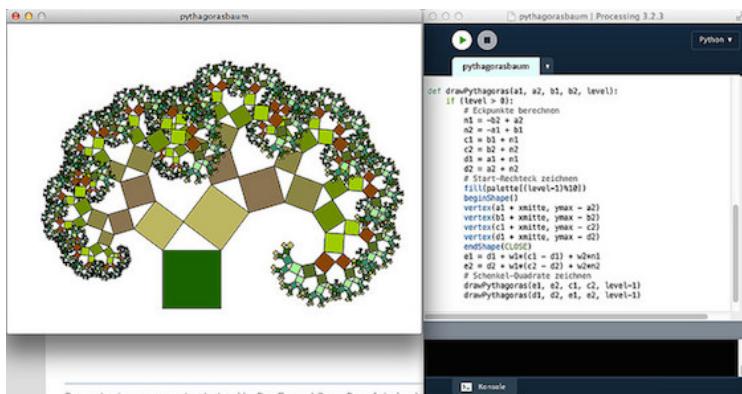
Ansonsten ist der Quellcode leicht nachzuvollziehen. Die Abstands- und Längenwerte für `w` und `delta` habe ich durch Experimentieren herausgefunden, ebenso die Startposition von `x`.

Quellen

Die Zahlen der *Forschungsgruppe Wahlen* habe ich auf der Seite [wahlrecht.de](#) entnommen, dort sind viele weitere Umfrageergebnisse zu Bundes- und Landtagswahlen zu finden. Und den verwendeten Font [Open Sans](#) habe ich bei Google Fonts gefunden. Er steht unter der [Apache-Lizenz, Version 2](#). Ihr solltet nicht vergessen, die entsprechende Datei `OpenSans-Regular.ttf` in den `data`-Folder Eures Sketches zu schieben, damit `Processing.py` den Font auch finden kann.

Der Baum des Pythagoras

Eine weitere Ikone der fraktalen Geometrie ist der [Pythagoras-Baum](#). Er geht zurück auf den niederländischen Ingenieur und späteren Mathematiklehrer *Albert E. Bosman* (1891–1961). Er entwarf während des 2. Weltkrieges in seiner Freizeit an einem Zeichenbrett, an dem er sonst U-Boot-Pläne zeichnete, geometrische Muster. Seine Graphiken wurden 1957 in dem Buch »*Het wondere onderzoeksveld der vlakke meetkunde*« veröffentlicht.



Der Pythagoras-Baum beruht auf einer rekursiven Abbildung des Pythagoras-Lehrsatzes: Die beiden Quadrate auf den Katheten des rechtwinkligen Dreiecks dienen als Verzweigung, auf dem jedes Kathetenquadrat sich wiederum verzweigt.

Die Funktion `drawPythagoras`

Um die Funktion rekursiv aufrufen zu können, mußte ich sie aus der `draw()`-Funktion auslagern und sie in einen eigenen Aufruf packen:

```

def drawPythagoras(a1, a2, b1, b2, level):
    if (level > 0):
        # Eckpunkte berechnen
        n1 = -b2 + a2
        n2 = -a1 + b1
        c1 = b1 + n1
        c2 = b2 + n2
        d1 = a1 + n1
        d2 = a2 + n2
        # Start-Rechteck zeichnen
        fill(palette[(level-1)%10])
        with beginClosedShape():
            vertex(a1 + xmitte, ymax - a2)
            vertex(b1 + xmitte, ymax - b2)
            vertex(c1 + xmitte, ymax - c2)
    
```

```

    vertex(d1 + xmitte, ymax - d2)
e1 = d1 + w1*(c1 - d1) + w2*n1
e2 = d2 + w1*(c2 - d2) + w2*n2
# Schenkel-Quadrate zeichnen
drawPythagoras(e1, e2, c1, c2, level-1)
drawPythagoras(d1, d2, e1, e2, level-1)

```

Zum Zeichnen der einzelnen Quadrate habe ich nicht die `rect()`-Funktion genutzt, sondern *Shapes*, mit denen sich Punkte zu einem beliebigen Gebilde oder Polygon zusammenfassen lassen. Hierzu müssen sie erst einmal mit `with beginClosedShape()` geklammert werden. Darin werden dann mit `vertex(x, y)` nacheinander die einzelnen Punkt aufgerufen, die (im einfachen Fall) durch Linien miteinander verbunden werden sollen. Mit `beginClosedShape` teile ich dem Sketch auch mit, daß das entstehende Polygon auf jeden Fall geschlossen werden soll, ein einfaches `with beginShape()` würde es offen lassen.

Der Aufruf ist rekursiv: Nachdem zuerst das Grundquadrat gezeichnet wurde, werden die rechten und die linken Schenkelquadrate gezeichnet, die dann wieder als Grundquadrate für den nächsten Rekursionslevel fungieren.

Processing (und damit auch der Python-Mode von Processing) ist gegenüber Rekursionstiefen relativ robust. Die benutzte Rekursionstiefe von 12 wird klaglos abgearbeitet, auch Rekursionstiefen bis 20 sind – genügend Geduld vorausgesetzt – kein Problem. Bei einer Rekursionstiefe von 22 verließ mich aber auf meinem betagten MacBook Pro die Geduld.

Die Farben

Für die Farben habe ich eine Palette in einer Liste zusammengestellt, die der Reihe nach die Quadrate einfärbt. Da die Liste nur 10 Elemente enthält, habe ich mit `fill(palette[(level-1)%10])` dafür gesorgt, daß nach 10 Leveln die Palette wieder von vorne durchlaufen wird.

Der Quellcode

Da die eigentliche Aufgabe des Programms in die Funktion `drawPythagoras()` ausgelagert wurde, ist der restlich Quellcode von erfrischender Kürze:

```
palette = [color(189,183,110), color(0,100,0), color(34,139,105),
           color(152,251,152), color(85,107,47), color(139,69,19),
           color(154,205,50), color(107,142,35), color(139,134,7),
           color(139, 115, 85)]  
  
xmax = 600  
xmitte = 300  
ymax = 440  
  
level = 12  
w1 = 0.36    # Winkel 1  
w2 = 0.48    # Winkel 2  
  
def setup():  
    size(640, 480)  
    background(255)  
    strokeWeight(1)  
    noLoop()  
  
def draw():  
    drawPythagoras(-(xmax/10), 0, xmax/20, 0, level)  
  
def drawPythagoras(a1, a2, b1, b2, level):  
    if (level > 0):  
        # Eckpunkte berechnen  
        n1 = -b2 + a2  
        n2 = -a1 + b1  
        c1 = b1 + n1  
        c2 = b2 + n2  
        d1 = a1 + n1  
        d2 = a2 + n2  
        # Start-Rechteck zeichnen  
        fill(palette[(level-1)%10])  
        with beginClosedShape():
```

```

    vertex(a1 + xmitte, ymax - a2)
    vertex(b1 + xmitte, ymax - b2)
    vertex(c1 + xmitte, ymax - c2)
    vertex(d1 + xmitte, ymax - d2)
    e1 = d1 + w1*(c1 - d1) + w2*n1
    e2 = d2 + w1*(c2 - d2) + w2*n2
    # Schenkel-Quadrat zeichnen
    drawPythagoras(e1, e2, c1, c2, level-1)
    drawPythagoras(d1, d2, e1, e2, level-1)

```

Auch wenn es nicht nötig gewesen wäre, ich mag es einfach (und es dient der Übersichtlichkeit), wenn ich meine Processing.py-Sketch mit `def setup()` und `def draw()` gliedere. Mit `noLoop()` habe ich dann dafür gesorgt, daß die `draw()`-Schleife nur einmal abgearbeitet wird.

Erweiterungen und Änderungen

Einen »symmetrischen« Pythagoras-Baum erhält man übrigens, wenn man die beiden Winkel-Konstanten `w1` und `w2` jeweils auf 0.5 setzt.

Credits

Den rekursiven Algorithmus habe ich einem Pascal-Programm aus Jürgen Plate: *Computergrafik: Einführung – Algorithmen – Programmentwicklung*, München (Franzis) 2. Auflage 1988, Seiten 460-462 entnommen. Und die Geschichte des Baumes steht in dem schon mehrfach erwähnten Buch von Dieter Hermann, *Algorithmen für Chaos und Fraktale*, Bonn (Addison-Wesley) 1944 auf den Seiten 170f.

Kapitel 8

Text(verarbeitung) in Processing.py

Mit `print()` oder `println()` kann man in Processing.py jede Ausgabe in das Konsolenfenster bringen, aber was ist, wenn der Text im Graphikfenster ausgegeben werden soll? Ich gehe erst einmal ganz naiv daran:

```
font = None
tt = "Zwölf Boxkämpfer jagen Eva quer über den großen Sylter Dei

def setup():
    size(800, 100)
    font = createFont("American Typewriter", 20)
    textFont(font)

def draw():
    background(255)
    fill(0)
    text(tt, 25, 50)
```

In der ersten Zeile teile ich Processing.py mit, daß ich die Variable `font` verwenden will und belege sie erst einmal mit dem Wert `none`. Das erspart mir ein oder sogar zwei Global-Statements. Die

Stringvariable `tt` bekommt meinen Text zugewiesen. In `setup()` mache ich ein langes, schamles Fenster auf (mein Text ist ja ziemlich lang) und dann teile ich mit `createFont()` Processing.py mit, daß ich den Font *American Typewriter* in der Größe von 20 Pixeln verwenden will und weise ihn der Variablen `font` zu. Zu guter Letzt lege ich noch fest, daß eben mein `textFont font` ist.

In `draw()` lege ich einen weißen Hintergrund und eine schwarze Füllfarbe fest und lasse dann mit der Funktion `text()` den Text in das Fenster zeichnen. `text()` besitzt drei Parameter, zuerst den zu schreibenden (oder besser: zeichnenden) Text, dann die x- und die y-Koordinate des Textbeginns.

Das sieht eigentlich alles ganz einfach aus, aber wenn Ihr den Sketch ausführen lasst, erlebt Ihr Euer blaues Wunder:

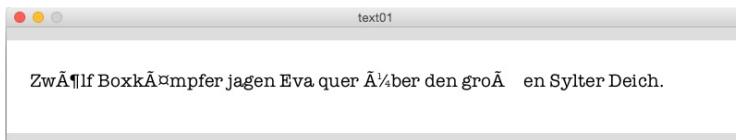


Abbildung 8.1: Screenshot

So verstümmelt habt Ihr Euch das sicher nicht vorgestellt. Die Ursache ist einfach und ärgerlich. Das Processing.py zugrundeliegende Python ist ein Jython (also die Java-Version von Python) und entspricht der Python-Version 2.7. Diese ist leider nicht *out of the box* UTF-8 fähig, ein Umstand, der in der (meist englischsprachigen) Literatur geflissentlich verschwiegen wird¹. Dabei ist er so leicht zu beheben. Ein vor einem String vorangestelltes `u` teilt Python 2.7 mit, daß dieser String ein UTF-8-String ist. Im Sketch ist also lediglich die Zeile

```
tt = "Zwölf Boxkämpfer jagen Eva quer über den großen Sylter Deich."
```

in

```
tt = u"Zwölf Boxkämpfer jagen Eva quer über den großen Sylter Deich."
```

¹Das ist keine Einschränkung, denn der Fensterrand ist ja im Grunde auch nichts anderes als eine undurchdringliche Mauer und im Zweifelsfall macht man das Spielfeld einfach um die Mauerdicke größer.

und schon wird der Text wie gewünscht ausgegeben:



Abbildung 8.2: Screenshot

Es gibt eine weitere, kleine Ungereimtheit im Umgang mit UTF-8 in Processing.py. Im Haupt-Tab, in dem das ausführbare Programm steht (das ist der Tab, der die Endung .pyde bekommt), kann man – wie gezeigt – ohne große Probleme im Programmtext Umlaute unterbringen, während der Code in den anderen Tabs (die unter .py gespeichert werden) strenger mit dem Programmierer umgeht: Wenn nicht in der ersten Zeile

```
# coding=utf-8
```

steht, meckert die IDE gnadenlos, selbst wenn Umlaute nur in den Kommentaren vorkommen.

!!! tip “Pangramm” Der Text mit den zwölf Boxern ist übrigens ein [Pangramm](#), ein Satz, der alle Buchstaben des (in diesem Falle deutschen) Alphabets enthält. Früher wurden sie benutzt, um zum Beispiel Schreibmaschinen nach einer Reparatur zu testen. Heute nutze ich ihn, um festzustellen, ob ein Font auch alle Umlaute des deutschen Alphabets enthält. Das bekannteste englische Pangramm ist der Satz »The quick brown fox jumps over the lazy dog«.

Als die Pangramme laufen lernten

Während in der Funktion `text()` die y-Koordinate immer die Grundlinie des Textes ist, kann man mit `textAlign()` festlegen, ob die x-Koordinate die rechte Kante (`RIGHT`), die linke Kante (`LEFT`) oder die Mitte (`CENTER`) des Textes betrifft. Das möchte ich ausnutzen, um eine Parade der Pangramme zu programmieren. Als erstes lege ich eine Liste mit Pangrammen an (der oben

verlinkte Wikipedia-Artikel ist voll von ihnen). Und damit es auch ein wenig bunt wird, habe ich eine gleichlange Liste mit Farben zusammengestellt. Im Endeffekt soll das dann so aussehen:



Abbildung 8.3: Screenshot

Der Sketch selber ist dadurch ein wenig länger geworden, aber das betrifft in der Hauptsache nur die beiden Listen:

```
font = None
pangramme = [u"Zwölf Boxkämpfer jagen Eva quer über den großen Sylter
              u"Jörg bäckt quasi zwei Haxenfüße vom Wildpony.",
              u"Falsches Üben von Xylophonmusik quält jeden größeren Ze
              u"Schweißgequält zündet Typograph Jakob verflucht öde Pan
              u"Vom Ödipuskomplex maßlos gequält, übt Wilfried zyklisci
              u"Asynchrone Bassklänge vom Jazzquintett sind nix für sp

colors = ["#cd0000", "#008b00", "#ffff00", "#a52a2a", "#ff00ff", "#00

def setup():
    global x, index
    frame.setTitle("Parade der Pangramme")
    size(800, 100)
    font = createFont("American Typewriter", 24)
    textAlign(LEFT)
    x = width
    index = 0

def draw():
    global x, index
    background(0)
    fill(colors[index])
    text(pangramme[index], x, 60)
```

```
x -= 3
w = textWidth(pangramme[index])
if (x < -w):
    x = width
index = (index+1) % len(pangramme)
```

Mit `textAlign(LEFT)` und `x = width` habe ich festgelegt, daß der Text im ersten Schritt am rechten Fensterrand beginnt und quasi ins Leere geschrieben wird. Bei jedem Durchlauf wird `x` um drei dekrementiert und so beginnt das erste Pangramm von rechts nach links durch das Fenster zu scrollen. Ist der Text aus dem sichtbaren Bereich des Fenster verschwunden (`x < -w`), dann wird `index` um einen erhöht und das nächste Pangramm beginnt seine Parade. Damit der Index nicht irgendwann überläuft wird er Modulo der Länge der Liste der Pangramme berechnet. Und da ich in weiser Voraussicht die Länge der Farbliste gleich der Länge der Liste der Pangramme entworfen habe, passiert auch bei den Farben nichts.

Font, Font, Font

Jetzt bleibt nur noch eins zu tun. Auf meinem Rechner läuft der Sketch ohne Probleme, da ich weiß, daß auf meinen Rechner der Font *American Typewriter* vorhanden ist. Dies muß aber nicht auf jedem anderen Rechner der Fall sein (falls also bei Euch die Sketche nicht laufen, tauscht einfach *American Typewriter* mit einem anderen Font, der auf Eurem Rechner vorhanden ist, aus). Wenn ich die `.ttf`-Datei des Fonts in den `data`-Ordner des Sketches kopiere (das geht am einfachsten, wenn ich die Datei auf das Editor-Fenster der IDE schiebe), würde der Sketch – wenn ich ihn weitergebe – überall funktionieren. Aber *American Typewriter* unterliegt mit Sicherheit dem Urheberrecht und eine Weitergabe ist vermutlich verboten oder mit hohen Kosten verbunden.

Aber es gibt ja eine Menge freier Fonts im Web und die größte Quelle dieser freien Fonts ist [Google Fonts](#). Dort habe ich mir den Font [Barrio](#) heruntergeladen, der unter der [Open Font Licence](#) zu nutzen ist.



Abbildung 8.4: Screenshot

Selbstverständlich habe ich mich vorher vergewissert, daß der Font auch die von mir gewünschten deutschen Umlaute enthält. Nachdem ich die Fontdatei dem Sketch hinzugefügt hatte, war eigentlich nur noch eine Zeile im Programm zu ändern:

```
font = createFont("Barrio-Regular.ttf", 64)
```

Barrio ist ein Display-Font, der nur ab einer gewissen Größe wirkt. Daher habe ich ihn auf 64 gesetzt und dann die y-Koordinate etwas weiter nach unten geschoben. Der vollständige und endgültige Sketch der Pangramm-Parade sieht daher nun so aus:

```
font = None
pangramme = [u"Zwölf Boxkämpfer jagen Eva quer über den großen Sylter
             u"Jörg bäckt quasi zwei Haxenfüße vom Wildpony.",
             u"Falsches Üben von Xylophonmusik quält jeden größeren Z
             u"Schweißgequält zündet Typograph Jakob verflucht öde Pany
             u"Vom Ödipuskomplex maßlos gequält, übt Wilfried zyklisc
             u"Asynchrone Bassklänge vom Jazzquintett sind nix für sp

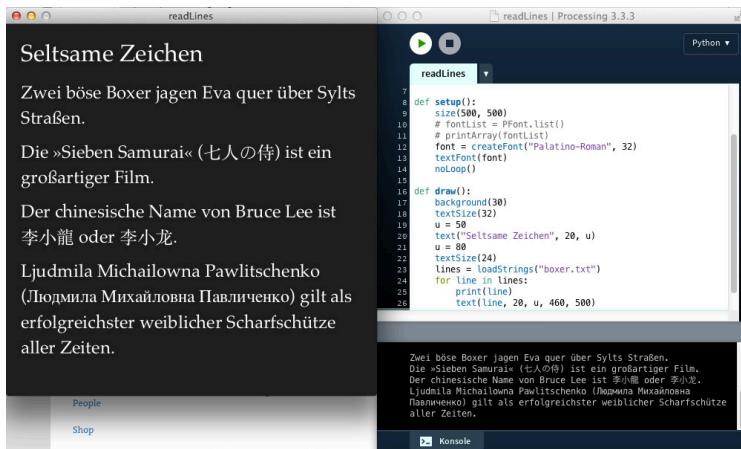
colors = ["#cd0000", "#008b00", "#ffff00", "#a52a2a", "#ff00ff", "#00

def setup():
    global x, index
    frame.setTitle("Parade der Pangramme")
    size(800, 100)
    font = createFont("Barrio-Regular.ttf", 64)
    textFont(font)
    x = width
    index = 0
```

```
def draw():
    global x, index
    background(0)
    fill(colors[index])
    textAlign(LEFT)
    text(pangramme[index], x, 80)
    x -= 3
    w = textWidth(pangramme[index])
    if (x < -w):
        x = width
        index = (index+1) % len(pangramme)
```

Wenn Ihr noch mehr über Strings, Text und Fonts in Processing.py wissen wollt, *Daniel Shiffman* hat dazu ein [nettes Tutorial](#) verfaßt, daß auch mir bei meinen Erkundungen sehr geholfen hat.

UTF-8-Text aus Dateien lesen

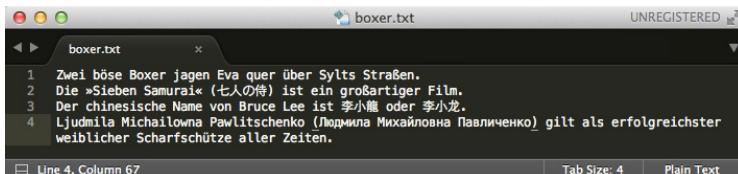


In der Reference für Processing 3 steht bei allen Datei-Operationen, [so auch bei `loadStrings\(\)`](#):

Starting with Processing release 0134, all files loaded and saved by the Processing API use UTF-8 encoding.

In previous releases, the default encoding for your platform was used, which causes problems when files are moved to other platforms.

Das ließ hoffen, daß man in Processing.py wenigstens an dieser Stelle ohne das (von mir) ungeliebte "utf-8-string" auskommen kann. Das wollte ich ausprobieren, also legte ich mir als erstes eine (UTF-8-) Textdatei mit diesem Inhalt an:



Das sieht doch schon sehr gefährlich aus, in der ersten Zeile die bösen deutschen Umlaute, die zweite Zeile mit japanischen Schriftzeichen, die dritte enthält chinesische Glyphen und die letzte Zeile kyrillische (russische) Zeichen. Noch vor wenigen Jahren hätte das jeden Programmierer an den Rand des Wahnsinns gebracht, aber nun: Selbst dieser simple Dreizeiler

```

lines = loadStrings("boxer.txt")
for line in lines:
    print(line)

```

gibt den Text mit allen Sonderzeichen auf der Konsole aus. Und auch der Befehl `text(line, x, y, w, h)` hat keine Schwierigkeiten (einen UTF-8-fähigen Font vorausgesetzt) diesen Text in das Processing-Fenster zu zaubern. Hier das Progrämmchen, das obigen Screenshot produziert:

```

font = None

def setup():
    size(500, 500)
    # fontList = PFont.list()
    # printArray(fontList)
    font = createFont("Palatino-Roman", 32)

```

```
textFont(font)
noLoop()

def draw():
    background(30)
    textSize(32)
    u = 50
    text("Seltsame Zeichen", 20, u)
    u = 80
    textSize(24)
    lines = loadStrings("boxer.txt")
    for line in lines:
        print(line)
        text(line, 20, u, 460, 500)
        u += 80
```

Die beiden auskommentierten Zeilen listen in der Konsole alle auf dem System verfügbaren Fonts auf, mit dem Namen, in dem sie mit `createFont()` in Processing angesprochen werden können. Wenn man einen dieser Fonts verwendet, erspart das zwar einerseits die Installation eines Fonts im `data`-Ordner, macht aber auf der anderen Seite solch ein Skript weniger portabel, denn was ist, wenn der Empfänger diesen Font nicht installiert hat.

Keine Emojis

In einer ersten Version des Textes hatte ich auch noch ein paar Emojis hineingeschmuggelt. Hier wurde aber eine Grenze überschritten, Emojis wurden weder in der Konsole noch auf dem Canvas angezeigt (man kann sie auch nicht per *Copy & Paste*) in den Editor schmuggeln auch nicht mit `u" "`. Das gilt aber auch für den Java-Mode von Processing, Emojis sind erst in P5.js in der Welt von Processing vorgesehen.

Caveat

Auch wenn ich es natürlich schön finde, daß das ungeliebte `u"utf-8-string"` bei den Dateioperationen mit Processing-

Befehlen wegfällt, ist es natürlich inkonsistent. Denn Dateioperationen mit Python-Befehlen arbeiten natürlich weiterhin mit der besonderen UTF-8-Kodierung von Python 2.7, so zum Beispiel die Befehle um CSV- oder JSON-Dateien zu lesen und zu schreiben. Daher ist eine gewisse Vorsicht angebracht.

Spaß mit Processing.py: Rentenuhr

Was für Gründe sprechen eigentlich dafür, Processing.py statt des »normalen« Processings zu nutzen? Nun, zum einen können es persönliche Gründe sein: Ich mag zum Beispiel keine Programmiersprachen, die Blöcke mit geschweiften Klammern ({}) trennen und vermeide sie, wo es nur geht. Zum anderen komme ich aus der [Pascal](#)-Ecke (Pascal und Algol 68 waren meine ersten Programmiersprachen überhaupt) und mag daher Programme, die so etwas sind wie »ausführbarer Pseudocode«. Aber der wichtigste Grund ist, Processing.py ist eben nicht nur Processing, sondern auch Python. Und Python kommt »*batteries included*«, es bringt eine große Anzahl von Standard-Bibliotheken mit, die man auch in Processing.py nutzen kann. Ich möchte das am Beispiel des Python-Moduls `datetime` einmal zeigen:



Als erstes habe ich den freien ([Open Font Licence](#)) Font [Coda Heavy](#) von Googles Seiten heruntergeladen, entpackt und ihn dem Skript zugänglich gemacht, indem ich die .ttf-Datei einfach auf das IDE-Fenster geschoben habe. Processing legt dann

automatisch im Skriptordner ein **data**-Vertzeichnis an und kopiert die Datei – wie auch alle Bild- oder Audio-Dateien dorthin. Die Skripte finden sie dann, zum Beispiel mit

```
font = createFont("Coda-Heavy.ttf", 96)
```

ohne eine spezielle Pfadangabe. Der zweite Parameter gibt die maximale Fontgröße vor. Am Anfang des Skriptes habe ich mit

```
import datetime as dt
```

das Python-Modul **datetime** aus der Standardbibliothek geladen und dann als erstes eine einfache Uhr gebastelt

```
myNow = dt.datetime.now()  
myHour = str(myNow.hour)  
myMinute = str(myNow.minute)  
mySecond = str(myNow.second)
```

und dann die **datetime**-Objekte in Strings verwandelt. Im eigentlichen Programm habe ich sie sogar noch ein wenig aufgehübscht und den einstelligen Sekunden und Minute eine fühlende Null verpaßt. Das könnt Ihr weiter unten im kompletten Quellcode nachlesen.

Jetzt kommt aber der eigentliche Gag: Mit den **datetime**-Objekten kann man nämlich rechnen! Und da ich am 31. Dezember 2018 in Rente gehe, wollte ich wissen, wieviele Tage ich noch ausharren muß

```
rente = dt.date(2018, 12, 31)  
heute = dt.date.today()  
differenz = rente - heute  
myDays = str(differenz.days)  
workingDays = float(myDays)/7.0 * 5  
workingDays = str(int(workingDays - 80))
```

und wieviele Tage davon Arbeitstage sind. Dazu habe ich einfach die Anzahl der Tage durch sieben geteilt und mit fünf multipliziert, was grob die Anzahl der Werkstage ergibt. Und da ich noch

20 Tage Resturlaub in dieses Jahr mitgeschleppt habe und mir pro Jahr auch noch je 30 Tage regulärer Urlaub zusteht, habe ich diese 80 Tage auch noch abgezogen. Die Feiertage habe ich nicht berücksichtigt, mir reicht diese grobe Schätzung.

Da die Differenz zweier `datetime`-Objekte wieder ein `datetime`-Objekt ist, muß die Umwandlung in einen *String* explizit mittels *Typecasting* vorgenommen werden und bei der Division durch sieben ist zu beachten, daß das Processing.py zugrundeliegende Jython ein Python 2.7 ist und deshalb bei einer Integer-Division alle Nachkommastellen abschneidet (zum Beispiel ergibt `13/7` eine 1, dieses – dokumentierte – Verhalten wurde in Python 3 geändert). Um das zu vermeiden, habe ich durch `7.0` geteilt und so eine Float-Division erzwungen und durch ein anschließendes Runden das Ergebnis doch wieder in eine Integer-Zahl verwandelt.

Jetzt das komplette Skript zum Nachlesen und Nachprogrammieren:

```
import datetime as dt

def setup():
    size(640, 480)
    font = createFont("Coda-Heavy.ttf", 96)
    textFont(font)

def draw():
    background("#000000")
    myNow = dt.datetime.now()
    myHour = str(myNow.hour)
    myMinute = str(myNow.minute).rjust(2, "0")
    mySecond = str(myNow.second).rjust(2, "0")
    myTime = myHour + " : " + myMinute + " : " + mySecond
    textSize(96)
    text(myTime, 60, 150)
    rente = dt.date(2018, 12, 31)
    heute = dt.date.today()
    differenz = rente - heute
    myDays = str(differenz.days)
    workingDays = float(myDays)/7.0 * 5
```

```
workingDays = str(int(workingDays - 80))
myText = u"Lieber Jörg, es sind nur noch " + myDays + \
u" Tage bis zu Deiner Rente!\nDas sind etwa " + \
workingDays + " Arbeits- tage. Das schaffst Du!"
textSize(32)
text(myText, 60, 200, 540, 300)
```

Wegen des Umlautes in meinem Vornamen, mußte ich mit `u"..."` die Umwandlung des Strings in einen UTF-8-String erzwingen (auch das ist Python 3 nicht mehr nötig), aber wie der obige Screenshot zeigt, wird dann der Umlaut auch brav angezeigt.

Die Funktion `text()` kann man in Processing einmal mit drei und einmal mit fünf Parametern aufrufen. Im ersten Fall über gibt man den Text und die x- und y-Koordinaten der linken Grundlinie des Textes. Im zweiten Fall kommen noch die Weite und die Höhe der Textbox hinzu. Damit erreicht man, daß ein langer String an den Textbox-Grenzen umgebrochen wird und der Text nicht aus dem Fenster herausläuft. Die Parameter habe ich durch einfaches Ausprobieren bekommen.

Kapitel 9

Bildmanipulation mit Processing.py

Jeder sein kleiner Warhol

Processing und damit auch Processing.py besitzt ein ganzes Arsenal von Filtern zur Bildmanipulation. Davon möchte ich zu Beginn zwei heraussuchen und damit ein kleines Programm erstellen, dessen Ergebnis ein wenig an die berühmten Siebdrucke des Pop-Art-Künstlers [Andy Warhol](#) erinnern soll.

Ich habe dazu ein Photo von unserem Sheltie Joey und mir genommen, das *Stefanie Radon* vor etwa vier Jahren von uns geschossen hatte, und das mein Facebook-Profil zierte und mit dem `filter(THRESHOLD, 0.55)` in eine reine schwarz-weiß-Zeichnung umgewandelt. `THRESHOLD` akzeptiert Parameter zwischen 0.0 und 1.0 – je kleiner der Wert, desto weniger wird angezeigt. Nach einigen Experimenten habe ich mich dann auf 0.55 festgelegt, das brachte in meinen Augen das brauchbarste Ergebnis für dieses Photo.

In der `draw()`-Funktion habe ich dann das Bild acht mal hintereinander in zwei Reihen gezeichnet und mit dem `tint(color)` jeweils in einer anderen Farbe eingefärbt. Ich habe



Abbildung 9.1: Warhol like

einige Zeit mit den Farben experimentieren müssen, bis ich das oben angezeigt Ergebnis bekam, mit dem ich nun zufrieden bin.

Der Quellcode

Der Quellcode ist einfach und leicht zu verstehen. Im `setup()` habe ich das Bild geladen und in eine schwarz-weiß-Version umgewandelt, in `draw()` habe ich dann die acht unterschiedlich eingefärbten Versionen erstellt. Dabei habe ich eine Schleife über die Liste `palette[]` der von mir ausgewählten Farben laufen lassen:

```
palette = [color(205, 133, 63), color(124, 205, 124),
           color(255, 140, 0), color(255, 20, 147),
           color(238, 238, 0), color(224, 102, 255),
           color(151, 255, 255), color(205, 200, 177)]  
  
def setup():
    global jojo
    size(640, 320)
    jojo = loadImage("jojo.jpg")
    jojo.filter(THRESHOLD, 0.55)
    noLoop()
```

```
def draw():
    global jojo
    background(51)
    for i in range(len(palette)):
        if (i < 4):
            row = 0
            j = i
        else:
            row = 160
            j = i - 4
        tint(palette[i])
        image(jojo, j*160, row)
```

Ressourcen

Natürlich könnt Ihr für Eure eigenen Experimente auch das Photo von [Joey und mir](#) nutzen – es steht schließlich auf Flickr und im Fratzenbuch, aber es wäre sicher mehr im Sinne von Andy Warhol, wenn Ihr Euch Eure eigene Bilder (aus-) sucht, die Ihr einfärben und serialisieren wollt.

Filter für die Bildverarbeitung

Processing und damit auch Processing.py bringen eine kleine Sammlung vorgefertigter Filter für die Bildmanipulation mit, die auf jedes Bild angewandt werden können. Die Filter haben folgende Syntax: Entweder

`filter(MODE)`

oder

`filter(MODE, param)`

Ob ein Filter einen zusätzlichen Parameter mitbekommen kann, hängt vom Filter ab. Wie die Filter wirken und ob und wie sie

einen Parameter mitbekommen, könnt Ihr der folgenden Tabelle entnehmen:

Filter

Originalbild (keinen Filter)

THRESHOLD, Parameter (optional) zwischen 0 und 1, Default 0.5

GRAY, keinen Parameter

Filter

INVERT, photographisch gesprochen das Negativ, keinen Parameter

POSTERIZE, zwischen 2 und 255, aber einen richtigen Effekt hat man nur mit nie

BLUR, je größer der Wert, desto verschwommener wird das Bild. Der Parameter ist

Filter

ERODE, keinen Parameter

DILATE (das Gegenteil von ERODE), keinen Parameter

Filter können auch kombiniert werden, hier erst GRAY und dann POSTERI

Mit folgendem kleinen Sketch könnt Ihr mit den diversen Filtern spielen (die auskommentierten Teile habe ich für die *Thumbnails* in obiger Tabelle benötigt):

```
selectFilter = 8
```

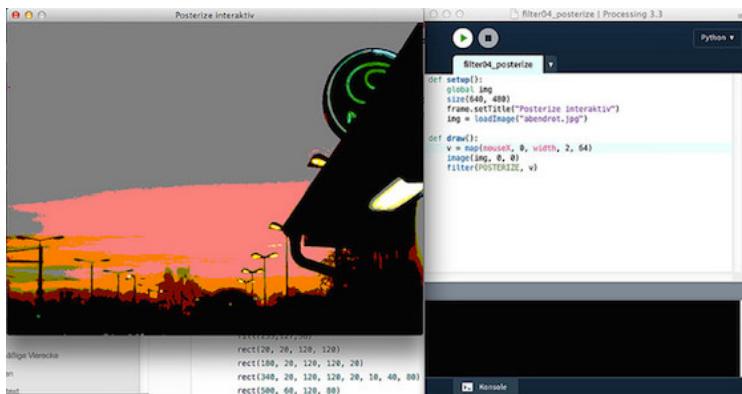
```
def setup():
    global img
    # Thumbnails
    # size(160, 120)
    # img = loadImage("abendrot-s.jpg")
    # Volle Größe
    size(640, 480)
    img = loadImage("abendrot.jpg")
    noLoop()

def draw():
    global img
    background(255, 127, 36)
    image(img, 0, 0)
    if (selectFilter == 1):
        filter(THRESHOLD, 0.55)
    elif (selectFilter == 2):
        filter(GRAY)
    elif (selectFilter == 3):
        filter(INVERT)
    elif (selectFilter == 4):
        filter(POTERIZE, 4)
    elif (selectFilter == 5):
        filter(BLUR, 6)
    elif (selectFilter == 6):
        filter(ERODE)
    elif (selectFilter == 7):
        filter(DILATE)
    elif (selectFilter == 8):
        filter(GRAY)
        filter(POTERIZE, 4)
    # save("filter020" + str(selectFilter) + ".jpg")
```

Einfach bei `selectFilter` den gewünschten Wert (zwischen 0 und 8) eingeben und dann den Sketch laufen lassen. Ihr seid natürlich eingeladen, bei den Filtern, die Parameter zulassen, mit diesen zu spielen.

Die letzte (auskommentierte) Zeile zeigt Euch, wie Ihr das Ergebnis abspeichern könnt. Das Format des Bildes erkennt Processing an der Endung.

Filter interaktiv



Noch besser könnt Ihr natürlich die Wirkung der diversen Filterparameter interaktiv mit der Maus erkunden. Ich habe als Beispiel dafür zwei kleine Sketche geschrieben, die einmal POSTERIZE und zum anderen THRESHOLD erkunden.

```
def setup():
    global img
    size(640, 480)
    frame.setTitle("Posterize interaktiv")
    img = loadImage("abendrot.jpg")

def draw():
    v = map(mouseX, 0, width, 2, 64)
    image(img, 0, 0)
    filter(PPOSTERIZE, v)
```

Da die hohen Werte bei POSTERIZE keinen interessanten Effekte mehr liefern, habe ich hier mithilfe der `map()`-Funktion den Parameter auf die Werte zwischen 2 und 64 begrenzt.

```
def setup():
    global img
    size(640, 480)
    frame.setTitle("Threshold interaktiv")
    img = loadImage("abendrot.jpg")
```

```
def draw():
    v = float(mouseX)/width
    image(img, 0, 0)
    filter(THRESHOLD, v)
```

THRESHOLD erwartet Werte zwischen 0.0 und 1.0. Daher habe ich einfach den `mouseX`-Wert durch die Breite des Fensters geteilt. Wegen der Integer-Division von Python 2.7 mußte ich einen der Werte explizit zu einem `float` konvertieren, um das gewünschte Ergebnis zu erhalten (denn sonst bekommt man nur den Wert Null). So aber wird das Bild, wenn die Maus ganz weit links ist, nur weiß, während es bei einer Mausposition ganz rechts im Fenster fast vollständig schwarz wird. Irgendwo dazwischen liegen die interessanten Ergebnisse. Ihr solltet dies mit diversen Bildern ausprobieren, um ein Gefühl für die zu erwartenden Effekte zu bekommen.

Pointillismus

[Pointillismus](#) bezeichnet eine Stilrichtung der Malerei, die zwischen 1889 und 1910 ihre Blütezeit hatte. Pointillistische Bilder bestehen aus kleinen regelmäßigen Farbtupfern in reinen Farben. Der Gesamt-Farbeindruck einer Fläche ergibt sich erst im Auge des Betrachters und aus einer gewissen Entfernung. So etwas in der Art kann man natürlich auch leicht in Processing.py nachbilden (wobei die möglichst reinen Farben in dem Beispielprogramm nur annähernd getroffen werden, weil es sich bei dem Ausgangsbild um eine handkolorierte Photographie vermutlich ebenfalls aus dem 19. Jahrhundert handelt¹).

Das Programmfenster zeigt links das Ausgangsbild. Rechts entsteht so langsam das aus Kreisen zufälliger Größe zusammengesetzte Zielbild. Dabei besitzen die Punkte einen Ausgangswert (`radius`) von sechs, der mit einem Zufallsfaktor zwischen 0.2 und 1.5 multipliziert wird. (Ich benutze im

¹Das ist keine Einschränkung, denn der Fensterrand ist ja im Grunde auch nichts anderes als eine undurchdringliche Mauer und im Zweifelsfall macht man das Spielfeld einfach um die Mauerdicke größer.



Abbildung 9.2: Nachkolorierter Akt

Programm die `randint()`-Funktion von Python und nicht die eingebaute `random()`-Funktion von Processing. Mir ist die Python-Funktion irgendwie sympathischer, aber das ist vermutlich Geschmackssache.)

Bei jedem Durchlauf der `draw()`-Schleife wird der Farbwert eines zufälligen Punktes im Ursprungsbild ermittelt und dann als Kreis (Punkt) im Zielbild eingezeichnet. Das Ergebnis gleicht dem Ursprungsbild, nur das es den Anschein erweckt, als würde man es durch eine Scheibe Strukturglas, wie sie manchmal Duschen- oder Badezimmertüren zieren, betrachten.

Der Quellcode

```
import random as r
radius = 6

def setup():
    global akt
    size(800, 640)
    akt = loadImage("akt.jpg")
    background(0)
    frameRate(600)

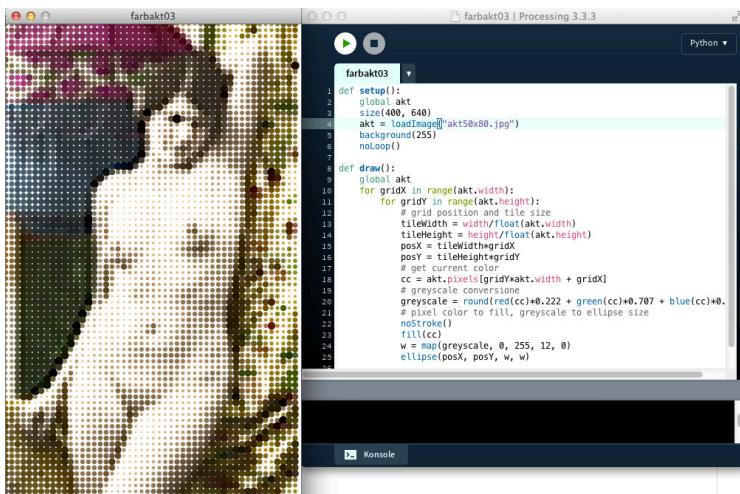
def draw():
    global akt
    image(akt, 0, 0)
    x = r.randint(0, akt.width - 1)
    y = r.randint(0, akt.height - 1)
    c = akt.pixels[x + y*akt.width]
    zufall = r.randint(2, 15)/10.0
    noStroke()
    fill(c)
    ellipse(x + 400, y, radius*zufall, radius*zufall)
```

Der Quellcode ist wieder schön kurz und lädt zum Experimentieren ein. Setzt man zum Beispiel die Konstante `radius = 3`, dann wirkt das Zielbild bedeutend realistischer. Und ein sehr seltsames Ergebnis bekommt man, wenn man die Zeile mit dem `noStroke()` auskommentiert.

Man muß natürlich nicht unbedingt Kreise zeichnen. Ein Quadrat oder ein Dreieck ergibt noch ganz andere Effekte. Spielt einfach mal ein wenig damit herum. Processing(.py) ist zum Spielen entworfen worden.

Noch mehr Pointillismus

Wenn ich ehrlich bin, kann das Ergebnis des Programms aus dem letzten Abschnitt weder ästhetisch noch im Sinne des Pointillismus wirklich überzeugen. Das liegt daran, daß im Programm jedes einzelne Pixel befragt und dann als vergrößerter Punkt wiedergegeben wird. So entsteht im Endeffekt so etwas wie ein verwaschenes Original, aber kein Raster. Daher habe ich – nach einer Idee aus dem wunderbaren Buch »Generative Gestaltung« (derzeit leider nur auf englisch [verfügbar](#)) – tatsächlich eine Rasterversion des Aktbildes programmiert und das Ergebnis überzeugt mich mehr:



Dafür habe ich zuerst das Bild, das im Original 400 x 640 Pixel groß war, auf 50 x 80 Pixel verkleinert um dann mit

```
tileWidth = width/float(akt.width)
tileHeight = height/float(akt.height)
```

```
posX = tileSize*width*gridX  
posY = tileSize*height*gridY
```

ein entsprechendes Raster für das immer noch 400 x 640 Pixel große Ausgabefenster zu schaffen. Mit der Formel

```
greyscale = round(red(cc)*0.222 + green(cc)*0.707 + blue(cc)*0.071)
```

habe ich danach die abgetasteten Farben in Graustufen gewandelt, die Gewichtungen habe ich dem oben erwähnten Buch »Generative Gestaltung« entnommen, die [Wikipedia](#) zum Beispiel nennt andere Gewichtungen, aber auch gleichverteilte Gewichtungen sind möglich und üblich. Hier gibt es also noch Raum für Experimente.

Mit

```
w = map(greyscale, 0, 255, 12, 0)
```

habe ich dann den Radius der Kreise in Abhängigkeit von der Graustufe bestimmt: Je dunkler die Graustufe, desto größer der Kreis. Den Wert 12 habe ich experimentell herausgefunden, auch hier ist ebenfalls noch Raum für Experimente. So bekommt man zum Beispiel auch ein nettes Ergebnis, wenn man die Zeile

```
fill(cc)
```

durch

```
fill(greyscale)
```

ersetzt. Der [Processing-Quellcode](#) aus »Generative Gestaltung« zeigt ebenfalls noch ein paar wirklich nette Möglichkeiten, was man mit so einem Grid alles anstellen kann.

Der Quellcode

Hier nun der vollständige Quellcode, er ist – wie fast immer – erfrischend kurz:

```
def setup():
    global akt
    size(400, 640)
    akt = loadImage("akt50x80.jpg")
    background(255)
    noLoop()

def draw():
    global akt
    for gridX in range(akt.width):
        for gridY in range(akt.height):
            # grid position and tile size
            tileSize = width/float(akt.width)
            tileHeight = height/float(akt.height)
            posX = tileSize*gridX
            posY = tileHeight*gridY
            # get current color
            cc = akt.pixels[gridY*akt.width + gridX]
            # greyscale conversion
            greyscale = round(red(cc)*0.222 + green(cc)*0.707 +
            # pixel color to fill, greyscale to ellipse size
            noStroke()
            fill(cc)
            w = map(greyscale, 0, 255, 12, 0)
            ellipse(posX, posY, w, w)
```


Kapitel 10

Animationen

Ein kleiner roter Luftballon

Die Idee zu diesem Tutorial kam mir, nachdem ich [ein Video](#) von *Daniel Shiffman* gesehen hatte, in dem er Pflanzen wie Blasen aufsteigen ließ. Dieser Sketch war eine Erweiterung eines anderen Sketches *Bubbles* ([Quellcode](#)), in dem er die dort verwendeten Kreise durch die Bilder von Blüten ersetzte. Ich dachte mir, so etwas ähnliches möchte ich auch einmal mit Processing.py programmieren und es sollte auch noch schöner aussehen. Zwar hatte ich zumindest eine der Blüten auch als PNG-Datei – es ist nämlich das Logo von *TextMate 2*, meines bevorzugten Texteditors –, aber ich dachte schon beim Anschauen des Videos sofort an Ballons und die bekommt man als Emoji geliefert. Nun ist aber Python 2.7 und damit auch Jython, das den Python-Mode von Processing antreibt, nicht gerade wirklich UTF-8-fest und Emoji-freundlich, also mußten Bilder her. Die Lösung sind die [Twemojis](#) von Twitter, ein vollständiger Emoji-Bilder-Satz in diversen Auflösungen und auch als SVG, der unter der unter der [CC-BY-4.0](#) Lizenz steht und frei verwendet werden kann. Dort habe ich mir erst einmal den Ballon als 72x72 Pixel großes, transparentes PNG herausgesucht

und dann zum Warmwerden damit diesen kleinen Sketch geschrieben:



Abbildung 10.1: Ballon

```
speed = 1.5

def setup():
    global balloon, x, y
    size(400, 200)
    balloon = loadImage("1f388.png")
    x = random(0, width-72)
    y = height

def draw():
    global balloon, x, y
    background(51)
    image(balloon, x, y)
    y -= speed
    if (y < -72):
        y = height
        x = random(0, width-72)
```

Damit zieht ein einsamer kleiner, roter Luftballon durch das Sketch-Fenster, der – wenn er oben am Fensterrand verschwindet – unten an einer anderen, zufälligen Position wieder auftaucht.

Viele, viele rote Luftballons

Doch da geht natürlich mehr. Ich wollte mehrere Ballons aufsteigen lassen und sie sollten sich auch ein wenig zufälliger bewegen. Und was macht man, wenn man mehrere ähnliche Objekt hat? Richtig, man erstellt eine Klasse für diese Objekte:



Abbildung 10.2: Screenshot

```
class Balloon():

    def __init__(self, dia, img):
        self.diameter = dia
        self.x = random(0, width - self.diameter)
        self.y = height
        self.diameter = dia
        self.img = img
        self.yspeed = random(0.5, 2)

    def move(self):
        self.y -= self.yspeed
        self.x = self.x + random(-2, 2)

    def display(self):
        image(self.img, self.x, self.y, self.diameter, self.diameter)

    def top(self):
        if (self.y <= 0):
            self.y = 0
```

Bilder in Processing funktionieren im Prinzip wie Rechtecke. Wird die Funktion `image(x, y)` nur mit zwei Parametern aufgerufen, wird das Bild an dieser Stelle in seiner vollen Größe

gezeigt. Ruft man hingegen `image(x, y, w, h)` auf, dann wird das Bild an dieser Stelle mit den Seitenlängen `w` und `h` gezeigt. Dabei wird das Bild im Zweifelsfalle auch proportional gestaucht oder gestreckt. Ihr könnt es einfach mal ausprobieren, indem Ihr ein Bild in `draw()` mit `image(0, 0, mouseX, mouseY)` aufruft.

Ich habe aber einfach dem Konstruktor der Klasse den Durchmesser des Bildes mitgegeben und eine Referenz auf das Bild, das zu laden ist. Dann wird mit `move()` das Bild bewegt und mit `display()` wird es in das Sketchfenster gezeichnet. Diese Konstruktion wird Euch in vielen Klassen in Processing begegnen.

Eine Besonderheit ist die Methode `top()`. Hier wird abgefragt, ob der Luftballon das obere Fenster erreicht hat und bleibt dann zitternd dort kleben.

Hier dann das Hauptprogramm,

```
from balloon import Balloon

numBalloons = 15
balloons = []

def setup():
    size(640, 320)
    i = 0
    balloon = loadImage("1f388.png")
    while (i < numBalloons):
        dia = random(24, 72)
        balloons.append(Balloon(dia, balloon))
        i += 1

def draw():
    background(51)
    for i in range(len(balloons)):
        balloons[i].move()
        balloons[i].display()
        balloons[i].top()
```

das dann dieses Bild erzeugt, das ich in diesem Screenshot zu Beginn festgehalten habe:



Abbildung 10.3: Screenshot

Ein wenig habe ich dabei gemogelt. Denn um überhaupt noch ein paar Ballons zu erwischen, die nicht an der Decke kleben, hatte ich die Geschwindigkeit drastisch reduziert.

Es kann nicht nur einen geben

Meine Idee war es aber, daß die Ballons vollständig den oberen Fensterrand passieren und dann an einer zufälligen Position und in einer zufälligen Größe unten wieder auftauchte, so daß die Illusion eines kontinuierlichen Ballonaufstiegs entsteht. Daher habe ich die Methode `top()` in der Klasse `Ballon` umgeschrieben:

```
def top(self):
    if (self.y <= -self.diameter):
        self.y = height + self.diameter
        self.x = random(0, width - self.diameter)
        self.diameter = random(24, 72)
        self.yspeed = random(0.5, 2)
```

Außerdem fand ich nur einen Ballon langweilig. *Dan Shiffman* hat in seinem oben erwähnten Video ja auch drei unterschiedliche Blüten genutzt. Also habe ich die Twemojis weiter geplündert und mir diese drei Vertreter ausgesucht:



Neben dem schon bekannten Ballon (`1f388.png`) ist es noch ein Halloween-Kürbis (`1f383.png`) und ein japanisches Windspiel (`1f390.png`), das zum Himmel aufsteigen soll:



Abbildung 10.4: Screenshot

Die Klasse `Balloon` mußte dafür nicht weiter geändert werden, aber im Hauptprogramm habe ich einige Erweiterungen durchgeführt.

```
from balloon import Balloon

numBalloons = 100
balloons = []

def setup():
    size(640, 320)
    i = 0
    balloon = loadImage("1f388.png")
    jackolantern = loadImage("1f383.png")
```

```
windchime = loadImage("1f390.png")
while (i < numBalloons):
    rand = random(10)
    if (rand < 1):
        img = jackolantern
    elif (rand < 8):
        img = balloon
    else:
        img = windchime
    dia = random(24, 72)
    balloons.append(Balloon(dia, img))
    i += 1

def draw():
    background(51)
    for i in range(len(balloons)):
        balloons[i].move()
        balloons[i].display()
        balloons[i].top()
```

Die Anzahl der »Ballons« habe ich großzügig auf 100 erhöht – man hat's ja. In `setup()` habe ich dann die Bilder der einzelnen Objekte geladen und in der `while`-Schleife sie erst einmal zufällig verteilt und dann die 100 Objekte in einer Liste erzeugt.

In Lorenzkirch ist Jahrmarkt

Zu guter Letzt habe ich noch den Hintergrund aufgehübscht und ihn mit einem Bild des [Jahrmarkts von Lorenzkirch](#) um 1900 versehen. Der vollständige Sketch sieht nun so aus:

```
from balloon import Balloon

numBalloons = 100
balloons = []

def setup():
    global jahrmarkt
    size(640, 320)
```

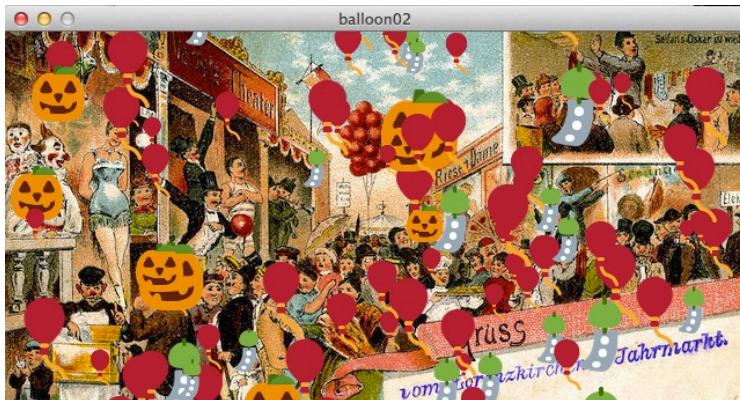


Abbildung 10.5: Screenshot

```

jahrmarkt = loadImage("jahrmarkt.jpg")
i = 0
balloon = loadImage("1f388.png")
jackolantern = loadImage("1f383.png")
windchime = loadImage("1f390.png")
while (i < numBalloons):
    rand = random(10)
    if (rand < 1):
        img = jackolantern
    elif (rand < 8):
        img = balloon
    else:
        img = windchime
    dia = random(24, 72)
    balloons.append(Balloon(dia, img))
    i += 1

def draw():
    global jahrmarkt
    background(jahrmarkt)
    # background(51)
    for i in range(len(balloons)):
        balloons[i].move()
        balloons[i].display()

```

```
balloons[i].top()
```

An der Klasse `Balloon` wurde nichts mehr geändert.

Credits

Neben den oben schon erwähnten Twemojijs, für die ich Twitter danke, habe ich das [Bild des Lorenzmarktes um 1900](#) den Wikimedia Commons entnommen. Es ist alt genug, daß es gemeinfrei ist und frei verwendet werden darf.



Abbildung 10.6: Screenshot

Kapitel 11

Spaß mit (SVG-) Shapes: Pinguine im Eismeer

Neben Bildern kann Processing (und damit auch Processing.py) auch SVG-Dateien laden und darstellen. Wie alle anderen Assets auch, müssen diese sich im `data`-Verzeichnis des Sketches befinden, damit Processing sie auch finden kann. Doch woher soll solch ein absoluter Kunstbananase wie ich SVG-Dateien finden, wenn er sie nicht stehlen will? Hier haben mir wieder die [Tweemojis](#) geholfen, der vollständige Emoji-Bilder-Satz von Twitter, der nicht nur in diversen Auflösungen, sondern auch als SVG-Dateien unter der [CC-BY-4.0 Lizenz](#) zur Verfügung steht und frei verwendet werden kann. Eines dieser Emojis ist ein Pinguin und mit dem werde ich nun ein wenig herumspielen.

SVG-Dateien sind Vektographiken und sie können verlustlos vergrößert und verkleinert werden. Das möchte ich im ersten Sketch vorführen:

```
def setup():
    global penguin
    size(400, 400)
```

```
penguin = loadShape("1f427.svg")
shapeMode(CENTER)

def draw():
    background(155)
    shape(penguin, width/2, height/2,
          map(mouseX, 0, width, 0, 800),
          map(mouseX, 0, width, 0, 800))
```

Die Größe des Pinguins ist nun von der Mauskoordinate-x abhängig und variiert von winzig klein bis riesengroß. Zu beachten ist, daß auch ein SVG-Shape diverse `shapeModes()` besitzen kann, der Default ist CORNER, bei dem die linke obere Ecke der Ankerpunkt ist, aber es gibt, wie bei anderen Shapes auch, die Modes CENTER und CORNERS.

Ein SVG-Shape wird mit `loadShape("dateiname.svg")` geladen und mit

```
shape(dateihandle, x, y)
```

oder

```
shape(dateihandle, x, y, w, h)
```

in das Sketch-Fenster gezeichnet. Im ersten Fall wird das SVG-Shape an den Punkten `x`, `y` in der Originalgröße gezeichnet, im zweiten Fall wird das SVG-Shape mit den Paramtern für die Weite und Höhe verkleinert oder vergrößert dargestellt.

Wenn Ihr diesen Sketch laufen lasst, seht Ihr, daß die Verkleinerung oder Vergrößerung tatsächlich absolut verlustfrei erfolgt.

Und nun das Eismeer

Wenn man den obige Sketch ein wenig erweitert, kann man etwas mehr Aktion in die Angelegenheit bringen. Ich habe dazu drei Pinguine in ein Eismeer gestellt, die, je nach Interpretation, ein wenig die Gegend erkunden oder sich einfach nur die Füße vertreten.



Abbildung 11.1: Screenshot



Abbildung 11.2: Screenshot

```
easing = 0.05
offset = 0

def setup():
    global penguin, landscape
    size(640, 400)
    penguin = loadShape("1f427.svg")
    landscape = loadImage("eismeer.jpg")

def draw():
    global offset
    background(landscape)

    targetOffset = map(mouseX, 0, width, -100, 100)
    offset += (targetOffset - offset)*easing
    smallerOffset = offset*0.7
    smallestOffset = smallerOffset * -0.5
    shape(penguin, 60 + offset, 160, 160, 160)
    shape(penguin, 260 + smallerOffset, 130, 80, 80)
    shape(penguin, 520 + smallestOffset, 220, 120, 120)
```

Es passiert eigentlich immer noch nicht viel, aber wenn man etwas mehr Aktion wünscht, dann kommt man wohl nicht darum herum, eine Pinguin-Klasse anzulegen und die einzelnen Pinguin-Objekte in einer Liste zu verwalten, so wie ich es zum Beispiel im letzten Tutorial mit den Ballons gemacht hatte.

Wartet, da ist noch mehr

Die Funktion `shape()` kann nicht nur SVG-Dateien darstellen, sondern auch dreidimensionale OBJ-Dateien. Dafür muß natürlich der Sketch im P3D-Mode laufen.

Credits



Abbildung 11.3: Das Eismeer

Das Hintergrundbild des zweiten Sketches ist ein Gemälde des deutschen, romantischen Malers [Caspar David Friedrich](#). Dieser ist 1840 gestorben, also hinreichend lange tot, so daß das Bild gemeinfrei ist und ohne Lizenzkosten verwendet werden kann.



Kapitel 12

Objekte und Klassen mit Kitty

Hallo Hörnchen – Hallo Kitty revisited

Nachdem ich am Wochenende mal wieder an [PyGame](#) verzweifelt bin (aus irgendwelchen Gründen funktionierte die Tastaturabfrage nicht), habe ich beschlossen, mich doch eher [Processing.py](#) zuwenden, dem Python-Mode für [Processing](#). Ziel sollte es sein, mein vierteiliges PyGame-Tutorial vom Mai dieses Jahres in Processing.py zu implementieren. Erfreulich war, daß ich mir den [ersten Teil](#) gleich schenken konnte, denn

```
def setup():
    size(640, 480)

def draw():
    background(0, 80, 125)
```

erzeugt bereits ein leeres, blaues Fenster. Also habe ich gleich den [zweiten Teil](#) in Angriff genommen und das »*Horn Girl*« aus dem von *Daniel Cook (Danc)* in seinem Blog [Lost Garden](#) unter einer freien Lizenz ([CC BY 3.0 US](#)) zu Verfügung gestellten Tileset [Planet Cute](#) in das Fenster gezaubert:

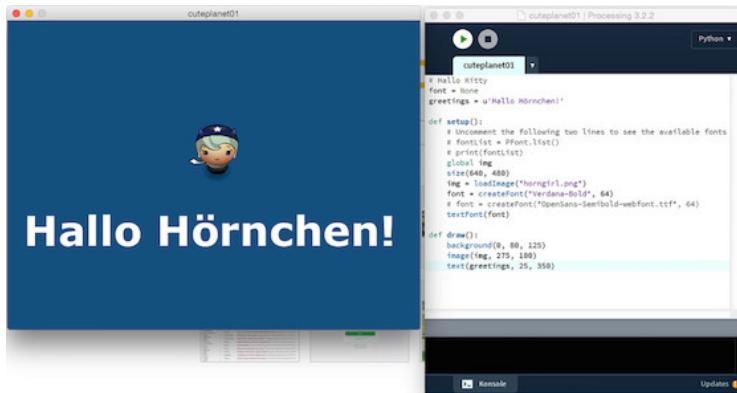


Abbildung 12.1: Hallo Hörnchen!

Zur Vorbereitung habe ich erst einmal das Bild der jungen Dame auf das Editorfenster der Processing-IDE geschoben. Falls noch nicht vorhanden, erzeugt Processing dann automatisch ein **data**-Verzeichnis und legt das Bild (aber auch Schriften oder andere Dateien) darin ab. Processing und damit auch Processing.py finden alles in diesem Verzeichnis ohne daß eine genauere Pfadangabe nötig ist. Und so ist auch das fertige Programm von erfrischender Kürze:

```

font = None
greetings = u'Hello Hörnchen!'

def setup():
    global img
    size(640, 480)
    img = loadImage("horngirl.png")
    font = createFont("Verdana-Bold", 64)
    textFont(font)

def draw():
    background(0, 80, 125)
    image(img, 275, 100)
    text(greetings, 25, 350)

```

Mehr ist nicht nötig, um obigen Screenshot zu bekommen. Vergleiche ich diese 14 Zeilen mit den 34 Zeilen der PyGame-Version, dann frage ich mich schon, warum ich nicht früher zu Processing.py gewechselt bin¹.

An der zweiten Zeile kann man es erkennen: Processing.py basiert auf [Jython](#) und ist damit kompatibel zu Python 2.7.5, aber nicht zu Python 3. Daher muß man Unicode-Strings (zum Beispiel mit deutschen Umlauten) explizit mit dem vorangestellten u markieren, sonst bekommt man seltsame Zeichen im Programmfenster angezeigt.

Processing(.py) kann mit Fonts im TrueType- (.ttf), OpenType- (.otf) und in einem eigenen Bitmap-Format, genannt VLW, umgehen. Natürlich findet es alle auf dem eigenen Rechner installierte Fonts, mit

```
fontList = PFont.list()
print(fontList)
```

kann man sich diese in der Konsole anzeigen lassen. Wenn man den Sketch allerdings weitergeben will, ist es sinnvoll, einen Font mitzugeben², da man nicht sicher sein kann, ob der gewählte Systemfont auf dem anderen Rechner vorhanden ist. Dafür schiebt man eine entsprechende Font-Datei einfach ebenfalls auf das Editorfenster der IDE, damit sie dem **data**-Ordner hinzugefügt wird. Ich habe testweise mal die Datei **OpenSans-Semibold-webfont.ttf** installiert, die entsprechende Zeile im Programm hieße dann:

```
font = createFont("OpenSans-Semibold-webfont.ttf", 64)
```

Der zweite Parameter der Funktion **createFont()** benennt die gewünschte Größe des Fonts. Mehr ist zu diesem ersten Sketch

¹Das ist keine Einschränkung, denn der Fensterrand ist ja im Grunde auch nichts anderes als eine undurchdringliche Mauer und im Zweifelsfall macht man das Spielfeld einfach um die Mauerdicke größer.

²Natürlich sollte man sicherstellen, daß man diese Fonts auch verwerten darf, aber im Netz findet man viele Fonts zur freien Verwendung. Gute Anlaufstellen dafür sind zum Beispiel [Google Fonts](#), die [\(Open\) Font Library](#) oder [The League of Moveable Type](#).

in Processing.py eigentlich nicht zu sagen. Im nächsten Schritt wird es darum gehen, die junge Dame über das Fenster zu bewegen. Nach meinen Erfahrungen mit PyGame werde ich sie nicht nur mit der Maus, sondern auch mit der Tastatur steuern. *Still digging!*

Moving Kitty

Im zweiten Teil meiner kleinen Erkundung von [Processing.py](#), dem Python-Mode von [Processing](#), möchte ich die im ersten Teil auf den Monitor gezauberte *Kitty* mithilfe der Pfeiltasten der Tastatur sich über den Monitor bewegen lassen.

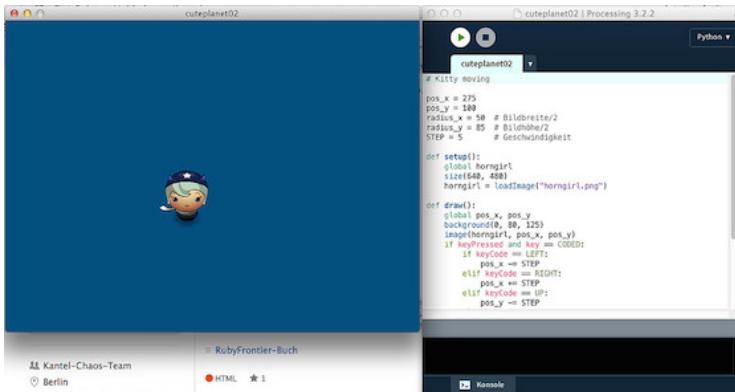


Abbildung 12.2: Moving Kitty

In Processing gehören die Pfeiltasten wie einige andere auch zu den **coded keys**, weil sie sich nicht einem Buchstaben zuordnen lassen und haben daher einen speziellen Namen. Die Pfeiltasten heißen LEFT, RIGHT, UP und DOWN, andere **coded keys** sind zum Beispiel ALT, CONTROL oder SHIFT. Diese müssen in Processing wie in Processing.py gesondert abgefragt werden, und zwar so:

```
if keyPressed and key == CODED:
    if keyCode == LEFT:
```

während die »normalen« Tasten so abgefragt werden können:

```
if keyPressed:  
    if key == 'b' or key == 'B':
```

Das ist eigentlich alles, was man wissen muß, um das Programmchen zu verstehen. Wenn Kitty den linken Rand des Fensters erreicht hat, taucht sie am rechten Rand wieder auf und umgekehrt. Genauso habe ich mit oben unten verfahren. Die Variablen `radius_x` und `radius_y` sorgen dafür, daß *Kitty* vollständig vom Bildschirm verschwunden ist, bevor sie am anderen Ende wieder auftaucht (ich mag keine halben Kittys) und mit `STEP` bestimmt Ihr die Geschwindigkeit, mit der Kitty über den Bildschirm wuselt. Hier der vollständige Quellcode zum nachprogrammieren:

```
pos_x = 275  
pos_y = 100  
radius_x = 50 # Bildbreite/2  
radius_y = 85 # Bildhöhe/2  
STEP = 5      # Geschwindigkeit  
  
def setup():  
    global horngirl  
    size(640, 480)  
    horngirl = loadImage("horngirl.png")  
  
def draw():  
    global pos_x, pos_y  
    background(0, 80, 125)  
    image(horngirl, pos_x, pos_y)  
    if keyPressed and key == CODED:  
        if keyCode == LEFT:  
            pos_x -= STEP  
        elif keyCode == RIGHT:  
            pos_x += STEP  
        elif keyCode == UP:  
            pos_y -= STEP  
        elif keyCode == DOWN:  
            pos_y += STEP  
        if pos_x > width + radius_x:  
            pos_x = -radius_x
```

```
elif pos_x < -2*radius_x:  
    pos_x = width + radius_x  
if pos_y < -2*radius_y:  
    pos_y = height  
elif pos_y > height:  
    pos_y = -radius_y
```

Kitty alias »*Horn Girl*« stammt wieder aus dem von *Daniel Cook (Danc)* in seinem Blog [Lost Garden](#) unter einer [freien Lizenz \(CC BY 3.0 US\)](#) zu Verfügung gestellten Tileset [Planet Cute](#). Aber Ihr könnt natürlich auch jedes andere Bild nehmen, das gerade auf Eurer Festplatte herumliegt.

Klasse Kitty!

Nach den [ersten beiden](#) Teilen meiner kleinen Erkundung von [Processing.py](#), dem [Python](#)-Mode von [Processing](#) und vermutlich die einzige, derzeit aktiv gepflegte Alternative zu [PyGame](#), möchte ich erst einmal ein wenig aufräumen und daran erinnern, daß Akteure eines Computerspiels programmiertechnisch am besten in Klassen aufgehoben sind. Daher habe ich auch *Kitty* eine eigene Klasse spendiert:

```
# coding=utf-8  
  
class Kitty(object):  
    def __init__(self, tempX, tempY):  
        self.x = tempX  
        self.y = tempY  
        self.radiusX = 50 # Bildbreite/2  
        self.radiusY = 85 # Bildhöhe/2  
  
    def loadPic(self):  
        self.img = loadImage("horngirl.png")  
  
    def move(self):  
        self.x = mouseX - self.radiusX  
        self.y = mouseY - self.radiusY
```

```
def display(self):
    image(self.img, self.x, self.y)
```

Klassen kann man in Processing der Übersicht halber in separaten Dateien unterbringen, die in der IDE jeweils einen eigenen Reiter bekommen (siehe Screenshot).

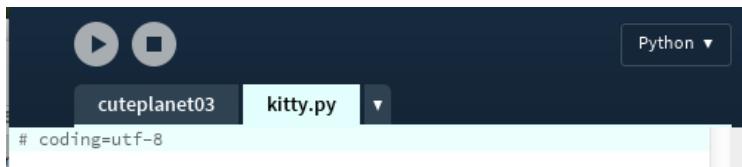


Abbildung 12.3: Klasse Kitty

Hierbei ist jedoch zu beachten, daß im Gegensatz zu Processing und P5.js (jeweils aus anderen Gründen) die Klasse nicht automatisch dem Quelltext der Applikation bei der Ausführung hinzugefügt wird. Sie ist wenn sie nicht im Quelltext der Applikation steht – wie in Python üblich – ein Modul und muß gesondert mit

```
from kitty import Kitty
```

importiert werden. Und da sie ein reines Python2- (oder genauer Jython-) Modul ist, sollte man auch nicht vergessen `# coding=utf-8` in die erste Zeile der Datei schreiben, denn sonst bekommt man Probleme mit dem ö im Kommentar (*Bildhöhe*).

Den Konventionen folgend, habe ich dem Objekt *Kitty* neben der eigentlichen Initialisierung drei Funktionen spendiert, nämlich `loadPic()`, `move()` und `display()`. Die beiden letzteren hätte man auch in einer Funktion zusammenfassen können (beispielsweise `update()` wie bei PyGame üblich), aber da die Philosophie sein sollte, jeder Aktivität eine eigene Funktion zu spendieren, bin ich der Konvention gefolgt³.

³Das ist keine Einschränkung, denn der Fensterrand ist ja im Grunde auch nichts anderes als eine undurchdringliche Mauer und im Zweifelsfall macht man das Spielfeld einfach um die Mauerdicke größer.

Ansonsten ist zu dem Programm nichts weiter zu sagen. Es zeigt einfach eine *Kitty* die der Maus hinterherrennt. Und dadurch, daß fast die gesamte Logik in die Klasse *Kitty* ausgelagert wurde, ist das Hauptprogramm von erfrischender Kürze:

```
from kitty import Kitty

kitty = Kitty(275, 100)

def setup():
    size(640, 480)
    kitty.loadPic()

def draw():
    background(0, 80, 125)
    kitty.move()
    kitty.display()
```

So muß es ja auch sein.



Abbildung 12.4: Horm Girl

Kitty alias »*Horn Girl*« stammt wieder aus dem von *Daniel Cook (Danc)* in seinem Blog [Lost Garden](#) unter einer [freien Lizenz \(CC BY 3.0 US\)](#) zu Verfügung gestellten Tileset [Planet Cute](#).

»Cute Planet« mit Processing.py

Im [letzten Teil](#) meiner kleinen Tutorial-Reihe zu [Processing.py](#), dem [Python](#)-Mode für [Processing](#), hatte ich ja eine Klasse erstellt. Auf den ersten Blick erschien sie nicht besonders nützlich,

da ich im eigentlichen Sketch ja nur eine Instanz der Klasse erzeugt hatte. So sah das schon ein wenig nach mehr Schreibaufwand ohne großen Nutzen aus. Um die Skeptiker zu überzeugen, werde ich in diesem Tutorial wieder eine Klasse erstellen, von der es im Sketch dann aber vier Instanzen geben wird. Vier Raumschiffe werden im Anschluß über den Bildschirm wuseln.



Abbildung 12.5: Cute Planet

Also erst einmal die Klasse selber, ich habe sie aus naheliegenden Gründen **Spaceship** genannt (auch wenn ein Planet ja im eigentlichen Sinne des Wortes kein Raumschiff ist, aber wie Ihr später sehen werdet, in »Space Cute« schon):

```
class Spaceship():

    def __init__(self, pic, posX, posY):
        self.pic = pic
        self.x = posX
        self.y = posY
        self.dx = 0

    def loadPic(self):
        self.img = loadImage(self.pic)

    def move(self):
        self.x += self.dx
```

```
if self.x >= width + 120:  
    self.x = -120  
    self.y = random(height-120)  
elif self.x < -120:  
    self.x = width + 120  
    self.y = random(height-120)  
  
def display(self):  
    image(self.img, self.x, self.y)
```

Der Konstruktor der Klasse verlangt die URL eines Bildes, das das Raumschiff (oder den Planeten) auf dem Monitor darstellt und eine initiale Position, auf der es im Fenster erscheinen soll.

Dann gibt es die Funktion `loadPic()`, die dieses Bild dann lädt. Die Bilder stammen wieder aus dem von *Daniel Cook (Danc)* in seinem Blog [Lost Garden](#) unter einer [freien Lizenz \(CC BY 3.0 US\)](#) zu Verfügung gestellten Tileset [Planet Cute](#). Ich habe sie mit dem [Bildverarbeitungsprogramm meiner Wahl](#) zurechtgeschnitten und auf eine Größe von 120x120 Pixeln heruntergerechnet und sie dann durch einfaches Schieben auf das Editor-Fenster der Processing IDE in den `data`-Ordner des Sketches transportiert. So findet Processing (und damit auch Processing.py) sie ohne zusätzliche Pfadangabe.



Dann folgt die Funktion `move()`, die das Herzstück der Klasse darstellt. Hier werden die einzelnen Raumschiffe bewegt und wenn sie die Grenzen des Fenster verlassen haben, von der gegenüberliegenden Seite von einer zufällig gewählten Position wieder zurück ins Fenster geschickt. Die Funktion `display()` kümmert sich dann um die Darstellung des Raumschiffs.

Nun das Hauptprogramm: Dank der Klasse `Spaceship` ist es kurz und übersichtlich geblieben.

```
from spaceship import Spaceship

planet = Spaceship("planet.png", 500, 350)
rocket = Spaceship("rocketship.png", 300, 300)
octopussy = Spaceship("octopussy.png", 400, 150)
beetle = Spaceship("beetleship.png", 200, 100)

ships = [planet, rocket, octopussy, beetle]

def setup():
    size(640, 480)
    planet.loadPic()
    planet.dx = 1
    rocket.loadPic()
    rocket.dx = 10
    octopussy.loadPic()
    octopussy.dx = -5
    beetle.loadPic()
    beetle.dx = 5

def draw():
    background(0, 80, 125)
    for i in range(len(ships)):
        ships[i].move()
        ships[i].display()
```

Als erstes wird die Klasse `Spaceship` importiert und der Variablen `spaceship` zugewiesen. Dann werden vier *Spaceships*« erzeugt und einer Variablen zugewiesen, die den Konstruktor der Klasse aufruft. Dann wird noch eine Liste erstellt, die alle vier

»Raumschiffe« enthält. Im `setup()` laden dann alle vier ihre Bilder und bekommen (mit `dx`) eine Geschwindigkeit verpaßt.

Das war es dann scho fast: In `draw()` wird dann nur noch eine Schleife durchlaufen, die für jedes der »Raumschiffe« die Funktionen `move()` und `display()` aufruft. Wenn Ihr nun den Sketch laufen läßt, werdet Ihr sehen, daß im Weltall rund um den Planeten »Space Cute« ein Verkehr wie auf dem Kudamm herrscht. Stellt Euch mal vor, ich hätte noch mehr Instanzen der Klasse `Spaceship` erzeugt.

Fluffy Planet

Nachdem ich nun ein paar Tage mit Orks und Fraktalen herumgespielt habe, fühle ich mich gestärkt genug, um mit meinen Processing.py-Experimenten mit den niedlichen Figürchen aus dem von *Daniel Cook (Danc)* in seinem Blog [Lost Garden](#) unter einer [freien Lizenz \(CC BY 3.0 US\)](#) zu Verfügung gestellten Tileset [Planet Cute](#) fortzufahren.



Abbildung 12.6: Screenshot

Ich möchte aus dem [bisher erlernten](#) ein kleines Spielchen entwickeln: Die süße *Octopussy* ist mit ihrem Raumschiff auf den Weg nach Hause. Dabei wird sie allerdings von den Brüdern *Rocketboy* und dem *Käferjungen* attackiert, die versuchen, sie zu

rammen. Zwar hat Octopussys Raumschiff Schutzschilder, die die Karambolage abfangen, aber es sind nur fünf. Die Gegner haben keine Schutzschilder und werden nach jeder Kollision in das Weltall zurückgeworfen, aus dem sie nach einer gewissen Zeit aber wieder auftauchen. Es ist also so eine Art [Autoscooter](#) im Weltenraum

Um ihre Schutzschilder zu erneuern, muß Octopussy möglichst oft die kleinen Planeten überfliegen, denn diese laden ihre Schutzschilder bis zur maximalen Anzahl von fünf wieder auf.

Im ersten Schritt habe ich nur eine simple Spielmechanik integriert: Octopussy kann ihr Schiff mit den Pfeiltasten nach unten und oben bewegen, mit jedem Druck auf die Tasten erhöht sich ihre Geschwindigkeit, während die gegenüberliegende Taste sie dann jeweils wieder um den gleichen Betrag erniedrigt. Natürlich gibt es eine Maximalgeschwindigkeit und an den Bildrändern oben und unten wird sie rigoros auf Null gesetzt. Die Gegner kommen von links nach rechts in geraden Bahnen und Octopussy muß ihnen ausweichen, aber dennoch versuchen, die kleinen Planeten zu überfliegen.

Um dem lästigen Umstand entgegenzuwirken, daß die Tastaturlbefehle nur greifen, wenn das Graphikfenster von Processing.py den Fokus besitzt, wird das Spiel erst gestartet, wenn zum ersten Mal eine Taste bewegt wurde, bis dahin bleibt es im Startzustand eingefroren.

Quellcode

Jetzt aber der Quellcode, zuerst das Modul `spaceship.py` mit den Klassen `Spaceship` und der daraus abgeleiteten Klasse `Octopussy`:

```
class Spaceship():

    def __init__(self, pic, posX, posY):
        self.pic = pic
        self.x = posX
        self.y = posY
        self.dx = 0
```

```
    self.dy = 0

def loadPic(self):
    self.img = loadImage(self.pic)

def move(self):
    self.x += self.dx
    if self.x >= width + 120:
        self.x = -120
        self.y = random(height-120)
    elif self.x < -120:
        self.x = width + 120
        self.y = random(height-120)

def display(self):
    image(self.img, self.x, self.y)

class Octopussy(Spaceship):

    def move(self):
        self.y += self.dy
        if self.dy >= 5:
            self.dy = 5
        elif self.dy <= -5:
            self.dy = -5
        if self.y < 0:
            self.y = 0
            self.dy = 0
        if self.y > height - 120:
            self.y = height - 120
            self.dy = 0
```

Das dürfte für Euch, die Ihr diese Serie schon verfolgt, nichts Neues mehr sein. Die benötigten Sprites findet Ihr auch [hier](#).

Auch das Hauptprogramm birgt wenig Überraschungen:

```
# Spaceship
from spaceship import Spaceship, Octopussy
```

```
octopussy = Octopussy("octopussy.png", 800, 150)
planet = Spaceship("planet.png", 500, 350)
rocketboy1 = Spaceship("rocketship.png", 300, 300)
rocketboy2 = Spaceship("rocketship.png", -300, 250)
beetle = Spaceship("beetleship.png", 200, 100)

ships = [planet, rocketboy1, rocketboy2, beetle]
fps = 30

def setup():
    size(920, 480)
    frameRate(fps)
    planet.loadPic()
    rocketboy1.loadPic()
    rocketboy2.loadPic()
    beetle.loadPic()
    octopussy.loadPic()
    rocketboy1.dx = rocketboy2.dx = beetle.dx = planet.dx= 0
    octopussy.dx = 0
    octopussy.dy = 0

def draw():
    background(0, 80, 125)
    for i in range(len(ships)):
        ships[i].move()
        ships[i].display()
    octopussy.move()
    octopussy.display()

def keyPressed():
    if keyPressed and key == CODED:
        planet.dx = 2
        rocketboy1.dx = 8
        rocketboy2.dx = 10
        beetle.dx = 6
        if keyCode == UP:
            octopussy.dy -= 1
        elif keyCode == DOWN:
            octopussy.dy += 1
```

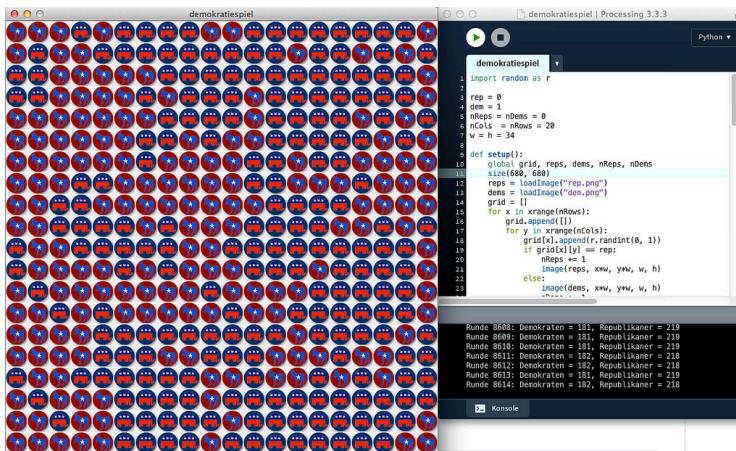
Im zweiten Teil dieses Tutorials möchte ich dann eine Kollisionserkennung implementieren und zeigen, wie man die Schutzschilder ab- und wieder aufbaut und die gegnerischen Raumschiffe ins All schleudert. *Still digging!*

Kapitel 13

Zelluläre Automaten

Das Demokratie-Spiel

Spätestens seit der Wahl von Donald Trump zum Präsidenten der USA fragen sich ja einige, wie es dort mit der Demokratie bestellt sei. Dazu passend ein Spiel, das *Peter Donelly* vom *University College of Swansea* in Wales und *Domenic Welsh* von der *Oxford University* schon in den 80er Jahren des letzten Jahrhunderts untersucht hatten. Populär wurde es dann durch eine Veröffentlichung von *Alexander K. Dewdney* in der *Scientific American* und in der deutschen Schwesterzeitschrift *Spektrum der Wissenschaft*. Er nannte das Spiel »WAEHLER«:



Winkler und *Manfred Eigen* schon 1975 in ihrem Buch »Das Spiel« vorgestellt hatten. Auch wenn die Regeln leicht abgewandelt sind, das Ergebnis ist stets das gleiche. Es überlebt immer nur eine Partei. Das ändert sich übrigens auch nicht, wenn man das Feld mit mehr als zwei Parteien beim Start füllt. Also ist nicht das amerikanische Wahlsystem die alleinige Ursache des Übels.

Der Code

Der Processing.py-Code ist *straight forward*. Lediglich die Behandlung der Randbedingungen ist allgemeiner gehalten, als unbedingt nötig. Damit sind bei Abwandlungen auch andere Nachbarschaften als die Moore-Umgebung möglich. Er folgt einem [Processing- \(Java-\) Code](#), den ich vor Jahren schon einmal programmiert hatte.

```
import random as r

rep = 0
dem = 1
nReps = nDems = 0
nCols = nRows = 20
w = h = 34

def setup():
    global grid, reps, dems, nReps, nDems
    size(680, 680)
    reps = loadImage("rep.png")
    dems = loadImage("dem.png")
    grid = []
    for x in xrange(nRows):
        grid.append([])
        for y in xrange(nCols):
            grid[x].append(r.randint(0, 1))
            if grid[x][y] == rep:
                nReps += 1
                image(reps, x*w, y*w, w, h)
            else:
```

```
        image(dems, x*w, y*w, w, h)
nDems += 1
println("Start: Demokraten = " + str(nDems) + ", Republikaner = " + str(nReps))

def draw():
    global reps, dems, nReps, nDems
    actorX = r.randint(0, nRows - 1)
    actorY = r.randint(0, nCols - 1)
    selection = r.randint(0, 7)
    if selection == 0:
        neighborX = actorX
        neighborY = actorY - 1
    elif selection == 1:
        neighborX = actorX + 1
        neighborY = actorY - 1
    elif selection == 2:
        neighborX = actorX + 1
        neighborY = actorY
    elif selection == 3:
        neighborX = actorX + 1
        neighborY = actorY + 1
    elif selection == 4:
        neighborX = actorX
        neighborY = actorY + 1
    elif selection == 5:
        neighborX = actorX - 1
        neighborY = actorY + 1
    elif selection == 6:
        neighborX = actorX - 1
        neighborY = actorY
    elif selection == 7:
        neighborX = actorX - 1
        neighborY = actorY - 1
    else:
        println("Irgend etwas ist gewaltig schiefgelaufen!")

    # Prüfung der Ränder:
    if neighborX < 0:
        neighborX = nRows + neighborX
    neighborX = neighborX % nRows
```

```

if neighborY < 0:
    neighborY = nCols + neighborY
    neighborY = neighborY % nCols

# Neuzeichnen des Spielfelds:
if grid[neighborX][neighborY] == dem:
    if grid[actorX][actorY] != dem:
        nDems += 1
        nReps -= 1
        grid[actorX][actorY] = dem
        image(dems, actorX*w, actorY*w, w, h)
    else:
        if grid[actorX][actorY] != rep:
            nReps += 1
            nDems -= 1
            grid[actorX][actorY] = rep
            image(rep, actorX*w, actorY*w, w, h)
    println("Runde " + str(frameCount) + ": Demokraten = " + str(nDems))
    println("Republikaner = " + str(nReps))

if nDems == 0:
    println("Die Republikaner haben nach " + str(frameCount))
    noLoop()
if nReps == 0:
    println("Die Demokraten haben nach " + str(frameCount))
    noLoop()

```

Wer das Spiel selber nachprogrammieren möchte, hier gibt es auch noch die beiden Icons für die Republikaner (Elephant) und Demokraten (Esel):



Caveat

Ungeduldige sollten erst einmal mit einem 10x10-Gitter beginnen (**size(340, 340)**). Dann hat man in der Regel spätestens nach 20.000 Runden ein Ergebnis. Oder es kann sehr schnell gehen: Ich hatte auf diesem kleinen Gitter auch schon nach unter

2.000 Runden die absolute Herrschaft einer Partei erreicht. Auf einem 20x20-Gitter wie hier kann es durchaus 200.000 Runden und mehr dauern, bis die Diktatur kommt. Aber auf so einem großen Spielfeld erkennt man natürlich die stabilen »Inseln gleicher Meinung« sehr viel besser.

Es gibt sicher einen Schwellwert, der – wenn unterschritten – kein Zurück zur Macht mehr erlaubt. Aber er ist sehr klein: Ich habe es schon erlebt, daß sich Populationen, die unter die 10-Prozent-Marke gerutscht waren, sich wieder berappelten und im Endeffekt die Macht ergriffen.

Das ist das erste aus einer Reihe von (geplanten) Processing.py-Programmen, die sich mit Simulationen auf einem Gitter (aka »zelluläre Automaten«) beschäftigen.

Literatur

- A.K. Dewdney: *Wie man erschießt. Fünf leichte Stücke für WHILE-Schleifen und Zufallsgenerator, oder: lebensechte Simulationen von Zombies, Wählern und Warteschlangen*, in: Immo Diener (Hg.): *Computer-Kurzweil*, Heidelberg (Spektrum der Wissenschaft, Reihe: Verständliche Forschung) 1988
- Manfred Eigen, Ruthild Winkler: *Das Spiel. Naturgesetze steuern den Zufall*, München (Piper), 1975 (unveränderte Taschenbuchausgabe 1985)

Frösche und Schildkröten oder: Wie entsteht Segregation?

[Mitchel Resnick](#) erzählt uns eine nette Geschichte: In einem Teich lebten Frösche und Schildkröten in trauter Eintracht zusammen. Jeder Frosch lebt auf einer Seerose und hat auf den acht benachbarten anliegenden Seerosen je vier Frösche und je vier Schildkröten als Nachbarn. (Man erkennt leicht, daß es sich um quadratische Seerosen mit einer [Moore-Nachbarschaft](#) handelt.) Doch

eines Tages kommt ein böser Sturm auf und wirbelt alles durcheinander und auch etliche Frösche und Schildkröten kommen (zu gleichen Teilen) dabei um. Als sich der Sturm gelegt hat, versuchen die Tiere sich wieder zu organisieren und es sich auf den Seerosen gemütlich zu machen. Sie sind jedoch nur glücklich, wenn sie mindestens drei Nachbarn haben, die der gleichen Spezies angehören, ansonsten versuchen sie, eine andere, freie Seerose zu besiedeln. Und was passiert dabei? Es entstehen Kolonien, die nur von Schildkröten und andere Kolonien, die nur von Fröschen bevölkert werden. Eine vorbeifliegende Eule wundert sich und fragt einen Frosch, ob sie sich denn nicht mehr lieb haben würden. »Doch, wir haben uns lieb wie eh und je. Nur ... es passiert einfach, daß wir zusammenziehen, unter der einzigen Voraussetzung, daß wir mindestens drei Nachbarn unserer Spezies haben wollen. Und den Schildkröten geht es genauso.«

Schauen wir uns das doch einfach einmal an:



Segregationsspiel (Startzustand)

Resnick hat das natürlich in [StarLogo](#) programmiert, ich habe ein leicht abgewandeltes Processing.py-Programm geschrieben, mit dem man das Verhalten untersuchen kann. Beim Start verteilen wir zufällig die Schildkröten und Frösche zu gleichen Teilen auf einem 40x40-Raster, wobei etwa 30 Prozent leer bleiben, damit sich die Viecher auch bewegen können. In jedem Durchlauf

wird zufällig ein Bewohner ausgewählt und er wird gefragt, ob er glücklich sei. Glücklich ist er nur, wenn er wenigstens drei Nachbarn hat, die der gleichen Spezies angehören. Ist er glücklich, bleibt er da sitzen wo er ist. Ist er unglücklich, sucht er zufällig in der Nachbarschaft in seiner Sprungdistanz (ja, in meiner Geschichte können auch Schildkröten springen) eine Seerose aus. Ist diese Seerose frei, springt er dahin, hoffend, dort glücklich zu werden. Ist das Feld nicht frei, bleibt er hocken und hofft auf eine neue Chance, wieder ausgewürfelt zu werden.



*Segregationsspiel (nachdem es ungefähr eine halbe Stunde gelau-
fen ist)*

Läßt man diese Simulation nun laufen, stellt man fest, daß sich tatsächlich Cluster gleicher Spezies bilden. Die kleinste stabile Einheit ist ein Quadrat mit der Kantenlänge zwei – hier hat jeder mindestens drei Nachbarn. Außerdem ist eine Flucht von den Rändern weg zu beobachten. Hier habe ich einfach angenommen, daß das Wasser so flach ist, daß dort keine Seerosen gedeihen – die Ränder werden also nicht periodisch fortgesetzt. Und so hat man an den Rändern natürlich weniger Nachbarn und die Chance, glücklich zu sein, ist geringer.

Außerdem kann es vorkommen, daß einzelne Tiere regelrecht von der benachbarten Spezies eingekesselt werden und sie nicht mehr fliehen können. Die Armen sind zu einem ewigen Unglücklichsein verdammt. Hier hilft nur, die maximale Sprungdistanz zu erhöhen.

Überhaupt: Obige Screenshots stammen von einem Sketch mit einer Sprungdistanz von zwei, nachdem der Sektch etwa eine halbe Stunde gelaufen war. Es passiert nicht mehr viel. Die meisten zufällig ausgewählten in einer Runde sind glücklich und verharren auf ihren Platz. Nur noch wenige Exemplare einer Gattung irren umher und suchen Anschluß. Andere sind enteder vom Rand des Teiches oder von den Spezies der anderen Art eingekesselt und zu ewigem unglücklich sein verdammt.

Erhöht man aber den Wert der Sprungdistanz (zum Beispiel auf fünf), dann geht nicht nur die Clusterbildung schneller vorstatten, sondern auch die Zahl der eingekesselten Tiere geht massiv zurück.

Was uns dieses einfache Programm über die tatsächliche Segregation erzählt, überlasse ich aber der Phantasie meiner Leserinnen und Leser.

Der Quellcode

```
import random as r

empty = 0
frog = 1
turtle = 2

nRows = 40
nCols = 40
w = h = 16

jumpsize = 2

def setup():
    global grid, frogs, turts
    size(640, 640)
    frogs = loadImage("frog.png")
    turts = loadImage("turtle.png")
    grid = []
    for x in xrange(nRows):
        grid.append([])
```

```

    for y in xrange(nCols):
        grid[x].append(r.randint(0, 2))
    # Für den Screenshot des Anfangszustandes
    # noLoop()

def draw():
    global grid, frogs, turts
    noStroke()
    background(0, 80, 125)

    for x in xrange(nRows):
        for y in xrange(nCols):
            if grid[x][y] == empty:
                fill(0, 80, 125)
                rect(x*w, y*h, w, h)
            elif grid[x][y] == frog:
                image(frogs, x*w, y*h, w, h)
            elif grid[x][y] == turtle:
                image(turts, x*w, y*h, w, h)
            else:
                println("Etwas ist falsch im Staate Lilypond!")

actorX = r.randint(0, nRows-1)
actorY = r.randint(0, nCols-1)
# Lebt hier jemand?
if grid[actorX][actorY] > 0:
    # Und ist er glücklich?
    happy = isHappy(grid[actorX][actorY], actorX, actorY)
    # Wenn nicht, dann möglichst weg von hier
    if not(happy):
        newX = r.randint(-jumpsize, jumpsize)
        newY = r.randint(-jumpsize, jumpsize)
        newX += actorX
        newY += actorY
        # Liegt mein Ziel noch im Teich?
        if ((newX >= 0) and (newX < nRows) and (newY >= 0) and (newY < nCols)):
            if grid[newX][newY] == empty:
                grid[newX][newY] = grid[actorX][actorY]
                grid[actorX][actorY] = empty

```

```
def isHappy(animal, x, y):
    happy = 0
    if (y-1 > 0) and (grid[x][y-1] == animal):
        happy += 1
    if (x+1 < nRows) and (y-1 > 0) and (grid[x+1][y-1] == animal):
        happy += 1
    if (x+1 < nRows) and (grid[x+1][y] == animal):
        happy += 1
    if (x+1 < nRows) and (y+1 < nCols) and (grid[x+1][y+1] == animal):
        happy += 1
    if (y+1 < nCols) and (grid[x][y+1] == animal):
        happy += 1
    if (x-1 > 0) and (y+1 < nCols) and (grid[x-1][y+1] == animal):
        happy += 1
    if (y+1 < nCols) and (grid[x][y+1] == animal):
        happy += 1
    if (x-1 > 0) and (grid[x-1][y] == animal):
        happy += 1
    if happy >= 3:
        return True
    else:
        return False
```

Die Bilder von Frosch und Schildkröte habe ich den [Twitter-Emojis](#) entnommen und hier sind sie noch einmal, damit Ihr das Spiel nachprogrammieren könnt:



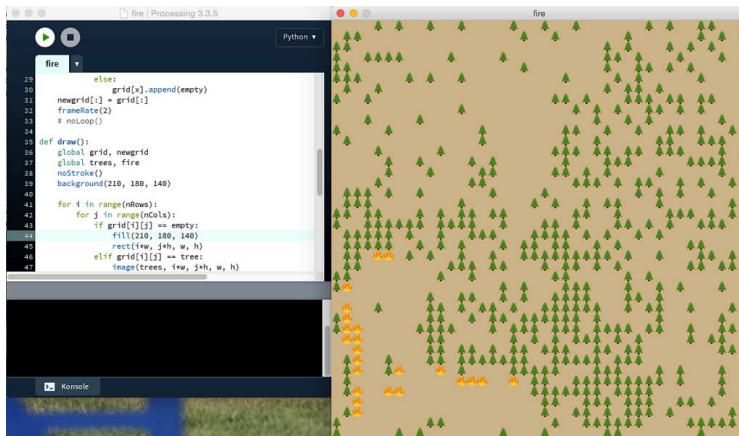
In Processing (Java) hatte ich vor Jahren diese Simulation auch schon einmal [programmiert](#).

Literatur

- Mitchel Resnick: *Turtles, Termites, and Traffic Jams: Explorations in Massively Parallel Microworlds*, Cambridge, MA (MIT Press) 1997, p. 81 - 88

Der Waldbrand-Simulator

Bei dem [Demokratie-Spiel](#) und bei den [Experimenten mit den Fröschen und Schildkröten](#) änderte sich pro Durchlauf jeweils nur das Verhalten einer Zelle in Abhängigkeit von ihren direkten Nachbarn und war für den weiteren Verlauf der Simulation verantwortlich. Bei den meisten Simulationen mit zellulären Automaten jedoch wird der neue Wert *aller* Zellen in Abhängigkeit von den Nachbarn betrachtet und neu berechnet. Dafür muß man dann zwei Arrays anlegen, eines, das die aktuellen und eines das die zukünftigen Werte beinhaltet. Ich möchte das mal am Beispiel einer beliebten Simulation zeigen, der Simulation eines Waldbrandes mit einem zellulären Automaten.



Die Regeln dieser Simulation folgen der Beschreibung, die *Daniel Scholz* in seinem Buch »Pixelspiele«¹ gegeben hat:

Kein Spiel ohne Regeln

Für alle Zellen x_{ij} gelten folgende Regeln:

1. Befindet sich x_{ij} im Zustand leer (*empty*), dann wächst auf

¹Das ist keine Einschränkung, denn der Fensterrand ist ja im Grunde auch nichts anderes als eine undurchdringliche Mauer und im Zweifelsfall macht man das Spielfeld einfach um die Mauerdicke größer.

x_{ij} mit einer Wahrscheinlichkeit a ein Baum, so daß der Zustand von x_{ij} im nächsten Schritt *tree* ist.

2. Befindet sich x_{ij} im Zustand Baum (*tree*), und mindestens eine Zelle in der Nachbarschaft ist im Zustand Feuer (*burning*), dann brennt auch x_{ij} , so daß der Zustand von x_{ij} im nächsten Schritt auch *burning* ist.
3. Automatischer Ablauf: Falls x_{ij} ein Baum ist und keiner seiner Nachbarn brennt, dann wird x_{ij} mit einer Wahrscheinlichkeit von g von einem Blitz getroffen, so daß x_{ij} im nächsten Schritt ebenfalls *burning* ist.
4. Interaktive Version: Wird die Zelle x_{ij} mit dem Mauszeiger angeklickt, dann wird x_{ij} von einem Blitz getroffen und der Zustand von x_{ij} ist im nächsten Schritt *burning*.
5. Befindet sich die Zelle x_{ij} im Zustand *burning*, dann erlischt das Feuer und der Zustand von x_{ij} ist im nächsten Schritt *empty*.
6. Trifft keine der obigen Regeln zu, dann verändert sich der Zustand der Zelle x_{ij} im nächsten Schritt nicht.

Als Nachbarschaft wird die **Von-Neumann-Nachbarschaft** angenommen, Nachbarn sind also nur die direkten Zellen oben und unten sowie rechts und links, das heißt, jede Zelle hat genau vier Nachbarn.

Als Randbedingung wurde ein geschlossener Rand gewählt, daß heißt die Zellen am Rande des Feldes werden in der Simulation gar nicht berücksichtigt, sie bleiben auf ewig, wie sie sind. Daher habe ich bei der Initialisierung des Feldes mit

```
if (x > 0) and (y > 0) and (x < nRows-1) and (y < nC
    grid[x].append(tree)
else:
    grid[x].append(empty)
```

dafür gesorgt, daß die Randfelder immer leer sind. Das hat auf den Simulationsverlauf keinen Einfluß, aber es störte mich besonders bei der Darstellung mit den Emojis, daß auf den Rändern anfangs immer ein paar Bäume dumm herum standen, wie noch im obigen Screenshot dokumentiert, der während einer frühen Phase der Realisierung dieser Simulation entstand.

Die Realisierung

Oben habe ich es schon angesprochen, für die erste Version der Waldbrandsimulation habe ich wieder [Twitters Twemojis](#) geplündert und hier sind die Bildchen vom Baum und vom Feuer, damit Ihr die Simulation nachprogrammieren könnt:



Wenn Ihr die Bilder herunterladet, beachtet bitte, daß ich, weil es schon ein anderes Baumbildchen gab, dieses hier `tree2.png` nennen mußte. Im Sketch heißt es aber `tree.png`, Ihr müßt also entweder die Bezeichnung im Sketch ändern (nicht gut) oder einfach den Namen des Bildchens ändern (besser). Es sind winzige, 16x16 Pixel große Bildchen und das Spielfeld habe ich diesen Ausmaßen angepaßt:

```
def setup():
    global trees, fire
    size(640, 640)
    background(210, 180, 140)
    trees = loadImage("tree.png")
    fire = loadImage("fire.png")
    for x in range(nRows):
        grid.append([])
        newgrid.append([])
        for y in range(nCols):
            # Randbedingungen
            if (x > 0) and (y > 0) and (x < nRows-1) and (y < nCols-1):
                grid[x].append(tree)
            else:
                grid[x].append(empty)
    newgrid[:] = grid[:]
    frameRate(2)
```

Eine `frameRate()` von 2 ist durchaus ausreichend, sonst läuft die Simulation so schnell, daß Ihr gar nichts nachvollziehen könnt.

Aber ganz zu Beginn habe ich `randint` für die Zufallszahlen importiert, ein paar Konstanten gesetzt und die beiden Arrays initialisiert:

```
from random import randint

empty = 0
tree = 1
burning = 20

a = 40
g = 1

nRows = 40
nCols = 40
w = h = 16

grid = []
newgrid = []
```

Die `draw()`-Funktion ist in allen Simulationen gleich,

```
draw():
    global grid, newgrid
    global trees, fire
    noStroke()
    background(210, 180, 140)

    for i in range(nRows):
        for j in range(nCols):
            if grid[i][j] == empty:
                fill(210, 180, 140)
                rect(i*w, j*h, w, h)
            elif grid[i][j] == tree:
                image(trees, i*w, j*h, w, h)
            elif grid[i][j] == burning:
                image(fire, i*w, j*h, w, h)
    calcNext()
```

der Unterschied für den per Zufall generierten Blitzschlag, respektive den durch Nutzereingabe verursachten Blitz, liegt in der Funktion `calcNext()`. Hier erst einmal die nicht interaktive Version:

```

calcNext():
    global grid, newgrid
    newgrid[:] = grid[:]
    # Next Generation
    for i in range(1, nRows-1):
        for j in range(1, nCols-1):
            if grid[i][j] == burning:
                newgrid[i][j] = empty
                # Brennt ein Nachbar?
                if grid[i-1][j] == tree:
                    newgrid[i-1][j] = burning
                if grid[i][j-1] == tree:
                    newgrid[i][j-1] = burning
                if grid[i][j+1] == tree:
                    newgrid[i][j+1] = burning
                if grid[i+1][j] == tree:
                    newgrid[i+1][j] = burning
            elif grid[i][j] == empty:
                if randint(0, 10000) < a:
                    newgrid[i][j] = tree
            if grid[i][j] == tree:
                # Schlägt ein Blitz ein?
                if (random(10000) < g):
                    newgrid[i][j] = burning
    grid[:] = newgrid[:]

```

Mit `for i in range(1, nRows-1)` und `for j in range(1, nCols-1)` habe ich die Randfelder von der Abfrage ausgeschlossen und somit die Randbedingung »geschlossener Rand« erfüllt.

In den vorletzten zwei Zeilen wird die Wahrscheinlichkeit abgefragt, ob ein Blitz einschlägt und wenn diese (geringe) Wahrscheinlichkeit zutrifft, dann wird das Feld x_{ij} für den nächsten Zustand auf brennend (*burning*) gesetzt. In der interaktiven Variante entfallen diese beiden Zeilen, dafür kommt noch die Funktion `mousePressed()` hinzu,

```

def mousePressed():
    newgrid[mouseX/16][mouseY/16] = burning

```

die einfach für die Zelle, in der die Maus klickt, den neuen Zustand auf *burning* setzt.

Der Quellcode (1)

Bevor ich weitermache, erst einmal den Quellcode des vollständigen Programmes in der interaktiven Version:

```
from random import randint

empty = 0
tree = 1
burning = 20

a = 40
g = 1

nRows = 40
nCols = 40
w = h = 16

grid = []
newgrid = []

def setup():
    global trees, fire
    size(640, 640)
    background(210, 180, 140)
    trees = loadImage("tree.png")
    fire = loadImage("fire.png")
    for x in range(nRows):
        grid.append([])
        newgrid.append([])
        for y in range(nCols):
            # Randbedingungen
            if (x > 0) and (y > 0) and (x < nRows-1) and (y < nC
```

```
newgrid[:] = grid[:]
frameRate(2)
# noLoop()

def draw():
    global grid, newgrid
    global trees, fire
    noStroke()
    background(210, 180, 140)

    for i in range(nRows):
        for j in range(nCols):
            if grid[i][j] == empty:
                fill(210, 180, 140)
                rect(i*w, j*h, w, h)
            elif grid[i][j] == tree:
                image(trees, i*w, j*h, w, h)
            elif grid[i][j] == burning:
                image(fire, i*w, j*h, w, h)
    calcNext()

def calcNext():
    global grid, newgrid
    newgrid[:] = grid[:]
    # Next Generation
    for i in range(1, nRows-1):
        for j in range(1, nCols-1):
            if grid[i][j] == burning:
                newgrid[i][j] = empty
                # Brennt ein Nachbar?
                if grid[i-1][j] == tree:
                    newgrid[i-1][j] = burning
                if grid[i][j-1] == tree:
                    newgrid[i][j-1] = burning
                if grid[i][j+1] == tree:
                    newgrid[i][j+1] = burning
                if grid[i+1][j] == tree:
                    newgrid[i+1][j] = burning
            elif grid[i][j] == empty:
                if randint(0, 10000) < a:
```

```
newgrid[i][j] = tree  
grid[:] = newgrid[:]  
  
def mousePressed():  
    newgrid[mouseX/16][mouseY/16] = burning
```

Für die automatisch ablaufende Fassung müßt Ihr einfach nur die oben erwähnten zwei Zeilen vor

```
grid[:] = newgrid[:]
```

in die Funktion `calcNext()` einfügen und die Funktion `mousePressed()` löschen.

Ein größerer Wald

Die obige Simulation läuft schon sehr zufriedenstellend ab, aber um Muster zu erkennen, muß man den »Wald« doch weiter vergrößern. Ich habe in einer neuen Version dieser Simulation das Spielfeld daher auf 280x160 Zellen erweitert. Damit mein Monitor nicht gesprengt wird, sind diese Zellen jetzt nur noch 2x2 Pixel groß, was zu einer Fenstergröße von 560x320 führt.

Die Zellen werden jetzt nicht mehr durch Emojis dargestellt, sondern durch kleine Rechtecke in verschiedenen Farben. Da ich leicht farbenblind bin, habe ich mir die Farben aus einer Tabelle mit Farbnamen zusammengesucht, ein leeres Feld sollte daher ockerfarben dargestellt werden, ein Feld mit einem Baum dunkelgrün und ein brennendes Feld in einem leuchtenden rot. Diese Angaben sind ohne Gewähr, aber Ihr könnt die Farben ja im Zweifelsfalle selber Euren Wünschen anpassen.

Beispielsimulation

Generation 50

Generation 100

Generation 150

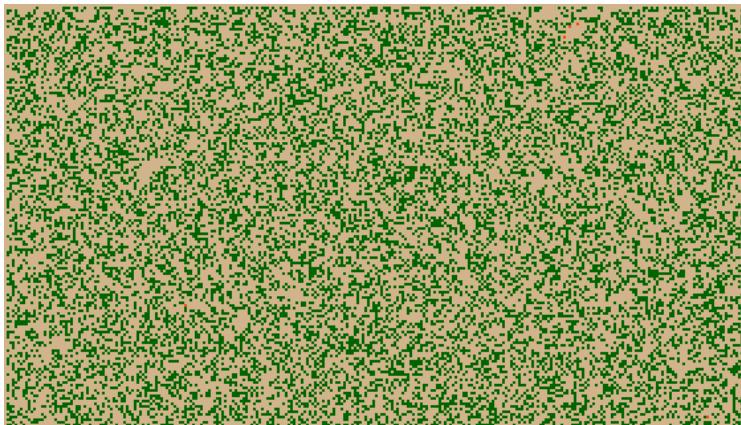


Abbildung 13.1: Generation 50

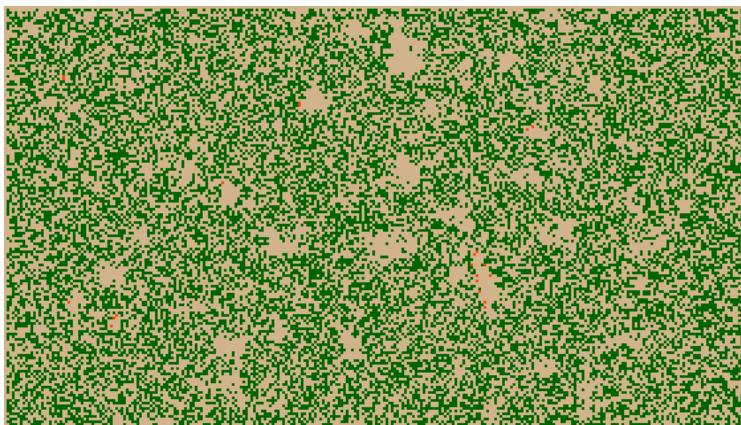


Abbildung 13.2: Generation 100

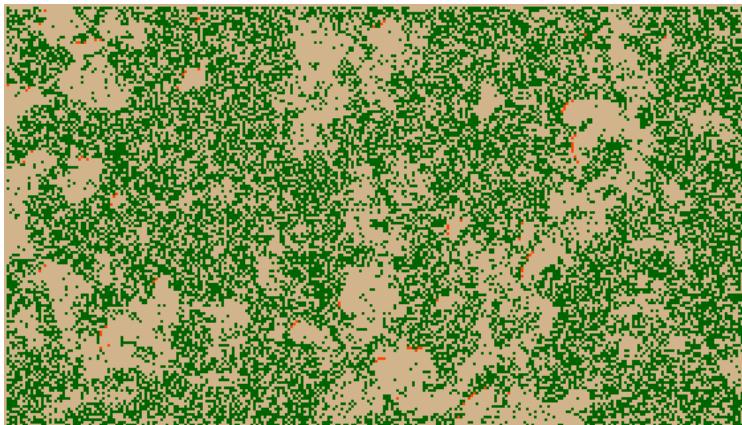


Abbildung 13.3: Generation 150

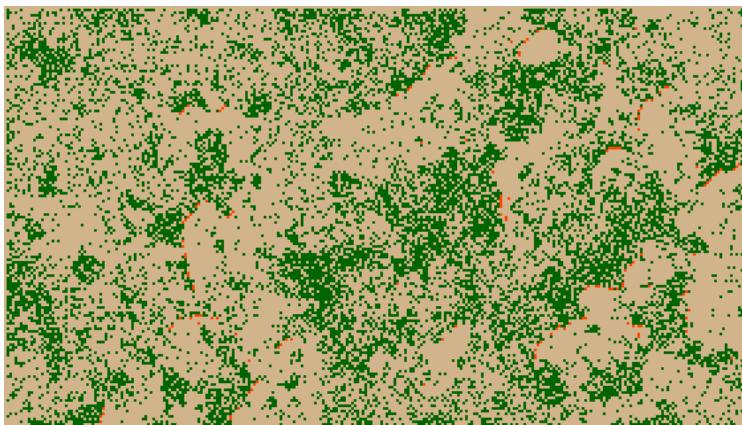


Abbildung 13.4: Generation 200

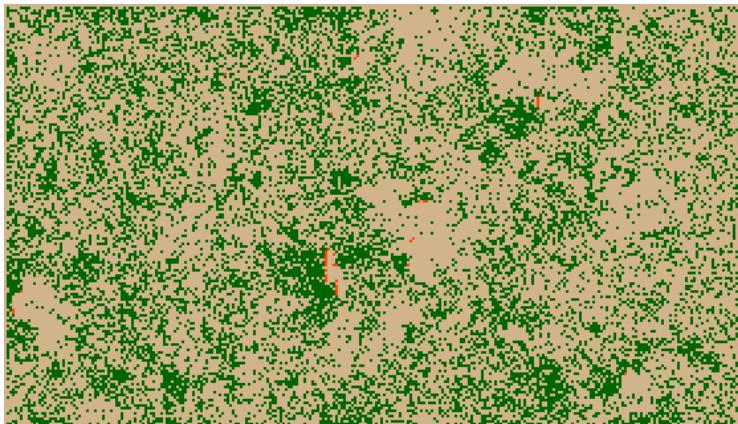


Abbildung 13.5: Generation 250

Generation 200

Generation 250

Generation 300

Wenn Ihr diese Simulation über einen gewissen Zeitraum laufen läßt, dann erkennt Ihr, daß sich im Laufe der Zeit ein gewisses, wenn auch schwankendes Gleichgewicht zwischen Wald und freier Fläche einstellt. Dieses Gleichgewicht soll sich sogar relativ unabhängig von den gewählten Parametern einstellen (das ist allerdings in der Literatur umstritten). In den obigen Screenshots, die ich mit

```
if (frameCount % 50) == 0:  
    print(frameCount)  
    saveFrame("frames/fire-gen-####.png")
```

erstellt habe, könnt Ihr sehen, wie sich die Simulation nach je 50 Schritten verändert hat und nach einer ruhigen Anfangsphase des Wachstums scheint der Gleichgewichtszustand tatsächlich erreicht. Generation 150 und Generation 300 sind sich sehr ähnlich, dazwischen brennt der Wald erst heftiger (Generation 200) und erlebt dann wieder eine Phase des Wachstums (Generation 250).

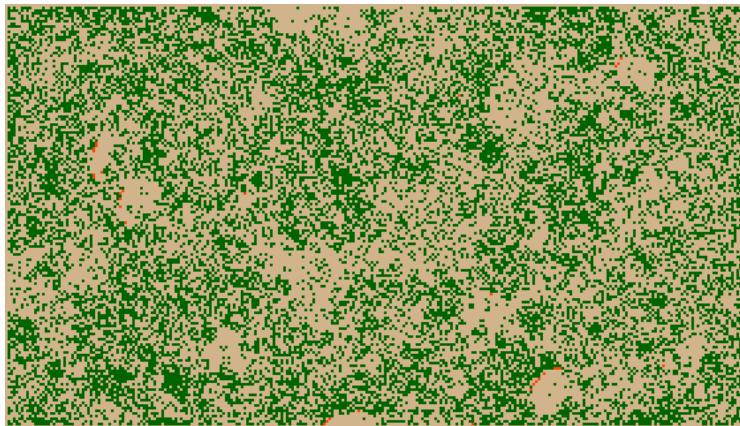


Abbildung 13.6: Generation 300

Der Quellcode (2)

Probiert es – auch mit anderen Paramtern für `a` und `g` einfach mal aus. Darum hier der komplette Quellcode dieser Version:

```
from random import randint

empty = 0
tree = 1
burning = 20

a = 40
g = 1

nRows = 280
nCols = 160
w = h = 2

grid = []
newgrid = []

def setup():
    global trees, fire
```

```
size(560, 320)
background(210, 180, 140)
for x in range(nRows):
    grid.append([])
    newgrid.append([])
    for y in range(nCols):
        # Randbedingungen
        if (x > 0) and (y > 0) and (x < nRows-1) and (y < nCols-1):
            grid[x].append(tree)
        else:
            grid[x].append(empty)
newgrid[:] = grid[:]
frameRate(10)
# noLoop()

def draw():
    global grid, newgrid
    global trees, fire
    noStroke()
    background(210, 180, 140)
    for i in range(nRows):
        for j in range(nCols):
            if grid[i][j] == empty:
                fill(210, 180, 140)
                rect(i*w, j*h, w, h)
            elif grid[i][j] == tree:
                fill(0, 100, 0)
                rect(i*w, j*h, w, h)
            elif grid[i][j] == burning:
                fill(255, 69, 0)
                rect(i*w, j*h, w, h)
    if (frameCount % 50) == 0:
        print(frameCount)
        saveFrame("frames/fire-gen-####.png")
    calcNext()

def calcNext():
    global grid, newgrid
    newgrid[:] = grid[:]
    # Next Generation
```

```
for i in range(1, nRows-1):
    for j in range(1, nCols-1):
        if grid[i][j] == burning:
            newgrid[i][j] = empty
            # Brennt ein Nachbar?
            if grid[i-1][j] == tree:
                newgrid[i-1][j] = burning
            if grid[i][j-1] == tree:
                newgrid[i][j-1] = burning
            if grid[i][j+1] == tree:
                newgrid[i][j+1] = burning
            if grid[i+1][j] == tree:
                newgrid[i+1][j] = burning
        elif grid[i][j] == empty:
            if randint(0, 10000) < a:
                newgrid[i][j] = tree
        if grid[i][j] == tree:
            # Schlägt ein Blitz ein?
            if (random(10000) < 1):
                newgrid[i][j] = burning
grid[:] = newgrid[:]
```

Er unterscheidet sich nicht grundlegend von den vorherigen Versionen, daher solltet Ihr ihn durchaus nachvollziehen können.

Caveat

Ich glaube nicht wirklich, daß diese Simualtion ein realistisches Bild von Waldbränden liefert, dazu fehlen zum Beispiel Parameter für die Windrichtung, manche Bäume brennen leichter als andere und vieles mehr. Aber es ist eine nette Spielerei und Ihr seid durchaus aufgefordert, die fehlenden Parameter einzufügen und damit zu spielen. Der Aufsatz »[Simulating the World in Emojis](#)« den *Nicky Case* im Januar 2016 veröffentlichte, gibt dafür – aber auch für weitere Simualtionen – nette Anregungen.

Kapitel 14

3D mit Processing.py

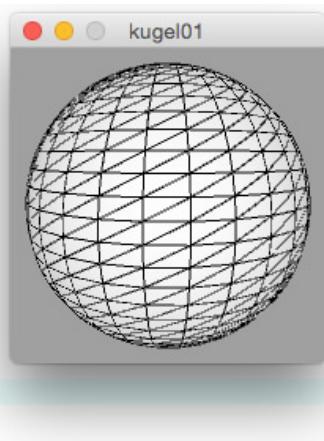
Kugeln und Kisten

Processing und damit auch Processing.py besitzt die Möglichkeit, sehr einfach 3D-Objekte zu erzeugen, allerdings sind als Primitive nur eine Kugel und eine Kiste (`sphere()` und `box()`) vorgesehen. Als Erstes möchte ich zeigen, wie man schnell eine sich drehende Kugel damit zaubert:

```
a = 0

def setup():
    size(200, 200, P3D)

def draw():
    global a
    background(160)
    lights()
    translate(width/2, height/2, 0)
    sphereDetail(30)
    with pushMatrix():
        rotateX(radians(-10))
        rotateY(a)
        a += 0.01
        sphere(80)
```



Um mit Processing in drei Dimensionen zu arbeiten, muß man

das bei der Initialisierung des Fensters dem Programm mitteilen:

```
def setup():
    size(200, 200, P3D)
```

Eigentlich teilt man Processing auch mit, wenn man in zwei Dimensionen hantieren will, nur ist P2D einfach der Default und kann entfallen.

Dann besitzt Processing eine einfache Methode, die 3D-Landschaft auszuleuchten, nämlich `lights()`. Und ähnlich wie den Kreisen und Ellipsen ist auch bei einer Kugel per Default, der Ursprung der Koordinaten die Mitte. Daher habe ich mit

```
translate(width/2, height/2, 0)
```

die x- und y-Achsen des Koordinatensystems in die Mitte des Fensters gelegt. Mit `sphereDetail(n)` wird die Anzahl der Dreiecke bestimmt, aus denen die Kugel zusammengesetzt werden soll. Je mehr Dreiecke, desto »runder« die Kugel, aber auch um so größer die Rechenzeit. Bei diesem einfachen Programm spielt das noch keine Rolle, die Zahl 30 ist eher dem Umstand geschuldet, daß die Kugel vor lauter Dreiecken sonst nicht mehr zu erkennen ist.

Und dann kommt wieder das geniale `with`-Statement zu Einsatz:

```
with pushMatrix():
    rotateX(radians(-10))
    rotateY(a)
    a += 0.01
    sphere(80)
```

Mit `rotateX()` wird die Kugel ein wenig geneigt und mit `rotateY()` dreht sie sich um die eigene Achse. Einfacher kann man eine sich bewegende Kugel in 3D eigentlich gar nicht programmieren.

Der Quellcode

Hier noch einmal der komplette Quellcode des Sketches zum Nachbauen:

```
a = 0

def setup():
    size(200, 200, P3D)

def draw():
    global a
    background(160)
    lights()
    translate(width/2, height/2, 0)
    sphereDetail(30)
    with pushMatrix():
        rotateX(radians(-10))
        rotateY(a)
        a += 0.01
        sphere(80)
```

Caveat

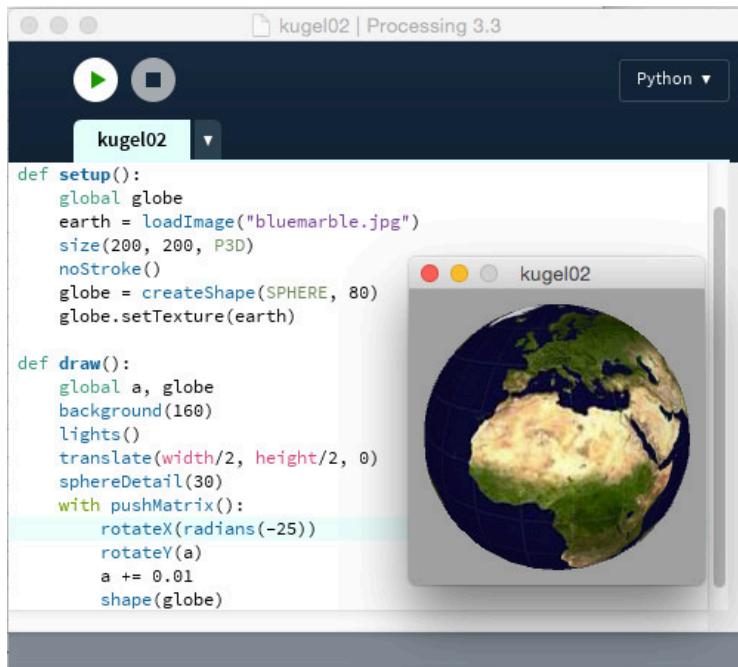
Jetzt kommt aber das Salz in der Suppe: `box()` wie auch `sphere()` lassen sich nicht mit Texturen versehen. Dafür muß man sich mit Vertizes seine eigenen 3D-Objekte bauen.

Und es geht doch: Kugeln und Texturen

Ich hatte doch [hier behauptet](#), daß man die einfachen 3D-Primitive `sphere()` und `box()` nicht mit Texturen versehen kann und man darum dann eigene 3D-Objekte bauen müsse. Nun gibt es jedoch einen einfachen Weg, diese Beschränkung zu umgehen. Denn der Befehl `createShape()` erzeugt nicht nur ein Objekt, sondern er kann auch Parameter übernehmen. Und so kann man mit

```
earth = loadImage("bluemarble.jpg")
noStroke()
globe = createShape(SPHERE, 80)
globe.setTexture(earth)
```

auf einfachste Weise einen *Shape* erzeugen, den man mit Texturen versehen kann.



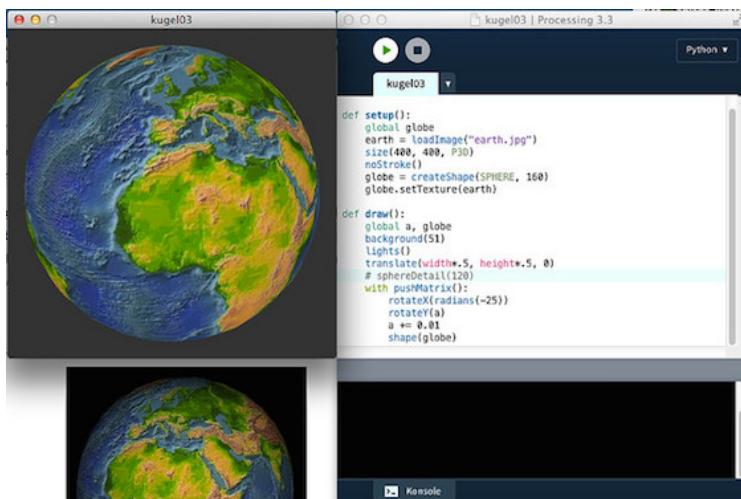
Hier der vollständige Sketch, der uns diese Erdkugel erzeugt:

```
a = 0

def setup():
    global globe
    earth = loadImage("bluemarble.jpg")
    size(200, 200, P3D)
    noStroke()
    globe = createShape(SPHERE, 80)
    globe.setTexture(earth)
```

```
def draw():
    global a, globe
    background(160)
    lights()
    translate(width/2, height/2, 0)
    sphereDetail(30)
    with pushMatrix():
        rotateX(radians(-25))
        rotateY(a)
        a += 0.01
        shape(globe)
```

Und noch eine Textur



Und hier noch einmal die Erdkugel mit einer anderen Textur, die ich [hier gefunden](#) habe. Schaut man genau hin, entdeckt man, daß die Erde an der Datumsgrenze einen Riß aufweist – ein Phänomen, daß ich hin und wieder schon beobachtet, für daß ich allerdings bis jetzt noch keine Erklärung habe.

Der Quellcode wurde nur geringfügig geändert, aber der Vollständigkeit halber hier noch einmal:

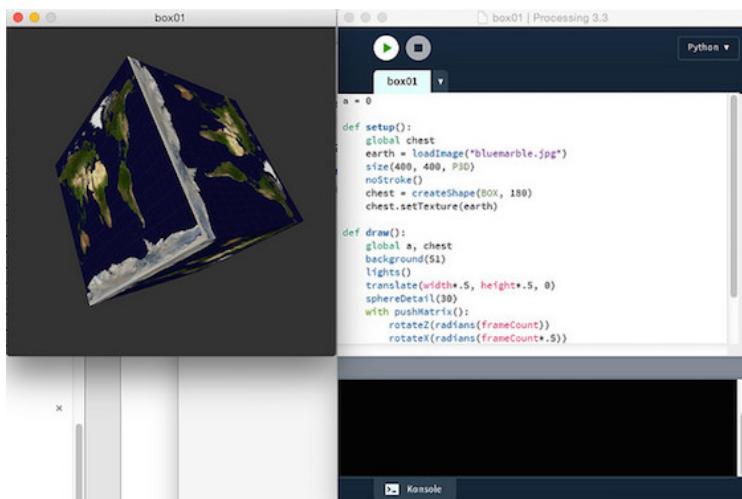
```
a = 0

def setup():
    global globe
    earth = loadImage("earth.jpg")
    size(400, 400, P3D)
    noStroke()
    globe = createShape(SPHERE, 160)
    globe.setTexture(earth)

def draw():
    global a, globe
    background(51)
    lights()
    translate(width*.5, height*.5, 0)
    # sphereDetail(120)
    with pushMatrix():
        rotateX(radians(-25))
        rotateY(a)
        a += 0.01
        shape(globe)
```

Die Erde ist eine Kiste

Natürlich kann man das, was ich [hier](#) mit einer Kugel angestellt habe, auch mit einer Kiste (in Processing BOX genannt) anstellen. Der einzige Unterschied ist, daß die Textur jeweils komplett auf alle sechs Seiten der Box abgebildet wird.



Aber dann hat man den Beweis: Die Erde ist eine Kiste!

Quellcode

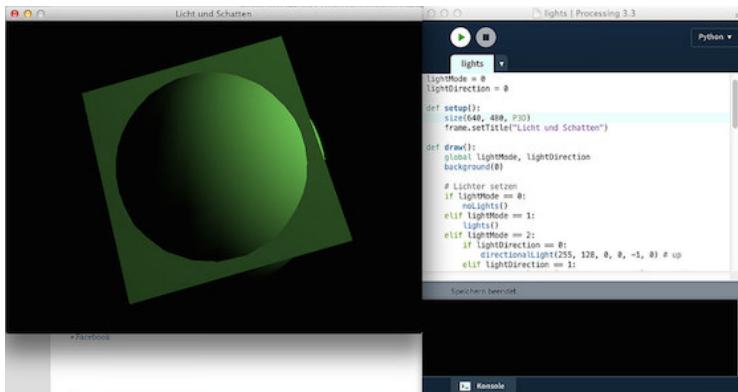
```
a = 0

def setup():
    global chest
    earth = loadImage("bluemarble.jpg")
    size(400, 400, P3D)
    noStroke()
    chest = createShape(BOX, 180)
    chest.setTexture(earth)

def draw():
    global a, chest
    background(51)
    lights()
    translate(width*.5, height*.5, 0)
    sphereDetail(30)
    with pushMatrix():
        rotateZ(radians(frameCount))
        rotateX(radians(frameCount*.5))
```

```
rotateY(radians(a))
a += 0.01
shape(chest)
```

Licht und Schatten



Bei dreidimensionalen Applikationen gilt für jede Software genau wie im wirklichen Leben: »Ohne Licht sehen Sie nichts!« Das ist bei den spezialisierten Programmen wie [Blender](#) oder [PoVRay](#) genau so, wie auch in Processing.py. Daher möchte ich in folgendem Skript zeigen, welche Möglichkeiten der Beleuchtung es in Processing gibt und welche Auswirkung sie auf die Szene haben.

Dazu habe ich eine Kugel verschachtelt in einer Box erzeugt und sie in ein 3D-Fenster gesetzt. Sie ist im Grunde farblos, nur für eine Belichtung (`lights()`) habe ich der Kugel eine hell- und der Box eine dunkelblaue Farbe verpaßt.

Bevor ich die Kugel und die Box zeichnen lasse, überprüfe ich, welche Beleuchtungsfunktion aktuell angewählt ist. Processing kennt sechs Beleuchtungsfunktionen. Diese sind

1. `noLights()`: Diese schaltet alle Beleuchtung aus und die dreidimensionalen Objekte wirken zweidimensional. Diese Funktion kann benutzt werden, um dreidimensionale Objekte mit zweidimensionalen Zeichnungen zu kombinieren.

2. `lights()`: Das ist die einfachste Beleuchtungsfunktion, die die Umgebung in ein neutrales, ambientes Licht taucht. Sie kann immer erst einmal für den Test der dreidimensionalen Objekte eingesetzt verwendet werden, bevor man sich an spektakulärere Beleuchtungsmodelle wagt.
3. `directionalLight(v1, v2, v3, nx, ny, nz)`: Diese Beleuchtungsfunktion besitzt sechs Parameter. Die ersten drei geben die Farbwerte an (es können je nach gewähltem Farbmodus entweder RGB- oder HSB-Werte sein). Die letzten drei Werte geben jeweils die Richtung des Lichtes aus der x-, y-, und/oder z-Richtung an. Direkte Richtungen sind 0, -1, 0 nach oben, 0, 1, 0 nach unten, 1, 0, 0 nach rechts und -1, 0, 0 nach links. Analog sind die Werte für »Licht von vorne« und »Licht von hinten« einzustellen und durch Kombinationen der drei Parameter bekommt man auch Licht aus beliebigen Richtungen.
4. `ambientLight(v1, v2, v3)` taucht die Umgebung in ein ambientes Licht in der mit `v1, v2, v3` spezifizierten Farbe (RGB oder HSB). Ambientes Licht wird meist mit anderen Lichtquellen kombiniert, um zum Beispiel die Schlagschatten der anderen, gerichteten Lichtquellen aufzuhellen.
5. `pointLight(v1, v2, v3, x, y, z)` setzt ein punktförmiges Licht in den Farben `v1, v2, v3` aus der Position `x, y, z`.
6. `spotLight(v1, v2, v3, x, y, z, nx, ny, nz, angle, concentration)` ergibt ein kegelförmiges Licht in den Variablen `v1, v2, v3` von der Quelle `x, y, z` in die Richtung `nx, ny, nz` mit dem Winkel `angle` und der Intensität `concentration`.

Alle Beleuchtungsfunktionen müssen innerhalb der `draw()`-Funktion aufgerufen werden. Werden sie stattdessen in der `setup()`-Funktion aufgerufen, sind sie nur beim ersten Durchlauf wirksam. Daher habe ich das auch im Sketch so gehalten, wobei je nach gewähltem `lightMode` die Beleuchtung gesetzt wird.

Licht aus – Spot an!

Die Beleuchtung kann man während der Sketch läuft mit der Tastatur auswählen. Die Tasten sind sprechend gewählt:

```
def keyPressed():
    global lightMode, lightDirection
    if key == "n":
        lightMode = 0          # no lights
    elif key == "l":
        lightMode = 1          # lights
    elif key == "d":
        lightMode = 2          # directional light
    elif key == "a":
        lightMode = 3          # ambient light
    elif key == "p":
        lightMode = 4          # point light
    elif key == "s":
        lightMode = 5          # spot light
```

!!! warning “Warnung” Bevor man die Tasten drückt, sollte man darauf achten, daß das Graphikfenster von Processing.py im Vordergrund ist, also den Fokus besitzt. Denn sonst tippt man versehentlich gnadenlos Buchstaben in sein Skript und wundert sich, warum es anschließend nicht mehr läuft. Ich verstehe nicht ganz, warum im Python-Mode das Ausgabefenster beim Start des Programmes nicht automatisch den Fokus bekommt, wie das im Java-Mode von Processing der Fall ist?

Ist das direktionale Licht ausgewählt kann man zusätzlich noch mit den Pfeiltasten die Richtung des Lichthes bestimmen.

Quellcode

Wenn man bedenkt, daß in diesem Programm doch einiges passt, ist der Quellcode immer noch sehr kurz. Das Nachvollziehen sollte auch keine besondere Mühe machen, schließlich wird Python ja oft und zu Recht als lauffähiger Pseudocode bezeichnet.

```
lightMode = 0
lightDirection = 0

def setup():
    size(640, 480, P3D)
    frame.setTitle("Licht und Schatten")

def draw():
    global lightMode, lightDirection
    background(0)

    # Lichter setzen
    if lightMode == 0:
        noLights()
    elif lightMode == 1:
        lights()
    elif lightMode == 2:
        if lightDirection == 0:
            directionalLight(255, 128, 0, 0, -1, 0) # up
        elif lightDirection == 1:
            directionalLight(0, 255, 0, 1, 0, 0)      # right
        elif lightDirection == 2:
            directionalLight(255, 0, 255, 0, 1, 0)   # down
        elif lightDirection == 3:
            directionalLight(0, 255, 255, -1, 0, 0) # left
    elif lightMode == 3:
        ambientLight(0, 255, 255)
    elif lightMode == 4:
        pointLight(255, 255, 0, 100, height*0.3, 100)
    elif lightMode == 5:
        spotLight(128, 255, 128, 800, 20, 300, -1, .25, 0, PI, 2
    else:
        noLights()

    # Kugel und Box zeichnen
    with pushMatrix():
        translate(width/2, height/2)
        with pushMatrix():
            rotateY(radians(frameCount))
            fill(255)
```

```
    if lightMode == 1:
        fill(151, 255, 255)
        noStroke()
        sphere(160)
    with pushMatrix():
        rotateZ(radians(frameCount))
        rotateX(radians(frameCount/2.0))
        fill(255)
        if lightMode == 1:
            fill(0, 0, 139)
        noStroke()
        box(240)

def keyPressed():
    global lightMode, lightDirection
    if key == "n":
        lightMode = 0                      # no lights
    elif key == "l":
        lightMode = 1                      # lights
    elif key == "d":
        lightMode = 2                      # directional light
    elif key == "a":
        lightMode = 3                      # ambient light
    elif key == "p":
        lightMode = 4                      # point light
    elif key == "s":
        lightMode = 5                      # spot light

    if key == CODED:
        if keyCode == UP:
            lightDirection = 0
        elif keyCode == RIGHT:
            lightDirection = 1
        elif keyCode == DOWN:
            lightDirection = 2
        elif keyCode == LEFT:
            lightDirection = 3
```

Credits

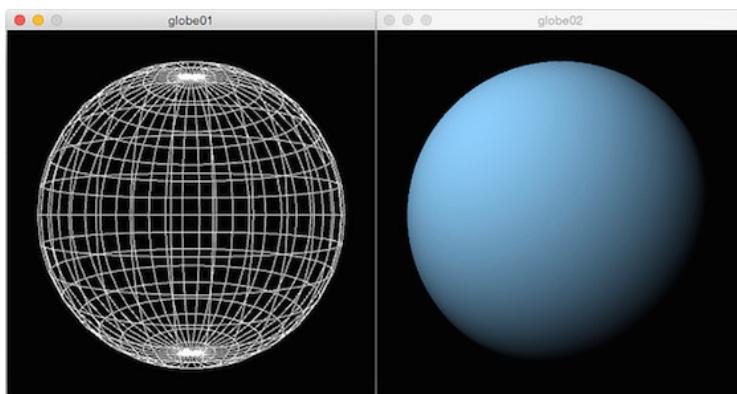
Dieses Beispielprogramm folgt einer Idee aus dem Buch »[Processing 2: Creative Programming Cookbook](#)« von Jan Vantomme. Ich habe sie geringfügig überarbeitet und vom Processing 2 Java-Mode in den Python-Mode von Processing 3 umgeschrieben.

Literatur

- Jan Vantomme: [*Processing 2: Creative Programming Cookbook*](#), Birmingham (*Packt Publishing*), 2012

Einen Globus basteln

Wenn ich – wie im [vorletzten Beispiel](#) – eine dreidimensionale Kugel programmiere, denke ich immer an den Leuchtglobus, der vor Jahrzehnten auf meinem Kinderzimmerschreibtisch stand und dem ich die Liebe zur Geographie verdanke. Es war ein Leuchtglobus, ausgeschaltet zeigte er eine topographische Weltkarte und eingeschaltet wurde er bunt und zeigte jeden Staat in einer anderen Farbe. So etwas ähnliches, nämlich einen Globus mit wechselnden Darstellungen möchte ich nun in Processing.py programmieren.



Um das zu erreichen, müssen Texturen über eine Kugel gelegt werden und das ist bei den einfachen, mit `sphere()` erzeugten

Kugeln leider nicht möglich, wir müssen uns schon eine eigene Kugel aus `Vertices` (Eckpunkten) selber basteln. Wir entwickeln sie in der Funktion `makeSphere()` aus einzelnen, aus Rechtecken zusammengesetzten Streifen in Polarkoordinaten, die dann mit Hilfe der Sinus- und Cosinus-Funktion in kartesische Koordinaten umgerechnet werden.

Um die einzelnen Rechteckstücke zu bekommen, müssen wir dem `beginShape()` den Modus `QUAD_STRIP` mitgeben. Eine anderer möglicher Modus wäre `TRIANGLE_STRIP`, Ihr könnt das ja mal testweise in den Sketch unten einsetzen, statt aus Streifen wird dann die Kugel aus Dreiecken zusammengesetzt.

Quellcode Globe 01

```
a = 0.0

def setup():
    global globe
    size(400, 400, P3D)
    globe = makeSphere(150, 10)

def draw():
    global globe, a
    background(0)
    translate(width/2, height/2)
    with pushMatrix():
        # rotateX(radians(0))
        rotateY(a)
        a += 0.01
        shape(globe)

def makeSphere(r, step):
    s = createShape()
    s.beginShape(QUAD_STRIP)
    s.noFill()
    s.stroke(255)
    s.strokeWeight(1)
    i = 0
    while i < 180:
```

```
    sini = sin(radians(i))
    cosi = cos(radians(i))
    sinip = sin(radians(i + step))
    cosip = cos(radians(i + step))
    j = 0
    while j <= 360:
        sinj = sin(radians(j))
        cosj = cos(radians(j))
        s.vertex(r*cosj*sini, r*-cosi, r*sinj*sini)
        s.vertex(r*cosj*sinip, r*-cosip, r*sinj*sinip)
        j += step
    i += step
s.endShape()
return s
```

Der nächste Schritt wäre dann, dieser Kugel eine Farbe zu verpassen. Ich habe dies in dem Sketch unten mit `fill(135, 206, 250)` versucht, das ein leichtes Blau erzeugt. Außerdem habe ich mit `pointLight(255, 255, 255, -250, -250, 500)` eine dramatische Beleuchtung gesetzt, die aber in den weiteren Fassungen wieder durch das einfache `lights()`, das für eine gleichmäßige Ausleuchtung sorgt, wieder ersetzt wird.

Doch das alleine reicht nicht aus. Wenn Ihr den Sketch so laufen läßt, werdet Ihr feststellen, daß die Kugel irgendwie »eckig« wirkt. Die Übergänge zwischen den einzelnen Quads sind deutlich zu erkennen. Und außerdem scheint sie sich an dem 0°-respektive 360°-Linie zu überlappen oder eine kleine Lücke zu klaffen. Um dies zu ändern, benutzt man in der Computergraphik den **Normalenvektor** zur Glättung von Kanten und auch in Processing ist dies mit der Methode `normal()` schon vorgesehen. Die Normalen sind Vektoren, die senkrecht auf einem Punkt stehen und diese müssen vor dem `vertex()`-Aufruf gesetzt werden.

Quellcode Globe 02

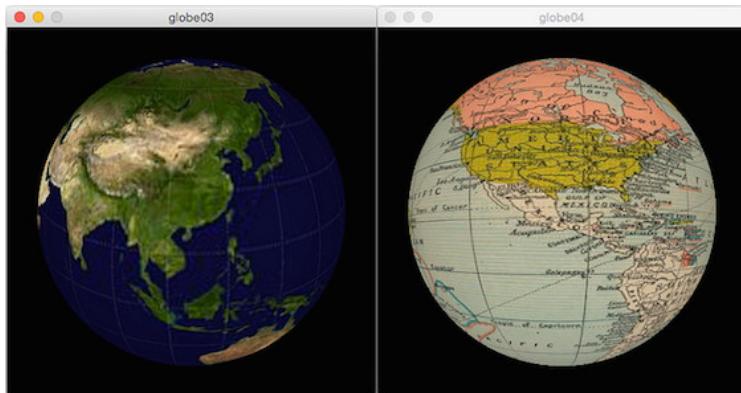
```
a = 0.0

def setup():
    global globe
```

```
size(400, 400, P3D)
globe = makeSphere(150, 5)

def draw():
    global globe, a
    background(0)
    translate(width/2, height/2)
    pointLight(255, 255, 255, -250, -250, 500)
    # lights()
    with pushMatrix():
        rotateX(radians(-30))
        rotateY(a)
        a += 0.01
        shape(globe)

def makeSphere(r, step):
    s = createShape()
    s.beginShape(QUAD_STRIP)
    s.fill(135, 206, 250)
    s.noStroke()
    s.strokeWeight(1)
    i = 0
    while i < 180:
        sini = sin(radians(i))
        cosi = cos(radians(i))
        sinip = sin(radians(i + step))
        cosip = cos(radians(i + step))
        j = 0
        while j <= 360:
            sinj = sin(radians(j))
            cosj = cos(radians(j))
            s.normal(cosj*sini, -cosi, sinj*sini)
            s.vertex(r*cosj*sini, r*-cosi, r*sinj*sini)
            s.normal(cosj*sinip, -cosip, sinj*sinip)
            s.vertex(r*cosj*sinip, r*-cosip, r*sinj*sinip)
            j += step
        i += step
    s.endShape()
    return s
```



Nun ist aber alles schön glatt und es können aus den Kugeln Globen werden. Für den ersten Globus habe ich mir eine der legendären [Blue-Marble-Karten der NASA](#) in der Größe von 640x320 Pixeln von den Seiten der Wikipedia gezogen ([Download-Link](#)) und sie `bluemarble01.jpg` genannt. Diese Bilder sind in einer [Projektion](#), die noch am ehesten das Aufziehen auf eine Kugel erlauben.

Die Funktion `vertex()` kann nun nicht nur in der Form

```
vertex(x, y, z)
```

sondern auch in der Form

```
verttext(x, y, z, u, v)
```

aufgerufen werden, wobei dann `u` und `v` die horizontalen respektive vertikalen Koordinaten für das Texture-Mapping sind. Und so wird aus der Kugel tatsächlich ein Globus mit einem Abbild unseres wunderschönen blauen Planeten.

Quellcode Globe 03

```
a = 0.0  
  
def setup():
```

```
global globe
size(400, 400, P3D)
world = loadImage("bluemarble01.jpg")
globe = makeSphere(150, 5, world)
frameRate(30)

def draw():
    global globe, a
    background(0)
    translate(width/2, height/2)
    lights()
    with pushMatrix():
        rotateX(radians(-25))
        rotateY(a)
        a += 0.01
        shape(globe)

def makeSphere(r, step, tex):
    s = createShape()
    s.beginShape(QUAD_STRIP)
    s.texture(tex)
    s.noStroke()
    i = 0
    while i < 180:
        sini = sin(radians(i))
        cosi = cos(radians(i))
        sinip = sin(radians(i + step))
        cosip = cos(radians(i + step))
        j = 0
        while j <= 360:
            sinj = sin(radians(j))
            cosj = cos(radians(j))
            s.normal(cosj*sini, -cosi, sinj*sini)
            s.vertex(r*cosj*sini, r*-cosi, r*sinj*sini,
                      tex.width-j*tex.width/360.0, i*tex.height/180.0)
            s.normal(cosj*sinip, -cosip, sinj*sinip)
            s.vertex(r*cosj*sinip, r*-cosip, r*sinj*sinip,
                      tex.width-j*tex.width/360.0, (i+step)*tex.height)
            j += step
        i += step
```

```
s.endShape()  
return s
```

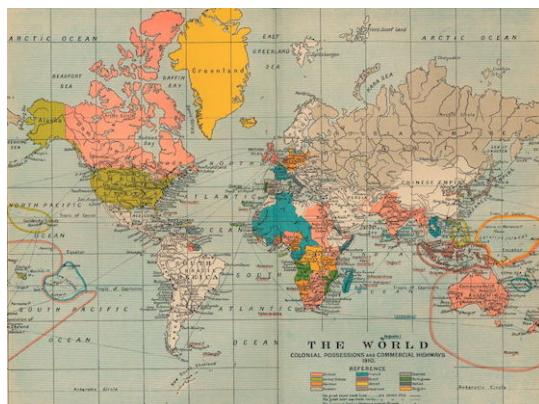


Abbildung 14.1: Weltkarte 1910

Zum Schluß bin ich dann übermüdig geworden und habe eine Weltkarte aus dem Jahre 1910 genommen und sie als Texture auf den Globus gelegt. Wie Ihr leicht erkennen könnt, eignet sich die in dieser Karte verwendete, sehr eurozentrische Projektion nicht besonders, um sie gerecht auf eine Kugel zu projizieren (Grönland besitzt in etwa die Fläche von Afrika). Um wenigstens den Schein zu wahren, habe ich die Kugel mit `rotateX(radians(5))` so geneigt, daß der Effekt in den Hintergrund tritt. Nordeuropa, Grönland und Kanada verschwinden dadurch ein bißchen hinter dem Horizont und durch den Bauch der Kugel erscheint dann auch Afrika wieder größer.

Quellcode Globe 04

```
a = 0.0  
  
def setup():  
    global globe  
    size(400, 400, P3D)  
    world = loadImage("world-map-1910.jpg")
```

```
globe = makeSphere(150, 5, world)
frameRate(30)

def draw():
    global globe, a
    background(0)
    translate(width/2, height/2)
    lights()
    with pushMatrix():
        rotateX(radians(5))
        rotateY(a)
        a += 0.01
        shape(globe)

def makeSphere(r, step, tex):
    s = createShape()
    s.beginShape(QUAD_STRIP)
    s.texture(tex)
    s.noStroke()
    i = 0
    while i < 180:
        sini = sin(radians(i))
        cosi = cos(radians(i))
        sinip = sin(radians(i + step))
        cosip = cos(radians(i + step))
        j = 0
        while j <= 360:
            sinj = sin(radians(j))
            cosj = cos(radians(j))
            s.normal(cosj*sini, -cosi, sinj*sini)
            s.vertex(r*cosj*sini, r*-cosi, r*sinj*sini,
                      tex.width-j*tex.width/360.0, i*tex.height/180.0)
            s.normal(cosj*sinip, -cosip, sinj*sinip)
            s.vertex(r*cosj*sinip, r*-cosip, r*sinj*sinip,
                      tex.width-j*tex.width/360.0, (i+step)*tex.height,
                      )
            j += step
        i += step
    s.endShape()
    return s
```

Credits

Noch mehr mit Globen stellt *Nikolaus Gradwohl* in seinem lebenswerten, von kreativen Ideen geradezu überlaufenden Buch »[Processing 2: Creative Coding Hotshot](#)« auf den Seiten 163 bis 215 an. Er lässt sie in Neonfarben leuchten und projiziert die Daten von Logfiles darauf. Ich habe in Teilen von seinem Code und seinen Ideen profitiert, ihn verändert und ihn von Processings Java-Mode in den Python-Mode überführt.

Literatur

- Nikolaus Gradwohl: [*Processing 2: Creative Coding Hotshot*](#), Birmingham (*Packt Publishing*) 2013

Kapitel 15

Einen eigenen Wetterbericht mit OpenWeatherMap



Abbildung 15.1: OpenWeatherMap Logo

OpenWeatherMap

OpenWeatherMap bietet aktuelle und freie ([CC BY-SA 4.0](#)) Wetterdaten von mehr als 200.000 Stationen weltweit (sagt die [Wikipedia](#)), die über eine frei nutzbare [JSON-API](#) abgefragt werden können – allerdings wird eine Registrierung und ein API-Key benötigt und um einen Rücklink auf OpenWeatherMap gebeten. Mit dem freien API-Key darf man 60 Anfragen in der Minute stellen und es stehen einem die

- Current Weather API, die
- 5 day/3 hour forecast API und die
- Weather maps API

zur Verfügung. Benötigt man mehr, muß man eine der kommerziellen Lizenzen nutzen. Man kann die APIs nach Städtenamen, Geo-Koordinaten oder Städte-IDs abfragen. Ich habe für mich mal ein paar relevante Orte herausgesucht:

- Berlin Tempelhof,DE – Geo-Koordinaten [52.4769, 13.4103] (das ist die nächste Station an meinem Wohnort)
- Berlin Steglitz Zehlendorf,DE – Geo-Koordinaten [52.4348, 13.2418]
- Schmargendorf,DE – Geo-Koordinaten [52.4752, 13.2907] (ich weiß nicht, welche von den beiden näher an meiner Arbeitsstelle stationiert sind, Schmargendorf steht übrigens für Berlin Dahlem)
- Berlin Koepenick,DE – Geo-Koordinaten [52.4425, 13.5823] (da ist »unser« Hundeplatz) und
- Berlin,DE – Geo-Koordinaten [52.5244, 13.4105] (Berlin Mitte)

Per Default kommen die Antwort in Englisch und die Temperaturangaben in Kelvin. Will man sie in Deutsch und °Celsius haben, muß man der URL noch die Parameter `&lang=de` und `&units=metric` mitgeben. Ein Aufruf für Berlin-Tempelhof sähe dann so aus:

<http://api.openweathermap.org/data/2.5/weather?q=Berlin%20Tempelhof,DE&lang=de&units=metric>

Die APPID habe ich mir ausgedacht, Ihr müßt Euch schon selber eine besorgen (die dann auch viel komplizierter und länger ist). Wird die Anfrage mit einer gültigen APPID abgeschickt, bekommt Ihr eine Antwort der Art:

```
{"coord":  
  {"lon":13.41,"lat":52.48},  
  "weather":[{"id":500,"main":"Rain","description":"leichter R  
  "base":"stations",  
  "main":  
    {"temp":8,"pressure":992,"humidity":93,"temp_min":8,"tem  
  },  
  "visibility":9000,  
  "wind":{"speed":5.7,"deg":210}, "clouds":{"all":90},  
  "dt":1487857800,  
  "sys":  
    {"type":1,"id":4892,"message":0.0029,"country":"DE","sun  
  },  
  "id":7290253,"name":"Berlin Tempelhof","cod":200  
}
```

Die Antwort kommt in einer Zeile, ich habe sie nur der besseren Lesbarkeit wegen umgebrochen. Die Bedeutung der einzelnen Parameter bekommt Ihr auf [dieser Seite](#) erklärt und die *Weather Condition Codes* und Icons sind auf [dieser Seite](#) aufgeführt.

Die Wetterstation mit Processing.py

Um diese JSON-Daten nun mit Processing.py lesen zu können, kann man auf die Standardbibliothek zurückgreifen, die einmal mit `urllib2` einen einfachen Umgang mit dem Laden von Daten aus dem Netz erlaubt und zum anderen mit `json` ein Modul mitbringt, das den Umgang mit den JSON-Dateien vereinfacht.

```
import json  
import urllib2  
weatherUrl = "http://api.openweathermap.org/data/2.5/weather?q=B  
weatherData = json.load(urllib2.urlopen(weatherUrl))
```



Abbildung 15.2: Screenshot Wetterstation

Noch einmal: Den API-Key (APPID) habe ich mir ausgedacht, um den Code-Schnipsel oben zum Laufen zu bekommen, müßt Ihr Euch auf den Seiten von OpenWeatherMap schon einen eigenen API-Key besorgen.

```
{  
    u'visibility': 10000,  
    u'main': {  
        u'temp': 3,  
        u'pressure': 1007,  
        u'temp_max': 3,  
        u'temp_min': 3,  
        u'humidity': 74  
    },  
    u'clouds': {u'all': 40},  
    u'sys': {  
        u'country':  
        u'DE',  
        u'sunrise': 1487916142,  
        u'type': 1,  
    }  
}
```

Kapitel 15. Einen eigenen Wetterbericht mit OpenWeatherMap

```
        u'message': 0.0025,
        u'sunset': 1487954244,
        u'id': 4892
    },
    u'dt': 1487940600,
    u'coord': {u'lon': 13.41, u'lat': 52.48},
    u'weather': [
        {
            u'icon': u'13d', u'description': u'm\xe4\xdfiger Schnee', u'',
        }],
    u'name': u'Berlin Tempelhof',
    u'cod': 200,
    u'id': 7290253,
    u'base': u'stations',
    u'wind': {u'deg': 310, u'speed': 4.6}
}
```

Laßt Ihr Euch die `weatherData` aus obigem Codeschnipsel mal anzeigen (ich habe sie wieder der besseren Lesbarkeit wegen umgebrochen), dann seht Ihr, daß die JSON-Bibliothek die Antwort als *Dictionary* behandelt und Ihr damit nicht mehr auf die Reihenfolge bauen könnt. Es ist zwar alles vorhanden, was Ihr auch ganz oben in der ersten Antwort seht, aber in einer völlig anderen Reihenfolge. Außerdem hat die Bibliothek alle Strings als UTF-8-Strings gekennzeichnet.

Nun lassen sich aber die Werte in *Dictionaries* in Python mit Ihrem Key abfragen und die Keys können miteinander verketten werden. Wollt Ihr zum Beispiel den Wert des Dictionaries "`temp`", das Teil des Dictionaries "`main`" ist, abfragen, so ist dies mit

```
temp = weatherData["main"]["temp"]
```

möglich. In einigen Fällen beinhalten die JSON-Objekte aber auch Listen. Diese können aber ebenfalls verkettet werden und werden über ihren Index aufgerufen. Wollt Ihr zum Beispiel die Wetterbeschreibung ("`description`") aus dem Dictionary "`weather`" haben, so müßt Ihr folgendes programmieren:

```
wetter = weatherData["weather"][0]["description"]
```

So habe ich mir Stück für Stück alle Daten, die ich für mein kleines Wetterfenster haben wollte, zusammengeklaubt.

Schaut Ihr Euch die Daten, die eine Datums- und Zeitangabe betreffen, genauer an, werdet Ihr feststellen (oder in der Dokumentation nachlesen), daß diese – wie international üblich – als UTC-Timestamp kommen. Um dieses zu konvertieren, bildet das Modul `datetime` Hilfe an, zum Beispiel:

```
import datetime
sunrise = weatherData["sys"]["sunrise"]
lokalsunrise = datetime.datetime.fromtimestamp(sunrise).ctime()
```

Mit `fromtimestamp` wird der UTC-Stempel in eine lesbare Zeit verwandelt und das anschließende `ctime` sorgt dafür, daß dies in die lokale Rechnerzeit umgewandelt wird (in meinem Fall in UTC+1 oder während der Sommerzeit in UTC+2). Um die Sommerzeit kümmert sich `ctime` automatisch, da muß sich der Programmierer nicht weiter sorgen.

Das Wetter-Icon kommt natürlich auch von OpenWeatherMap und kann so geladen und angezeigt werden:

```
icon = weatherData["weather"][0]["icon"]
weatherIcon = loadImage("http://openweathermap.org/img/w/" + icon + ".png")
image(weatherIcon, 10, 260)
```

Das ist eigentlich alles, was der Programmierer wissen muß. Für die aktuelle Zeit habe ich ebenfalls das Modul `datetime` genutzt und die Berechnungen durchgeführt, die ich auch schon in dem Programm zur [Rentenuhr](#) genutzt hatte.

Um den Hauptsketch übersichtlich zu halten, habe ich die beiden Funktionen `getWeatherData()` und `getNow()` in ein eigenes Modul `getWeatherData.py` ausgelagert, das wie folgt aussieht:

```
# coding=utf-8
import json
import urllib2
import datetime
```

```
def getWeatherData():
    weatherUrl = "http://api.openweathermap.org/data/2.5/weather"
    weatherData = json.load(urllib2.urlopen(weatherUrl))

    # Temperatur
    temp = weatherData["main"]["temp"]
    myTemperatur = u"Temperatur: " + str(temp) + u"°C."
    text(myTemperatur, 10, 20)

    # Wetter-Beschreibung
    wetter = weatherData["weather"][0]["description"]
    myWetter = u"Wetter: " + wetter + "."
    text(myWetter, 10, 42)

    # Sonnenauf- und -untergang
    sunrise = weatherData["sys"]["sunrise"]
    sunset = weatherData["sys"]["sunset"]
    mySunrise = "Sonnenaufgang: " + datetime.datetime.fromtimestamp(sunrise).strftime("%H:%M")
    mySunset = "Sonnenuntergang: " + datetime.datetime.fromtimestamp(sunset).strftime("%H:%M")
    text(mySunrise, 10, 80)
    text(mySunset, 10, 102)

    # Luftdruck und -feuchtigkeit
    pressure = weatherData["main"]["pressure"]
    myPressure = "Luftdruck: " + str(pressure) + " hPa."
    text(myPressure, 10, 140)
    humidity = weatherData["main"]["humidity"]
    myHumidity = "Luftfeuchtigkeit: " + str(humidity) + " %."
    text(myHumidity, 10, 162)

    # Windgeschwindigkeit und Bewölkung
    wind = weatherData["wind"]["speed"]
    myWind = "Windgeschwindigkeit: " + str(wind) + " m/s."
    text(myWind, 10, 200)
    clouds = weatherData["clouds"]["all"]
    myClouds = u"Bewölkung: " + str(clouds) + " %."
    text(myClouds, 10, 222)

    # Wetter-Icon
```

```

icon = weatherData["weather"][0]["icon"]
weatherIcon = loadImage("http://openweathermap.org/img/w/" + icon)
image(weatherIcon, 10, 260)

# Abfragezeit und -ort
dt = weatherData["dt"]
station = weatherData["name"]
myDt = "Stand: " + datetime.datetime.fromtimestamp(dt).ctime() +
text(myDt, 10, 360)

def getNow():
    myNow = datetime.datetime.now()
    myHour = str(myNow.hour)
    myMinute = str(myNow.minute).rjust(2, "0")
    mySecond = str(myNow.second).rjust(2, "0")
    myTime = myHour + ":" + myMinute + ":" + mySecond
    text(u"Update: " + myTime, 10, 382)

```

Zum letzten Mal: Die APPID habe ich mir ausgedacht, mit dieser bekommt Ihr keine Daten von OpenWeatherMap.

Das Modul wirkt auf den ersten Blick schlimmer als es ist, denn eigentlich ist alles *straightforward*: Die Daten werden aus dem JSON-Objekt ausgelesen, dann wird ein String erzeugt und zum Schluß wird dieser String mit Hilfe der `text()`-Funktion angezeigt.

So ist das Hauptprogramm wieder sehr kurz geraten:

```

from getWeatherData import getWeatherData, getNow

def setup():
    size(600, 400)
    background(0)
    frame.setTitle (u"Jörgs Wetterstation")
    font = createFont("American Typewriter", 18)
    textAlign(CENTER)
    textSize(18)
    getWeatherData()
    getNow()
    frameRate(1)

```

```
def draw():
    if(second() == 0):
        background(0)
        getWeatherData()
        getNow()
```

In der `setup()`-Funktion rufe ich zu Initialisierung genau einmal die Wetterdaten ab. Nun darf man in der kostenlosen Lizenz die Daten maximal 60 mal in der Minute abrufen. Mit `frameRate(1)` alleine schafft man das nicht (ich habe sie auch nur darauf gesetzt, um den Rechner nicht unnötig zu belasten). Darum werden die weiteren Abfragen nur gestartet, wenn die Sekunde auf Null steht, das heißt es gibt nur einen Aufruf in der Minute. Damit habe ich die Lizenz der API mehr als eingehalten.

Eigentlich könnte man die Daten noch seltener abrufen: Die von mir angefragte Station *Berlin Tempelhof* gibt nur jede halbe Stunde (um x:20 Uhr und um x:50 Uhr) ihre Daten weiter und es kann noch bis zu einer weiteren halben Stunde dauern, bis die Daten bei OpenWeatherMap eingepflegt und abrufbar sind. Im schlimmsten Fall kann es zu Verzögerungen bis zu einer Stunde kommen, manchmal sind die neuen Daten aber auch überraschend schnell da.

Sicher kann man die Wetterstation optisch noch ein wenig aufpeppen und man kann auch mehrere Wetterstationen abfragen oder die API zur Wettervorhersage nutzen, aber als Beispiel, wie man JSON-Daten per API aus dem Netz holt und aufbereitet, ist dieser Sketch völlig ausreichend. Alles weitere bleibt der Phantasie meiner Leserinnen und Leser überlassen.

Kapitel 16

WordCram: Processing.py und eine Processing (Java) Bibliothek



Abbildung 16.1: Screenshot

Wie man eine Python-Bibliothek in Processing.py nutzt, [hatte ich ja schon gezeigt](#). Das geht so einfach, wie man es auch von »normalen« Python-Programmen gewohnt ist – solange die Pakete *pure Python* sind. Doch wie sieht es aus, wenn man eine Bibliothek nutzen will, die für den Java-Mode von Processing in Java geschrieben wurde? Um dies zu testen, hatte ich mir die Bibliothek [WordCram](#), die die beliebten Wortwolken (*Word Clouds*) erzeugt, ausgesucht und heruntergeladen. Bevor man damit irgendetwas anstellen kann, muß man die Bibliothek entpacken und in den **libraries**-Ordner im Processing-Verzeichnis ablegen. Falls Ihr nicht mehr wißt, wo Euer Processing-Verzeichnis liegt, findet Ihr es im Processing-Menü unter **Einstellungen -> Sketchbook Pfad**.

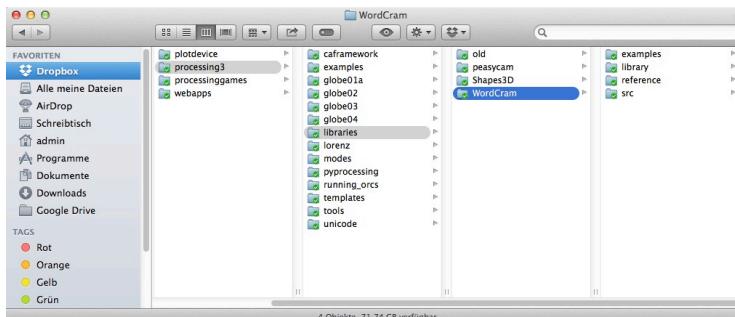


Abbildung 16.2: Screenshot

Da ich von mehreren Rechnern an mehreren Standorten meine Processing.py-Sketches bearbeite, liegt dieser Ordner bei mir in der Dropbox. Das ist nicht unbedingt immer eine gute Idee, manche Libraries bestehen aus Tausenden von Dateien und da kann das Synchronisieren schon mal eine gewisse Zeit in Anspruch nehmen.

Wenn Ihr dies erledigt habt, findet die Processing-IDE unter **Sketch -> Library importieren** die Bibliothek und sie kann mit einem Klick in Euren Sketch eingefügt werden:

```
add_library('WordCram')
```

Natürlich könnt Ihr diese Zeile auch einfach selber eintippen.

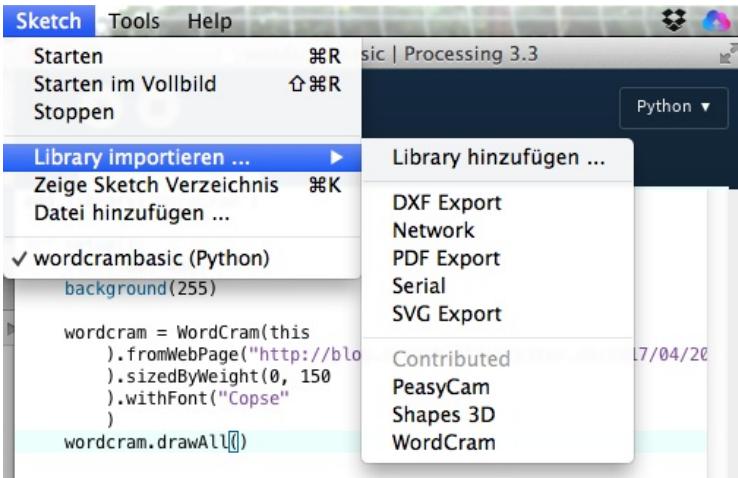


Abbildung 16.3: Screenshot

Um das Teil zu testen, habe ich diesen kleinen Sketch geschrieben:

```
add_library('WordCram')

def setup():
    size(700, 400)
    background(255)

    wordcram = WordCram(this
        ).fromWebPage("http://blog.schockwellenreiter.de/2017/04
        ).sizedByWeight(0, 150
        ).withFont("Copse"
    )
    wordcram.drawAll()
```

Der ruft [diese Seite](#) auf und stellt ihren Inhalt – wie obiger Screenshot zeigt – als Wortwolke dar.

Zwei Dinge sind noch zu beachten: Erstens verlangen viele Processing (Java) Bibliotheken eine Referenz auf das aktuelle PApplet-Objekt – so auch WordCram. Im Java-Mode für

Processing wird dafür das eingebaute `this`-Keyword verwendet. Python allerdings kennt kein `this`-Schlüsselwort, aber `Processing.py` stellt automatisch eine globale Variable namens `this` zur Verfügung, die für diese Zwecke verwendet werden kann.

Zweitens kann man in Python wegen der besonderen Bedeutung der Einrückung nicht so, wie es in Java üblich ist, die fortlaufende Punkt-Notierung an dem Punkt umbrechen, wenn Ihr es so versucht

```
wordcram = WordCram(this)
    .fromWebPage("http://blog.schockwellenreiter.de/2017/04/201704")
    .sizedByWeight(0, 150)
    .withFont("Copse")
```

bekommt Ihr eine Fehlermeldung. Ich habe daher überall die schließende Klammer aus der darüberliegenden Zeile weggenommen und vor den Punkt gestellt. Das ist nicht wirklich eine schöne Lösung, funktioniert aber und hält den Code einigermaßen leserlich. Vielleicht fällt mir dafür noch eine schönere Lösung ein.

Die WordCram-Bibliothek ist ziemlich mächtig, aber leider nicht besonders gut dokumentiert. Sie erkennt zum Beispiel selbstständig, daß mein Text auf Deutsch geschrieben ist und ist auch UTF-8-fest. Zudem scheint sie für viele Sprachen schon eine eingebaute Liste von Stopwörtern mitzubringen. Und man kann auch irgendwie die Wörter der Cloud einfärben, aber das ist nicht sehr intuitiv und die bisher erzielten Ergebnisse hatten mir nicht gefallen, so daß ich es bei der schwarz-weiß-Darstellung belassen habe. *Still digging!*

Kapitel 17

Running Orc mit Processing.py

Nach den vier Tutorials mit den Figuren aus *Cute Planet* zu Processing.py, dem Python-Mode für Processing, wurde mir das allmählich zu niedlich und ich beschloß, endlich mal wieder einen Ork durch das Bildschirmfenster wuseln zu lassen.

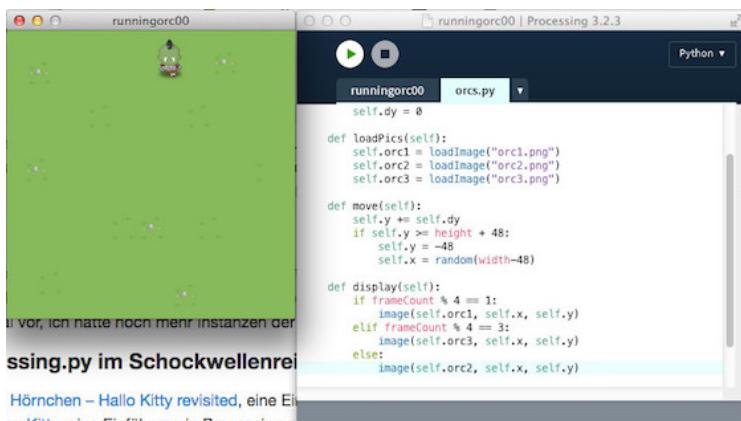


Abbildung 17.1: Schon wieder ein rennender Ork

Dafür habe ich erst einmal die Klasse `orc()` definiert und nach

der Initialisierung – wie in den anderen Tutorials auch schon – die Methoden `loadPics()`, `move()` und `display()` implementiert:

```
class Orc():

    def __init__(self, posX, posY):
        self.x = posX
        self.y = posY
        self.dy = 0

    def loadPics(self):
        self.orc1 = loadImage("orc1.png")
        self.orc2 = loadImage("orc2.png")
        self.orc3 = loadImage("orc3.png")

    def move(self):
        self.y += self.dy
        if self.y >= height + 48:
            self.y = -48
            self.x = random(width-48)

    def display(self):
        if frameCount % 4 == 1:
            image(self.orc1, self.x, self.y)
        elif frameCount % 4 == 3:
            image(self.orc3, self.x, self.y)
        else:
            image(self.orc2, self.x, self.y)
```

Die Figur des *Orcs* hatte ich ja schon häufiger verwendet, die Zeichnungen stammen aus dem freien ([CC-BY-SA](#)) [OpenPixels](#)-Projekt von [Silveira Neto](#). Der Einfachheit halber und damit Ihr das nachprogrammieren könnt, habe ich die drei animierten Bildchen hier noch einmal eingebunden:



Das Hauptskript war dank des Klasse `Orc()` dann wieder von erfrischender Kürze:

```
from orcs import Orc

orc = Orc(160, -48)

def setup():
    global bg
    bg = loadImage("field.png")
    frameRate(15)
    size(320, 320)
    orc.loadPics()
    orc.dy = 5

def draw():
    background(bg)
    orc.move()
    orc.display()
```

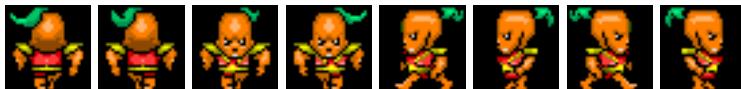
In `setup()` habe ich zuerst das Hintergrundbild eingebunden (es ist ebenfalls aus dem oben erwähnten OpenPixels-Projekt – Ihr könnt alternativ aber auch einfach einen grünen Hintergrund zeichnen) und dann den Orc angewiesen, seine Bilder zu laden. Und wie in den vorhergegangenen Tutorials auch wird in `draw()` zuerst der Hintergrund gezeichnet, dann der Ork bewegt und schließlich an seiner neuen Position angezeigt. *That's all!*

Wie es damit weitergeht, weiß ich noch nicht so genau. Zum einen habe ich die Idee, ganz viele Instanzen der Orks über das Spielfeld zu bewegen, eine andere Idee ist es, den Ork mittels der Pfeiltasten in alle vier Himmelsrichtungen laufen und auf Hindernisse reagieren zu lassen. Schauen wir mal ...

Running Orc in vier Richtungen

Heute möchte ich meine kleine Einführung in Processing.py, dem Python-Mode für Processing, damit fortsetzen, daß ich einen

kleinen Ork unter Benutzerführung und mit Hilfe der Pfeiltasten in allen vier Himmelsrichtungen über die Spielwiese wuseln lasse. Die Grundlagen hatte ich dafür ja schon im [hier](#) gelegt, der Unterschied aber ist, daß der kleine Ork sich tatsächlich bewegt und auch in alle Richtungen dreht. Dafür brauchte ich erst einmal diese acht Bildchen des kleinen Monsters:



Im Gegensatz zu dem Ork aus dem letzten Tutorial stammen diese Bildchen (bis auf die Hintergrund-Wiese) nicht aus dem [Open-Pixels](#)-Fundus von [Silveira Neto](#), sondern aus der ebenfalls freien ([CC BY 3.0](#)) [Sprite-Sammlung](#) von [Philipp Lenssen](#) (über 700 animierte Avatare in der klassischen Größe von 32x32 Pixeln). Und die Animationen setzen sich auch nur aus je zwei verschiedenen Bildchen zusammen, was zum einen Code und Speicher spart und zum anderen den Charakteren einen besonders wuseligen Eindruck verschafft, der an die Frühzeit der Computerspiele erinnert (aus der die Bilder auch stammen). Man benötigt so für jede der vier Himmelsrichtungen nur zwei Bilder, was dann zusammen obige acht Bildchen ergibt.

Als erstes habe ich dem Ork natürlich wieder eine eigene Klasse spendiert (in dem Tab/der Datei `orc2.py`), deren Quellcode nun schon bedeutend umfangreicher geworden ist:

```
class Orc():

    def __init__(self, posX, posY):
        self.x = posX
        self.y = posY
        self.dir = 1
        self.dx = 0
        self.dy = 0

    def loadPics(self):
        self.orcrt1 = loadImage("orcrt1.gif")
        self.orcrt2 = loadImage("orcrt2.gif")
        self.orcfr1 = loadImage("orcfr1.gif")
        self.orcfr2 = loadImage("orcfr2.gif")
```

```
self.orclf1 = loadImage("orclf1.gif")
self.orclf2 = loadImage("orclf2.gif")
self.orcbk1 = loadImage("orcbk1.gif")
self.orcbk2 = loadImage("orcbk2.gif")

def move(self):
    if self.dir == 0:
        if self.x >= width - 32:
            self.x = width - 32
            self.image1 = self.orcrt2
            self.image2 = self.orcrt2
    else:
        self.x += self.dx
        self.image1 = self.orcrt1
        self.image2 = self.orcrt2
    elif self.dir == 1:
        if self.y >= height - 32:
            self.y = height - 32
            self.image1 = self.orcfr2
            self.image2 = self.orcfr2
    else:
        self.y += self.dy
        self.image1 = self.orcfr1
        self.image2 = self.orcfr2
    elif self.dir == 2:
        if self.x <= 0:
            self.x = 0
            self.image1 = self.orclf2
            self.image2 = self.orclf2
    else:
        self.x -= self.dx
        self.image1 = self.orclf1
        self.image2 = self.orclf2
    elif self.dir == 3:
        if self.y <= 0:
            self.y = 0
            self.image1 = self.orcbk2
            self.image2 = self.orcbk2
    else:
        self.y -= self.dy
```

```
    self.image1 = self.orcbk1
    self.image2 = self.orcbk2

def display(self):
    if frameCount % 8 >= 4:
        image(self.image1, self.x, self.y)
    else:
        image(self.image2, self.x, self.y)
```

Im Konstruktor werden nur die Startposition festgelegt und ein paar Variablen initialisiert und mit Default-Werten versehen. Danach werden die acht Bildchen geladen. Die eigentliche Logik liegt in der Funktion `move()`: Erreicht der Orc einer der Fensterränder, bleibt er einfach stehen. Der visuelle Eindruck wird dadurch erreicht, daß die beiden zu swappenden Bilder identisch sind. Ansonsten bewegt er sich in der angesagten Richtung weiter, indem `dx` oder `dy` zu der akutellen Position addiert oder abgezogen werden.

Die Funktion `display()` ist dann für die Darstellung zuständig: Ist der `frameCount % 8 >= 4`, dann wird das erste Bild gezeichnet, ansonsten das zweite Bild. Durch diesen Modulo-Trick bin ich noch einmal daran vorbeigekommen, einen Timer implementieren zu müssen, aber irgendwann wird kein Weg mehr daran vorbeiführen.

Das Hauptprogramm ist immer noch vergleichsweise kurz und übersichtlich geraten:

```
from orc2 import Orc

orc = Orc(160, -32)

def setup():
    global bg
    bg = loadImage("field.png")
    frameRate(30)
    size(320, 320)
    orc.loadPics()
    orc.dx = 2
```

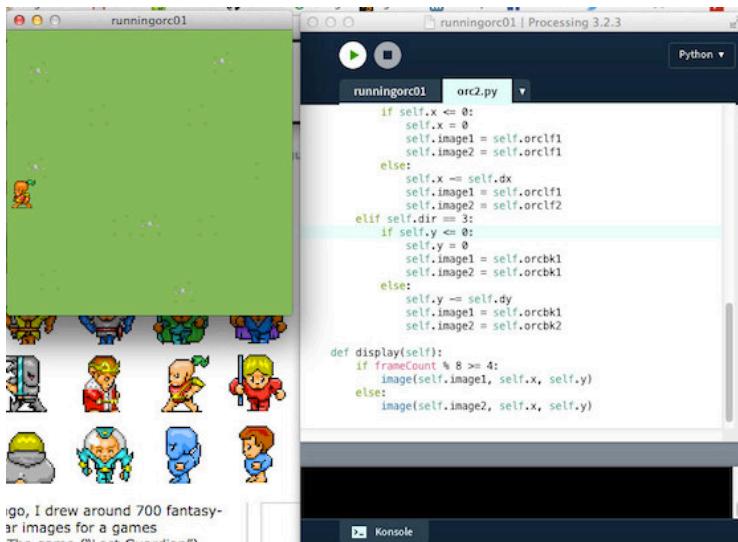


Abbildung 17.2: Running Orc 4

```

orc.dy = 2

def draw():
    background(bg)
    orc.move()
    orc.display()

def keyPressed():
    if keyPressed and key == CODED:
        if keyCode == RIGHT:
            orc.dir = 0
        elif keyCode == DOWN:
            orc.dir = 1
        elif keyCode == LEFT:
            orc.dir = 2
        elif keyCode == UP:
            orc.dir = 3

    if self.x <= 0:
        self.x = 0
        self.image1 = self.orclf1
        self.image2 = self.orclf1
    else:
        self.x -= self.dx
        self.image1 = self.orclf1
        self.image2 = self.orclf2
    elif self.dir == 3:
        if self.y <= 0:
            self.y = 0
            self.image1 = self.orcbk1
            self.image2 = self.orcbk1
        else:
            self.y -= self.dy
            self.image1 = self.orcbk1
            self.image2 = self.orcbk2

    def display(self):
        if frameCount % 8 >= 4:
            image(self.image1, self.x, self.y)
        else:
            image(self.image2, self.x, self.y)

```

Die Klasse `Orc` wird importiert und initialisiert. Danach wird das Hintergrundbild geladen (Ihr könnt auch hier wieder alternativ

einfach einen grünen Hintergrund zeichnen) und die Fenstergröße festgelegt. Dann wird die Funktion `orc.loadPics()` aufgerufen und die horizontale und vertikale Geschwindigkeit auf je zwei Pixel pro Frame-Durchlauf bestimmt.

Die `draw()`-Routine ist immer noch einfach: Erst wird der Hintergrund gezeichnet, dann der Ork bewegt und danach ebenfalls in das Fenster gezeichnet.

Neu ist die Funktion `keyPressed()`, die während des gesamten Programmablaufs die Tastatur überwacht. Sie überprüft, welche der Pfeiltasten gedrückt wurden und weist ihnen dementsprechend eine Himmelsrichtung zu. Per Konvention fängt man normalerweise im Osten an (`orc.dir = 0`), um dann über den Süden (`orc.dir = 1`) und den Westen (`orc.dir = 2`) zum Norden (`orc.dir = 3`) zu gelangen.

Beachtet bitte, daß die Abfrage der Tastatur erst greift, wenn das Programmfenster den Fokus besitzt. Leider passiert das bei Processing.py nicht automatisch beim Programmstart, Ihr müßt einmal mit der Maus in das Fenster klicken.

Das ist alles. Erfreut Euch auch an dem kleinen Gag, den *Philipp Lenssen* seinem Ork verpaßt hat: Das Haarschwänzchen wedelt fröhlich hin und her.

Ork mit Kollisionserkennung

Nachdem ich im letzten Abschnitt gezeigt hatte, wie man einen kleinen Ork mit Hilfe der Pfeiltasten in allen vier Himmelsrichtungen über das Bildschirmfenster jagen kann, bis er am Fensterrand stehenbleibt, möchte ich Euch nun zeigen, wie man eine generelle Kollisionserkennung implementiert. Dafür habe ich erst einmal eine Oberklasse namens `Sprite` eingeführt, die das Verhalten, das allen `Sprites` gemein ist, festlegt und von der alle `Sprites` erben sollen (zur Bedeutung und Herkunft des Begriffs `Sprite` informiert die Wikipedia).

Die Klasse `Sprite` sieht in [Processing.py](#) erst einmal so aus:

```
class Sprite(object):
```



Abbildung 17.3: Screenshot

```

def __init__(self, posX, posY):
    self.x = posX
    self.y = posY
    self.dir = 1
    self.dx = 0
    self.dy = 0

def checkCollision(self, otherSprite):
    if (self.x < otherSprite.x + tw and otherSprite.x < self.x + th
        and self.y < otherSprite.y + th and otherSprite.y < self.y + tw):
        println("Kollision")
        return True
    else:
        return False

```

Das Objekt wird initialisiert und die Startposition festgelegt. Dann werden noch ein paar Variablen mit Defaultwerten besetzt. Da es durchaus Sprites geben kann, die sich gar nicht bewegen, sind dx und dy mit 0 vorbelegt.

Momentan die wichtigste Funktion ist die Funktion `checkCollision(self, otherSprite)`. Darin wird geprüft, ob sich die umgebenden Rechtecke der Sprites (in diesem Falle ist das die Bildgröße (tw und th sind jeweils 32 Pixel) überlappen, denn dann liegt eine

Kollision vor. Dazu ist es für eine einigermaßen »realistische« Darstellung natürlich wichtig, daß die Sprite-Zeichnung das Rechteck möglichst vollständig ausfüllt. In diesem Falle nehme ich das einfach mal an (mehr dazu weiter unten). Die Klasse `Orc` erbt nun natürlich von `Sprite`:

```
class Orc(Sprite):

    def loadPics(self):
        self.orcrt1 = loadImage("orcrt1.gif")
        self.orcrt2 = loadImage("orcrt2.gif")
        self.orcfr1 = loadImage("orcfr1.gif")
        self.orcfr2 = loadImage("orcfr2.gif")
        self.orclf1 = loadImage("orclf1.gif")
        self.orclf2 = loadImage("orclf2.gif")
        self.orcbk1 = loadImage("orcbk1.gif")
        self.orcbk2 = loadImage("orcbk2.gif")

    def move(self):
        if self.dir == 0:
            if self.x >= width - tileSize:
                self.x = width - tileSize
                self.image1 = self.orcrt2
                self.image2 = self.orcrt2
            else:
                self.x += self.dx
                self.image1 = self.orcrt1
                self.image2 = self.orcrt2
        elif self.dir == 1:
            if self.y >= height - tileSize:
                self.y = height - tileSize
                self.image1 = self.orcfr2
                self.image2 = self.orcfr2
            else:
                self.y += self.dy
                self.image1 = self.orcfr1
                self.image2 = self.orcfr2
        elif self.dir == 2:
            if self.x <= 0:
                self.x = 0
```

```
        self.image1 = self.orclf2
        self.image2 = self.orclf2
    else:
        self.x -= self.dx
        self.image1 = self.orclf1
        self.image2 = self.orclf2
    elif self.dir == 3:
        if self.y <= 0:
            self.y = 0
        self.image1 = self.orcbk2
        self.image2 = self.orcbk2
    else:
        self.y -= self.dy
        self.image1 = self.orcbk1
        self.image2 = self.orcbk2

def display(self):
    if frameCount % 8 >= 4:
        image(self.image1, self.x, self.y)
    else:
        image(self.image2, self.x, self.y)
```

hat sich aber ansonsten gegenüber dem letzten Tutorial nicht verändert. Da ja nun die Kollisionsüberprüfung getestet werden muß, habe ich ein weiteres, unbewegliches Sprite konstruiert, das ich aus naheliegenden Gründen `Wall` genannt habe. Auch `Wall` erbt natürlich von `Sprite`:

```
class Wall(Sprite):

    def loadPics(self):
        self.pic = loadImage("wall.png")

    def display(self):
        image(self.pic, self.x, self.y)
```

Da sich die Mauer nicht bewegt, besitzt `Wall` natürlich auch keine `move()`-Methode, sondern wird nur angezeigt. Ganz oben in die ersten drei Zeilen des Tabs `sprites.py` habe noch ein paar Konstanten initialisiert:

```
tw = 32
th = 32
tileSize = 32
```

Das war erst einmal das Modul `sprites.py`. Das Hauptprogramm, das ich `obstacles` genannt habe, ist immer noch von erfrischender Kürze und dank der Objekte kaum verändert:

```
tileSize = 32

from sprites import Orc, Wall

def setup():
    global bg
    bg = loadImage("field.png")
    frameRate(30)
    size(320, 320)
    global orc
    orc = Orc(8*tileSize, 0)
    orc.loadPics()
    orc.dx = 2
    orc.dy = 2
    global wall1
    wall1 = Wall(5*tileSize, 3*tileSize)
    wall1.loadPics()

def draw():
    global moving
    background(bg)
    wall1.display()
    orc.move()
    orc.display()
    orc.checkCollision(wall1)

def keyPressed():
    if keyPressed and key == CODED:
        if keyCode == RIGHT:
            orc.dir = 0
        elif keyCode == DOWN:
```

```
orc.dir = 1
elif keyCode == LEFT:
    orc.dir = 2
elif keyCode == UP:
    orc.dir = 3
```

Neu ist lediglich das Mauerfragment `wall1` und das nun als letztes in der `draw()`-Funktion mit `orc.checkCollision(wall1)` überprüft wird, ob unser Ork mit der Mauer kollidiert. Im Falle einer Kollision wird bisher allerdings »Kollision« in das Terminalfenster geschrieben. Das zeigt, daß der Algorithmus funktioniert, mehr aber noch nicht.

Um dies zu ändern, habe ich erst einmal das `println("Kollision")` in der Klasse `Sprite` gelöscht und – um auf ein Problem aufmerksam zu machen – die Klasse `Tree` als weiteres, unbewegliches Objekt hinzugefügt:

```
class Tree(Sprite):

    def loadPics(self):
        self.pic = loadImage("tree.png")

    def display(self):
        image(self.pic, self.x, self.y)
```

Bis auf das andere Bildchen unterscheidet sie sich nicht von der Klasse `Wall`. Baum und Mauer (sowie die neue Hintergrundkachel) habe ich dem freien ([CC BY 3.0](#)) Angband-Tilesets von [dieser Site](#) entnommen und mit dem Editor [Tiled](#) zurechtgeschnitten. Hier die Bildchen auch für Euch, damit Ihr das Beispiel nachprogrammieren könnt:



Das Hintergrundbild habe ich in *Tiled* aus der Graskachel erzeugt. Die Bilder des Orks könnt Ihr im letzten Abschnitt finden.

Die Datei im Tab `sprites.py` hat sich sonst nicht weiter verändert, aber eine wesentliche Veränderung hat im Hauptpro-

gramm stattgefunden. Hier heißtt es nun zwischen `orc.move()` und `orc.display()`:

```
if orc.checkCollision(wall1) or orc.checkCollision(tree1):
    if orc.dir == 0:
        orc.x -= orc.dx
    elif orc.dir == 1:
        orc.y -= orc.dy
    elif orc.dir == 2:
        orc.x += orc.dx
    elif orc.dir == 3:
        orc.y += orc.dy
    orc.image1 = orc.image2
```

Jetzt wird also überprüft, ob eine Kollision mit dem Mauerfragment oder mit dem Baum stattgefunden hat. Hat eine stattgefunden, wird der Orc einfach auf die vorherige Position zurückgesetzt und die beiden Bilder – wie wir es schon mit der Kollision mit den Rändern hatten – auf ein Bild gesetzt, so daß es aussieht, als ob der Ork stehen bleiben würde und auf Eure nächste Eingabe wartet.

Hier nun den kompletten Sketch zum Nachbauen. Erst einmal das Hauptprogramm `obstacles02`:

```
tileSize = 32
from sprites import Orc, Wall, Tree

def setup():
    global bg
    bg = loadImage("ground0.png")
    frameRate(30)
    size(320, 320)
    global orc
    orc = Orc(8*tileSize, 0)
    orc.loadPics()
    orc.dx = 2
    orc.dy = 2
    global wall1
    wall1 = Wall(5*tileSize, 3*tileSize)
```

```
wall1.loadPics()
global tree1
tree1 = Tree(3*tileSize, 7*tileSize)
tree1.loadPics()

def draw():
    background(bg)
    wall1.display()
    tree1.display()
    orc.move()
    if orc.checkCollision(wall1) or orc.checkCollision(tree1):
        if orc.dir == 0:
            orc.x -= orc.dx
        elif orc.dir == 1:
            orc.y -= orc.dy
        elif orc.dir == 2:
            orc.x += orc.dx
        elif orc.dir == 3:
            orc.y += orc.dy
        orc.image1 = orc.image2

    orc.display()

def keyPressed():
    if keyPressed and key == CODED:
        if keyCode == RIGHT:
            orc.dir = 0
        elif keyCode == DOWN:
            orc.dir = 1
        elif keyCode == LEFT:
            orc.dir = 2
        elif keyCode == UP:
            orc.dir = 3
```

Und dann das Modul `sprites.py`, das ich in einem separaten Tab untergebracht habe:

```
tw = 32
th = 32
```

```
tileSize = 32

class Sprite(object):

    def __init__(self, posX, posY):
        self.x = posX
        self.y = posY
        self.dir = 1
        self.dx = 0
        self.dy = 0

    def checkCollision(self, otherSprite):
        if (self.x < otherSprite.x + tw and otherSprite.x < self.x + tw
            and self.y < otherSprite.y + th and otherSprite.y < self.y + th):
            return True
        else:
            return False

class Orc(Sprite):

    def loadPics(self):
        self.orcrt1 = loadImage("orcrt1.gif")
        self.orcrt2 = loadImage("orcrt2.gif")
        self.orcfr1 = loadImage("orcfr1.gif")
        self.orcfr2 = loadImage("orcfr2.gif")
        self.orclf1 = loadImage("orclf1.gif")
        self.orclf2 = loadImage("orclf2.gif")
        self.orcbk1 = loadImage("orcbk1.gif")
        self.orcbk2 = loadImage("orcbk2.gif")

    def move(self):
        if self.dir == 0:
            if self.x >= width - tileSize:
                self.x = width - tileSize
                self.image1 = self.orcrt2
                self.image2 = self.orcrt2
        else:
            self.x += self.dx
            self.image1 = self.orcrt1
            self.image2 = self.orcrt2
```

```
        elif self.dir == 1:
            if self.y >= height - tileSize:
                self.y = height - tileSize
                self.image1 = self.orcfr2
                self.image2 = self.orcfr2
            else:
                self.y += self.dy
                self.image1 = self.orcfr1
                self.image2 = self.orcfr2
        elif self.dir == 2:
            if self.x <= 0:
                self.x = 0
                self.image1 = self.orclf2
                self.image2 = self.orclf2
            else:
                self.x -= self.dx
                self.image1 = self.orclf1
                self.image2 = self.orclf2
        elif self.dir == 3:
            if self.y <= 0:
                self.y = 0
                self.image1 = self.orcbk2
                self.image2 = self.orcbk2
            else:
                self.y -= self.dy
                self.image1 = self.orcbk1
                self.image2 = self.orcbk2

    def display(self):
        if frameCount % 8 >= 4:
            image(self.image1, self.x, self.y)
        else:
            image(self.image2, self.x, self.y)

class Wall(Sprite):

    def loadPics(self):
        self.pic = loadImage("wall.png")

    def display(self):
```

```
    image(self.pic, self.x, self.y)

class Tree(Sprite):

    def loadPics(self):
        self.pic = loadImage("tree.png")

    def display(self):
        image(self.pic, self.x, self.y)
```

Wenn Ihr nun ein wenig damit herumspielt, werdet Ihr eine kleine Ungenauigkeit bemerken. Nähert sich der Ork von rechts oder von links der Tanne, dann sieht es so aus, als ob er ziemlich weit davor stehenbleiben würde. Das liegt daran, daß sowohl die Seitenansichten des Ork wie auch die der Tanne die 32-Pixel Breite nicht besonders gut ausfüllen. Abhilfe könnte man schaffen, indem man die umgebenden Rechtecke schmäler macht. Das ist noch relativ einfach zu implementieren, macht den Quellcode aber dennoch komplizierter und unübersichtlicher. Da ich aber erst einmal nur das Prinzip der Kollisionserkennung mit überlappenden Rechtecken deutlich machen wollte, dachte ich, daß man im Sinne der Klarheit mit diesem kleinen Handicap leben kann.

Ein Ork im Labyrinth

Nachdem ich im letzten Abschnitt erfolgreich eine Kollisionserkennung implementiert hatte, wollte ich nun das alles auf die Spitze treiben und den kleinen Ork durch ein Labyrinth (genauer: einen Irrgarten) bewegen. Und natürlich sollte er nur dort laufen können, wo es keine Hindernisse gab. Im Endeffekt sollte das Ergebnis so aussehen:

Die einzelnen Klassen im zweiten Reiter (`sprites2.py`) blieben gegenüber dem letzten Abschnitt nahezu unverändert. Lediglich die Klasse `Tree` habe ich durch die Klasse `Lava` ersetzt und der Klasse `Wall` ein anderes Kachelbild verpaßt, was aber beides nur kosmetische Gründe hat. Hier die notwendigen Kacheln:

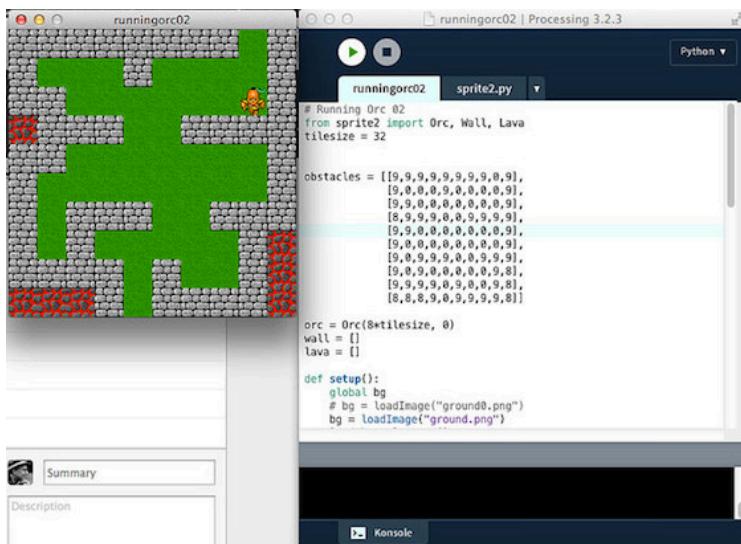
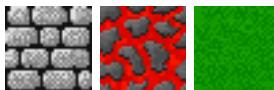


Abbildung 17.4: Ein Ork im Labyrinth



Das Labyrinth habe ich in [Tiled](#) entworfen und die Bilder dafür wieder dem freien ([CC BY 3.0](#)) Angband-Tilesets von [dieser Site](#) entnommen.

Die größten Änderungen gab es im Hauptprogramm. Zu den üblichen Vorbelegungen kam ein zweidimensionales Array `obstacles` hinzu. Dies habe ich mir zurechtgebastelt, in dem ich in *Tiled* das Tileset als CSV exportiert habe und dann in dem [Text-Editor meines Vertrauens](#) mit *Suchen und Ersetzen* die Zahlen ein wenig vereinfacht hatte:

```

obstacles = [[9,9,9,9,9,9,9,9,0,9],
             [9,0,0,0,9,0,0,0,0,9],
             [9,9,0,0,0,0,0,0,0,9],
             [8,9,9,9,0,0,9,9,9,9],
             [9,9,0,0,0,0,0,0,0,9],
             [9,0,0,0,0,0,0,0,0,9],
             [9,9,9,9,0,0,0,0,0,9],
             [9,0,0,0,0,0,0,0,0,9],
             [9,9,9,9,0,0,0,0,0,9],
             [8,8,8,9,0,9,9,9,9,8]]]

```

```
[9,0,9,9,9,0,0,9,9,9],  
[9,0,9,0,0,0,0,0,9,8],  
[9,9,9,9,0,9,0,0,9,8],  
[8,8,8,9,0,9,9,9,9,8]]
```

Wenn Ihr das mit dem Screenshot oben vergleicht, vermutet Ihr sicher sehr schnell, daß die 9 für ein Mauerstück und die 8 für Lava steht, während Null einfach der Fußboden ist. Dann habe ich den Ork und zwei leere Listen, die Mauer und Lava aufnehmen sollen, initialisiert:

```
orc = Orc(8*tilesize, 0)  
wall = []  
lava = []
```

Die `setup()`-Funktion sieht nun so aus:

```
def setup():  
    global bg  
    bg = loadImage("ground0.png")  
    loadObstaclesData()  
    for i in range(len(wall)):  
        wall[i].loadPics()  
    for i in range(len(lava)):  
        lava[i].loadPics()  
    frameRate(30)  
    size(320, 320)  
    orc.loadPics()  
    orc.dx = 2  
    orc.dy = 2
```

Sie ruft die Funktion `loadObstaclesData()` auf, die für die Belegung der beiden Listen `wall` und `lava` zuständig ist

```
def loadObstaclesData():  
    for y in range(10):  
        for x in range(10):  
            if obstacles[y][x] == 9:  
                wall.append(Wall(x*tilesize, y*tilesize))
```

```
    elif obstacles[y][x] == 8:  
        lava.append(Lava(x*tilesize, y*tilesize))
```

und lädt anschließend die entsprechenden Bilder für die Hindernisse.

In der `draw()`-Funktion wird erst das Hintergrundbild geladen, das nur aus einer grünen Grasfläche belegt und dann werden die einzelnen *Obstacles* eingezeichnet:

```
def draw():  
    background(bg)  
    for i in range(len(wall)):  
        wall[i].display()  
    for i in range(len(lava)):  
        lava[i].display()  
    orc.move()  
    for i in range(len(wall)):  
        if orc.checkCollision(wall[i]):  
            if orc.dir == 0:  
                orc.x -= orc.dx  
            elif orc.dir == 1:  
                orc.y -= orc.dy  
            elif orc.dir == 2:  
                orc.x += orc.dx  
            elif orc.dir == 3:  
                orc.y += orc.dy  
            orc.image1 = orc.image2  
    orc.display()
```

Die Bewegung des Orcs wurde aus dem letzten Tutorial unverändert übernommen. Da der Ork niemals mit einem Lava-Feld kollidieren kann (er trifft immer vorher auf eine Mauer) reichte es, die Kollisionsüberprüfung auf die Mauerteile zu beschränken.

Damit ist das Prinzip erklärt, doch es geht noch einfacher. Denn man kann sich das Neuzeichnen der einzelnen Hindernisse bei jedem Durchlauf natürlich ersparen, wenn man sie in dem Hintergrundbild mit aufgenommen hat. Also habe ich das Hintergrundbild mit allen Mauern und dem Lava aus *Tiled* exportiert

und als Hintergrund geladen. Dann sieht die Funktion `setup()` so aus:

```
def setup():
    global bg
    bg = loadImage("ground.png")
    loadObstaclesData()
    frameRate(30)
    size(320, 320)
    orc.loadPics()
    orc.dx = 2
    orc.dy = 2
```

Statt `ground0.png` heißt das Hintergrundbild nun nur noch `ground.png` und enthält nicht nur den Rasen, sondern das gesamte Labyrinth. Auch die `draw()`-Funktion ist um vier Zeilen kürzer geworden:

```
def draw():
    background(bg)
    orc.move()
    for i in range(len(wall)):
        if orc.checkCollision(wall[i]):
            if orc.dir == 0:
                orc.x -= orc.dx
            elif orc.dir == 1:
                orc.y -= orc.dy
            elif orc.dir == 2:
                orc.x += orc.dx
            elif orc.dir == 3:
                orc.y += orc.dy
            orc.image1 = orc.image2
    orc.display()
```

Man erspart sich so die `laodPics()` wie auch die `display()`-Aufrufe der *Obstacle Sprites*, was auch einiges an Rechenzeit spart.

Zum Abschluß das vollständige Programm. Zuerst das Paket `sprite2.py`, das ich in einem separaten Tab untergebracht habe:

```
tw = 32
th = 32
tileSize = 32

class Sprite(object):

    def __init__(self, posX, posY):
        self.x = posX
        self.y = posY
        self.dir = 1
        self.dx = 0
        self.dy = 0

    def checkCollision(self, otherSprite):
        if (self.x < otherSprite.x + tw and otherSprite.x < self.x + tw
            and self.y < otherSprite.y + th and otherSprite.y < self.y + th):
            return True
        else:
            return False

class Orc(Sprite):

    def loadPics(self):
        self.orcrt1 = loadImage("orcrt1.gif")
        self.orcrt2 = loadImage("orcrt2.gif")
        self.orcfr1 = loadImage("orcfr1.gif")
        self.orcfr2 = loadImage("orcfr2.gif")
        self.orclf1 = loadImage("orclf1.gif")
        self.orclf2 = loadImage("orclf2.gif")
        self.orcbk1 = loadImage("orcbk1.gif")
        self.orcbk2 = loadImage("orcbk2.gif")

    def move(self):
        if self.dir == 0:
            if self.x >= width - tileSize:
                self.x = width - tileSize
                self.image1 = self.orcrt2
                self.image2 = self.orcrt1
        else:
            self.x += self.dx
```

```
        self.image1 = self.orcrt1
        self.image2 = self.orcrt2
    elif self.dir == 1:
        if self.y >= height - tileSize:
            self.y = height - tileSize
            self.image1 = self.orcfr2
            self.image2 = self.orcfr2
        else:
            self.y += self.dy
            self.image1 = self.orcfr1
            self.image2 = self.orcfr2
    elif self.dir == 2:
        if self.x <= 0:
            self.x = 0
            self.image1 = self.orclf2
            self.image2 = self.orclf2
        else:
            self.x -= self.dx
            self.image1 = self.orclf1
            self.image2 = self.orclf2
    elif self.dir == 3:
        if self.y <= 0:
            self.y = 0
            self.image1 = self.orcbk2
            self.image2 = self.orcbk2
        else:
            self.y -= self.dy
            self.image1 = self.orcbk1
            self.image2 = self.orcbk2

    def display(self):
        if frameCount % 8 >= 4:
            image(self.image1, self.x, self.y)
        else:
            image(self.image2, self.x, self.y)

class Wall(Sprite):

    def loadPics(self):
        self.pic = loadImage("wall2.png")
```

```
def display(self):
    image(self.pic, self.x, self.y)

class Lava(Sprite):

    def loadPics(self):
        self.pic = loadImage("lava.png")

    def display(self):
        image(self.pic, self.x, self.y)
```

Es hat jetzt schon einiges an Länge angenommen, dafür ist aber das Hauptprogramm immer noch recht kurz:

```
from sprite2 import Orc, Wall, Lava
tilesize = 32
```

```
obstacles = [[9,9,9,9,9,9,9,9,0,9],
             [9,0,0,0,9,0,0,0,0,9],
             [9,9,0,0,0,0,0,0,0,9],
             [8,9,9,9,0,0,9,9,9,9],
             [9,9,0,0,0,0,0,0,0,9],
             [9,0,0,0,0,0,0,0,0,9],
             [9,0,9,9,9,0,0,9,9,9],
             [9,0,9,0,0,0,0,0,9,8],
             [9,9,9,9,0,9,0,0,9,8],
             [8,8,8,9,0,9,9,9,9,8]]
```

```
orc = Orc(8*tilesize, 0)
wall = []
lava = []

def setup():
    global bg
    bg = loadImage("ground.png")
    loadObstaclesData()
    frameRate(30)
    size(320, 320)
```

```
orc.loadPics()
orc.dx = 2
orc.dy = 2

def draw():
    background(bg)
    orc.move()
    for i in range(len(wall)):
        if orc.checkCollision(wall[i]):
            if orc.dir == 0:
                orc.x -= orc.dx
            elif orc.dir == 1:
                orc.y -= orc.dy
            elif orc.dir == 2:
                orc.x += orc.dx
            elif orc.dir == 3:
                orc.y += orc.dy
            orc.image1 = orc.image2
    orc.display()

def keyPressed():
    if keyPressed and key == CODED:
        if keyCode == RIGHT:
            orc.dir = 0
        elif keyCode == DOWN:
            orc.dir = 1
        elif keyCode == LEFT:
            orc.dir = 2
        elif keyCode == UP:
            orc.dir = 3

def loadObstaclesData():
    for y in range(10):
        for x in range(10):
            if obstacles[y][x] == 9:
                wall.append(Wall(x*tilesize, y*tilesize))
            elif obstacles[y][x] == 8:
                lava.append(Lava(x*tilesize, y*tilesize))
```

Wenn Ihr das Programm laufen läßt, werdet Ihr feststellen, daß

ich kurz vor dem Ausgang unten eine kleine Gemeinheit eingebaut habe und es gar nicht so einfach ist, den Ork dorthin zu lotsen. Er will partout 32 Pixel breit sein und macht sich nicht schmäler, daher muß man die Drehung nach unten genau abpassen. Aber es ist nicht unmöglich, ich habe es probiert und geschafft.

Die Bilder des Orks stammen aus der ebenfalls freien ([CC BY 3.0](#)) [Sprite-Sammlung von Philipp Lenssen](#) (über 700 animierte Avatare in der klassischen Größe von 32x32 Pixeln) und Ihr könnt sie in [diesem Tutorial][15] finden.

Alle Quelltexte und Bilder gibt es übrigens auch immer aktuell im [GitHub-Repo](#) zu dieser kleinen Tutorial-Reihe.

Der autonome Ork

Nun sind Orks, wie ich sie in den letzten Beiträgen über den Bildschirm habe wuseln lassen, normalerweise nicht die Figuren, mit denen der Spieler spielt. Er leitet seinen Held, einen *Hero* durch die Spielwelt. Orks und andere Monster hingegen sind meist computergesteuerte Spielfiguren, sogenannte NPCs (*Non Player Characters*). Daher habe ich in dieser Folge einen spielbaren Helden eingebaut und der Ork bewegt sich mehr oder weniger autonom durch das Spielfenster.

Dafür habe ich dann erst einmal die Klasse Hero in den zweiten Tab (den ich dieses Mal `sprite3.py` genannt habe) eingefügt:

```
class Hero(Sprite):  
  
    def loadPics(self):  
        self.mnv1rt1 = loadImage("mnv1rt1.gif")  
        self.mnv1rt2 = loadImage("mnv1rt2.gif")  
        self.mnv1fr1 = loadImage("mnv1fr1.gif")  
        self.mnv1fr2 = loadImage("mnv1fr2.gif")  
        self.mnv1lf1 = loadImage("mnv1lf1.gif")  
        self.mnv1lf2 = loadImage("mnv1lf2.gif")  
        self.mnv1bk1 = loadImage("mnv1bk1.gif")  
        self.mnv1bk2 = loadImage("mnv1bk2.gif")
```



Abbildung 17.5:

```
def move(self):
    if self.dir == 0:
        if self.x >= width - tileSize:
            self.x = width - tileSize
            self.image1 = self.mnv1rt2
            self.image2 = self.mnv1rt2
        else:
            self.x += self.dx
            self.image1 = self.mnv1rt1
            self.image2 = self.mnv1rt2
    elif self.dir == 1:
        if self.y >= height - tileSize:
            self.y = height - tileSize
            self.image1 = self.mnv1fr2
            self.image2 = self.mnv1fr2
        else:
            self.y += self.dy
            self.image1 = self.mnv1fr1
            self.image2 = self.mnv1fr2
    elif self.dir == 2:
        if self.x <= 0:
            self.x = 0
            self.image1 = self.mnv1lf2
```

```
        self.image2 = self.mnv1lf2
    else:
        self.x -= self.dx
        self.image1 = self.mnv1lf1
        self.image2 = self.mnv1lf2
elif self.dir == 3:
    if self.y <= 0:
        self.y = 0
        self.image1 = self.mnv1bk2
        self.image2 = self.mnv1bk2
    else:
        self.y -= self.dy
        self.image1 = self.mnv1bk1
        self.image2 = self.mnv1bk2

def display(self):
    if frameCount % 8 >= 4:
        image(self.image1, self.x, self.y)
    else:
        image(self.image2, self.x, self.y)
```

Sie unterscheidet sich – bis auf die Bildchen – kaum von der bisherigen Orc-Klasse. Auch diese Bildchen stammen aus der ebenfalls freien ([CC BY 3.0](#)) [Sprite-Sammlung von Philipp Lenssen](#) (über 700 animierte Avatare in der klassischen Größe von 32x32 Pixeln) und hier sind sie, damit Ihr sie herunterladen und verwenden könnt:



Den Hintergrund habe ich wieder mit Tiled erstellt und die Bilder dafür wieder dem ebenfalls freien ([CC BY 3.0](#)) Angband-Tilesets von [dieser Site](#) entnommen. Nach einem Export als CSV-Datei und ein wenig *Suchen und Ersetzen* kam dann dieses Terrain zustande:

Wie man leicht sieht, haben alle Hindernisse einen Wert > 5 , wodurch man sie recht einfach in einer Liste zusammenfassen kann, was im Anschluß auch die Kollisionserkennung erleichtert:

```
def loadObstaclesData():
    for y in range(10):
        for x in range(20):
            if terrain[y][x] > 5:
                obstacles.append(Obstacle(x*tilesize, y*tilesize))
```

Und da alle Hindernisse ja schon im Hintergrundbild eingezeichnet sind, braucht man sie auch nicht mehr einzeln zu zeichnen und zu lokalisieren. Es reicht, wenn man die Position eines Hindernisses kennt, egal ob es ein Sumpf, ein Fels, ein Baum, eine Tanne oder eine Mauer ist.

Die Funktion `keyPressed()` ändert jetzt nicht mehr die Laufrichtung des Orks, sondern die unseres Helden. Der Ork bewegt sich selbstständig und ändert die Richtung, sobald er auf eine Hindernis trifft

```
orc.move()
for i in range(len(obstacles)):
    if orc.checkCollision(obstacles[i]):
        if orc.dir == 0:
            orc.x -= orc.dx
            orc.dir = int(random(4))
        elif hero.dir == 1:
            orc.y -= orc.dy
            orc.dir = int(random(4))
        elif hero.dir == 2:
            orc.x += orc.dx
```

```
        orc.dir = int(random(4))
    elif hero.dir == 3:
        orc.y += orc.dy
        orc.dir = int(random(4))
    orc.image1 = orc.image2
```

oder eines der Fensterränder erreicht hat.

Jetzt der Vollständigkeit halber das ganze Skript. Erst einmal alles, was ich in dem Reiter `sprites3.py` eingetippt habe:

```
tw = 32
th = 32
tileSize = 32

class Sprite(object):

    def __init__(self, posX, posY):
        self.x = posX
        self.y = posY
        self.dir = 1
        self.dx = 0
        self.dy = 0

    def checkCollision(self, otherSprite):
        if (self.x < otherSprite.x + tw and otherSprite.x < self.x + th
            and self.y < otherSprite.y + th and otherSprite.y < self.y + tw):
            return True
        else:
            return False
    class Hero(Sprite):

        def loadPics(self):
            self.mnv1rt1 = loadImage("mnv1rt1.gif")
            self.mnv1rt2 = loadImage("mnv1rt2.gif")
            self.mnv1fr1 = loadImage("mnv1fr1.gif")
            self.mnv1fr2 = loadImage("mnv1fr2.gif")
            self.mnv1lf1 = loadImage("mnv1lf1.gif")
            self.mnv1lf2 = loadImage("mnv1lf2.gif")
            self.mnv1bk1 = loadImage("mnv1bk1.gif")
```

```
self.mnv1bk2 = loadImage("mnv1bk2.gif")

def move(self):
    if self.dir == 0:
        if self.x >= width - tileSize:
            self.x = width - tileSize
            self.image1 = self.mnv1rt2
            self.image2 = self.mnv1rt2
    else:
        self.x += self.dx
        self.image1 = self.mnv1rt1
        self.image2 = self.mnv1rt2
elif self.dir == 1:
    if self.y >= height - tileSize:
        self.y = height - tileSize
        self.image1 = self.mnv1fr2
        self.image2 = self.mnv1fr2
    else:
        self.y += self.dy
        self.image1 = self.mnv1fr1
        self.image2 = self.mnv1fr2
elif self.dir == 2:
    if self.x <= 0:
        self.x = 0
        self.image1 = self.mnv1lf2
        self.image2 = self.mnv1lf2
    else:
        self.x -= self.dx
        self.image1 = self.mnv1lf1
        self.image2 = self.mnv1lf2
elif self.dir == 3:
    if self.y <= 0:
        self.y = 0
        self.image1 = self.mnv1bk2
        self.image2 = self.mnv1bk2
    else:
        self.y -= self.dy
        self.image1 = self.mnv1bk1
        self.image2 = self.mnv1bk2
```

```
def display(self):
    if frameCount % 8 >= 4:
        image(self.image1, self.x, self.y)
    else:
        image(self.image2, self.x, self.y)

class Orc(Sprite):

    def loadPics(self):
        self.orcrt1 = loadImage("orcrt1.gif")
        self.orcrt2 = loadImage("orcrt2.gif")
        self.orcfr1 = loadImage("orcfr1.gif")
        self.orcfr2 = loadImage("orcfr2.gif")
        self.orclf1 = loadImage("orclf1.gif")
        self.orclf2 = loadImage("orclf2.gif")
        self.orcbk1 = loadImage("orcbk1.gif")
        self.orcbk2 = loadImage("orcbk2.gif")

    def move(self):
        if self.dir == 0:
            if self.x >= width - tileSize:
                self.x = width - tileSize
                self.dir = int(random(4))
            else:
                self.x += self.dx
                self.image1 = self.orcrt1
                self.image2 = self.orcrt2
        elif self.dir == 1:
            if self.y >= height - tileSize:
                self.y = height - tileSize
                self.y -= self.dy
                self.dir = int(random(4))
            else:
                self.y += self.dy
                self.image1 = self.orcfr1
                self.image2 = self.orcfr2
        elif self.dir == 2:
            if self.x <= 0:
                self.x = 0
                self.dir = int(random(4))
```

```

        else:
            self.x -= self.dx
            self.image1 = self.orclf1
            self.image2 = self.orclf2
        elif self.dir == 3:
            if self.y <= 0:
                self.y = 0
                self.dir = int(random(4))
            else:
                self.y -= self.dy
                self.image1 = self.orcbk1
                self.image2 = self.orcbk2

    def display(self):
        if frameCount % 8 >= 4:
            image(self.image1, self.x, self.y)
        else:
            image(self.image2, self.x, self.y)

class Obstacle(Sprite):

    def loadPics(self):
        self.pic = loadImage("wall.png")

    def display(self):
        image(self.pic, self.x, self.y)

```

Es ist noch umfangreicher geworden, aber eigentlich ist alles aus den vorherigen Tutorials bekannt. Die Klasse `Obstacle()` ist eigentlich überflüssig, da ihre Methoden nicht benötigt werden, man könnte stattdessen direkt die Klasse `Sprite()` nutzen. Sie schafft in meinen Augen aber mehr Klarheit und daher habe ich sie dennoch – mit Dummy-Methoden – stehen lassen.

Auch das Hauptprogramm wird langsam umfangreicher, ist aber immer noch übersichtlich. Es sieht nun so aus:

```
# Hero 01
from sprites3 import Hero, Orc, Obstacle
```

```
tilesize = 32

terrain = [[0,0,0,0,0,0,8,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,7],  
           [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,7,7],  
           [0,0,0,0,0,8,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,7],  
           [0,0,0,0,0,8,8,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],  
           [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,6,6],  
           [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,6,6,0],  
           [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,6,6,0,0],  
           [6,9,0,9,9,0,0,0,0,0,0,0,0,0,0,0,0,8,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],  
           [9,6,9,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],  
           [6,6,6,6,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]]  
  
hero = Hero(16*tilesize, 0)  
orc = Orc(4*tilesize, 0)  
obstacles = []  
  
def setup():  
    global bg  
    bg = loadImage("terrain.png")  
    loadObstaclesData()  
    frameRate(30)  
    size(640, 320)  
    hero.loadPics()  
    orc.loadPics()  
    hero.dx = 2  
    hero.dy = 2  
    orc.dx = 2  
    orc.dy = 2  
  
def draw():  
    background(bg)  
    hero.move()  
    for i in range(len(obstacles)):  
        if hero.checkCollision(obstacles[i]):  
            if hero.dir == 0:  
                hero.x -= hero.dx  
            elif hero.dir == 1:  
                hero.y -= hero.dy  
            elif hero.dir == 2:  
                hero.x += hero.dx  
            elif hero.dir == 3:  
                hero.y += hero.dy
```

```
        hero.x += hero.dx
    elif hero.dir == 3:
        hero.y += hero.dy
        hero.image1 = hero.image2
hero.display()
orc.move()
for i in range(len(obstacles)):
    if orc.checkCollision(obstacles[i]):
        if orc.dir == 0:
            orc.x -= orc.dx
            orc.dir = int(random(4))
        elif hero.dir == 1:
            orc.y -= orc.dy
            orc.dir = int(random(4))
        elif hero.dir == 2:
            orc.x += orc.dx
            orc.dir = int(random(4))
        elif hero.dir == 3:
            orc.y += orc.dy
            orc.dir = int(random(4))
        orc.image1 = orc.image2
orc.display()

def keyPressed():
    if keyPressed and key == CODED:
        if keyCode == RIGHT:
            hero.dir = 0
        elif keyCode == DOWN:
            hero.dir = 1
        elif keyCode == LEFT:
            hero.dir = 2
        elif keyCode == UP:
            hero.dir = 3

def loadObstaclesData():
    for y in range(10):
        for x in range(20):
            if terrain[y][x] > 5:
                obstacles.append(Obstacle(x*tilesize, y*tilesize))
```

Caveat

Wenn Ihr das Programm laufen läßt, werdet Ihr feststellen, daß der Ork manchmal kleine Tänzchen veranstaltet oder sogar durch Mauern gehen kann. Und aus der linken, oberen Ecke findet er auch manchmal schwer wieder heraus. Das erste liegt daran, daß ich nicht verhindert habe, daß er nach einer Kollisionserkennung zufällig die gleiche Richtung noch einmal auswählt. Das kann man zum Beispiel verhindern, in dem man bei jeder Kollisionserkennung jeweils eine Liste der »zulässigen« Richtungen erstellt und nur daraus die neue Richtung heraussuchen läßt.

Für das zweite Problem bräuchte der Ork einfach mehr »Intelligenz«. Bisher wechselt er die Richtung nur, wenn er auf einer Hindernis trifft. Anders sähe es aus, wenn man zufallsgesetziert nach etwa jedem 20. Schritt eine Richtungsänderung vornimmt. Die Laufrichtungen des Ork würden dann noch unvorhersehbarer.

Drei Orks und ein Held

In diesem Tutorial, in dem erstmalig auch mehrere Orks auftreten, habe ich die Unstimmigkeiten aus dem [letzten Teil dieser Reihe](#) beseitigt. Die Ränder-Behandlung habe ich dadurch vereinfacht, daß nun die ganze Spielwelt eingezäunt ist¹ und die Tänzchen vor Hindernissen habe ich dadurch eliminiert, daß ich Listen der zulässigen Richtungsänderungen angelegt habe und nur diese per Zufall auswählen lasse:

```
for i in range(len(orc)):
    orc[i].move()
    for j in range(len(wall)):
        if orc[i].checkCollision(wall[j]):
            if orc[i].dir == 0:
                orc[i].x -= orc[i].dx
```

¹Das ist keine Einschränkung, denn der Fensterrand ist ja im Grunde auch nichts anderes als eine undurchdringliche Mauer und im Zweifelsfall macht man das Spielfeld einfach um die Mauerdicke größer.



Abbildung 17.6: Drei Orks

```
        legalMove = [1, 2, 3]
        orc[i].dir = legalMove[int(random(3))]
    elif orc[i].dir == 1:
        orc[i].y -= orc[i].dy
        legalMove = [0, 2, 3]
        orc[i].dir = legalMove[int(random(3))]
    elif orc[i].dir == 2:
        orc[i].x += orc[i].dx
        legalMove = [0, 1, 3]
        orc[i].dir = legalMove[int(random(3))]
    elif orc[i].dir == 3:
        orc[i].y += orc[i].dy
        legalMove = [0, 1, 2]
        orc[i].dir = legalMove[int(random(3))]
    orc[i].display()
```

Außerdem habe ich in der Klasse `Orc` (im Modul `sprite2.py`) den Orks einen zufälligen Richtungswechsel verpaßt, damit sie nicht nur bei einer Kollision mit Hindernissen ihre Richtung ändern und so ihre Bewegungen unvorhersehbarer werden.

```
def move(self):
    if frameCount % int(random(30, 120)) == 0:
        if self.dir == 0:
            legalMove = [1, 2, 3]
            self.dir = legalMove[int(random(3))]
        elif self.dir == 1:
            legalMove = [0, 2, 3]
            self.dir = legalMove[int(random(3))]
        elif self.dir == 2:
            legalMove = [0, 1, 3]
            self.dir = legalMove[int(random(3))]
        elif self.dir == 3:
            legalMove = [0, 1, 2]
            self.dir = legalMove[int(random(3))]
```

Fragt mich nicht, wie ich auf die Werte 30, 120 gekommen bin. Ich habe einfach ein wenig experimentiert und diese brachten in meinen Augen das ansprechendste Ergebnis.

Das einzige sonstige neue ist, daß ich die drei Orks in einer Liste zusammengefaßt habe, so daß sie – wie das Code-Fragment ganz oben zeigt – in einer Schleife abgehandelt werden können.

Der Quellcode

Daher erst einmal der vollständige Quellcode, damit Ihr das Beispiel auch nachvollziehen und -programmieren könnt. Erst einmal das Modul `sprite2.py`, das ich wieder in einem separaten Tab in der Processing-IDE untergebracht habe:

```
tw = 32
th = 32
tileSize = 32

class Sprite(object):

    def __init__(self, posX, posY):
        self.x = posX
        self.y = posY
        self.dir = 1
        self.dx = 0
        self.dy = 0

    def checkCollision(self, otherSprite):
        if (self.x < otherSprite.x + tw and otherSprite.x < self.x + tw
            and self.y < otherSprite.y + th and otherSprite.y < self.y + th):
            return True
        else:
            return False

class Hero(Sprite):

    def loadPics(self):
        self.mnv1rt1 = loadImage("mnv1rt1.gif")
        self.mnv1rt2 = loadImage("mnv1rt2.gif")
        self.mnv1fr1 = loadImage("mnv1fr1.gif")
        self.mnv1fr2 = loadImage("mnv1fr2.gif")
        self.mnv1lf1 = loadImage("mnv1lf1.gif")
```

```
self.mnv1lf2 = loadImage("mnv1lf2.gif")
self.mnv1bk1 = loadImage("mnv1bk1.gif")
self.mnv1bk2 = loadImage("mnv1bk2.gif")

def move(self):
    if self.dir == 0:
        self.x += self.dx
        self.image1 = self.mnv1rt1
        self.image2 = self.mnv1rt2
    elif self.dir == 1:
        self.y += self.dy
        self.image1 = self.mnv1fr1
        self.image2 = self.mnv1fr2
    elif self.dir == 2:
        self.x -= self.dx
        self.image1 = self.mnv1lf1
        self.image2 = self.mnv1lf2
    elif self.dir == 3:
        self.y -= self.dy
        self.image1 = self.mnv1bk1
        self.image2 = self.mnv1bk2

def display(self):
    if frameCount % 8 >= 4:
        image(self.image1, self.x, self.y)
    else:
        image(self.image2, self.x, self.y)

class Orc(Sprite):

    def loadPics(self):
        self.orcrt1 = loadImage("orcrt1.gif")
        self.orcrt2 = loadImage("orcrt2.gif")
        self.orcfr1 = loadImage("orcfr1.gif")
        self.orcfr2 = loadImage("orcfr2.gif")
        self.orclf1 = loadImage("orclf1.gif")
        self.orclf2 = loadImage("orclf2.gif")
        self.orcbk1 = loadImage("orcbk1.gif")
        self.orcbk2 = loadImage("orcbk2.gif")
```

```
def move(self):
    if frameCount % int(random(30, 120)) == 0:
        if self.dir == 0:
            legalMove = [1, 2, 3]
            self.dir = legalMove[int(random(3))]
        elif self.dir == 1:
            legalMove = [0, 2, 3]
            self.dir = legalMove[int(random(3))]
        elif self.dir == 2:
            legalMove = [0, 1, 3]
            self.dir = legalMove[int(random(3))]
        elif self.dir == 3:
            legalMove = [0, 1, 2]
            self.dir = legalMove[int(random(3))]
    if self.dir == 0:
        self.x += self.dx
        self.image1 = self.orcrt1
        self.image2 = self.orcrt2
    elif self.dir == 1:
        self.y += self.dy
        self.image1 = self.orcfr1
        self.image2 = self.orcfr2
    elif self.dir == 2:
        self.x -= self.dx
        self.image1 = self.orclf1
        self.image2 = self.orclf2
    elif self.dir == 3:
        self.y -= self.dy
        self.image1 = self.orcbk1
        self.image2 = self.orcbk2

def display(self):
    if frameCount % 8 >= 4:
        image(self.image1, self.x, self.y)
    else:
        image(self.image2, self.x, self.y)

class Wall(Sprite):

    def loadPics(self):
```

```
self.pic = loadImage("wall.png")  
  
def display(self):  
    image(self.pic, self.x, self.y)
```

Es ist gegenüber dem letzten Mal ein wenig einfacher geworden, weil die Ränderbehandlung entfallen ist. Das Hauptprogramm hat allerdings an Komplexität deutlich zugenommen:

```
from sprite2 import Hero, Orc, Wall  
tilesize = 32  
  
dungeon = [[9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,7,7,9],  
           [8,9,0,0,0,9,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,9],  
           [8,9,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,9],  
           [9,9,9,9,9,9,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,9,9,9,9],  
           [9,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,9,0,0,0,9],  
           [9,0,0,0,0,0,0,0,9,9,0,0,0,0,0,0,0,0,0,9,0,0,0,0,9],  
           [9,0,0,0,0,0,0,9,0,0,0,0,0,0,0,0,0,0,0,9,0,0,0,0,9],  
           [9,9,9,9,9,9,9,9,9,0,0,0,0,0,0,0,0,0,9,0,0,0,0,9],  
           [8,9,0,0,0,0,0,0,9,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,9],  
           [8,9,0,0,0,0,0,9,0,0,0,0,0,0,0,0,0,0,0,0,9,0,0,0,9],  
           [8,9,0,0,0,0,0,9,0,0,9,9,9,9,9,9,9,9,9,9,9,9,9,9],  
           [9,9,0,0,0,0,0,9,0,9,0,9,0,0,0,0,0,0,0,0,0,0,0,0,9],  
           [9,0,0,0,0,0,0,0,9,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,9],  
           [9,9,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,5],  
           [8,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9]]  
  
wall = []  
hero = Hero(18*tilesize, 13*tilesize)  
orc = []  
orc.append(Orc(2*tilesize, 12*tilesize))  
orc.append(Orc(3*tilesize, 2*tilesize))  
orc.append(Orc(16*tilesize, 4*tilesize))  
  
def setup():  
    global bg  
    bg = loadImage("dungeon.png")  
    loadDungeonData()  
    frameRate(30)
```

```
size(640, 480)
hero.loadPics()
hero.dx = 2
hero.dy = 2
hero.dir = 2
for i in range(len(orc)):
    orc[i].loadPics()
    orc[i].dx = 2
    orc[i].dy = 2
    orc[i].dir = 0

def draw():
    background(bg)
    hero.move()
    for j in range(len(wall)):
        if hero.checkCollision(wall[j]):
            if hero.dir == 0:
                hero.x -= hero.dx
            elif hero.dir == 1:
                hero.y -= hero.dy
            elif hero.dir == 2:
                hero.x += hero.dx
            elif hero.dir == 3:
                hero.y += hero.dy
            hero.image1 = hero.image2
    hero.display()

    for i in range(len(orc)):
        orc[i].move()
        for j in range(len(wall)):
            if orc[i].checkCollision(wall[j]):
                if orc[i].dir == 0:
                    orc[i].x -= orc[i].dx
                    legalMove = [1, 2, 3]
                orc[i].dir = legalMove[int(random(3))]
                elif orc[i].dir == 1:
                    orc[i].y -= orc[i].dy
                    legalMove = [0, 2, 3]
                orc[i].dir = legalMove[int(random(3))]
```

```
        elif orc[i].dir == 2:
            orc[i].x += orc[i].dx
            legalMove = [0, 1, 3]
            orc[i].dir = legalMove[int(random(3))]
        elif orc[i].dir == 3:
            orc[i].y += orc[i].dy
            legalMove = [0, 1, 2]
            orc[i].dir = legalMove[int(random(3))]
    orc[i].display()

def loadDungeonData():
    for y in range(15):
        for x in range(20):
            if dungeon[y][x] >= 5:
                wall.append(Wall(x*tilesize, y*tilesize))

def keyPressed():
    if keyPressed and key == CODED:
        if keyCode == RIGHT:
            hero.dir = 0
        elif keyCode == DOWN:
            hero.dir = 1
        elif keyCode == LEFT:
            hero.dir = 2
        elif keyCode == UP:
            hero.dir = 3
```

Den Raum habe ich wieder in [Tiled](#) erstellt und einmal als Bild und einmal als CSV-Datei exportiert. Aus dieser CSV-Datei habe ich dann obiges Array gebastelt, aus dem man die Struktur des *Dungeon* ziemlich gut ablesen kann. Damit das mit dem Ablesen aber auch wirklich funktioniert, mußte ich gegenüber dem gewohnten Brauch x und y vertauschen (weil man sonst den Kopf immer schräg legen müßte).

Meditieren mit den Orks

Ich habe mir bei der Platzierung der Orks beim Programmstart etwas gedacht. Läßt man das Programmchen nämlich eine Weile

laufen, dann werdet Ihr feststellen, daß die beiden Orks unten kaum Probleme haben, ihrem ursprünglichen Raum zu entkommen, während der Ork in dem kleinen Zimmerchen rechts wie ein im Zoo eingesperrter Tiger meist ziemlich lange dort auf und ab tigert, bis er endlich entkommen kann (irgendwann entkommt aber jeder). Um mir das anzuschauen, manövriere ich den Helden gerne in das kleine Räumchen oben links und lasse ihn dort einfach stehen (noch passiert ja nichts, wenn er von einem Ork entdeckt wird).

Wenn man den Sketch dann lange genug laufen läßt, verirrt sich hin und wieder auch ein Ork zurück in das Gefängniszimmer und braucht natürlich ebenfalls seine Zeit, bis er wieder entkommt. Ein chinesisches Restaurant in der Nähe meines Arbeitsplatzes hat kleine Aquarien mit Guppies im Gastraum. Wenn ich dort essen gehe, setze ich mich gerne in die Nähe der Aquarien und schaue den Fischen beim Umherwieseln zu. Ihre Bewegungen sind denen der Orks in diesem Skript ziemlich ähnlich und daher wirkt dieser Sketch ähnlich meditativ auf mich. Glaubt mir, ich habe gestern abend fast eine Stunde vor dem Rechner gesessen und den Orks ganz entspannt beim Wuseln zugesehen.

Credits

Die Bilder des Helden und der Orks entstammen wieder der freien ([CC BY 3.0](#)) [Sprite-Sammlung von Philipp Lenssen](#) (über 700 animierte Avatare in der klassischen Größe von 32x32 Pixeln). Den Hintergrund habe ich wie bei den anderen Beispielen auch mit [Tiled](#) erstellt und die Tiles dem ebenfalls freien ([CC0](#)) [Dungeon Crawl Tileset](#) entnommen.

Alle Skripte und Assets zu dieser kleinen [Processing.py](#)-Serie könnt Ihr natürlich [auf GitHub](#) finden. Also habt Spaß damit.

Kapitel 18

Exkurs Rauhnächte: Spaß mit Processing.py

Jeden Winter in den [Rauhnächten](#) treffen sich die Geister mit den Engeln, um gemeinsam zu tanzen und ihrer Freude Ausdruck zu verleihen, daß die Tage nun wieder länger werden. Ich habe das mal in einem kleinen Processing.py-Sketch nachempfunden.

Hier treffen sich Geist und Engelchen vor dem Tor einer Waldkirche oder -kapelle um anmutig Euren Mauszeiger zu umtanzen. Dabei habe ich eine Technik verwendet, die [Easing](#) genannt wird. Dabei folgt zwar der *Sprite* dem Mauszeiger, doch bei jedem Durchlauf wird die Distanz zwischen dem Mauszeiger und dem Sprite berechnet und mit einer kleinen Konstante (zum Beispiel 0.05) multipliziert.

```
easing1 = 0.01  
easing2 = 0.05
```

[...]

```
targetX = mouseX
```



Abbildung 18.1: Screenshot

```
targetY = mouseY  
[...]  
  
engelX += (targetX - engelX) * easing1  
engelY += (targetY - engelY) * easing1  
  
[...]  
  
ghostX += (targetX - ghostX) * easing2  
ghostY += (targetY - ghostY) * easing2
```

Durch die beiden unterschiedlichen Konstanten `easing1` und `easing2` habe ich erreicht, daß die beiden Sprites in unterschiedlichen Geschwindigkeiten und Abständen den Mauszeiger umtanzen.

Das vollständige Programm

```
easing1 = 0.01  
easing2 = 0.05
```

```
ghostX = 240
ghostY = 200
engelX = 200
engelY = 240

def setup():
    global bg, ghost, engel
    bg = loadImage("koken.jpg")
    ghost = loadImage("ghost.png")
    engel = loadImage("engel.png")
    frameRate(30)
    size(560, 320)

def draw():
    global ghostX, ghostY, engelX, engelY
    background(bg)
    targetX = mouseX
    targetY = mouseY

    engelX += (targetX - engelX) * easing1
    if engelX >= (width - 36):
        engelX = width - 36
    elif engelX <= 0:
        engelX = 0;
    engelY += (targetY - engelY) * easing1
    if engelY >= (height - 36):
        engelY = height - 36
    elif engelY <= 0:
        engelY = 0
    image(engel, engelX, engelY)

    ghostX += (targetX - ghostX) * easing2
    if ghostX >= (width - 36):
        ghostX = width - 36
    elif ghostX <= 0:
        ghostX = 0;
    ghostY += (targetY - ghostY) * easing2
    if ghostY >= (height - 36):
        ghostY = height - 36
    elif ghostY <= 0:
```

```
ghostY = 0  
image(ghost, ghostX, ghostY)
```

Wie Ihr seht, ist da eigentlich nicht viel mehr. Außer dem *Easing* habe ich mit den Randabfragen nur noch dafür gesorgt, daß die beiden Sprites bei ihrem Tänzchen das Programmfenster nicht verlassen.

Maus versus Tastatur

Die Abfrage der Mausposition funktioniert bei Processing(.py) im Gegensatz zur Tastaturabfrage auch dann, wenn das Programmfenster nicht den Fokus besitzt, sondern auch, wenn die IDE oder andere Fenster noch im Vordergrund sind. Denn die IDE muß sich die Maus ja auch nicht mit dem Programmfenster teilen, die Tastatur aber doch.

Caveat

Meine ursprüngliche Idee war, statt der Bilder Emojis für Geist und Engel einzusetzen, und zwar diese: und , wie ich eine ähnliche Idee schon einmal in einem [P5.js-Sketch](#) umgesetzt hatte. Dann fiel mir jedoch ein, daß Processing.py ja auf [Jython](#) aufsetzt und es daher mit der UTF-8-Unterstützung im Allgemeinen und der Nutzung von Emojis im Besonderen schwierig werden kann (ein aktuelles Jython ist zwar nahezu kompatibel zu Python 2.7, aber eben nicht zu Python 3). Daher habe ich auf die freien ([CC-BY 4.0](#)) [Twemoji-Graphiken von Twitter](#) zurückgegriffen. Hier sind sie, falls Ihr das Beispiel nachprogrammieren wollt.



Der Geist sieht zwar nicht ganz so fröhlich aus, wie das Emoji von Apple und anderen, aber es ist vielleicht realistischer. Wenn Männer tanzen (müssen), dann verziehen sie halt oft schmerhaft ihr Gesicht.

Weitere Credits

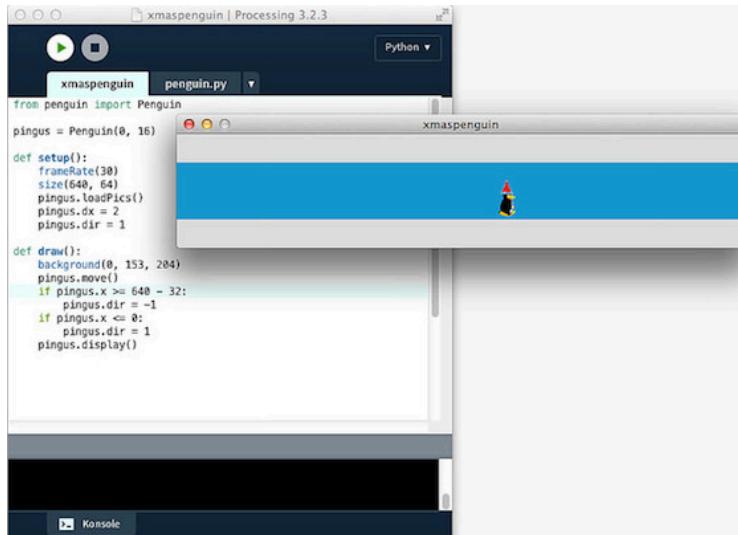


Das Tor zur Waldkirche ist ein Ausschnitt aus einem [Gemälde von Edmund Koken](#), das – da der Maler 1872 verstorben ist – hinreichend gemeinfrei sein dürfte, so daß Ihr das Bild gefahrlos verwenden könnt.

Kapitel 19

Exkurs: Walking Pingus

Die älteren unter Euch können sich sicher noch an das Computerspiel [Lemminge](#) von 1991 erinnern, in dem man eine Horde kleiner, aber dummer Geschöpfe mit grünen Haaren und blauem Anzug, die immer stur geradeaus liefen, davon abhalten mußte, ins Verderben zu rennen und sie zum rettenden Ausgang führen. *Ingo Runke* hatte einen freien (GPL) Klon gebastelt, den er in Anspielung auf das Linus-Maskottchen *Tux Pingus* nannte und in dem man – statt der Lemming – kleine Pinguine retten mußte. Die Pinguine bewegten sich in jede Richtung mit 8 Bildern und da ich mal etwas anderes als zappelige Orks mit [Processing.py](#) auf den Bildschirm bringen wollte, habe ich mich mal an den Pinguinen versucht.



Nach den bisherigen Helden- und Orks-Tutorials ist es nur eine Fingerübung. Auf eine Oberklasse **Sprites** habe ich dieses Mal verzichtet, der Pingus muß ja nur mit den Fensterrändern kommunizieren. Wie schon bisher existieren neben der Initialisierung drei Methoden, nämlich `loadPics()`, `move()` und `display()`. Die ersten beiden Methoden sind eigentlich trivial und nur deswegen so umfangreich, weil sie jeweils mit 16 Bildchen umgehen müssen. Lediglich bei der `display()`-Methode muß man aufpassen und rückwärts zählen, da andersherum die Schleife nach dem ersten Male immer sofort verlassen wird:

```

def display(self):
    if frameCount % 32 >= 28:
        image(self.image1, self.x, self.y)
    elif frameCount % 32 >= 24:
        image(self.image2, self.x, self.y)
    elif frameCount % 32 >= 20:
        image(self.image3, self.x, self.y)
    elif frameCount % 32 >= 16:
        image(self.image4, self.x, self.y)
    elif frameCount % 32 >= 12:
        image(self.image5, self.x, self.y)
    elif frameCount % 32 >= 8:
  
```

```
        image(self.image6, self.x, self.y)
elif frameCount % 32 >= 4:
    image(self.image7, self.x, self.y)
else:
    image(self.image8, self.x, self.y)
```

Die einzelnen Bilder habe ich wieder mit [Tiled](#) aus dem Spritesheet ausgeschnitten. Dabei ist zu beachten, daß die einzelnen Pinguine eine Tilegröße von 32x44 Pixeln besitzen.



Abbildung 19.1: Spritesheet

Der Quellcode

Wie gesagt, es ist nur eine kleine Fingerübung. Hier erst einmal das Modul `penguin.py`, das nur die Klasse `Penguin` enthält:

```
class Penguin(object):

    def __init__(self, posX, posY):
        self.x = posX
        self.y = posY
        self.dir = 0
        self.dx = 0

    def loadPics(self):
        # nach rechts laufen
        self.pingrt1 = loadImage("pingrt1.png")
        self.pingrt2 = loadImage("pingrt2.png")
```

```
self.pingrt3 = loadImage("pingrt3.png")
self.pingrt4 = loadImage("pingrt4.png")
self.pingrt5 = loadImage("pingrt5.png")
self.pingrt6 = loadImage("pingrt6.png")
self.pingrt7 = loadImage("pingrt7.png")
self.pingrt8 = loadImage("pingrt8.png")
# nach links laufen
self.pinglft1 = loadImage("pinglft1.png")
self.pinglft2 = loadImage("pinglft2.png")
self.pinglft3 = loadImage("pinglft3.png")
self.pinglft4 = loadImage("pinglft4.png")
self.pinglft5 = loadImage("pinglft5.png")
self.pinglft6 = loadImage("pinglft6.png")
self.pinglft7 = loadImage("pinglft7.png")
self.pinglft8 = loadImage("pinglft8.png")

def move(self):
    if self.dir == 1:
        self.x += self.dx
        self.image1 = self.pingrt1
        self.image2 = self.pingrt2
        self.image3 = self.pingrt3
        self.image4 = self.pingrt4
        self.image5 = self.pingrt5
        self.image6 = self.pingrt6
        self.image7 = self.pingrt7
        self.image8 = self.pingrt8
    elif self.dir == -1:
        self.x -= self.dx
        self.image1 = self.pinglft1
        self.image2 = self.pinglft2
        self.image3 = self.pinglft3
        self.image4 = self.pinglft4
        self.image5 = self.pinglft5
        self.image6 = self.pinglft6
        self.image7 = self.pinglft7
        self.image8 = self.pinglft8

def display(self):
    if frameCount % 32 >= 28:
```

```
        image(self.image1, self.x, self.y)
    elif frameCount % 32 >= 24:
        image(self.image2, self.x, self.y)
    elif frameCount % 32 >= 20:
        image(self.image3, self.x, self.y)
    elif frameCount % 32 >= 16:
        image(self.image4, self.x, self.y)
    elif frameCount % 32 >= 12:
        image(self.image5, self.x, self.y)
    elif frameCount % 32 >= 8:
        image(self.image6, self.x, self.y)
    elif frameCount % 32 >= 4:
        image(self.image7, self.x, self.y)
    else:
        image(self.image8, self.x, self.y)
```

Das Hauptprogramm ist extrem kurz, aber der Pinguin watschelt ja auch nur von links nach rechts und wieder zurück:

```
from penguin import Penguin

pingus = Penguin(0, 16)

def setup():
    frameRate(30)
    size(640, 64)
    pingus.loadPics()
    pingus.dx = 1
    pingus.dir = 1

def draw():
    background(0, 153, 204)
    pingus.move()
    if pingus.x >= 640 - 32:
        pingus.dir = -1
    if pingus.x <= 0:
        pingus.dir = 1
    pingus.display()
```

Im Gegensatz zu den Orks aus den vorherigen Programmen bewegt sich Pingus mit jedem Frame nur einen Pixel weiter. Denn durch die vielen Bilder ist die Bewegung doch so exakt, daß es bei schnellerem Vorangehen aussieht, als ob Pingus auf Eis schlittert (bei Pinguinen sicher nicht unüblich, aber in diesem Fall nicht gewollt). Es ist eben kein *Running Ork* sondern nur ein *Walking Pingus*.

Wenn Ihr das nachprogrammiert und laufen läßt, werdet Ihr sehen, daß das schon sehr nett aussieht, besonders auch wie die Zipfelmütze des kleinen Pinguins im Takt hin und her wippt.

Pingus Links

Wenn Ihr Pingus spielen wollt, das Spiel gibt es trotz seines Alters immer noch [hier für Windows, Mac und Linux zum freien Download](#). Auf meinen Macs läuft es auch noch, macht Spaß und die [Quellen könnt Ihr auf GitHub](#) finden.

Kapitel 20

Das Avoider Game

Game Stage 1

Zum Abschluß meiner kleinen, geplanten Tutorial-Reihe zu Processing.py, dem Python-Mode von Processing, möchte ich mit und für Euch ein kleines, vollständiges Spieleprojekt programmieren. Es basiert zum einen auf dem »[AS3 Avoider Game Tutorial](#)«, das *Michael James Williams* für ActionScript 3 und Flash geschrieben hat und das *Michael Haungs* dann in seinem Buch »[Creative Greenfoot](#)« nach Java und Greenfoot portierte. Zum anderen habe ich es noch mit Ideen aus einem Programm aus dem wundervollen Buch »[Game Programming](#)«, einem PyGame-Tutorial von *Andy Harris* aufgepeppt, in dem ein Postflieger Inseln anfliegen, aber Wolken vermeiden muß.

Die Spiel-Idee



Ziel des Spiels ist es, daß der Held seinen von oben herabregnenden Feinden ausweichen muß. Doch in diesem Spiel ist nichts so, wie es scheint: Die Feinde sind lächelnde Smileys und unser Held ist ein häßlicher Totenkopfschädel. Im ersten Stadium möchte ich nur dieses einfache Spieleprinzip und einen *Highscore* implementieren, in den nächsten Tutorials möchte ich dieses mit weiteren Variationen und Ideen zu einem interessanteren Spiel ausbauen.

Das Sprite-Modul

Wie schon mehrmals praktiziert, lege ich erst einmal einen separaten Tab **spite.py** an, der in der Hauptsache die Klasse **Sprite** und die davon abgeleiteten Unterklassen **Skull** und **Smiley** beherbergt:

```

from random import randint

tw = th = 36

class Sprite():
    def __init__(self, posX, posY):

```

```
        self.x = posX
        self.y = posY
        self.dx = 0
        self.dy = 0
        self.score = 0
        self.over = False

    def checkCollision(self, otherSprite):
        if (self.x < otherSprite.x + tw and otherSprite.x < self.x
            and self.y < otherSprite.y + th and otherSprite.y < self.y):
            return True
        else:
            return False

class Skull(Sprite):

    def loadPics(self):
        self.im1 = loadImage("skull.png")

    def move(self):
        self.x = mouseX
        if self.x <= 0:
            self.x = 0
        elif self.x >= width-tw:
            self.x = width - tw

    def display(self):
        image(self.im1, self.x, self.y)

class Smiley(Sprite):

    def loadPics(self):
        self.im0 = loadImage("smiley0.png")
        self.im1 = loadImage("smiley1.png")
        self.im2 = loadImage("smiley4.png")

    def move(self):
        self.over = False
        self.y += self.dy
        if self.y >= height:
```

```

        self.over = True
        self.y = -randint(50, 250)
        self.x = randint(0, width-tw)
        self.dy = randint(2, 10)

    def display(self):
        if (self.y > -30) and (self.y <= 200):
            image(self.im0, self.x, self.y)
        elif (self.y > 200) and (self.y <= 360):
            image(self.im1, self.x, self.y)
        elif (self.y > 360):
            image(self.im2, self.x, self.y)

```

Für die Bilder der Akteure habe ich mich wieder bei den freien [Twitter Emojis](#) bedient und hier sind sie, damit Ihr das Spiel nachprogrammieren könnt:



Die Bilder sind jeweils 36x36 Pixel groß, das habe ich in den Variablen `tw` und `th` festgehalten. Unser armer Held, der den grinsenden Smiley ausweichen muß, soll mit der Maus gesteuert werden. Dabei kann er sich nur horizontal bewegen, sein vertikaler Standort ist im Programm festverdrahtet. Mit den Zeilen

```

if self.x <= 0:
    self.x = 0
elif self.x >= width-tw:
    self.x = width - tw

```

ist sichergestellt, daß er das Spielfeld nicht heimlich verlassen kann, um sich den Grinsebacken zu entziehen. Diese grinsen tatsächlich nicht immer: Fröhlich stürzen sie herab, strecken auf der Höhe unseres Helden die Zunge heraus, um dann, wenn sie merken, daß sie ihn nicht getroffen haben, mit verärgertem Gesicht in die Tiefe zu stürzen. Dazu wird ihnen je nach Y-Koordinate in der `display()`-Funktion das entsprechende Bildchen zugewiesen:

```
def display(self):
    if (self.y > -30) and (self.y <= 200):
        image(self.im0, self.x, self.y)
    elif (self.y > 200) and (self.y <= 360):
        image(self.im1, self.x, self.y)
    elif (self.y > 360):
        image(self.im2, self.x, self.y)
```

Die Smileys stürzen natürlich nicht ins Bodenlose. Ich wollte mir den Stress ersparen und die Smiley-Objekte löschen zu müssen, nachdem sie das Spielfeld verlassen haben. Stattdessen habe ich im Hauptprogramm die Anzahl der Smileys konstant gesetzt (es sind zehn) und diese jedesmal, wenn sie das Spieldatenfenster verlassen haben, habe ich sie an einer zufälligen Position weit oberhalb des Fensters wieder neu positioniert:

```
self.y = -randint(50, 250)
self.x = randint(0, width-tw)
self.dy = randint(2, 10)
```

Mit der letzten Zeile wird ihnen dabei auch noch zufällig eine neue Geschwindigkeit zugewiesen, so daß der Spieler nicht merkt, daß er es immer wieder mit den gleichen Akteuren zu tun hat.

Das Hauptprogramm

Nun zum Hauptprogramm. Es ist zwar nicht ganz so kurz geraten, wie einige andere, die ich hier schon vorgestellt hatte, aber eigentlich immer noch übersichtlich:

```
from random import randint
from sprite import Skull, Smiley

w = 640
tw = th = 36
noSmileys = 10
startGame = True
playGame = False
```

```
gameOver = False

skull = Skull(w/2, 320)
smiley = []
for i in range(noSmileys):
    smiley.append(Smiley(randint(0, w-tw), randint(50, 250)))

def setup():
    skull.score = 0
    size(640, 480)
    frameRate(30)
    skull.loadPics()
    for i in range(len(smiley)):
        smiley[i].loadPics()
        smiley[i].dy = randint(2, 10)
    font = loadFont("ComicSansMS-32.vlw")
    textFont(font, 32)

def draw():
    global startGame, playGame, gameOver
    background(0, 0, 0)
    text("Score: " + str(skull.score), 10, 32)
    if startGame:
        text("Klick to Play", 200, height/2)
        if mousePressed:
            startGame = False
            playGame = True
    elif playGame:
        skull.move()
        for i in range(len(smiley)):
            if skull.checkCollision(smiley[i]):
                playGame = False
                gameOver = True
        skull.display()
        for i in range(len(smiley)):
            smiley[i].move()
            if smiley[i].over:
                skull.score += 1
            smiley[i].display()
```

```
elif gameOver:  
    text("Game Over!", 200, height/2)
```

Für dieses Spiel habe ich mir mal erlaubt, die allgemein verpönte Schrift `Comic Sans` zu verwenden, denn nichts ist hier so, wie es scheint: Das Böse ist gut und das Gute ist böse. Die entsprechende `.vlw`-Datei habe ich mit dem Tool »Schrift erstellen« (in `Tools -> Schrift erstellen ...`) erzeugt und wie die Bildchen in den `data`-Folder des Sketches abgelegt.

Nach dem Import der Klassen `Skull` und `Smiley` habe ich die entsprechenden Objekte erzeugt und ihnen ihre Startposition zugewiesen. Im `setup()` werden dann die Bilder geladen und den Smileys je eine eigene, zufällige Geschwindigkeit (`dy`) zugewiesen.

Etwas komplizierter ist die `draw()`-Funktion aufgebaut. Wegen der Eigenheit von `Processing.py`, daß das Programm zwar aus der IDE heraus startet, das Programmfenster aber dann noch nicht den Fokus besitzt (den hat nach wie vor die IDE), war es notwendig, einen Startbildschirm vorzuschalten, der das Spiel erst nach einem Mausklick startet (und damit dem Programmfenster auch den Fokus gibt). Und natürlich sollte es auch einen »Game Over«-Bildschirm geben. Daher habe ich drei globale Zustandsvariablen (`startGame`, `playGame` und `gameOver`) definiert und je nach ihrem Zustand werden dann die jeweiligen Bildschirme angezeigt.

Jeder Smiley, der das Fenster verläßt, ohne mit dem Schädel zu kollidieren, erhöht den Score des Spielers um einen Punkt. Dazu wurde die Variable `over` schon in der Klasse `Smiley` erzeugt, die jedesmal, wenn ein Smiley das Fenster verläßt

```
if self.y >= height:  
    self.over = True
```

auf `True` gesetzt wird. Dies wird aber beim nächsten Druchlauf in

```
def move(self):  
    self.over = False
```

sofort wieder zurückgesetzt. Im Hauptprogramm wird dann in den Zeilen

```
if smiley[i].over:  
    skull.score += 1
```

der Zustand abgefragt und der Score entsprechend hochgesetzt.

Das Programm ist tatsächlich spielbar. Passt der Spieler nicht auf und kollidiert mit einem der Smileys, dann ist es unbarmherzig zu Ende und es heißt »Game Over!«

Stage 2

Nun ist es an der Zeit, das *Avoider Game*, das ich [hier begonnen](#) hatte ein wenig aufzupeppen und auch ein bißchen *Refactoring* vorzunehmen. Zum einen war es ja bisher sehr unnachgiebig und hat bei jeden Kontakt mit einem Smiley unseren Helden sofort sterben lassen. Nun möchte ich ihm ein paar Leben mehr spendieren. Und zum anderen habe ich aus Bequemlichkeit einige Initialisierungen in der Klasse **Sprite** vorgenommen, die dort eigentlich nicht hingehörten, da sie redundant waren. Diese habe ich nun in die abgeleiteten Klassen verfrachtet. Dazu mußte ich aber die `__init__()`-Methode jeweils überschreiben, so daß ich in den abgeleiteten Klassen `super()` aufrufen mußte, um die `__init__()`-Methode der Oberklasse auch aufzurufen. Ich will das mal an einem Beispiel zeigen. Die Klasse **Sprite** sieht nun so aus:

```
class Sprite(object):  
    def __init__(self, posX, posY):  
        self.x = posX  
        self.y = posY  
  
    def checkCollision(self, otherSprite):  
        if (self.x < otherSprite.x + tw and otherSprite.x < self.x + th  
            and self.y < otherSprite.y + th and otherSprite.y < self.y + th):  
            return True
```

```
    else:  
        return False
```

Sie hat nur noch eine minimale Initialisierung und besitzt auch nur noch die Methode `checkCollision()`, da nur diese an die daraus abgeleiteten Klassen vererbt wird. Die Klasse `Skull` hingegen und ihre `__init__()`-Methode sieht nun so aus:

```
class Skull(Sprite):  
  
    def __init__(self, posX, posY):  
        super(Skull, self).__init__(posX, posY)  
        self.score = 0  
        self.health = 0
```

Der Aufruf der `super()`-Methode ist so Python 2.7 spezifisch, in Python 3 wurde sie vereinfacht, aber Processing.py beruht nun mal auf Jython und Jython ist (noch?) Processing 2.7. Damit der `super()`-Aufruf funktioniert, muß übrigens das Eltern-Objekt von `object` abgeleitet werden, sonst kann Processing.py den Typ nicht erkennen.

Die Variablen `score` und `health` sind nur für das Objekt `Skull` von Bedeutung und wurden daher vom Eltern-Objekt in das abgeleitete Objekt verschoben.

Das Spiel

Um das Spiel angenehmer für den Spieler zu machen, bekam der Schädel ein paar Leben spendiert, die mit Herzchen symbolisiert werden, und außerdem bekam der *Game-Over-Screen* die Möglichkeit, von hier aus das Spiel noch einmal zu starten. Dafür mußte ich der Klasse `Smiley`, deren `__init__()`-Methode nun so aussieht,

```
def __init__(self, posX, posY):  
    super(Smiley, self).__init__(posX, posY)  
    self.outside = False
```

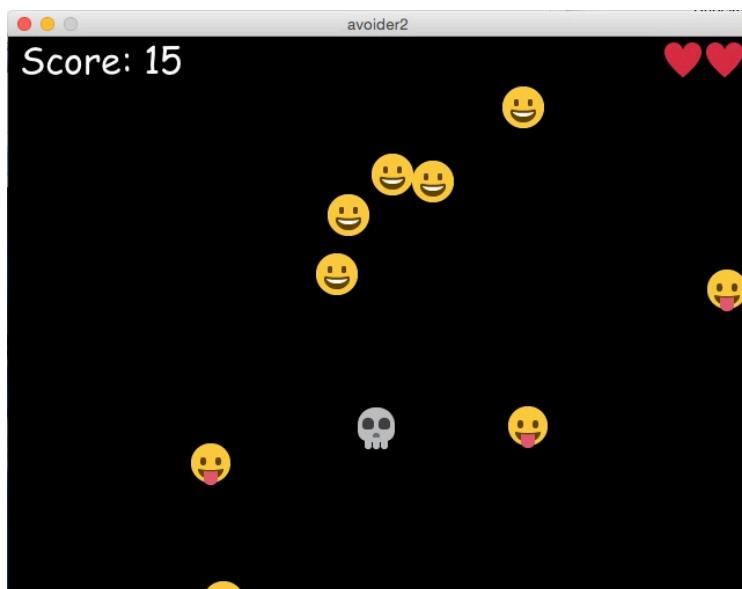


Abbildung 20.1: Screenshot

eine `reset()`-Methode verpassen, die die Möglichkeit gibt, zu Beginn eines neuen Spiels auch die Smileys wieder oberhalb des oberen Bildschirmrandes zu katapultieren, von der sie dann fröhlich wieder herabfallen können. Sie ist ganz einfach gehalten, da die Berechnung der neuen Positionen im Hauptprogramm abläuft:

```
def reset(self, posX, posY):
    self.x = posX
    self.y = posY
```

In der Initialisierung habe ich noch die Variable `over` in `outside` geändert. Auch wenn es nur Kosmetik ist, der Name schien mir verständlicher auszudrücken, was die Variable machen soll. Ansonsten hat es in dem Reiter `sprite.py` keine weiteren Veränderungen gegeben.

Das Hauptprogramm

Alle anderen Veränderungen fanden im Hauptprogramm statt, das ein komplettes *Refactoring* erfahren hat. Die `draw()`-Schleife sieht nun so aus:

```
def draw():
    global heart
    background(0, 0, 0)
    text("Score: " + str(skull.score), 10, 32)
    for i in range(skull.health):
        image(heart, width - i*tw - tw - 2, 2)
    if startgame:
        startGame()
    elif playgame:
        playGame()
    elif gameover:
        gameOver()
```

Nach der Definition des Hintergrundes wird ein *HUD (Head-Up-Display)* gezeichnet der in allen drei Bildschirmen gleich ist.

Damit die Herzchen, obwohl von links nach rechts gezeichnet, immer in der rechten oberen Ecke kleben, sieht die Berechnung der Position etwas seltsam aus, aber es ist einfach nur die Weite des Bildschirms, abzüglich der Weite der Herzchen (in diesem Fall `tw = 36`) multipliziert mit der Anzahl der Herzchen und versehen mit einem Abstand von je zwei Pixeln.

Die einzelnen Bildschirme (Startbildschirm, das eigentliche Spiel und den Game-Over-Bildschirm) habe ich dann in eigene Funktionen verschoben und so aus der Hauptschleife ausgelagert. Sie sehen nun so aus:

```
def startGame():
    global startgame, playgame
    text("Klick to Play", 200, height/2)
    if mousePressed:
        startgame = False
        playgame = True

def playGame():
    global playgame, gameover
    skull.move()
    for i in range(len(smiley)):
        if skull.checkCollision(smiley[i]):
            if skull.health > 0:
                skull.health -= 1
                smiley[i].reset(randint(0, w-tw), -randint(50, 250))
            else:
                playgame = False
                gameover = True
    skull.display()
    for i in range(len(smiley)):
        smiley[i].move()
        if smiley[i].outside:
            skull.score += 1
        smiley[i].display()

def gameOver():
    global playgame, gameover
    text("Game Over!", 200, height/2)
    text("Klick to play again.", 200, 300)
```

```
if mousePressed:  
    gameover = False  
    for i in range(len(smiley)):  
        smiley[i].reset(randint(0, w-tw), -randint(50, 250))  
    playgame = True  
    skull.health = 5
```

Zu `startGame()` ist eigentlich nichts zu schreiben, der Code sollte selbsterklärend sein.

Anders ist es bei `playGame()`. Da der Kontakt des Schädel mit einem Spieler nicht mehr zum sofortigen Spielende führt, muß bei Kontakt das Smiley »gelöscht« werden, das heißt es wird wieder an eine zufällige Stelle oberhalb des Bildschirms versetzt. Und bei jedem Kontakt bekommt der Spieler natürlich ein Leben und ein Herzchen abgezogen. Da ich schon soviel darüber geschrieben habe, hier erst einmal das Herzchen, damit Ihr das Spiel auch nachprogrammieren könnt:



Abbildung 20.2: Heart

Ähnliches gilt für den `gameOver`-Screen. Hier müssen *alle* Smileys wieder an eine zufällige Position oberhalb des Bildschirms katapultiert werden und natürlich erhält der Schädel auch alle seine Leben wieder zurück.

Der Quellcode

Zum vollen Verständnis und damit Ihr das Spiel auch vollständig nachprogrammieren könnt, hier der vollständige Quellcode. Erst einmal der Code im Reiter `sprite.py`:

```
from random import randint  
  
tw = th = 36
```

```
class Sprite(object):
    def __init__(self, posX, posY):
        self.x = posX
        self.y = posY

    def checkCollision(self, otherSprite):
        if (self.x < otherSprite.x + tw and otherSprite.x < self.x + tw
            and self.y < otherSprite.y + th and otherSprite.y < self.y + th):
            return True
        else:
            return False

class Skull(Sprite):

    def __init__(self, posY):
        super(Skull, self).__init__(posX, posY)
        self.score = 0
        self.health = 0

    def loadPics(self):
        self.im1 = loadImage("skull.png")

    def move(self):
        self.x = mouseX
        if self.x <= 0:
            self.x = 0
        elif self.x >= width-tw:
            self.x = width - tw

    def display(self):
        image(self.im1, self.x, self.y)

class Smiley(Sprite):

    def __init__(self, posY):
        super(Smiley, self).__init__(posX, posY)
        self.outside = False

    def loadPics(self):
```

```
self.im0 = loadImage("smiley0.png")
self.im1 = loadImage("smiley1.png")
self.im2 = loadImage("smiley4.png")

def move(self):
    self.outside = False
    self.y += self.dy
    if self.y >= height:
        self.outside = True
        self.y = -randint(50, 250)
        self.x = randint(0, width-tw)
        self.dy = randint(2, 10)

def display(self):
    if (self.y > -30) and (self.y <= 200):
        image(self.im0, self.x, self.y)
    elif (self.y > 200) and (self.y <= 360):
        image(self.im1, self.x, self.y)
    elif (self.y > 360):
        image(self.im2, self.x, self.y)

def reset(self, posX, posY):
    self.x = posX
    self.y = posY
```

Und dann das Hauptprogramm `avoider2`:

```
from random import randint
from sprite import Skull, Smiley

w = 640
tw = th = 36
noSmileys = 10
startgame = True
playgame = False
gameover = False

skull = Skull(w/2, 320)
smiley = []
for i in range(noSmileys):
```

```
smiley.append(Smiley(randint(0, w-tw), -randint(50, 250)))  
  
def setup():  
    global heart  
    skull.score = 0  
    skull.health = 5  
    size(640, 480)  
    frameRate(30)  
    skull.loadPics()  
    for i in range(len(smiley)):  
        smiley[i].loadPics()  
        smiley[i].dy = randint(2, 10)  
    font = loadFont("ComicSansMS-32.vlw")  
    textFont(font, 32)  
    heart = loadImage("heart.png")  
    # noCursor()  
    # cursor(HAND)  
  
def draw():  
    global heart  
    background(0, 0, 0)  
    text("Score: " + str(skull.score), 10, 32)  
    for i in range(skull.health):  
        image(heart, width - i*tw - tw - 2, 2)  
    if startgame:  
        startGame()  
    elif playgame:  
        playGame()  
    elif gameover:  
        gameOver()  
  
def startGame():  
    global startgame, playgame  
    text("Klick to Play", 200, height/2)  
    if mousePressed:  
        startgame = False  
        playgame = True  
  
def playGame():  
    global playgame, gameover
```

```
skull.move()
for i in range(len(smiley)):
    if skull.checkCollision(smiley[i]):
        if skull.health > 0:
            skull.health -= 1
            smiley[i].reset(randint(0, w-tw), -randint(50, 250))
        else:
            playgame = False
            gameover = True
    skull.display()
for i in range(len(smiley)):
    smiley[i].move()
    if smiley[i].outside:
        skull.score += 1
    smiley[i].display()

def gameOver():
    global playgame, gameover
    text("Game Over!", 200, height/2)
    text("Klick to play again.", 200, 300)
    if mousePressed:
        gameover = False
        for i in range(len(smiley)):
            smiley[i].reset(randint(0, w-tw), -randint(50, 250))
        playgame = True
        skull.health = 5
```

Ich glaube, das *Refactoring* hat dem Quellcode gutgetan, er ist deutlich lesbarer und verständlicher geworden. Das Spiel ist so schon richtig gut spielbar, in einer nächsten Version möchte ich aber noch ein paar *Gimmicks* einbauen.

Stage 3: Sternenhimmel

Als nächstes wollte ich dem kleinen Avoider-Spiel ein wenig optische Tiefe verpassen. Daher habe ich einen Sternenhimmel inszeniert, bei dem die kleinen Sternen im fernen Hintergrund sich sehr langsam bewegen und die größeren Sterne etwas schneller.

So, wie wenn man bei einer Zugfahrt aus dem Fenster schaut, da scheinen die nahen Bäume auch schnell vorbeizufliegen, während der Wald im Hintergrund sich nur langsam bewegt. Diese Wahrnehmung nennt man [Bewegungsparallaxe](#) und sie wird besonders gerne in [Plattformspielen](#) angewandt.



Abbildung 20.3: Screenshot

Die Sterne

Um dies zu inszenieren, habe ich erst einmal im Reiter `sprite.py` eine Klasse `Star` angelegt:

```
class Star(object):

    def __init__(self, posX, posY, dia, speed):
        self.x = posX
        self.y = posY
        self.r = dia
        self.dy = speed
```

```

self.a = 255 # Transparency

def move(self):
    self.outside = False
    self.y += self.dy
    if self.y >= height:
        self.outside = True
        self.y = -2*self.r
        self.x = randint(0, width - 2*self.r)

def display(self):
    fill(255, 255, 255, self.a)
    noStroke()
    ellipse(self.x, self.y, self.r, self.r)

```

Ich hätte die Sterne natürlich auch von der Klasse `Sprite` ableiten können, aber da für sie ja keine Kollisionserkennung benötigt wird, hielt ich dies für *Overkill*. Da zumindest die größeren Sterne blinken sollen, bekommen sie eine Alpha-Kanal für Transparenz zugewiesen (`self.a`). Ansonsten bewegen sie sich genauso wie die Smileys von oben nach unten, nur viel, viel langsamer.

Jeder Stern wird mit seiner Position, seiner Größe und seiner Geschwindigkeit initialisiert. Per Default erhält er die größtmögliche Transparenz, das heißt, er ist strahlend weiß.

Im Hauptprogramm werden für die Sterne zwei Listen angelegt, eine (`bstar[]`) für die weit entfernten, kleinen Sterne und eine `nStar` für die größeren, näher erscheinenden Sterne. Das Auffüllen aller Listen habe ich in die `setup`-Funktion verschoben, dort wird nun die Funktion `loadData()` aufgerufen:

```

def loadData():
    for i in range(noSmileys):
        smiley.append(Smile(y=randint(0, w-tw), -randint(50, 250)))
    for i in range(nobStars):
        bStar.append(Star(randint(0, w-2), randint(2, h-2), 1, 0))
    for i in range(nonStars):
        nStar.append(Star(randint(0, w-4), randint(2, h-2), randint(1, 5)))

```

Die kleinen Sterne werden mit einem Durchmesser von 1 initialisiert, die größeren Sterne bekommen per Zufallszahl entweder

einen Durchmesser von 2 oder 3 zugewiesen. Interessant ist die Geschwindigkeit, mit der die Sterne sich bewegen: 0.1 per Frame für die kleinen, 0.2 per Frame für die großen. Processing kommt intern erstaunlich gut mit diesen dezimalen Werten bei der Positionierung zurecht, obwohl ja eigentlich nur ganzzahlige Pixel möglich sind.

Es gibt jeweils eine feste Anzahl von Sternen, wie bei den Smilies auch werden sie, wenn sie den unteren Bildrand passiert haben, wieder auf eine zufällige Position oberhalb des Fensters zurückversetzt.

Die Bewegung der Sterne findet natürlich in der Funktion `playGame()` statt, und zwar als erstes, bevor alle anderen Akteure gezeichnet werden (schließlich bilden sie den Hintergrund des Spiels):

```
for i in range(len(bStar)):
    bStar[i].move()
    bStar[i].display()
for i in range(len(nStar)):
    nStar[i].move()
    if (frameCount % randint(15, 30)) < randint(1, 15):
        nStar[i].a = 120
    else:
        nStar[i].a = 255
    nStar[i].display()
```

Die größeren Sterne sollen zusätzlich zur Bewegung auch noch Blinken, daher habe ich ihnen zufällige Intervalle zugewiesen, in denen der Alpha-Kanal auf 120 gesetzt wird (`nStar[i].a = 120`). Die Werte für die Zufallszahlen habe ich experimentell herausgefunden, Ihr könnt ruhig auch einmal andere Intervalle ausprobieren.

Der Quellcode

Und nun zum Nachvollziehen der vollständige Quellcode. Zuerst der Code aus dem Reiter `sprite.py`:

```
from random import randint

tw = th = 36

class Sprite(object):
    def __init__(self, posX, posY):
        self.x = posX
        self.y = posY

    def checkCollision(self, otherSprite):
        if (self.x < otherSprite.x + tw and otherSprite.x < self.x + tw
            and self.y < otherSprite.y + th and otherSprite.y < self.y + th):
            return True
        else:
            return False

class Skull(Sprite):
    def __init__(self, posX, posY):
        super(Skull, self).__init__(posX, posY)
        self.score = 0
        self.health = 0

    def loadPics(self):
        self.im1 = loadImage("skull.png")

    def move(self):
        self.x = mouseX
        if self.x <= 0:
            self.x = 0
        elif self.x >= width-tw:
            self.x = width - tw

    def display(self):
        image(self.im1, self.x, self.y)

class Smiley(Sprite):
    def __init__(self, posX, posY):
```

```
super(Smiley, self).__init__(posX, posY)
self.outside = False

def loadPics(self):
    self.im0 = loadImage("smiley0.png")
    self.im1 = loadImage("smiley1.png")
    self.im2 = loadImage("smiley4.png")

def move(self):
    self.outside = False
    self.y += self.dy
    if self.y >= height:
        self.outside = True
        self.y = -randint(50, 250)
        self.x = randint(0, width-tw)
        self.dy = randint(4, 10)

def display(self):
    if (self.y > -30) and (self.y <= 250):
        image(self.im0, self.x, self.y)
    elif (self.y > 250) and (self.y <= 320):
        image(self.im1, self.x, self.y)
    elif (self.y > 320):
        image(self.im2, self.x, self.y)

def reset(self, posX, posY):
    self.x = posX
    self.y = posY

class Star(object):

    def __init__(self, posX, posY, dia, speed):
        self.x = posX
        self.y = posY
        self.r = dia
        self.dy = speed
        self.a = 255 # Transparency

    def move(self):
        self.outside = False
```

```
        self.y += self.dy
        if self.y >= height:
            self.outside = True
            self.y = -2*self.r
            self.x = randint(0, width - 2*self.r)

def display(self):
    fill(255, 255, 255, self.a)
    noStroke()
    ellipse(self.x, self.y, self.r, self.r)
```

Außer dem schon oben besprochen Objekt `Star` gibt es hier nichts Neues. Aber auch im Hauptprogramm sind nur die erwähnten Änderungen neu:

```
from random import randint
from sprite import Skull, Smiley, Star

w = 640
h = 480
tw = th = 36
noSmileys = 10
nobStars = 30
nonStars = 15
startgame = True
playgame = False
gameover = False

skull = Skull(w/2, 320)
smiley = []
bStar = []
nStar = []

def setup():
    global heart
    skull.score = 0
    skull.health = 5
    size(640, 480)
    frameRate(30)
    loadData()
```

```
skull.loadPics()
for i in range(len(smiley)):
    smiley[i].loadPics()
    smiley[i].dy = randint(4, 10)
font = loadFont("ComicSansMS-32.vlw")
textFont(font, 32)
heart = loadImage("heart.png")
# noCursor()
# cursor(HAND)

def draw():
    global heart
    background(0, 0, 0)
    fill(255, 255, 255, 255)
    text("Score: " + str(skull.score), 10, 32)
    for i in range(skull.health):
        image(heart, width - i*tw - tw - 2, 2)
    if startgame:
        startGame()
    elif playgame:
        playGame()
    elif gameover:
        gameOver()

def loadData():
    for i in range(noSmileys):
        smiley.append(Smile(y=randint(0, w-tw), -randint(50, 250)))
    for i in range(nobStars):
        bStar.append(Star(randint(0, w-2), randint(2, h-2), 1, 0.1))
    for i in range(nonStars):
        nStar.append(Star(randint(0, w-4), randint(2, h-2), randint(2, 10), 0.1))

def startGame():
    global startgame, playgame
    text("Klick to Play", 200, height/2)
    if mousePressed:
        startgame = False
        playgame = True

def playGame():
```

```
global playgame, gameover
for i in range(len(bStar)):
    bStar[i].move()
    bStar[i].display()
for i in range(len(nStar)):
    nStar[i].move()
    if (frameCount % randint(15, 30)) < randint(1, 15):
        nStar[i].a = 120
    else:
        nStar[i].a = 255
    nStar[i].display()
skull.move()
for i in range(len(smiley)):
    if skull.checkCollision(smiley[i]):
        if skull.health > 0:
            skull.health -= 1
            smiley[i].reset(randint(0, w-tw), -randint(50, 250))
        else:
            playgame = False
            gameover = True
    skull.display()
for i in range(len(smiley)):
    smiley[i].move()
    if smiley[i].outside:
        skull.score += 1
    smiley[i].display()

def gameOver():
    global playgame, gameover
    text("Game Over!", 200, height/2)
    text("Klick to play again.", 200, 300)
    if mousePressed:
        gameover = False
        for i in range(len(smiley)):
            smiley[i].reset(randint(0, w-tw), -randint(50, 250))
        playgame = True
        skull.health = 5
```

Das Spiel ist schon recht spielbar geworden, durch die Sterne entsteht tatsächlich die Illusion von Tiefe und es ist auch nicht

einfach, den Schädel für längere Zeit an den herunterfallenden Smileys vorbei zu manövrieren. Irgendwann erwischt es einen immer.

Stage 4: PowerUp und PowerDown

Im vierten und letzten Teil meiner kleinen Serie über die Programmierung des Avoider-Spiels in Processing.py wollte ich das Spiel noch mit ein paar zusätzlichen Akteuren aufpeppen. Dazu habe ich *Power Items* eingeführt, die entweder dem Spieler zusätzliche Leben geben oder nehmen, also je ein *PowerUp* und ein *PowerDown*. Als besonderes Highlight bewegen diese sich auf anderen Wegen durch das Spieldfenster als die Smileys und sind daher etwas unberechenbarer für den Spieler. Gemäß dem Motto des Spiels, daß man niemanden trauen darf, das gut aussieht, ist das PowerUp, das dem Spieler ein weiteres Leben schenkt, ein grimmig aussehendes Gespenst und das PowerDown, das ihm ein Leben nimmt, ein lecker aussehendes Tassentörtchen.



Auch diese Bilder habe ich wieder den freien [Twitter Emojis](#) (*Twemojis*) entnommen und hier sind sie, damit Ihr das Spiel nachprogrammieren könnt.

Power Items

Als erstes habe ich im Reiter `sprite.py` eine Klasse `PowerItem` angelegt, die von `Sprite` erbt:

```
class PowerItem(Sprite):

    def __init__(self, posX, posY, tX, tY, eT):
        super(PowerItem, self).__init__(posX, posY)
        self.origX = posX
        self.origY = posY
```

```
    self.targetX = tX
    self.targetY = tY
    self.expireTime = eT
    self.duration = self.expireTime/2.0
    self.counter = 0
    self.pause = randint(10, 150)

def curveX(self, x):
    return x

def curveY(self, y):
    return y

def easing(self):
    self.counter += 1
    self.fX = self.fY = (self.counter)/float(self.duration)
    self.fX = self.curveX(self.fX)
    self.fY = self.curveY(self.fY)
    self.x = (self.targetX * self.fX) + (self.origX * (1.0 -
    self.y = (self.targetY * self.fY) + (self.origY * (1.0 -

def move(self):
    self.expireTime -= 1
    if self.expireTime < 0:
        self.pause -= 1
        if self.pause < 0:
            self.reset()

def display(self):
    # print(self.x, self.y)
    image(self.im1, self.x, self.y)

def reset(self):
    self.origX = randint(-150, width-tw)
    self.origY = -randint(50, 250)
    self.targetX = randint(tw, width-tw)
    self.targetY = randint(tw, height-tw)
    self.expireTime = self.duration*2.0
    self.counter = 0
    self.pause = randint(10, 150)
```

Die *Power Items* haben nur eine gewisse Lebensdauer und bewegen sich während ihrer Lebenszeit (`eT`) von der Startposition (`posX`, `posY`) zur Zielposition (`tX`, `tY`). Diese Parameter müssen daher dem Konstruktor übergeben werden.

Wie alle Akteure prasseln die *Power Items* zu Beginn des Spieles quasi gleichzeitig vom oberen Fensterrand auf den Spieler nieder, damit sich die Lage in den folgenden Runden entspannt, habe ich den einzelnen Items nach Ende ihren Lebens eine Pause verordnet, deren Länge vom Zufallszahlengenerator bestimmt wird, bevor sie wieder die Arena betreten dürfen.

Easing

Das Prinzip des *Easings* hatte ich [in diesem Beispiel](#) schon einmal eingeführt. Es war ein einfaches, lineares Easing, in dem die Figur immer langsamer wurde, je mehr sie sich dem Ziel näherte. Dieses lineare Easing ist auch in der Klasse `PowerItem` implementiert, aber so, daß es verändert werden kann, wenn die abgeleiteten Klassen die Methoden `curveX()` und/oder `curveY()` überschreiben. Außerdem wird die Geschwindigkeit und neue Position unter anderem auch von der Lebensdauer des *Power Items* beeinflußt.

In den von `PowerItem` abgeleiteten Klassen `Ghost` und `Cupcake` mußten also nur die entsprechenden Bildchen geladen und die Methode `curveY()` überschreiben:

```
class Ghost(PowerItem):  
  
    def loadPics(self):  
        self.im1 = loadImage("ghost.png")  
  
    def curveY(self, y):  
        return y**5  
  
class Cupcake(PowerItem):  
  
    def loadPics(self):
```

```
self.im1 = loadImage("cupcake.png")

def curveY(self, y):
    return 3*sin(3*y)
```

Im Falle des *Power Up*, des Gespenstes, bewegt sich das *Power Item* in einer exponentiellen Kurve von oben nach unten und wird immer schneller, je tiefer es fällt. Der Spieler muß sich schon beeilen, um mit diesem Item zu kollidieren, um ein zusätzliches Leben einzufangen. Dagegen habe ich mir im Falle des Tasentörtchens etwas Gemeines überlegt: Die einzelnen Törtchen bewegen sich auf einer übergroßen Sinuskurve durch das Geschehen. Daher kann es durchaus passieren, daß die Törtchen, nachdem sie das Fenster am unteren Rand verlassen haben, von dort auch wieder auftauchen und nach oben schießen. Das macht es dem Spieler schwieriger, ihnen auszuweichen. Also: Die Kollision mit den *Power Ups* ist schwierig, umgekehrt ist es schwer, den *Power Downs* auszuweichen. Schließlich soll es dem Spieler ja nicht zu einfach vorkommen.

Die jeweiligen Werte in der Methode `curveY()` habe ich durch wildes Experimentieren herausgefunden.

Das Hauptprogramm

Im Hauptprogramm sind die wichtigsten Änderungen in der Funktion `playGame()` vorgenommen worden, die folgende zusätzliche Zeilen erhielt:

```
for i in range(len(ghost)):
    ghost[i].easing()
    ghost[i].move()
    if ghost[i].checkCollision(skull):
        if skull.health < 5:
            skull.health += 1
            ghost[i].reset()
    ghost[i].display()
for i in range(len(cupcake)):
    cupcake[i].easing()
```

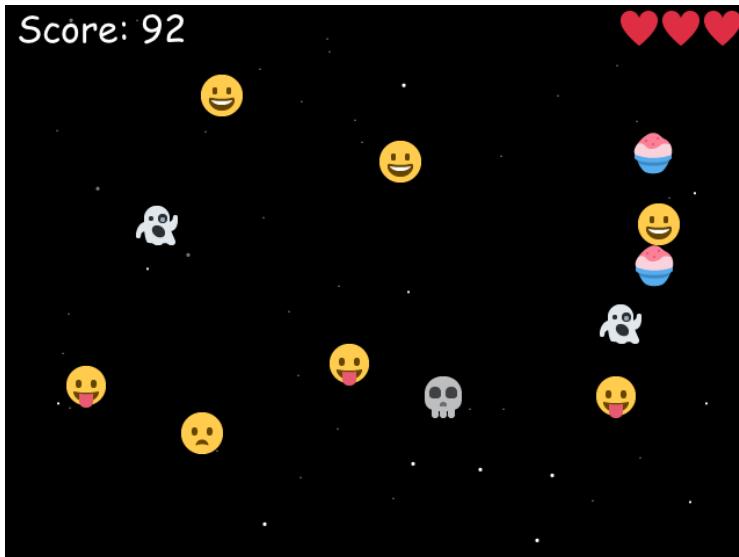


Abbildung 20.4: Screenshot

```
cupcake[i].move()
if cupcake[i].checkCollision(skull):
    skull.health -= 1
    cupcake[i].reset()
cupcake[i].display()
```

Für jedes *Power Item* wird erst das *Easing* berechnet, dann die neue Position bestimmt, überprüft ob es mit dem Spieler kollidiert und dann wird es angezeigt. Außerdem lasse ich als kleine Optimierung nicht mehr in jedem Frame den Spieler prüfen, ob er mit einem der Smileys kollidiert (das muß er nämlich jedes Mal mit *allen* Smileys machen), sondern nun überprüfen – wie bei den *Power Items* – die Smileys, ob sie mit dem Spieler kollidieren:

```
for i in range(len(smiley)):
    smiley[i].move()
    if smiley[i].checkCollision(skull):
        skull.health -= 1
```

```
smiley[i].reset(randint(0, w-tw), -randint(50, 250))
if smiley[i].outside:
    skull.score += 1
smiley[i].display()
```

Das Spiel startet in meiner Version mit zehn Smileys, drei Ge-
spenstern und fünf Tassentörtchen. Das sind 18 Akteure auf die
der Spieler aufpassen muß und das macht das Spiel schon ganz
schön schwierig, aber ohne daß es unfair wirkt oder gar unspiel-
bar ist.

Der Quellcode

Und nun – wie immer – der vollständige Quellcode, damit Ihr das
Spiel nachprogrammieren und nachvollziehen könnt. Als erstes
wieder der Code aus dem Reiter `sprite.py`:

```
from random import randint

tw = th = 36

class Sprite(object):
    def __init__(self, posX, posY):
        self.x = posX
        self.y = posY

    def checkCollision(self, otherSprite):
        if (self.x < otherSprite.x + tw and otherSprite.x < self.x + tw
            and self.y < otherSprite.y + th and otherSprite.y < self.y + th):
            return True
        else:
            return False

class Skull(Sprite):

    def __init__(self, posX, posY):
        super(Skull, self).__init__(posX, posY)
        self.score = 0
```

```
    self.health = 0

def loadPics(self):
    self.im1 = loadImage("skull.png")

def move(self):
    self.x = mouseX
    if self.x <= 0:
        self.x = 0
    elif self.x >= width-tw:
        self.x = width - tw

def display(self):
    image(self.im1, self.x, self.y)

class Smiley(Sprite):

    def __init__(self, posX, posY):
        super(Smiley, self).__init__(posX, posY)
        self.outside = False

    def loadPics(self):
        self.im0 = loadImage("smiley0.png")
        self.im1 = loadImage("smiley1.png")
        self.im2 = loadImage("smiley4.png")

    def move(self):
        self.outside = False
        self.y += self.dy
        if self.y >= height:
            self.outside = True
            self.y = -randint(50, 250)
            self.x = randint(0, width-tw)
            self.dy = randint(4, 10)

    def display(self):
        if (self.y > -30) and (self.y <= 250):
            image(self.im0, self.x, self.y)
        elif (self.y > 250) and (self.y <= 320):
            image(self.im1, self.x, self.y)
```

```
        elif (self.y > 320):
            image(self.im2, self.x, self.y)

    def reset(self, posX, posY):
        self.x = posX
        self.y = posY

class PowerItem(Sprite):

    def __init__(self, posX, posY, tX, tY, eT):
        super(PowerItem, self).__init__(posX, posY)
        self.origX = posX
        self.origY = posY
        self.targetX = tX
        self.targetY = tY
        self.expireTime = eT
        self.duration = self.expireTime/2.0
        self.counter = 0
        self.pause = randint(10, 150)

    def curveX(self, x):
        return x

    def curveY(self, y):
        return y

    def easing(self):
        self.counter += 1
        self.fX = self.fY = (self.counter)/float(self.duration)
        self.fX = self.curveX(self.fX)
        self.fY = self.curveY(self.fY)
        self.x = (self.targetX * self.fX) + (self.origX * (1.0 -
        self.y = (self.targetY * self.fY) + (self.origY * (1.0 -

    def move(self):
        self.expireTime -= 1
        if self.expireTime < 0:
            self.pause -= 1
            if self.pause < 0:
                self.reset()
```

```
def display(self):
    # print(self.x, self.y)
    image(self.im1, self.x, self.y)

def reset(self):
    self.origX = randint(-150, width-tw)
    self.origY = -randint(50, 250)
    self.targetX = randint(tw, width-tw)
    self.targetY = randint(tw, height-tw)
    self.expireTime = self.duration*2.0
    self.counter = 0
    self.pause = randint(10, 150)

class Ghost(PowerItem):

    def loadPics(self):
        self.im1 = loadImage("ghost.png")

    def curveY(self, y):
        return y**5

class Cupcake(PowerItem):

    def loadPics(self):
        self.im1 = loadImage("cupcake.png")

    def curveY(self, y):
        return 3*sin(3*y)

class Star(object):

    def __init__(self, posX, posY, dia, speed):
        self.x = posX
        self.y = posY
        self.r = dia
        self.dy = speed
        self.a = 255 # Transparency
```

```
def move(self):
    self.outside = False
    self.y += self.dy
    if self.y >= height:
        self.outside = True
        self.y = -2*self.r
        self.x = randint(0, width - 2*self.r)

def display(self):
    fill(255, 255, 255, self.a)
    noStroke()
    ellipse(self.x, self.y, self.r, self.r)
```

Und dann das eigentliche Hauptprogramm, das ebenfalls noch einmal an Umfang zugenommen hat:

```
from random import randint
from sprite import Skull, Smiley, Ghost, Cupcake, Star

w = 640
h = 480
tw = th = 36
noSmileys = 10
nobStars = 30
nonStars = 15
noGhost = 3
noCupcakes = 5
startgame = True
playgame = False
gameover = False

skull = Skull(w/2, 320)
smiley = []
bStar = []
nStar = []
ghost = []
cupcake = []

def setup():
```

```
global heart
size(640, 480)
frameRate(30)
loadData()
skull.score = 0
skull.health = 5
skull.loadPics()
for i in range(len(smiley)):
    smiley[i].loadPics()
    smiley[i].dy = randint(4, 10)
for i in range(len(ghost)):
    ghost[i].loadPics()
for i in range(len(cupcake)):
    cupcake[i].loadPics()
font = loadFont("ComicSansMS-32.vlw")
textFont(font, 32)
heart = loadImage("heart.png")
# noCursor()
# cursor(HAND)

def draw():
    global heart
    background(0, 0, 0)
    fill(255, 255, 255, 255)
    text("Score: " + str(skull.score), 10, 32)
    for i in range(skull.health):
        image(heart, width - i*tw - tw - 2, 2)
    if startgame:
        startGame()
    elif playgame:
        playGame()
    elif gameover:
        gameOver()

def loadData():
    for i in range(noSmileys):
        smiley.append(Smiley(randint(0, width-tw), -randint(50, 250)))
    for i in range(noGhost):
        ghost.append(Ghost(randint(-150, width-tw), -randint(50, 250)))
    for i in range(noCupcakes):
```

```
cupcake.append(Cupcake(randint(-150, width-tw), -randint(50, 150)))
for i in range(nobStars):
    bStar.append(Star(randint(0, width-2), randint(2, height-2), 2))
for i in range(nonStars):
    nStar.append(Star(randint(0, width-4), randint(2, height-4), 4))

def startGame():
    global startgame, playgame
    text("Klick to Play", 200, height/2)
    if mousePressed:
        startgame = False
        playgame = True

def playGame():
    global playgame, gameover
    for i in range(len(bStar)):
        bStar[i].move()
        bStar[i].display()
    for i in range(len(nStar)):
        nStar[i].move()
        if (frameCount % randint(15, 30)) < randint(1, 15):
            nStar[i].a = 120
        else:
            nStar[i].a = 255
        nStar[i].display()
    skull.move()
    if skull.health < 0:
        playgame = False
        gameover = True
    skull.display()
    for i in range(len(smiley)):
        smiley[i].move()
        if smiley[i].checkCollision(skull):
            skull.health -= 1
            smiley[i].reset(randint(0, w-tw), -randint(50, 250))
        if smiley[i].outside:
            skull.score += 1
        smiley[i].display()
    for i in range(len(ghost)):
        ghost[i].easing()
```

```
ghost[i].move()
if ghost[i].checkCollision(skull):
    if skull.health < 5:
        skull.health += 1
        ghost[i].reset()
    ghost[i].display()
for i in range(len(cupcake)):
    cupcake[i].easing()
    cupcake[i].move()
    if cupcake[i].checkCollision(skull):
        skull.health -= 1
        cupcake[i].reset()
    cupcake[i].display()

def gameOver():
    global playgame, gameover
    text("Game Over!", 200, height/2)
    text("Klick to play again.", 200, 300)
    if mousePressed:
        gameover = False
        for i in range(len(smiley)):
            smiley[i].reset(randint(0, w-tw), -randint(50, 250))
        for i in range(len(ghost)):
            ghost[i].reset()
        for i in range(len(cupcake)):
            cupcake[i].reset()
        playgame = True
        skull.health = 5
        skull.score = 0

def mousePressed():
    global playgame
    if playgame:
        saveFrame("frames/screenshot-####.png")
```

Screenshots

Bei diesem Spiel ist es nahezu unmöglich, mit den Bordmitteln des Betriebssystems noch aussagefähige Screenshots wie den

oben im Beitrag zu erstellen. Daher habe ich das mit Processing-eigenen Mitteln erledigt: Die Funktion `mousePressed()`

```
def mousePressed():
    global playgame
    if playgame:
        saveFrame("frames/screenshot-####.png")
```

schießt jedes Mal, wenn die linke Maustaste gedrückt wird, einen aktuellen Screenshot. Aus dem fertigen Spiel solltet Ihr diese Funktion natürlich wieder herausnehmen.

Das war es mit dem *Avoider Game*. Natürlich sind noch jede Menge Erweiterungen möglich und auch die Gestaltung des Start- und des Game-Over-Bildschirms kann sicher noch verschönert werden. Mir kam es aber darauf an, zu zeigen, wie in Processing.py mit einfachen Mitteln doch schnell ein ansprechendes Spiel programmiert werden kann. Alles weitere ist Eurer Phantasie überlassen.

Nachtrag: Avoider Game Stage 4a

Ich konnte es nicht lassen, nachdem ich zwei Nächte darüber geschlafen hatte, mußte ich doch noch einmal an das Avoider Game heran. Die *Power Ups* und *Power Downs* sollten jeweils zwei unterschiedliche Bildchen zugeordnet bekommen. Erreicht habe ich das mit der Python-eigenen Zufallsfunktion `choice()` aus der [Random-Bibliothek](#). So habe ich im Reiter `sprite.py` in der ersten Zeile `choice` importiert:

```
from random import randint, choice
```

Und dann in der Klasse `Ghost` die Methode `loadPics()` wie folgt geändert:

```
def loadPics(self):
    self.im1 = loadImage(choice(["ghost.png", "octo.png"]))
```

In der Klasse `Cupcake` sieht die gleiche Methode nun so aus:

```
def loadPics(self):
    self.im1 = loadImage(choice(["cupcake.png", "bier.png"]))
```

Hier sind die Bildchen für diejenigen unter Euch, die auch diese (letzte) Änderung nachprogrammieren wollen:



Auch diese Bilder entstammen den freien (CC-BY) Twitter Emojis (*Twemojis*).

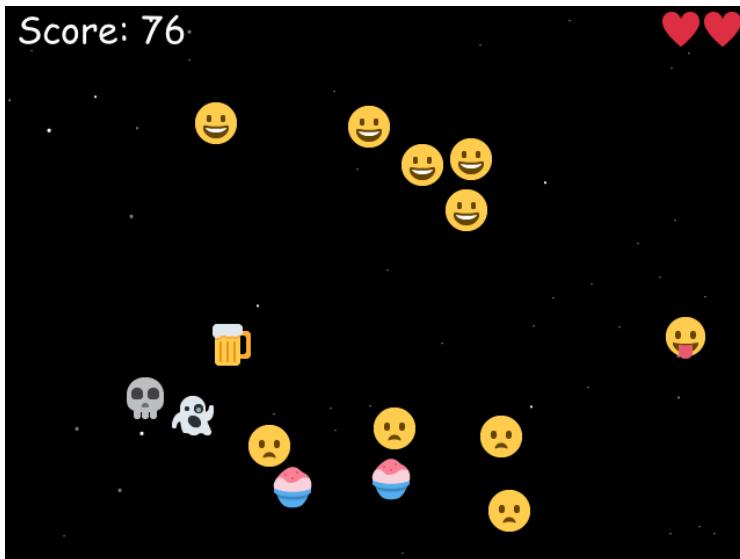


Abbildung 20.5:

Ich habe leider keinen Screenshot hinbekommen, auf denen alle verwendeten Bildchen zu sehen sind. So müßt Ihr mit obigem vorliebnehmen und mir glauben: Auch die Krake existiert!

Kapitel 21

Epilog

Kapitel 22

Anhang

Literaturverzeichnis

Index