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 dann aufgrund der Nachbarschaftsumgebung dynamisch anpassen können.

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, das 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:

Zeilenlänge = 1980 ; Spaltenlänge = 1080 ; Chance = 12

Dann erhält man ein Gitter, welches ein Full-HD Bild repräsentiert, mit einer Wahrscheinlichkeit von 1 in 12 das eine Zelle ein Aktivator wird.

Das Grid setzt sich dann aus einem zweidimensionalen Array zusammen, welches jeder einzelnen Position in diesem Gitter mit einem Objekt einer Zelle versieht

#CodeBlock (Z.42 - Z.56 aus Cellmatrix.py)

Eine Zelle wird am Start das Grid übergeben, in dem es sich befindet, die Position (mit Zeile und Spalte), einem Wert, welcher als interner Speicher genutzt wird, und der Tatsache, ob sie ein Aktivator darstellen soll oder ob sie de facto Tod ist.

Eine Zelle werden am Start folgende Werte übergeben: Grid, Zeile, Spalte, Wert und Activator. Bei dem Grid handelt es sich um die Referenz auf das Grid, in dem sich die Zelle befindet. Dies wird dazu genutzt um auf Nachbarzellen zugreifen zu können und um die verschiedenen Zählalgorithmen anzuwenden. Die Zeile und Spalte sind zu Orientierung, also das die Zelle weiß wo sie sich im Grid befindet. Wert ist eine Speichervariable, die zum Zählen genutzt wird und der Activator ist ein boolescher Wert, der angibt ob die Zelle lebt oder nicht.

#CodeBlock (Z.179 - Z.189 aus Cellmatrix.py)

Der Prinzipielle Ablauf zum Erstellen eines Patterns ist dabei der Folgende

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 bist keine Veränderungen mehr Vorkommen (Siehe dazu: Mögliche Verbesserungen & Ergänzungen)

Benutzung und Features

Generierung neuer Patterns

Möchte man nun ein neues Pattern generieren, dann 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.

Dies 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.

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.

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.

Kleiner Ausschnitt der Bilder wie die Schwarzweißen Punkte zum Leopardenmuster wurden

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.

Mögliche Verbesserungen & Ergänzungen

1. Andere Struktur

Eine mögliche Optimierung in der Struktur währe die Verwendung eines höherdimensionalen Arrays, welches 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 zu dieser Entwicklung mit den Zellen als erstes und danach wurden erst 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 Bildererstellen gehabt. Da nicht bekannt war in wie weit diese Umstellung sich zeitlich gelohnt hätte, konnte dieses Risiko nicht eingegangen werden und es wurde der sichere Weg bevorzugt, sodass wir zum Präsentationstermin fertige Bilder präsentieren konnten.

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. Was dann zur Folge hatte, dass die Umsetzung bis zur Präsentation erst einmal gestrichen war, da man auch bei dieser Verbesserung nicht sicher sein konnte, ob es sich der Kosten-Nutzen-Faktor tatsächlich lohnen würde diese Zeit zu investieren.

Eine mögliche Verbesserung in der Zukunft für dieses Problem könnte eine Aufteilung entsprechend der Kerne sein. So dass dynamisch vom Programm 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. Jenes 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.

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 nutzen können also wurde es ausgelassen und anstatt dessen die Möglichkeit hinzugefügt die Berechnung abzubrechen und zu einem späteren Zeitpunkt weiterführen zu können.

4. Modulo-Betrachtung

hiermit ist lediglich die Betrachtung des Grids als rumreichende Ebene gemeint. Also das die Zellen a 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.