Inhaltsverzeichnis

[Vorwort 2](#_Toc322106768)

[Definition: Zelluläre Automaten 3](#_Toc322106769)

[Geschichte 3](#_Toc322106770)

[Meine Simulation 4](#_Toc322106771)

[Das Programm 4](#_Toc322106772)

[Der Algorithmus 6](#_Toc322106773)

[Versuche & Analysen 8](#_Toc322106774)

[Versuche 8](#_Toc322106775)

[Quellen & Links 9](#_Toc322106776)

# Vorwort

Bevor ich zu der eigentlichen Facharbeit komme, hier einige Worte vorweg; Diese Facharbeit hat nicht den Anspruch zelluläre Automaten von Grund auf neu zu erfinden und zu erforschen, sie ist lediglich ein Versuch Zelluläre Automaten an einem Bespiel zu untersuchen und zu visualisieren. Die durchgeführte Simulation, stellt die Ausbreitung eines Waldbrandes anhand einer in Java geschriebenen Simulation dar. Die Simulation steht zum Download unter [L1]. Mit dieser Angabe, komme ich schon zu der Quellenangabe. Eine Quelle ist durch eine in eckigen Klammern geschriebene Zahlen- und Buchstabenkombination (z.B. Q5 für „Quelle 5“) angegeben. Die exakte Quelle oder der Link steht im Abschnitt „Quellen & Links“.

Dieses Dokument beginnt mit der Definition der Zellulären Automaten, danach folgt eine Einführung in die Idee und die Arbeitsweise meiner Simulation. Schließlich folgt eine Auswertung der Ergebnisse des Verhaltens.

Wichtiger Hinweis: Der abgedruckte Code kann vom tatsächlichen Code abweichen, da dieser Kommentare aufweist und z.T. nachträglich leicht verändert wurde. Die Funktionsweise ist aber stets identisch.

# Definition: Zelluläre Automaten

Der Wikipedia Definition nach sind Zelluläre Automaten eine Methode zur „Modellierung räumlich diskreter dynamischer Systeme, wobei die Entwicklung einzelner Zellen zum Zeitpunkt *t+1* primär von den Zellzuständen in einer vorgegebenen Nachbarschaft und vom eigenen Zustand zum Zeitpunkt *t* abhängt.“ [Q1] Ein zellulärer Automat besteht aus mehreren Zellen in Raum und einer oder mehreren Vorschriften die die Entwicklung bestimmten.

Jede Zelle befindet sich in einem aus einer bestimmten Anzahl von Zuständen (z.B. „An“ und „Aus“). Zusätzlich hat sie eine *Nachbarschaft* die häufig auch sie selbst einschließt und relativ von ihr besteht (z.B. „alle angrenzenden Zellen“). Jeder Zelle wird ein Anfangsstatus zugeordnet, der zum Zeitpunkt *t=0* gilt. Mit dem Fortschreiten der Zeit werden die Zellzustände den spezifizierten Regeln nach verändert.

Der Raum ist durch eine endliche Zahl von Dimensionen bestimmt und kann *endlich* oder *unendlich* sein.

## Geschichte

1940 arbeitete John von Neumann in Los Alamos an selbst-reproduzierenden Systemen. Zur gleichen Zeit untersuchte sein Kollege Stanislaw Ulam das Wachstum von Kristallen. Von Neumann’s beruhte auf der Idee, dass ein Roboter einen anderen baut. Als er daran arbeitete erkannte er die Schwierigkeit einen solchen Roboter zu bauen und ihn mit genügend Teilen zu versorgen. Ulam empfahl ihn sein System mathematisch zu abstrahieren so wie auch Ulam es tat. Der erste Zelluläre Automat war geboren. Der vorgestellte Automat hatte 29 Zustände und konnte sich immer wieder selbst reproduzieren. Dieses Design ist bekannt als „tesselation model“ (dt. Parkettierung).

1969 veröffentlichte Korad Zuse sein Buch „Rechnender Raum“. In diesem nimmt er an dass alle Naturgesetze bestimmten Regeln unterworfen ist und das gesamte Universum als Zellulärer Automat gesehen werden kann.

1970 wurde das von John Horton Conway entwickelte „Game of Life“ vorgestellt. Die Regeln lauteten: Hat eine Zelle zwei Nachbarn behält sie ihren Zustand. Hat sie drei Nachbarn wird sie schwarz. Liegen andere Zustände vor wird sie weiß. Trotz dieser einfachen Regeln ergeben sich viele verschiedene Verhaltensweisen zum Beispiel *Gleiter*, Anordnungen die sich von selbst durch den Raum bewegen.

1983 veröffentlichte Stephen Wolfram grundlegende Arbeiten zu „elementary cellular automata“ (dt. elementare Zellulären Automaten). Die Komplexität die diese simplen Regeln zeigten, ließen ihn annehmen, dass die Natur durch ähnliche Mechanismen gesteuert wird.

*Quelle: Wikipedia [Q1 & Q2]*

# Meine Simulation

Wie bereits erwähnt, stellt meine Simulation die Ausbreitung eines Brandes in einem Wald dar. Im Folgenden werde ich die Wirkungsweise des von mir entwickelten Programms und des dahintersteckenden Algorithmusses erläutern.

## Das Programm

Da ich das Programm mit der „lwjgl“ und „slick“ gebaut habe, war die Grundstrucktur bereits gegeben. Die Klasse ist folgendermaßen definiert:

**public** **class** simulation **extends** BasicGame**{**

Es folgen die leeren Methodenbehälter:

**public** simulation**()** **{**

**}**

**public** **void** init**(**GameContainer gc**)** **throws** SlickException **{**

**}**

**public** **void** update**(**GameContainer gc**,** **int** delta**)** **throws** SlickException **{**

**}**

**public** **void** render**(**GameContainer gc**,** Graphics g**)** **throws** SlickException **{**

**}**

Wie die Namen erahnen lassen, ist das erste der Konstruktor. Die zweite Methode wird einmal beim Starten des Programms aufgerufen. Die anderen Beiden Methoden werden immer nacheinander aufgerufen. Erst updatezur Berechnung des Ablaufes und dann render zum Zeichnen auf den Bildschirm.

Zuerst werden globale Variablen initialisiert.

**public** Image white**,**red**,**green**;**

**public** **int[][]** field**;**

**public** **int** width**,**height**,**scale**,**spread**;**

**public** **float** prob**;**

**public** TextField outputField**;**

**public** **int** mode**,** size**,** drawing**;**

Da die Simulation mit verschiedenen Parametern aufrufbar sein soll, ist dem Konstruktor eine Reihe von Parametern gegeben.

**public** simulation**(int** width**,** **int** height**,** **int** scale**,** **int** spread**,** **float** prob**)**

**{**

**super(**"simulation"**);**

**this.**width **=** width**;**

**this.**height **=** height**;**

**this.**scale **=** scale**;**

**this.**spread **=** spread**;**

**this.**prob **=** prob**;**

**this.**mode **=** 0**;**

**this.**size **=** 2**;**

**this.**drawing **=** 0**;**

**this.**field **=** **new** **int[**width**][**height**];**

**}**

Hier wird der Konstruktor der Klasse BasicGame aufgerufen und die anfangs erstellten Variablen mit Werten aus den Parametern gefüllt. Zuletzt wird ein 2-Dimensionales Array erstellt, welches das Feld darstellt.

Danach wird die Methode init() bestimmt. Diese wird beim Start einmal aufgerufen, sie ist also ideal um die Bilder die später als kleine Punkte dargestellt werden zu laden und das Feld Array mit Zahlen zu füllen. Hier ist anzumerken, dass die Bilder nur 20x20 große, einfarbige Flächen sind. In „slick“ ist es schneller Bilder als Bitmaps in den Speicher zu legen und von dort aus zu zeichnen, anstatt die Farben selbst zu zeichnen.

* Der hier angesprochene Code steht im Anhang „Die Methode init()“.

Zunächst werden drei weitere globale Variablen mit ihrem Namen entsprechenden Bildern/ Farben gefüllt. Darunter werden zwei Zählvariablen für die for Loops initialisiert. Die Loops füllen dann das Feld Array. Mit der gegeben Wahrscheinlichkeit starten die Zellen entweder im Zustand 2 oder 1. An dieser Stelle wäre eine Übersicht über die Zustände angebracht.

|  |  |
| --- | --- |
| 0 | Es steht kein Baum an dieser Stelle. Die Zelle ist weiß. |
| 1 | Es steht ein Baum an dieser Stelle. Die Zelle ist grün. |
| 2 | Der Baum in dieser Position steht in Flammen. Im nächsten Durchlauf wird er abgebrannt sein. Die Zelle ist rot. |

Die Wahrscheinlichkeit wird folgendermaßen berechnet: Die Funktion Math.random() gibt eine zufällige Zahl zwischen 0 und 1 zurück. Die Variable this.prob enthält die angegebene Wahrscheinlichkeit. Es wird überprüft ob Math.random() größer ist als 1 - this.prob. Somit lässt sich die Wahrscheinlichkeit präzise festlegen. Trifft diese Wahrscheinlichkeit nicht zu, wird der Zustand 1 in das Feld geschrieben.

Nach der Methode init() folgt die Methode update(). Diese ist dafür zuständig die Berechnungen bzw den Algorithmus auszuführen. Diese wird im Abschnitt „Der Algorithmus“ weiter erklärt. Nach der Methode update() folgt render(). Diese Methode ist dafür zuständig, das Feld auf den Bildschirm zu zeichnen.

**public** **void** render**(**GameContainer gc**,** Graphics g**)** **throws** SlickException **{**

**int** i**,**j**;**

**for** **(**i **=** 0**;** i **<** **this.**width**;** i**++)**

**{**

**for** **(**j **=** 0**;** j **<** **this.**height**;** j**++)**

**{**

**switch** **(this.**field**[**i**][**j**])**

**{**

**case** 1**:**

**this.**green**.**draw**(**i **\*** **this.**scale**,**j **\*** **this.**scale**,this.**scale**);**

**break;**

**case** 2**:**

**this.**red**.**draw**(**i **\*** **this.**scale**,**j **\*** **this.**scale**,this.**scale**);**

**break;**

**case** 0**:**

**this.**white**.**draw**(**i **\*** **this.**scale**,**j **\*** **this.**scale**,this.**scale**);**

**break;**

**}**

**}**

**}**

**}**

Zunächst werden die beiden Zählvariablen i und j deklariert. In den darauffolgenden for Schleifen wird das Feld-Array Zelle für Zelle durchlaufen und dem Zustand entsprechend entweder ein weißer, ein roter, oder ein grüner Punkt gemalt. Dabei wird auf die draw() Methode der Images zurückgegriffen.

## Der Algorithmus

Hier erläutere ich die update() Methode bzw. den Algorithmus der hinter der Berechnung der Zustandsänderung steht.

**public** **void** update**(**GameContainer gc**,** **int** delta**)** throws SlickException **{**

 **if** **(this.**mode **==** 1**)** **{**

 **int** i**,**j**;**

 **for** **(**i **=** 0**;** i **<** **this.**width**;** i**++)** **{**

 **for** **(**j **=** 0**;** j **<** **this.**height**;** j**++)** **{**

 **if** **(this.**field**[**i**][**j**]** **==** 2**)** **{**

 **int** c**;**

 **for** **(**c **=** 0**;** c **<=** **this.**spread**;** c**++)** **{**

 **double** rand **=** Math**.**random**()** **\*** 9**;**

 **int** column **=** **(int)**Math**.**round**(**rand **%** 3**);**

 **int** row **=** **(int)((**rand **-** column**)** **/** 3**)** **+** 1**;**

 column **+=** 1**;**

 **if** **(((**i **+** **(**row **-** 2**))** **>** 1**)** **&&** **((**i **+** **(**row **-** 2**))** **<** **this.**width**)** **&&** **((**j **+** **(**column **-** 2**))** **>** 1**)** **&&** **((**j **+** **(**column **-** 2**))** **<** **this.**height**))** **{**

 **if** **(this.**field**[**i **+** **(**row **-** 2**)][**j **+** **(**column **-** 2**)]** **==** 1**) {**

 **this.**field**[**i **+** **(**row **-** 2**)][**j **+** **(**column **-** 2**)]** **=** 2**;**

 **}**

 **}**

 **}**

 **this.**field**[**i**][**j**]** **=** 0**;**

 **}**

 **}**

 **}**

 **}**

 **else** **if** **(this.**mode **==** 0**)** **{**

 **if** **(**gc**.**getInput**().**isKeyDown**(**Input**.**KEY\_ENTER**))** **{**

 **this.**mode **=** 1**;**

 **}**

 **}**

**}**

Anfangs wird überprüft ob die Simulation bereits begonnen hat, oder ob sie sich noch im Zeichnermodus befindet. Ist dies nicht der Fall, werden die schon öfters verwendeten Zählvariablen i und j deklariert. Die Schleife ist wieder dafür zuständig auf jede einzelne Zelle zuzugreifen. Dann wird überprüft, ob die Zelle ein brennender Baum (Zustand 2)ist, oder nicht. Ist dies der Fall wird die Zählvariable c deklariert. In der for Schleife wird dann dem Parameter spread entsprechend gezählt auf welche bzw. auf wie viele angrenzende Zellen das Feuer überspringt. Dann wird eine zufällige Zahl zwischen 0 und 9 generiert. Durch die Rechnung Math.round(rand % 3) bekommt man dann entweder 0,1 oder 2 als Ergebnis. Dies ist die Spalte in der Tabelle der Umliegenden Zellen. Das Modell sieht in etwa so aus:

|  |  |  |
| --- | --- | --- |
| 1|1 | 2|1 | 3|1 |
| 1|2 | 2|2 | 3|2 |
| 1|3 | 2|3 | 3|3 |

Die erste Zahl stellt die Spalte dar, die zweite die Zeile. In dem Teil

**int** row **=** **(int)((**rand **-** column**)** **/** 3**)** **+** 1**;**

wird dann die Zeile berechnet. Indem von der zufälligen Zahl zwischen 0 und 8 die Spalte abgezogen wird und das Ergebnis durch 3 geteilt wird, liegt das Ergebnis immer zwischen 0 und 2. Durch Math.round() wird es dann auf die nächstliegende natürliche Zahl gerundet. Beispiel:

rand = 7,356

column = rand % 3 = 1,356 ≈ 1

rand – column = 6,356

6,356 / 3 = 2,119 ≈ 2

2 + 1 = 3

Nach dieser Berechnung wird noch überprüft, ob die betreffenden Zellen überhaupt existieren, oder die mittlere Zelle ganz am Rand ist. Ist dies nicht der Fall, bekommt die Zelle den Zustand 2 zugeordnet.

Nachdem die Schleife spead entsprechend oft durchgelaufen ist, wird die Ausgangszelle auf Zustand 0 gesetzt.

In diesem Algorithmus stecken versteckte Wahrscheinlichkeiten, die nicht außer Acht gelassen werden dürfen. So ist es beispielsweise möglich, dass das Ergebnis der Rechnung 2|2 zum Ergebnis hat. Diese Zelle ist allerdings die Ausgangszelle und hat ohnehin schon den Zustand 2. Die Wahrscheinlichkeit dazu beträgt 1:9 also ~11,1% Zudem ist es möglich, dass die errechnete Zelle am Rand des Feldes liegt. In dem Fall findet auch keine Veränderung statt. Die Wahrscheinlichkeit für diesen Fall berechnet sich folgendermaßen:

((Breite \* Höhe) / ((Breite + Höhe) \* 2)) \* $\frac{1}{3}$

Erklärung:

(Breite\*Höhe) ist die Anzahl aller Zellen. ((Breite + Höhe) \* 2) ist die Zahl der Zellen auf der Außenlinie (der Umfang). Das Ergebnis der Division von Umfang und Anzahl ist die Wahrscheinlichkeit, dass die aktuelle Zelle auf der Außenlinie ist. An jedem Punkt auf der Außenlinie besteht eine Chance von $\frac{1}{3}$ (33%), dass die errechnete Zelle außerhalb des Feldes liegt. So erklärt sich die Formel.

Kommt das if Statement am Anfang der Klassendefinition zu dem Ergebnis, dass sich das Programm noch im Zeichnermodus befindet, so wird lediglich überprüft ob die Taste Enter gedrückt ist. Trifft dies zu, wird der Zeichnermodus beendet und die eigentliche Simulation beginnt.

# Versuche & Analysen

In diesem Teil erläutere Ich die unterschiedlichen Versuche, ihre Ergebnisse und die Analysen dazu.

## Versuche

Quellen & Links

Q1: http://de.wikipedia.org/wiki/Zellul%C3%A4rer\_Automat (10.04.2012)

Q2: http://en.wikipedia.org/wiki/Cellular\_automaton (10.04.2012)

L1: http://download.jangxx.com/Random\_Files/facharbeit/simulation/