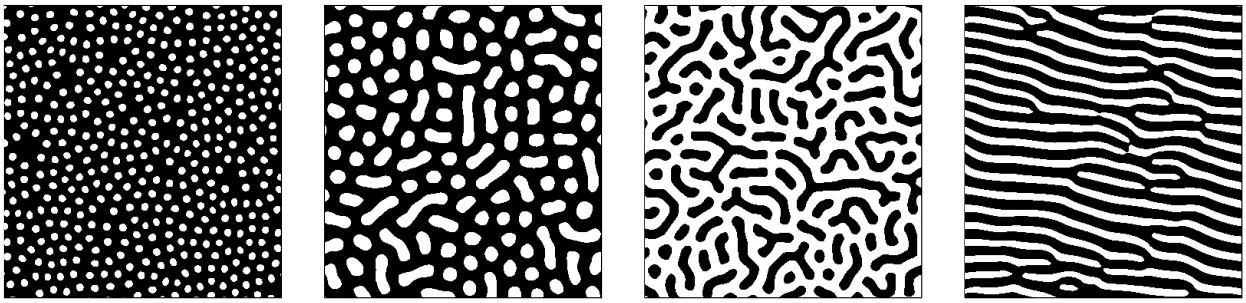
Turing ähnliche Muster aus zellulären Automaten

Einführung in die Visualisierung

Johanna Jansen, Jette von Postel, Julien Stengel



1 Einleitung

Auf der Bridges Finland Conference 2016 wurde eine Zusammenarbeit von Mathematikern und Künstlern zu Turing-ähnlichen Mustern vorgestellt. Im Rahmen des Seminars "Einführung in die Visualisierung" haben wir diese Muster analysiert und reproduziert.

2 Das Turing Modell

Der englische Mathematiker Alan Turing stellte 1952 einen Entwurf zur Modellierung stabiler räumlicher Muster durch ein System aus Chemikalien oder Morphogenen vor, die bei unterschiedlich schneller Diffusion durch ein Substrat miteinander reagieren. Im Sinne einer biologischen Modellanwendung wirkt eines dabei kurzfristig aktivierend, während das andere eine langfristige hemmende Wirkung hat.

Das Modell ist gegeben durch die Anzahl der Zellen $C_i,\ i = 1, ..., k$ und der Morphogene $M_j,\ j = 1, ..., l$, sowie die Diffusionsraten der verschiedenen Morphogene. Jede Zelle C_i enthält nun Angaben zur Menge jedes Morphogens M_j in dieser Zelle, $n_{C_i}(M_j)$. Diese Angaben ändern sich durch Reaktion untereinander und Diffusion mit jedem Zeitschritt. Um diese Änderung feststellen zu können, ist es genug, die Werte des gleichen Morphogens in den umliegenden Zellen $n_{C_i}(M_j),\ i = 1, ..., k$ zu kennen für eine Aussage zur Diffusion, sowie das Konzentrationsverhältnis aller Morphogene in der Zelle $n_{C_i}(M_j),\ j = 1, ..., l$ zur Betrachtung der chemischen Reaktionen.

Dieses an sich homogene System ist sehr anfällig für kleine Abweichungen, die es dann instabil machen und die Unregelmäßigkeiten wachsen lässt. Daraus entsteht ein neues stabiles System, was nicht mehr homogen ist, sondern verschiedene Muster aufweist.

Je mehr verschiedene Morphogene eine Zelle enthält, desto komplizierter wird das Reaktions-Diffusions-System, das damit zusammenhängt. Für Systeme mit zwei Komponenten gibt es bereits viel mehr mögliche Beobachtungen als im eindimensionalen und Turings Idee, dass ein zunächst stabiler Zustand durch Diffusion instabil wird, wird wichtig. Dies ist nur in einer bestimmten Klasse von Systemen möglich, den *activator-inhibitor systems*. Das bekannteste Beispiel dafür ist die FitzHugh-Nagumo Gleichung

$$\partial_t u = d_u^2 \nabla^2 u + f(u) - \sigma v,$$
$$\tau \partial_t v = d_v^2 \nabla^2 v + u - v$$

wobei $f(u) = \lambda u - u^3 - \kappa$ beschreibt wie Aktionspotential durch einen Nerv geleitet wird. Hier sind d_u, d_v, τ, σ und λ positive Konstanten.

Turing wollte mit diesem Morphogenese Modell eine biologische Erklärung für Muster im Fell von Wirbeltieren finden.

3 Das Young Modell

Das erste Modell einer Implementierung von Turings Aktivierungs-Hemmungs-Konzept stammt von D.A. Young und ist ein diskreter zellulärer Automat. Er ist eine diskretisierte Lösung einer verallgemeinerten Diffusionsgleichung.

Das Modell betrachtet eine Fläche aus gleichgroßen Zellen, von denen einige gefärbt sind und einige nicht. Die Verteilung kann durch einen einfachen Zufallsprozess generiert werden.

Die gefärbten Zellen produzieren ein aktivierendes und ein hemmendes Gen. Das aktivierende Gen M_1 stimuliert umliegende ungefärbte Zellen gefärbt zu werden und das hemmende Gen M_2 hindert weiter entfernte gefärbte Zellen daran, gefärbt zu bleiben. Zusammen induzieren sie ein morphogenes Feld $w(R)$, wobei R die Distanz zur gefärbten Zelle ist. Um die Zelle entsteht so ein innerer und ein äußerer Ring. In dem der inneren Ring hat $w(R)$ einen positiven Wert und im äußeren Ring einen negativen Wert.

In jedem Zeitschritt werden dann für jede Zelle mit Position $\mathbf{R}$ die Einflüsse ihrer Nachbarzellen mit Positionen $\mathbf{R}_i$ aufsummiert.

- Wenn $\sum_i w(|\mathbf{R} - \mathbf{R}_i|) > 0$, die Morphogen-Konzentration in der Zelle also positiv ist, dann wird bzw. bleibt die Zelle gefärbt.
- Wenn $\sum_i w(|\mathbf{R} - \mathbf{R}_i|) = 0$, dann bleibt die Zelle, wie sie ist.
- Wenn $\sum_i w(|\mathbf{R}-\mathbf{R}_i|) < 0$, die Morphogen-Konzentration in der Zelle also negativ ist, dann wird bzw. bleibt die Zelle ungefärbt.

4 Theoretische Umsetzung

Zur Umsetzung in die Praxis stehen das ursprüngliche Turing-Modell und das Young-Modell zur Auswahl. Da jedoch das Young-Modell unter anderem zur einfachen Berechnung von Rechnern entwickelt wurde, haben wir uns für diese Methode der Generierung entschieden. Man benötigt also eine Möglichkeit ein Gitter darzustellen, in denen Zellen miteinander interagieren und sich dann aufgrund der Nachbarschaftsumgebung dynamisch anpassen können. Dabei werden Nachbarschaften als Kreis- oder Ellipsenringe betrachtet, sodass lebende Zellen im inneren Radius aktivierend und lebende Zellen im äußeren Radius eine hemmende Wirkung auf ihre Nachbarn haben.

5 Praktische Umsetzung

Für die praktische Umsetzung haben wir uns für Python entschieden. Um genau zu sein, verwendeten wir den Spyder-Editor aus dem Anaconda-Paket, welcher uns im Kurs vorgestellt wurde. Zur Berechnung der Muster entwickelten wir zwei Klassen, mit denen wir jeweils das Grid und die Zellen dargestellt und simuliert haben. Das Grid wird darüber klassifiziert, dass es eine Zeilen- und Spaltenlänge hat. Zusätzlich wird es mit einer Chance versehen, mit der es am Anfang der Initialisierung eine Aktivatorzelle erstellt. Gibt man also als Startwerte für das Grid die folgenden Werte ein:

$$\text{Zeilenlänge} = 1980;\ \text{Spaltenlänge} = 1080;\ \text{Chance} = 12$$

Dann erhält man ein Gitter, das ein Full-HD Bild repräsentiert und in dem mit einer Wahrscheinlichkeit von 1 in 12 eine Zelle ein Aktivator wird. Das Grid setzt sich dann aus einem zweidimensionalen Array zusammen, das jede einzelne Position in diesem Gitter mit einem Objekt, in diesem Fall einer Zelle versieht.

```
class Grid:
        def __init__(self, zeilenlaenge, spaltenlaenge, chance):
                """ int zeilenl\"ange, int spaltenl\"ange, int chance
                        Stellt die Grundstruktur zur Erzeugung der Patterns"""

                self.zeilenlaenge = zeilenlaenge
                self.spaltenlaenge = spaltenlaenge
                self.grid = np.ndarray((zeilenlaenge, spaltenlaenge), \
                                        dtype = np.object)

        for z in range(zeilenlaenge):
                for s in range(spaltenlaenge):
                        self.grid[z][s] = Zelle(self, z, s, rd.randint(-1, 1), \
                                                randact(chance))
```

Einer Zelle wird am Start das Grid übergeben, in dem es sich befindet, die Position (mit Zeile und Spalte), ein Wert, der als interner Speicher genutzt wird und die Tatsache, ob sie einen Aktivator darstellen soll oder ob sie de facto tot ist. Einre Zelle werden am Start also folgende Werte übergeben: Grid, Zeile, Spalte, Wert und Activator. Diese werden genutzt, um auf Nachbarzellen zugreifen zu können und um die verschiedenen Zählalgorithmen anzuwenden. Die Zeile und Spalte sind zu Orientierung, also dass die Zelle weiß, wo sie sich im Grid befindet. Wert ist die Speichervariable, die zum Zählen genutzt wird und der Activator ist ein boolescher Wert, der angibt, ob die Zelle lebt oder nicht.

```
class Zelle:
        def __init__(self, grid, zeile, spalte, wert, activator):
                """" Grid grid, int zeile, int spalte, int wert, bool activator
                Zelle wird zur Bev\"olkerung des Grids benutzt"""

                self.grid = grid
                self.zeile = zeile
                self.spalte = spalte
                self.counter = 0
                self.wert = wert
                self.activator = activator
```

Der prinzipielle Ablauf zum Erstellen eines Patterns ist dabei wie folgend:

1. Erstelle Grid mit Zellen und bevölkere es zufällig mit Aktivatoren
2. Speichere dieses erste Grid als Bild zum späteren Nachvollziehen
3. Gehe zur ersten Zelle und überprüfe wie viele Aktivatoren sich in der Umgebung befinden - ein Beispiel hierzu befindet sich in Zeile 244 der Datei "CellMatrix.py"
4. Speichere diesen Wert und gehe zur nächsten Zelle
5. Wiederhole Schritt 3 und 4 solange, bis alle Zellen betrachtet wurden
6. Alle Zellen wechseln ihren Zustand anhand der gespeicherten Anzahl der Aktivatoren
7. Speichere das neue Grid als Bilddatei
8. Wiederhole Schritt 3 bis 7 solange, bis die gewünschte Anzahl der Iterationen erfüllt ist - oder bis keine Veränderungen mehr vorkommen (siehe dazu: Mögliche Verbesserungen & Ergänzungen)

6 Ergebnisse

Die entstandenen Bilder benötigten zwischen 12 und 20 Iterationen, um die Stabilität erkennbar zu machen.

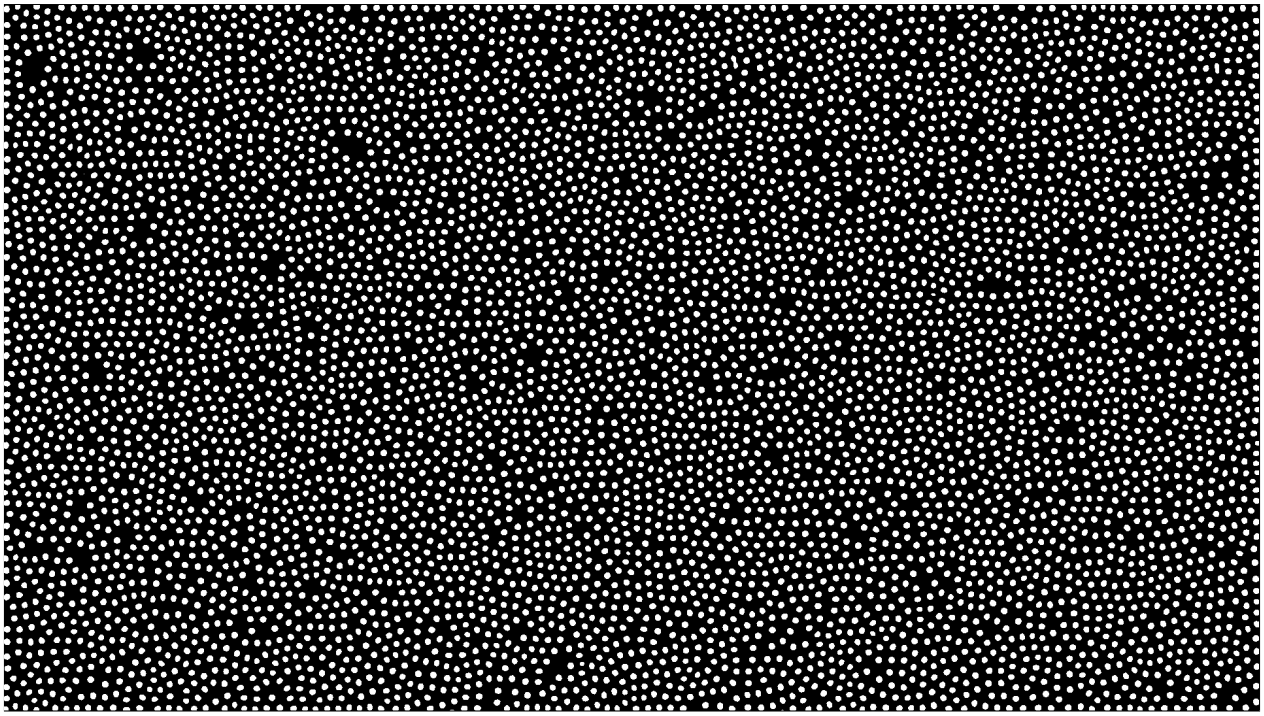


Figure 1: Es entstehen Punkte wenn der innere Radius 6, der äußere Radius 11 und die zufällige Chance einer Zelle zu Anfang gefärbt zu werden 1:20 beträgt.

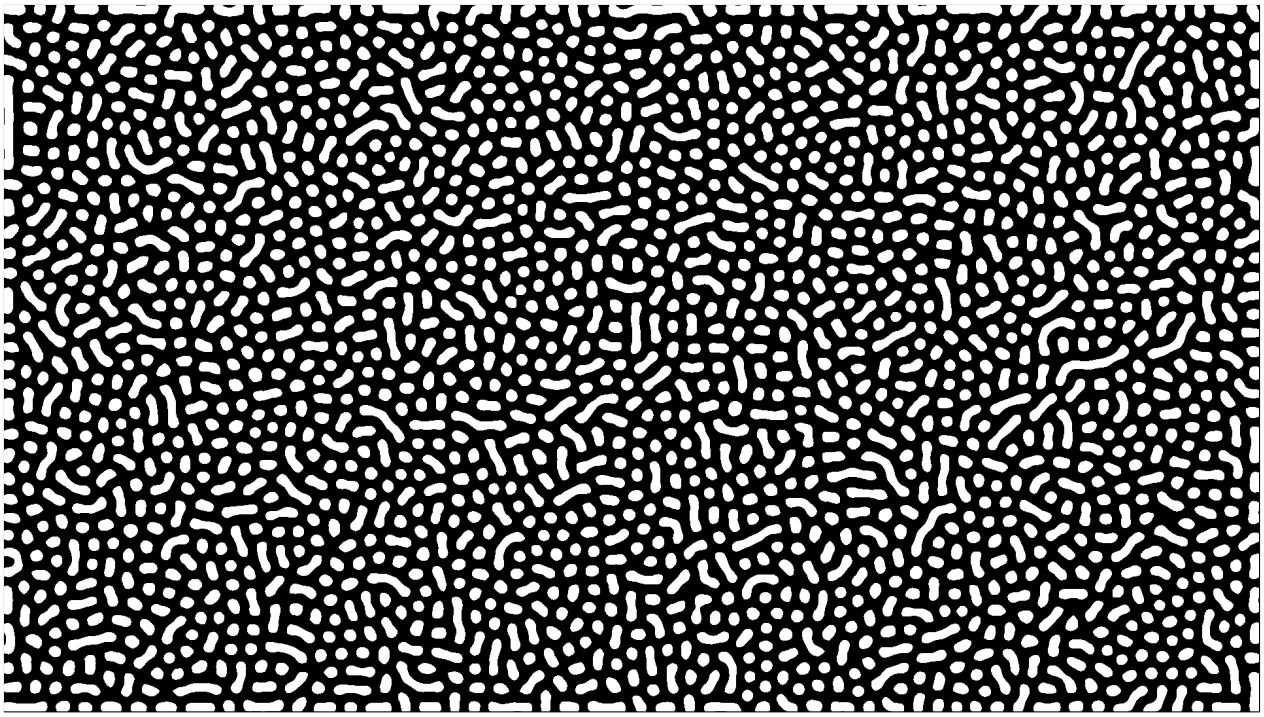


Figure 2: Es entstehen längere zusammnehängende Stücke wie Würmer, wenn der innere Radius 12, der äußere Radius 18 und die zufällige Chance einer Zelle zu Anfang gefärbt zu werden 1:16 beträgt.

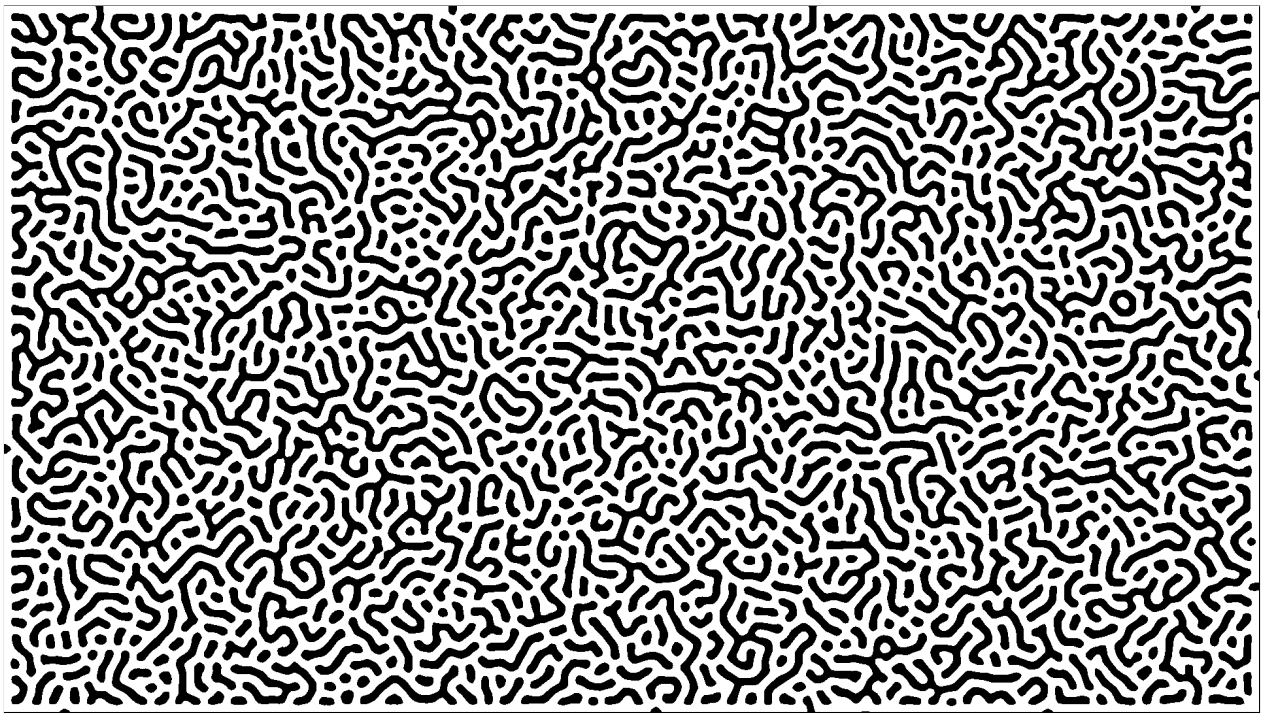


Figure 3: Es entstehen Punkte wenn der innere Radius 10, der äußere Radius 14 und die zufällige Chance einer Zelle zu Anfang gefärbt zu werden 1:12 beträgt.

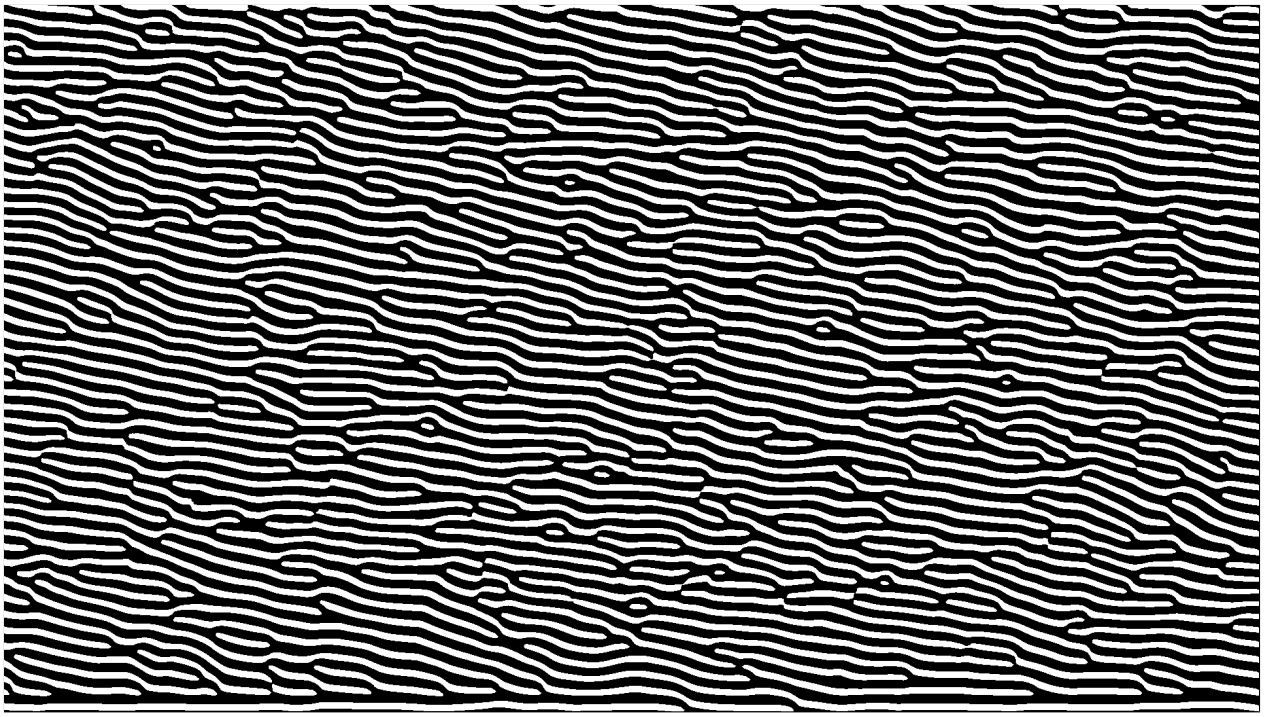


Figure 4: Es entstehen Punkte wenn die Ausmaße der Ellipsen 4 x 8 und 8 x 14 und die zufällige Chance einer Zelle zu Anfang gefärbt zu werden 1:15 betragen.

7 Benutzung und Features

7.1 Generierung neuer Patterns

Möchte man nun ein neues Pattern generieren, muss man die Datei "UI.py" öffnen, welche Abfragen in der Konsole stellt und anhand der Eingaben dann neue Patterns erstellt. Die dabei entstehenden Bilder werden in einem Unterordner namens "Generierte Bilder" gespeichert. Alle Iterationen eines Patterns werden in einem gemeinsamen Ordner gesammelt und durch die Informationen der Eingaben benannt.

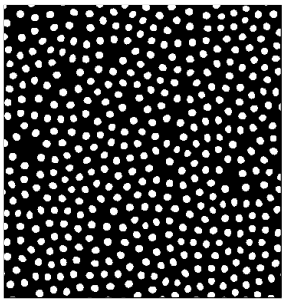
Das bedeutet, dass ein Ordner mit dem Namen "1920 x 1080 Modus 2 Chance 16 Ri 12 Ra 18" ein Bild mit der Auflösung 1920 x 1080 Pixel ist, das im Modus 2 generiert wurde. Der Modus 2 steht dabei für einen Kreis und Ri und Ra stehen dabei jeweils für Innen- und Außenradius; beide Werte sind in Pixeln angegeben. Desweiteren kann man an der Benennung erkennen, dass bei der Initiierung des Grids eine Wahrscheinlichkeit von 1 zu 16 bestand, dass eine Zelle ein Aktivator war. Die Bilder werden dann weiterhin mit dem Iterationsschritt benannt, sodass man diese leicht ordnen kann.

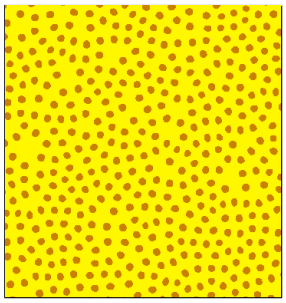
7.2 Fortsetzung einer Berechnung

Ein ergänzendes Feature ist die Möglichkeit vorhandene Bilder zu betrachten und anhand der Farben die Berechnungen fortzusetzen. Das ermöglicht sowohl die Möglichkeit Generierungen zu unterbrechen und zu einem späteren Zeitpunkt fortzufahren, ein Bild (beispielsweise in Paint) zu erstellen und dieses dann als Grundlage für den Algorithmus zu nehmen oder aber wechselnde Zonen und Radien auf das gleiche Bild anzuwenden.

7.3 Farbtausch

Zu guter Letzt gibt das kurze Skript "ColourSwapping.py" die Möglichkeit, Farben in einem zweifarbigen Bild auszutauschen. Dadurch kann man tierähnliche Farben in die Pattern-Generierung bringen. Dies hat eher einen veranschaulichenden Hintergrund als einen rein wissenschaftlichen.





Zum Testen der Hintergrundmethoden kann man noch die Datei "CellMatrix.py" direkt in der Konsole öffnen. Genaueres steht dazu in den Kommentaren ab Zeile 530.

8 Mögliche Verbesserungen & Ergänzungen

8.1 Andere Struktur

Eine mögliche Optimierung in der Struktur wäre die Verwendung eines höherdimensionalen Arrays, das dann die Nutzung der Unterklasse einer Zelle obsolet gemacht hätte. Wahrscheinlich handelt es sich dann um ein dreidimensionales Array, in denen dann die dritte Dimension zur Speicherung der Werte genommen worden wären anstatt des Zellen-Objektes. Dies hätte den weiteren Vorteil, dass pro Zelle zwei Integer weniger benötigt werden, da man der Zelle nicht mehr explizit sagen muss, wo sie sich befindet. Jedoch kam es als erstes zu der Entwicklung mit Zellen und danach erst wurden die Numpy-Arrays verwendet. Dies allein führte schon zu einer deutlichen Verringerung der Berechnungszeit, jedoch hätte man für mehr Umstrukturierung viele der Funktionen neu modellieren müssen und man hätte weniger Zeit zum Bilder erstellen gehabt. Da nicht bekannt war, in wie weit sich diese Umstellung zeitlich gelohnt hätte, konnte dieses Risiko nicht eingegangen werden und es wurde der sichere Weg bevorzugt, sodass wir zum Präsentationstermin fertige Bilder präsentieren konnten.

8.2 Multithreading

Der Gedanke zum Verwenden von Multithreading kam bereits relativ früh und wurde auch ein wenig getestet. Das Problem hierbei liegt allerdings in der Verwendung von Python und der Overhead, der beim Threading durch das Verschieben der Ergebnisse im Speicher generiert wird. Allerdings liegt dies auch daran, dass im ersten Versuch die Granulierung zum Erstellen der Threads zu klein gewählt wurde. So wurde mit jeder Zelle ein Thread generiert, welches bei größeren Bildern natürlich problematisch wird. Allerdings kam die Idee zur gröberen Granulierung erst zu spät während des Projektes auf. Das hatte dann zur Folge, dass die Umsetzung bis zur Präsentation erst einmal gestrichen war, da man auch bei dieser Verbesserung nicht sicher sein konnte, ob es sich tatsächlich lohnen würde, diese Zeit zu investieren. Eine mögliche Verbesserung dieses Problems in der Zukunft könnte eine Aufteilung entsprechend der Kerne sein, sodass dynamisch vom Pro-

gramm entschieden wird, in wie viele Teile das Grid zur Berechnung unterteilt wird. Erkennt das Programm einen Vierkernprozessor, so wird das Gitter geviertelt, beim Sechskern gesechstelt und so weiter. Das bringt allerdings nur etwas, wenn Python tatsächlich gut mit mehreren Threads skaliert und man sich sicher sein kann, dass das entsprechende Betriebssystem diese Aufgaben auch gleichmäßig auf die Kerne verteilt.

8.3 Konvergenz

Damit ist gemeint, dass die Pattern-Generierung solange laufen soll, bis es zu keiner weiteren Veränderung gegenüber der vorherigen Iteration mehr kommt. Dieses Feature wäre sehr sinnvoll zu ergänzen, da jedoch die Renderzeiten der einzelnen Bilder teilweise mehr als eine Stunde in Anspruch nahmen, hätte dieses Feature allerdings nie genutzt werden können. Deshalb wurde es ausgelassen und stattdessen die Möglichkeit hinzugefügt, die Berechnung abzubrechen und zu einem späteren Zeitpunkt weiterführen zu können.

8.4 Modulo-Betrachtung

Hiermit ist lediglich die Betrachtung des Grids als herumreichende Ebene gemeint, sodass die Zellen an einem Ende des Gitters auch die Zellen am anderen Ende des Gitters mit in Betracht ziehen. Dieses Feature wurde allerdings auch aufgrund von Zeitknappheit nicht weiter bearbeitet.