Processing.py lernen

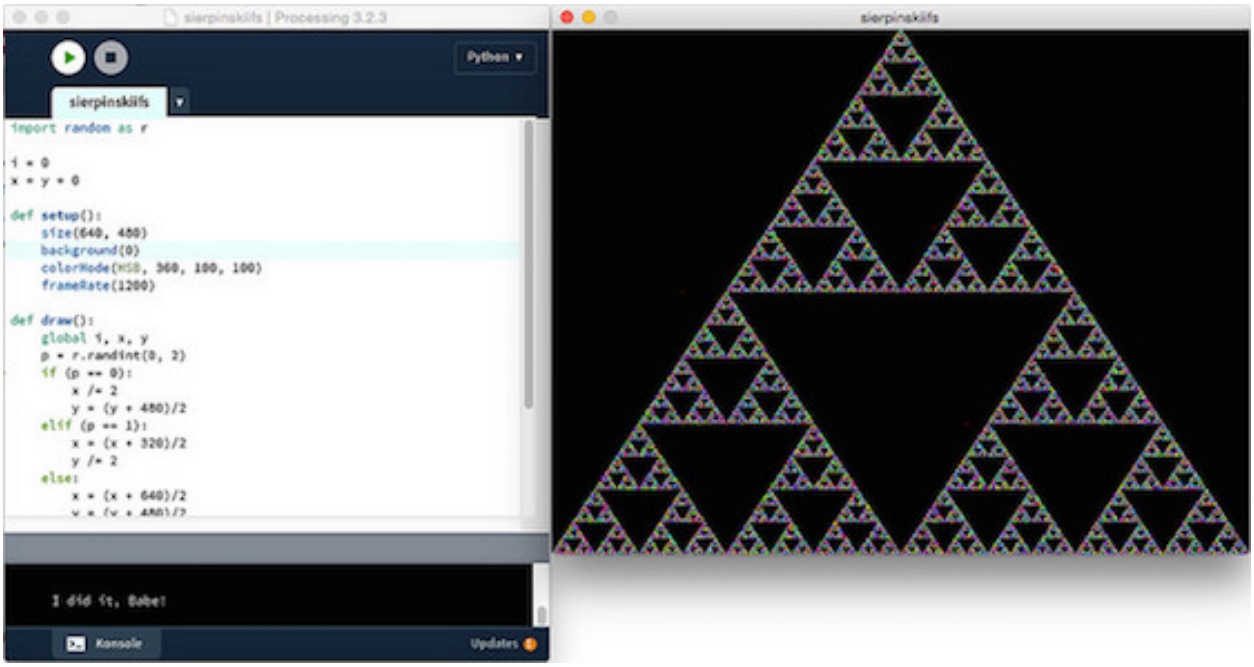
Einleitung

DDownload und Installation

JEP: Just Enough Python (Gerade genug Python)

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 *Wacław 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.

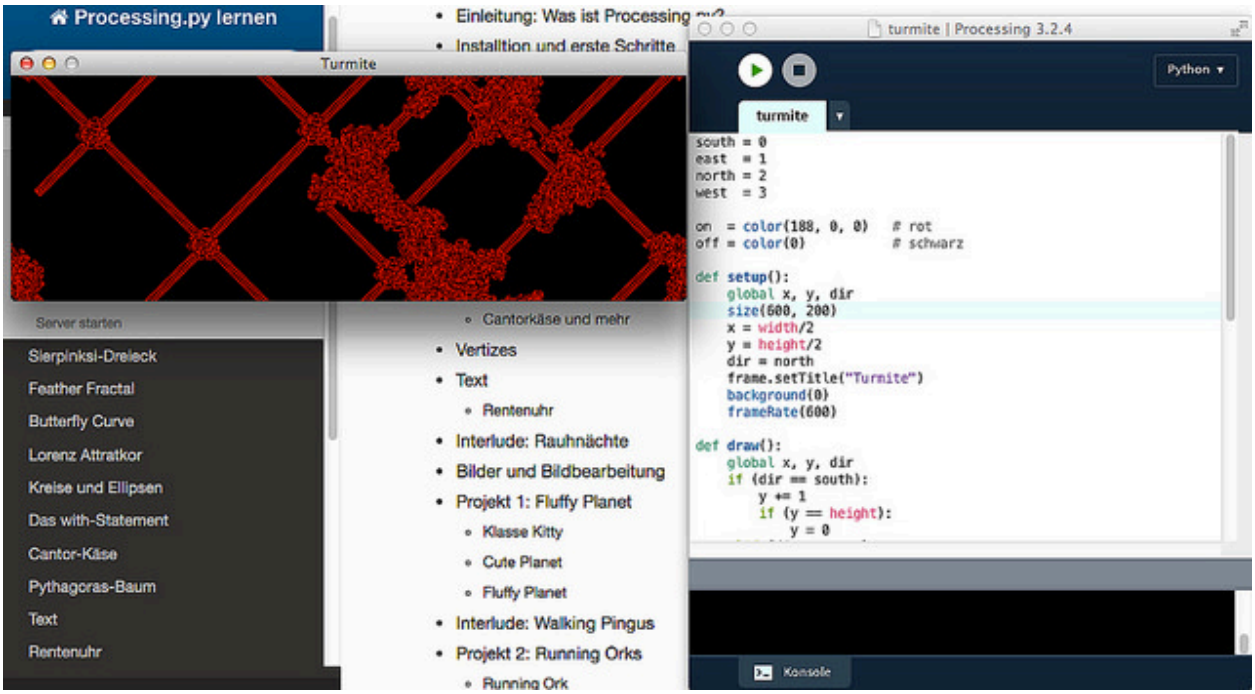
Aber man sieht sehr schön, wie sich das Dreieck zufällig, aber dennoch erkennbar, zusammensetzt. Je nach zufälligem Startwert liegen die ersten drei bis vier Punkte noch außerhalb des Fraktals, danach geht aber alles seinen geordneten Gang. Und an den Farben erkennt man, daß auch die Reihenfolge, in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

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, vorwärs, 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

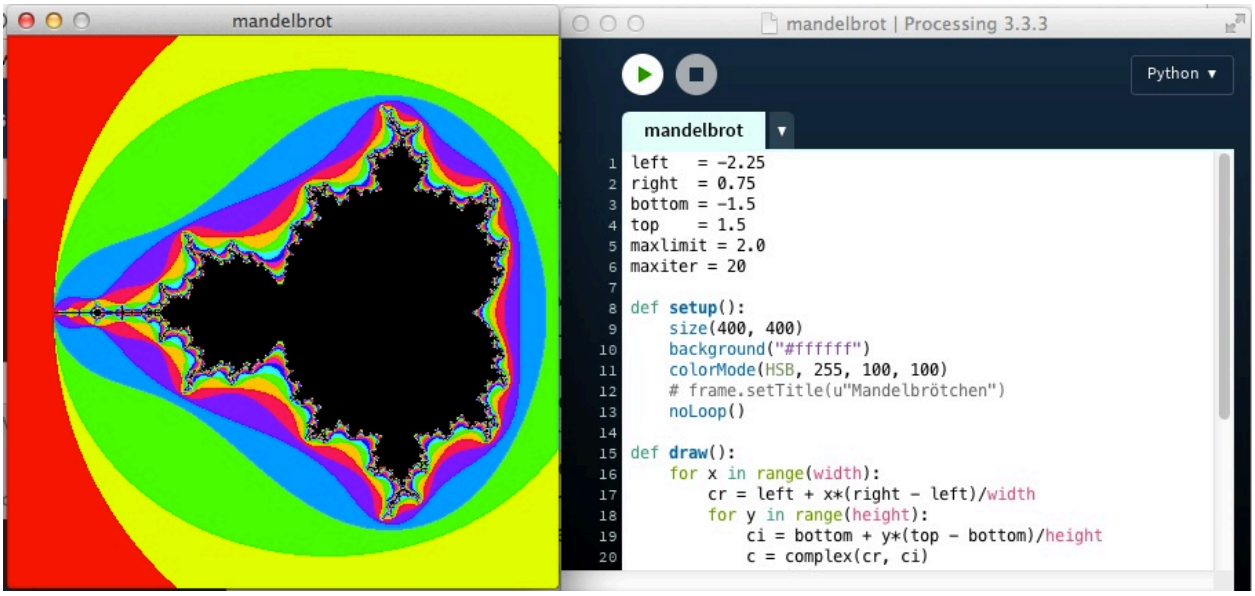


Abbildung 1: Screenshot

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

$$z_0 = 0 \tag{1}$$

$$\tag{2}$$

$$z_{n+1} = z_n^2 + c \tag{3}$$

$$\tag{4}$$

$$\tag{5}$$

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 kartesisschen 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. Zuert 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 Farbzuordnung, verlängert aber auch die Rechhenzeit 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()
```

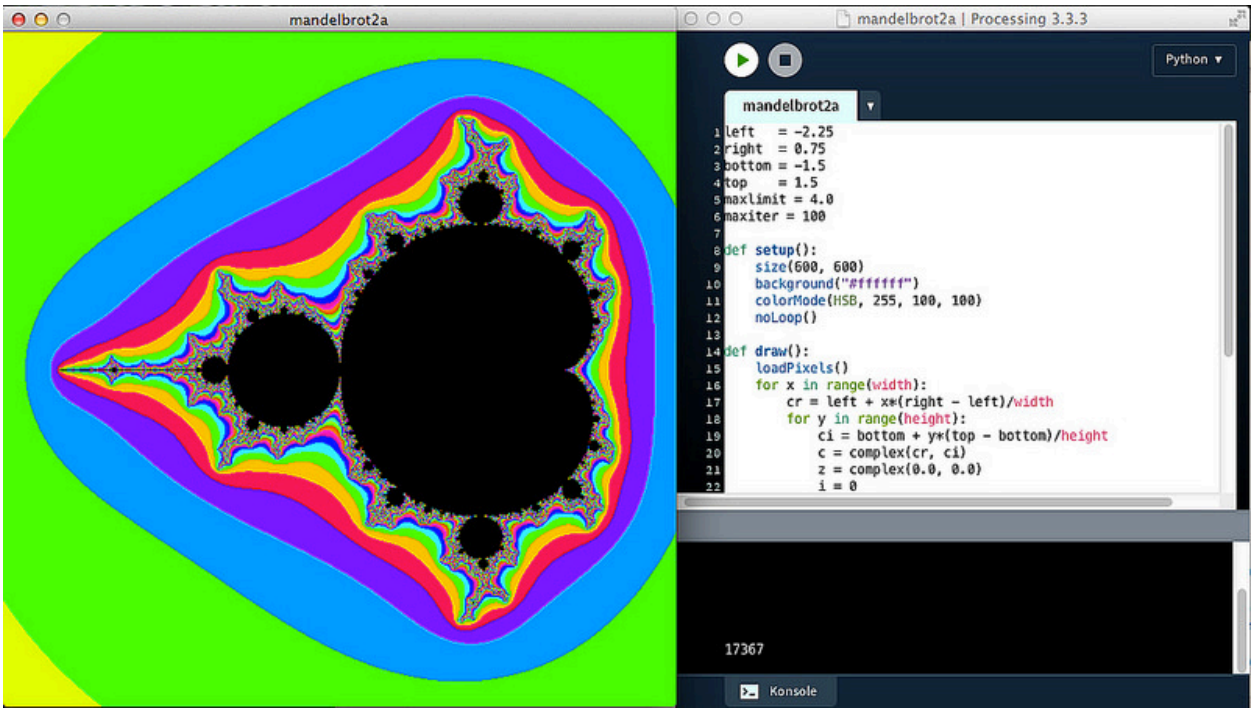
die Farbe an die gewünschte Stelle `x, y` zu setzen. Anschließend darf man nicht vergessen, mit

```
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 antreten 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
            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

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 von-

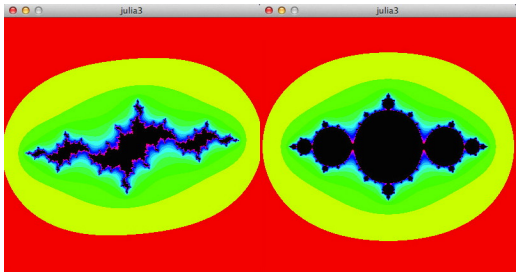


Abbildung 2: Screenshot

einander 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

$$z_0 = 0 \tag{6}$$
$$\tag{7}$$
$$z_{n+1} = z_n^2 + c \tag{8}$$
$$\tag{9}$$
$$\tag{10}$$

folgen, ist bei der Julia-Menge c konstant:

$$z_n^2 + c \tag{11}$$
$$\tag{12}$$
$$\tag{13}$$

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, 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)
```

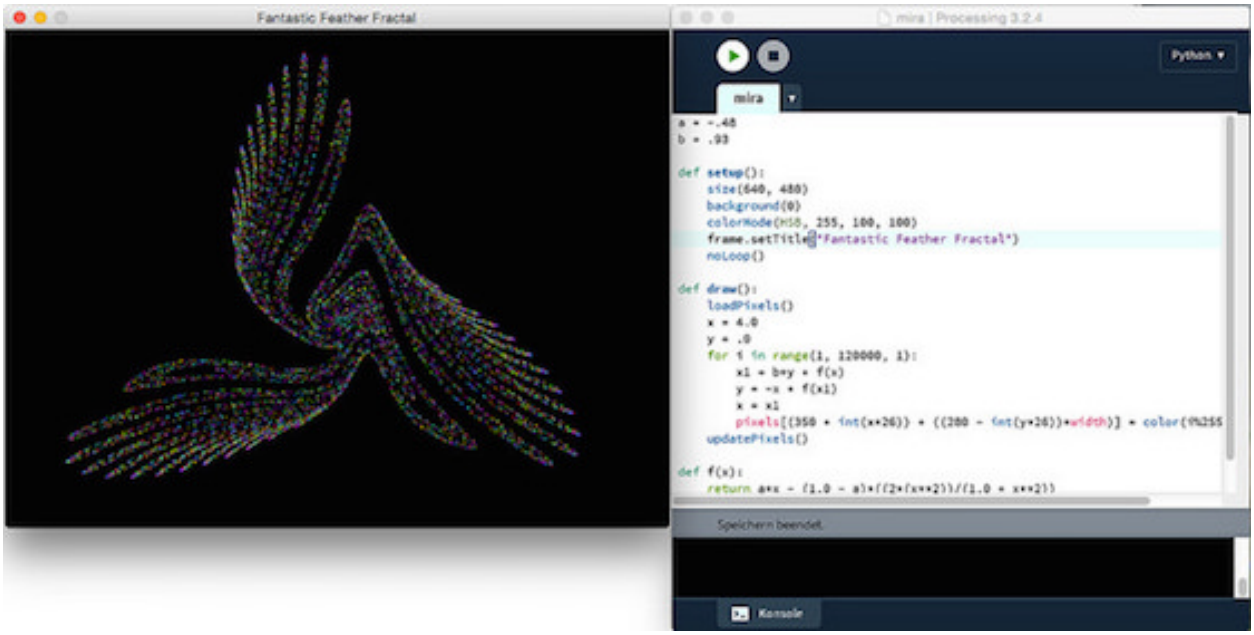
Das Setzen eines Pixels erfolgt genau umgekehrt:

```
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 laßt, werdet Ihr feststellen, daß das fertige Fraktal fast unmittelbar nach dem Aufruf im Graphikfenster erscheint.[1]



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 gefunden[2].

Der Quellcode

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

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

[1] Eine sehr schöne Einführung in das ungelöste Problem der Navier-Stokes-Gleichungen gibt es von *Florian Freistetter* in der 217. Folge seiner *Sternengeschichten*

[2] Und das schon vor langer Zeit, als der Monitor meines Rechners noch eine Auflösung von 640 x 480 Pixeln hatte.

```
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(i%255, 100, 100)
    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 zurückkomme, werden die Pixel-Arrays noch einmal ausführlich behandelt werden.

Literatur

- Clifford A. Pickover: *Mazes for the Mind. Computer s 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)

Anschauliche Mathematik: Die Schmetterlingskurve

Abbildung 3: 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

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

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:

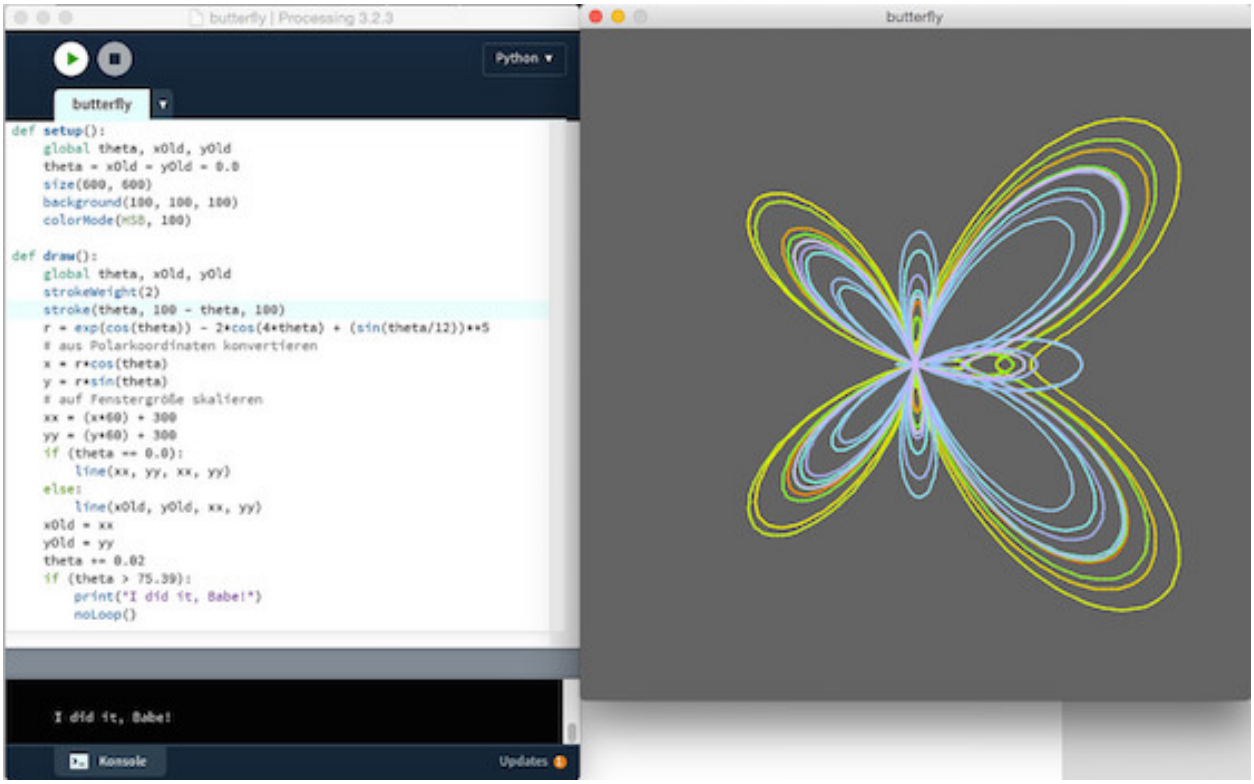
```
def setup():
    global theta, xOld, yOld
    theta = xOld = yOld = 0.0
    size(600, 600)
    background(100, 100, 100)
    colorMode(HSB, 100)

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 anspruchvollste 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*[3], nacheinander folgende Formlen 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 Schmetterling
```

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 Störungsgteil der Formel addiert oder subtrahiert: Der Schmetterling ist nahezu identisch, lediglich an der anderen Farbgebung erkennt Ihr, daß es zwei verschiedene Formeln sind.

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

Die Schmetterlingskurve und ähnliche Kurven wurden von *Temple Fay*[4] an der Universität von Southern Mississpi entwickelt. Sie eignen sich vorzüglich zum Experimentieren. So weist Pickover[5] darauf hin, daß die Kurve

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

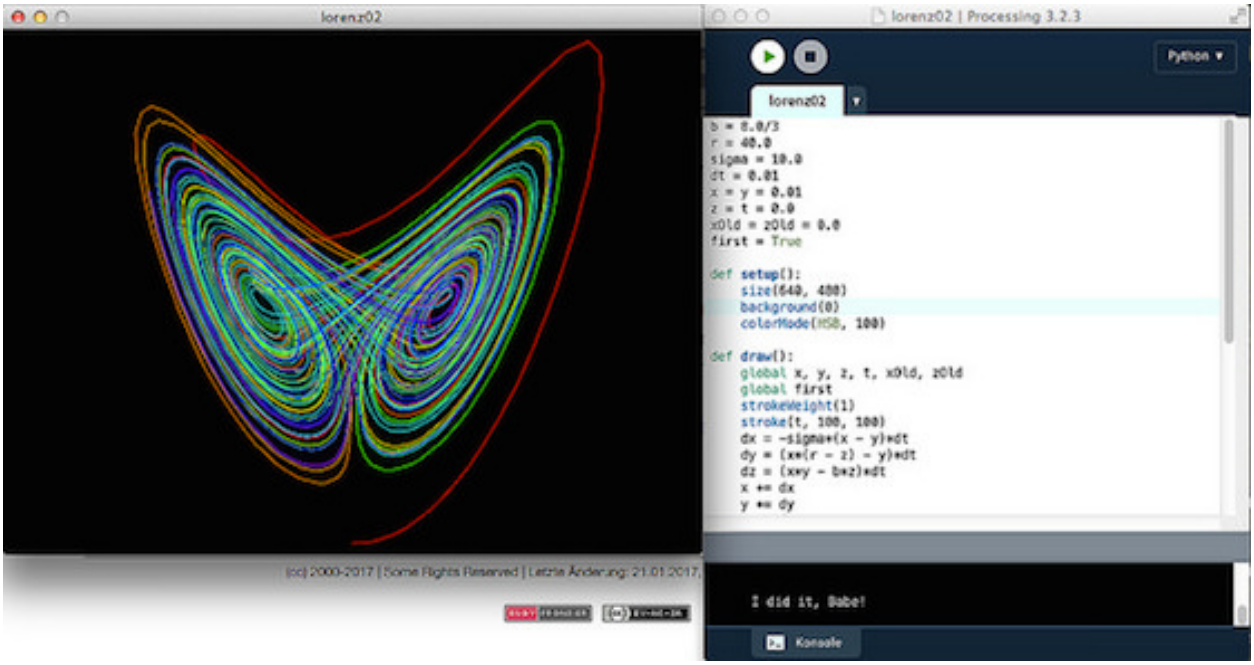
eine bedeutend größere Wiederholungsperiode besitzt. Ihr solltet Euch auch das ruhig einmal ansehen. Interessante Vergleiche mit der Originalschmetterlingskurve können Ihr auch ziehen, wenn Ihr mit

```
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:



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

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

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

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 [6] 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:

$$\frac{dx}{dt} = -\sigma(y - z) \tag{14}$$

$$\tag{15}$$

$$\frac{dy}{dt} = (\rho - z)x - y \tag{16}$$

$$\tag{17}$$

$$\frac{dz}{dt} = xy - \gamma z \tag{18}$$

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 Farbberechnugn 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
```

[6] Eine sehr schöne Einführung in das ungelöste Problem der Navier-Stokes-Gleichungen gibt es von *Florian Freistetter* in der 217. Folge seiner *Sternengeschichten*

```
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

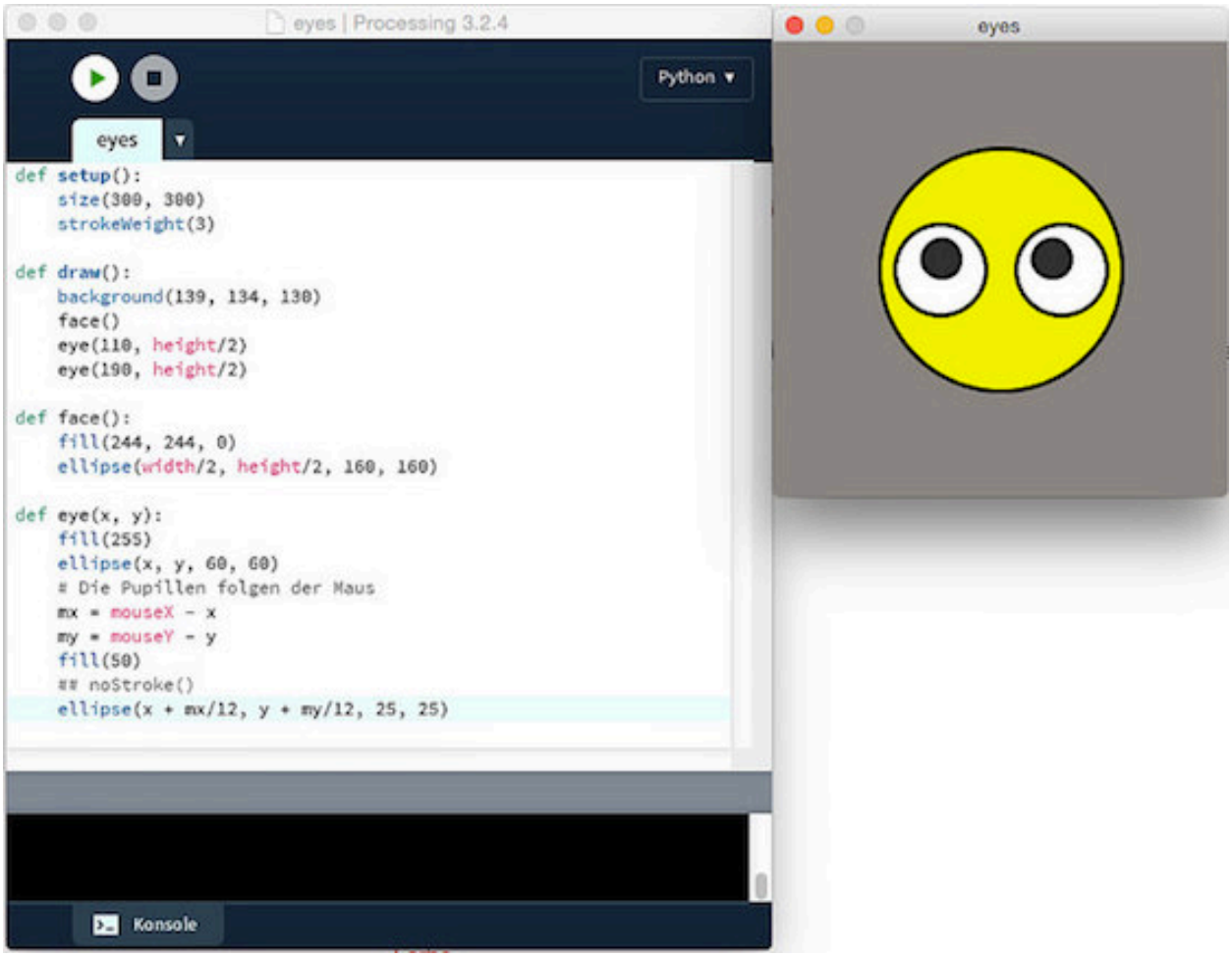
- Der *Lorenz Attractor* auf Wolfram MathWorld

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

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 grellroten 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 Funtion `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 Konfetti-Schnipsel auf den Bildschirm zaubern kann:

Und hier der Quellcode des Sketches in Processing.py:

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

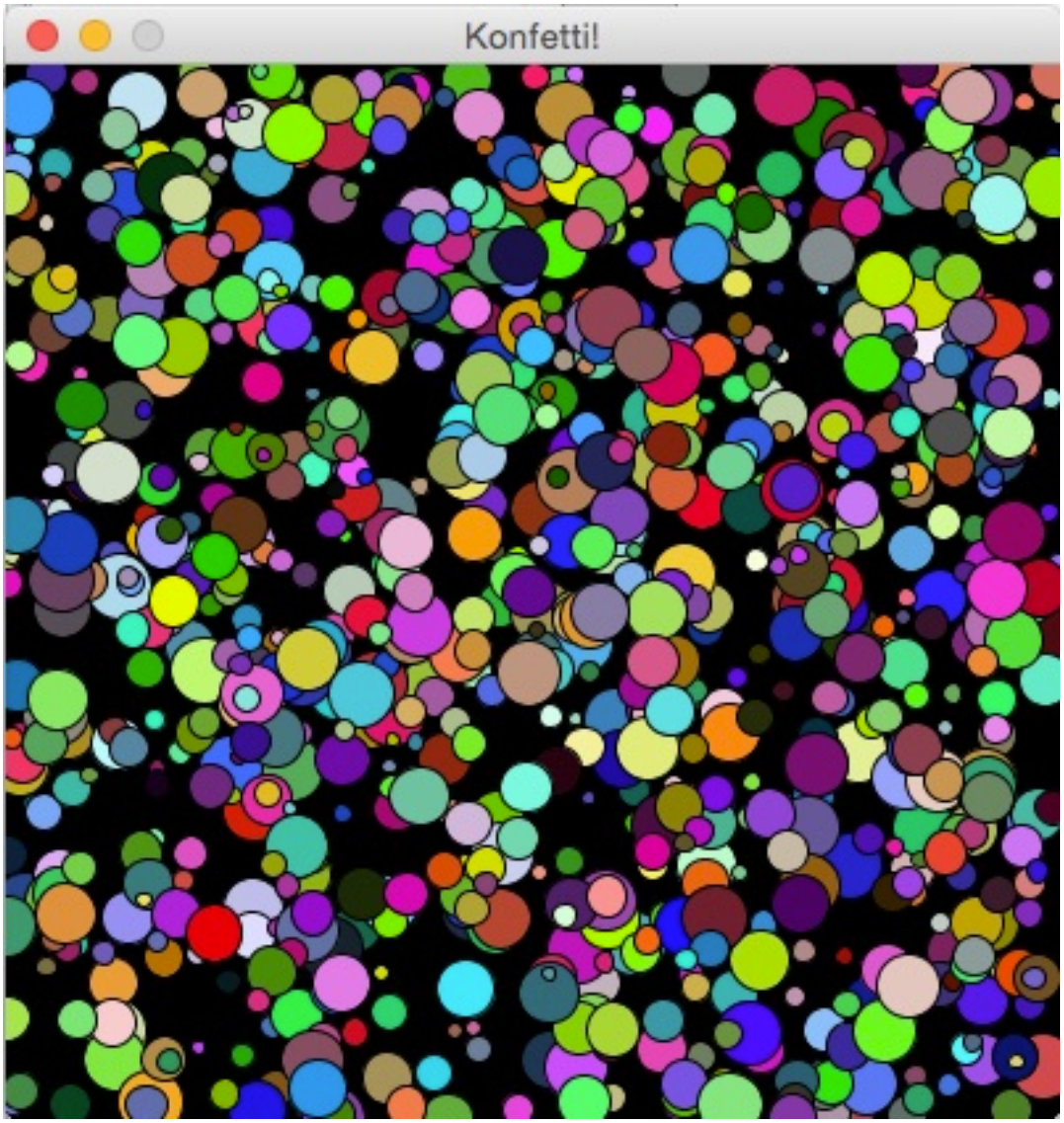


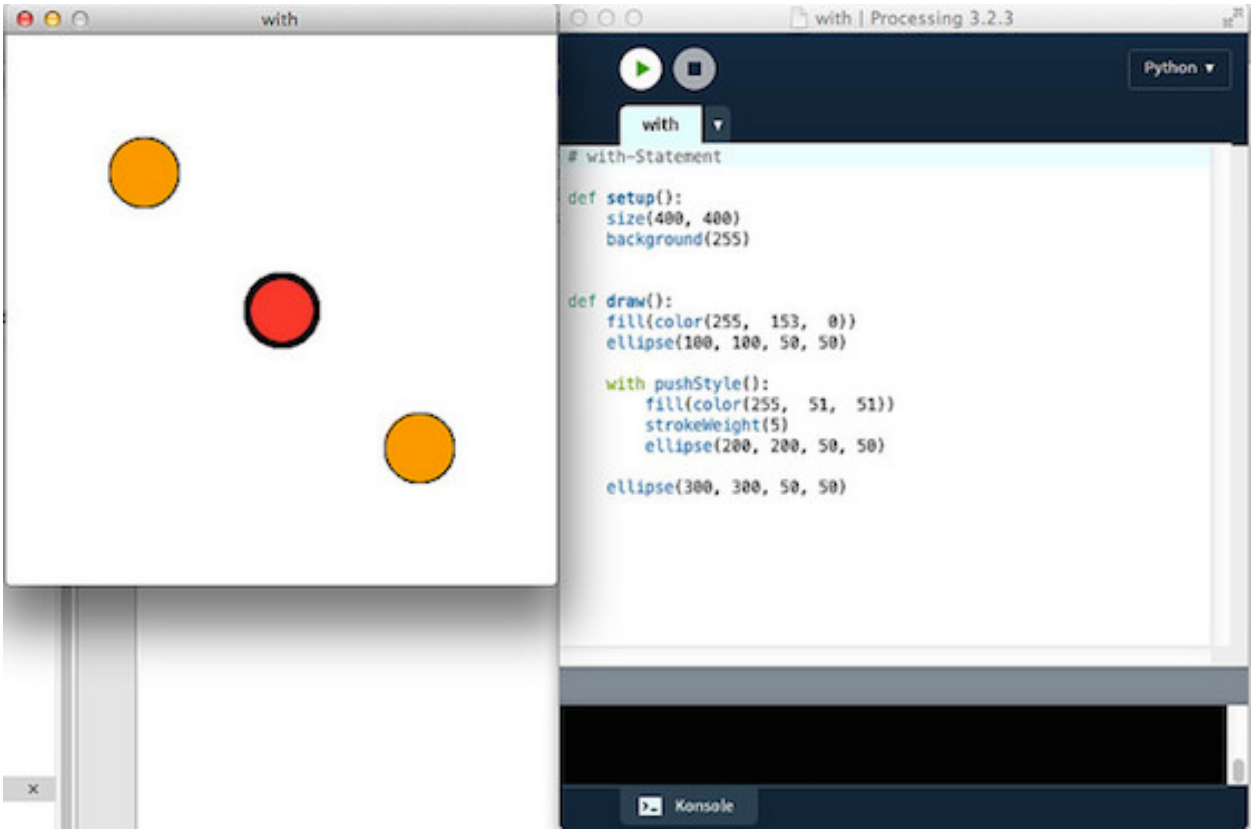
Abbildung 4: Screenshot

```
    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?

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.

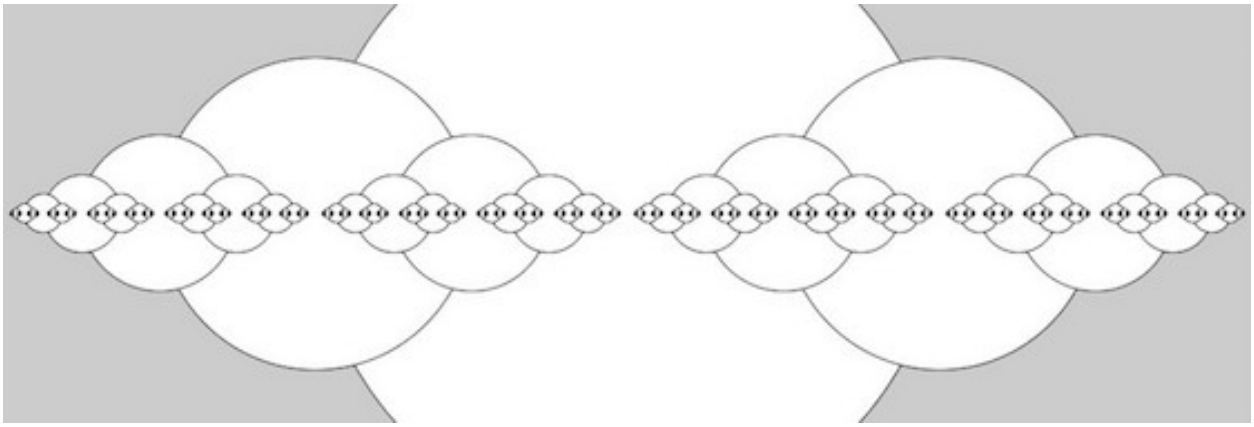
Da ich die Verwendung des `with`-Statements auch erst durch eines der mitgelieferten Beispielprogramme herausbekommen habe, hier eine (hoffentlich) komplette Liste der Möglichkeiten:

```
    with pushMatrix():          pushMatrix()
        translate(10, 10)       translate(10, 10)
        rotate(PI/3)            rotate(PI/3)
        rect(0, 0, 10, 10)      rect(0, 0, 10, 10)
    rect(0, 0, 10, 10)          popMatrix()
                                rect(0, 0, 10, 10)

with beginContour():             beginContour()
    doSomething()                doSomething()
                                 endContour()


with beginCamera():              beginCamera()
    doSomething()                doSomething()
                                 endCamera()
```

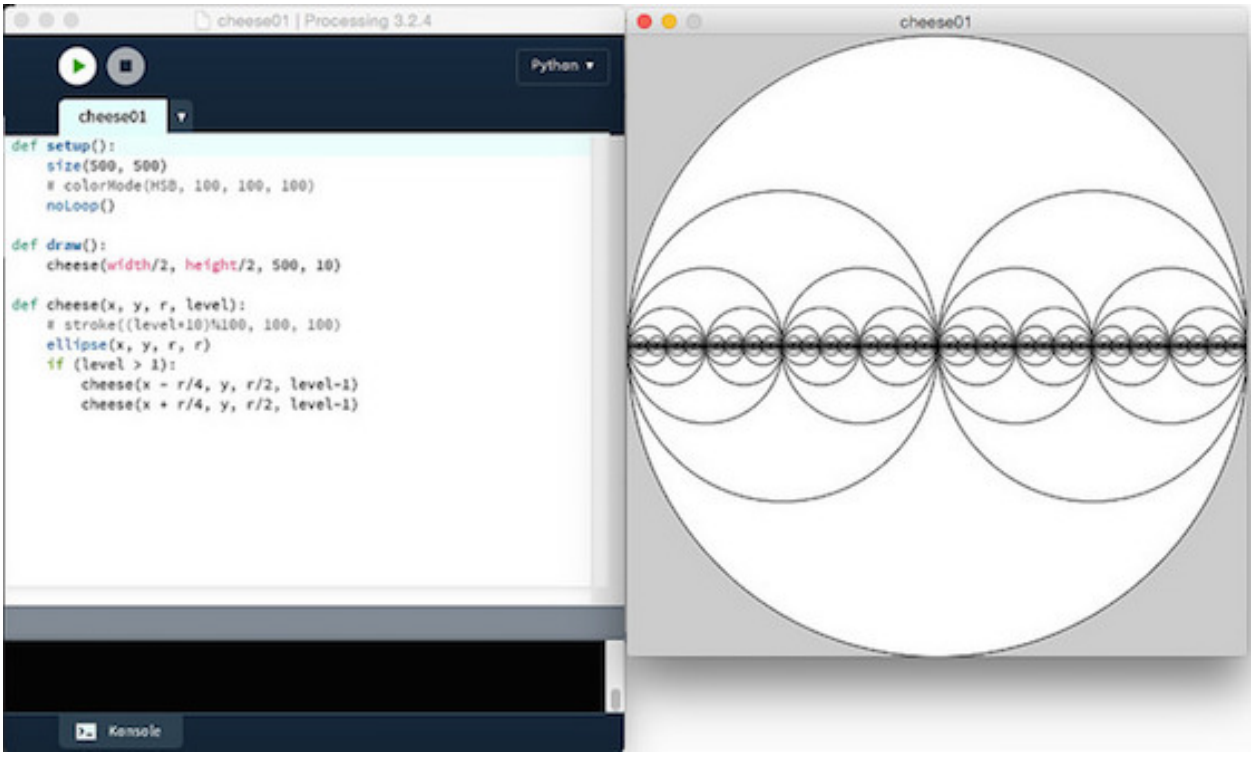
```
with beginPGL():                    beginPGL()
    doSomething()                   doSomething()
                                    endPGL()

with beginShape():                  beginShape()
    vertex(x, y)                    vertex(x, y)
    vertex(j, k)                    vertex(j,k)
                                    endShape()


with beginShape(TRIANGLES):         beginShape(TRIANGLES)
    vertex(x, y)                    vertex(x, y)
    vertex(j, k)                    vertex(x, y)
                                    endShape()

with beginClosedShape():            beginShape()
    vertex(x, y)                    vertex(x, y)
    vertex(j, k)                    vertex(j, k)
                                    endShape(CLOSED)
```

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 Porbleme mit der Rekursiontiefe 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 verwechseln. Setzt man dann den Algorithmus 1:1 um, zum Beispiel wie in diesem Sketch

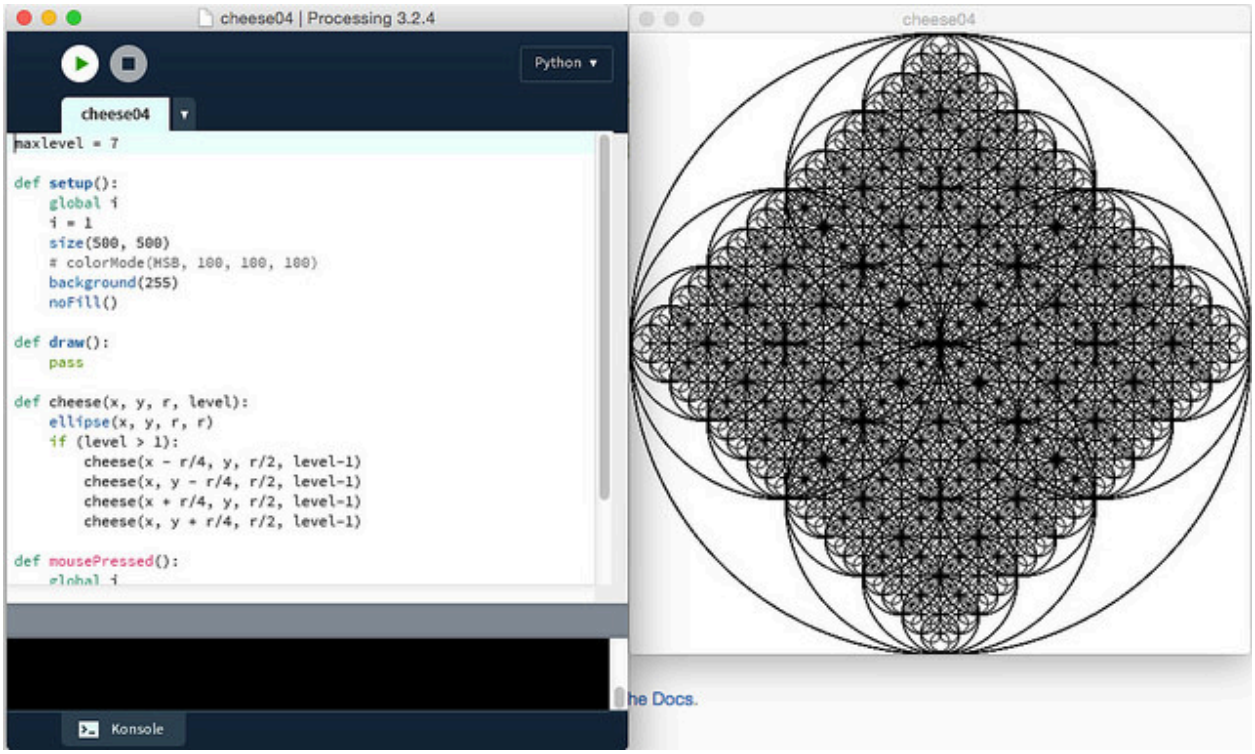
```
def setup():
    size(1000, 500)
```

```
    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 ziert. 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 Rekursionstiefe, 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 Rekursionstiefen 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.

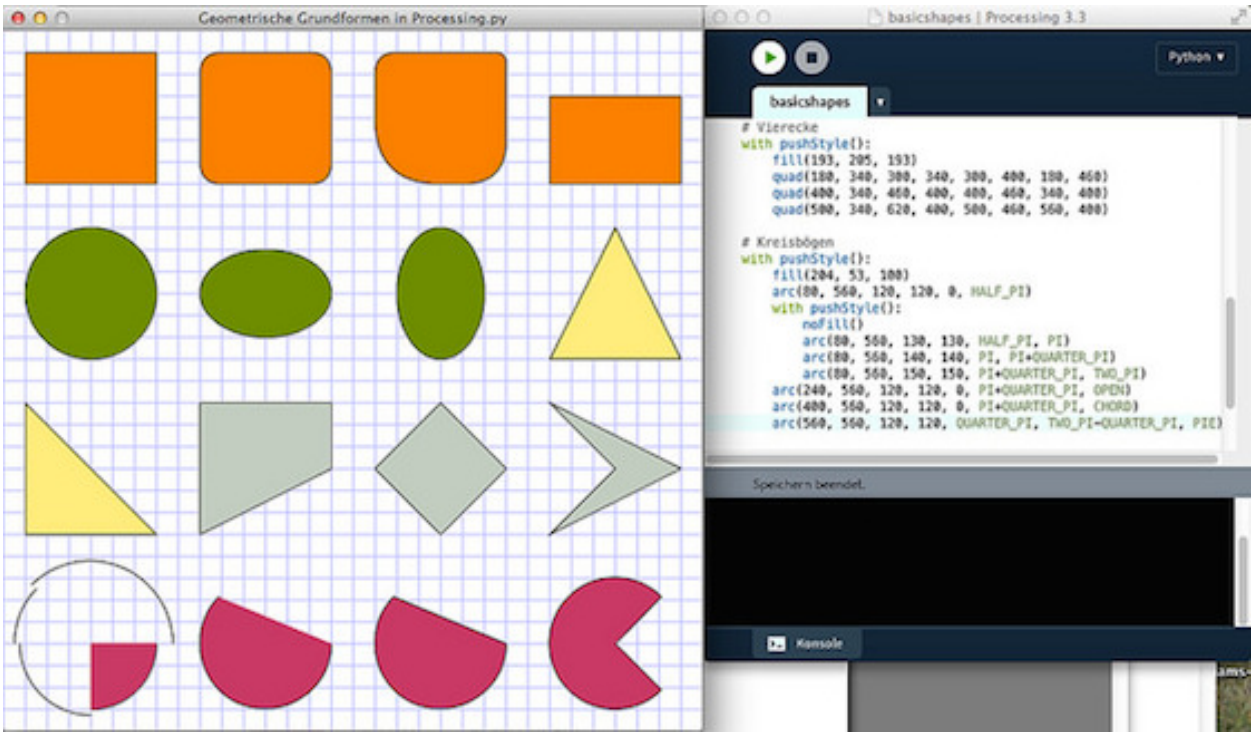
Interessant und neu für mich war, daß man – um überhaupt ein Zeichenfenster zu bekommen, in das man mit der Maus klicken konnte – eine leere `draw()`-Funktion benötigte. Eigentlich logisch, aber ich hatte vorher nie darüber nachgedacht.

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 Spielereien 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.

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, br, 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 Paramtern 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 bennen die ersten beiden Paramter 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 bennen 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 Paramter 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 bennenen 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 Dreieick 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-Koordinagen 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 Vierecken. Sie werden mit

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

erzeugt und auch hier sind es absolute Koordinaten und das Gebilde 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():
```

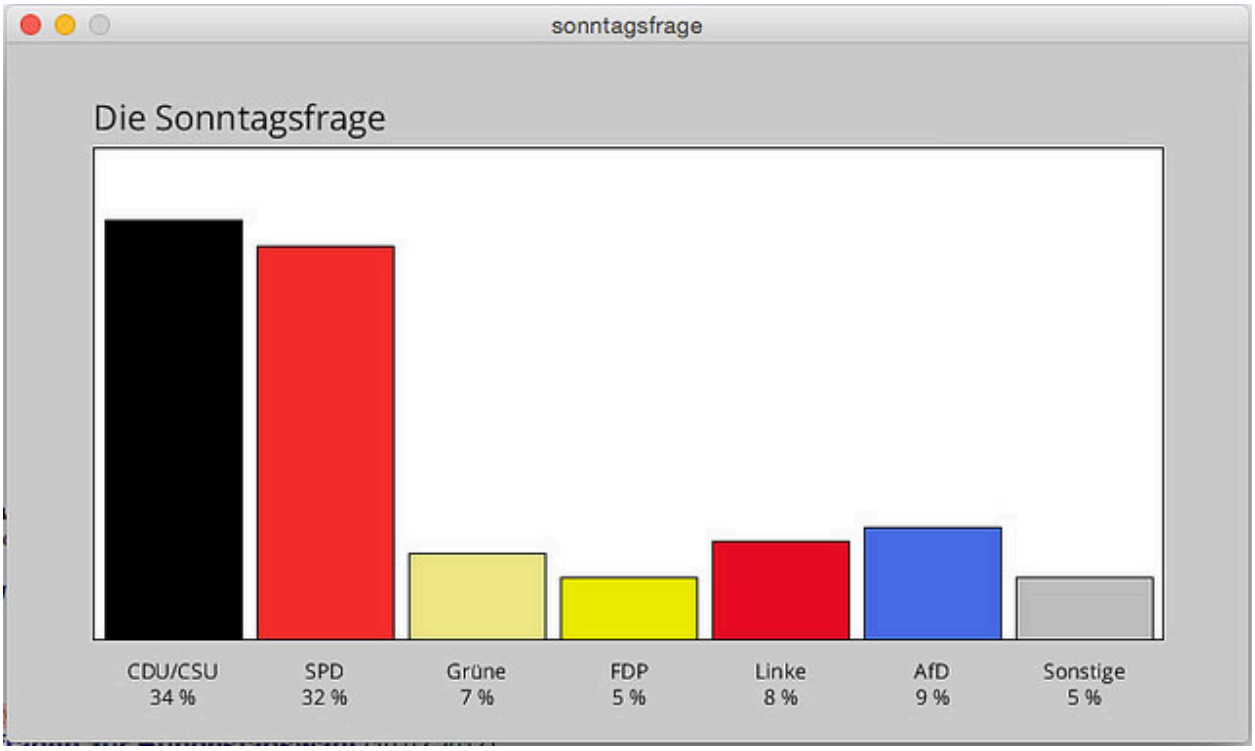
```
        fill(204, 53, 100)
        arc(80, 560, 120, 120, 0, HALF_PI)
        with pushStyle():
            noFill()
            arc(80, 560, 130, 130, HALF_PI, PI)
            arc(80, 560, 140, 140, PI, PI+QUARTER_PI)
            arc(80, 560, 150, 150, PI+QUARTER_PI, TWO_PI)
        arc(240, 560, 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 Cookbook« (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 einem 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", u"Grüne", "FDP", "Linke", "AfD", "Sonstige"]
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)
    textFont(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.