Gymnasium Arnoldinum Schuljahr 2011/12

Jahrgangsstufe: Q1

Facharbeit

Zelluläre Automaten

Am Beispiel Waldbrand

Name: Jan Scheiper
Kurs: M01, Herrn Walters
Abgabetermin: 24.04.2012
Bearbeitungszeit: 8 Wochen

Inhaltsverzeichnis

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 Information, 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 & Hilfsmittel“.

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 tatsächlich benutzen Code abweichen, da dieser Kommentare enthält und z.T. nachträglich leicht verändert wurde. Die Funktionsweise ist aber identisch.

Definition: Zelluläre Automaten

Der Wikipedia Definition zufolge 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 bestimmte Zahl von Dimensionen definiert 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 ihm sein System mathematisch zu abstrahieren, so wie auch er 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

Der Konstruktor

Da das Programm mit der „lwjgl“ („light weight java game library) und „slick“ gebaut ist, war die Grundstruktur 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 `update` zur Berechnung der Veränderung und dann `render` zum Zeichnen auf den Bildschirm.

Zuerst werden globale Variablen initialisiert.

```
public int[][] field, bkupField;
public int width,height,scale,spread;
public float prob;
public int mode, size, drawing, runs;
public int runCount = 0;
public boolean disp;
```

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,boolean
disp, int runs) {
   super("simulation");
   this.width = width;
   this.height = height;
   this.scale = scale;
```

```
    this.spread = spread;
    this.prob = prob;
    this.field = new int[width][height];
    this.bkupField = new int[width][height];
    this.mode = 0;
    this.size = 2;
    this.drawing = 0;
    this.disp = disp;
    this.runs = runs;
}
```

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 und eines mit dem selben Inhalt zum Backup.

Die Parameter in der Übersicht:

`this.width`	Die Breite des Feldes in Zellen.
`this.height`	Die Höhe des Feldes in Zellen.
`this.scale`	Die Skalierung (Wie viele Pixel ist eine Zelle groß)
`this.spread`	Die Ausbreitung (Pro Rechendurchlauf, auf wie viele anliegende Zellen springt das Feuer über
`this.prob`	Die Wahrscheinlichkeit (engl. prob[ability]) mit der bei der Generierung des Feldes ein Feld im Zustand 2 startet.
`this.mode`	Der Modus in dem sich die Simulation befindet. 0: Zeichenmodus; 1: Die Simulation läuft; 2: Die Simulation in durchgelaufen und wartet auf Beendigung.
`this.size`	Die Größe (Radius) des Zeichenstiftes in Zellen.
`this.drawing`	Der Modus („Die Farbe“) des Zeichenstiftes.
`this.field[][]`	Das Array, das alle Zellen enthält.
`this.bkupField[][]`	Das Array, das alle Zellen als Backup enthält um mehrfache Durchläufe mit denselben Anfangsbedingungen zu ermöglichen
`this.disp`	`true` oder `false`, ob das Feld angezeigt werden soll oder nicht. Aus Performancegründen ist es sinnvoll bei vielen Durchläufen die Anzeige zu deaktivieren.
`this.runs`	Die Anzahl der Durchläufe, die das Programm noch machen muss.
`runCount`	Die Anzahl der durchlaufenen update-Zyklen im aktuellen Durchlauf

Danach wird die Methode `init()` bestimmt. Diese wird beim Start einmal aufgerufen.

Die Methode init()

➔ Der hier behandelte Code steht im Anhang „Die Methode init()“.

Zunächst wird ein `MouseListener` hinzugefügt. Dieser erlaubt das Bearbeiten des Feldes, bevor die Simulation anfängt. In diesem Programm werden allerdings nur die Methoden `mouseWheelMoved`, `mouseClicked` und `mouseDragged` benötigt. Die anderen Methoden sind zwar deklariert, aber leer.

Die Methode `mouseWheelMoved` wird immer dann aufgerufen, wenn das Mausrad gedreht wird. Zuerst wird überprüft in welchem Modus sich die Simulation aktuell befindet. Ist sie im Zeichenmodus, wird überprüft ob das Mausrad vor- oder zurückgedreht wurde (`change` größer oder kleiner als 0). Je nach Ergebnis wird die Stiftgröße entweder um zwei Pixel vergrößert, oder verkleinert.

Die Methode `mouseClicked` wird immer dann aufgerufen, wenn der Benutzer die Maus klickt. In diesem Programm ist allerdings nur der Rechtsklick relevant, daher das `if` Statement. Das darauffolgende `switch` Statement dient dem Umschalten der Zeichenmodi. Ist der Modus 0, wird er auf 1 gesetzt. Ist der Modus 1, auf 2 und ist er 2, auf 0.

Die Methode `mouseDragged` wird immer dann aufgerufen, wenn die Maus mit gedrückter Maustaste bewegt wird. Die Methode bekommt die Parameter `oldx`, `oldy`, `newx` und `newy` übergeben. Die zwei erstgenannten sind die Pixelkoordinaten des Bewegungsanfangs; die letzten beiden sind die Pixelkoordinaten des Bewegungsendes. In diesem Programm werden allerdings nur die Koordinaten des Bewegungsendes berücksichtigt, das sich diese Methode so schnell updatet, dass die Anfangskoordinaten ohnehin „übermalt“ werden. Nach dem sichergestellt worden ist, dass sich das Programm im Zeichenmodus befindet, wird der Bereich um den Endpunkt der Bewegung mit der gewählten „Farbe“ gefüllt. Dazu laufen zwei `for`-Schleifen; eine für die x-Werte, die andere für die y-Werte um jeden Punkt in dem Bereich zu erreichen. Das längliche `if` Statement ist lediglich eine Absicherung, damit das Zeichnen an der Kante des Feldes möglich ist und keine Werte außerhalb der Arraygröße auftreten. Hierbei wird überprüft, ob die Koordinaten der nächsten Zelle innerhalb des Feldes liegen. Dazu wird die x-Koordinate der Position durch den Skalierungsfaktor geteilt und der Feldgröße gegenübergestellt. Der erste Schritt ist notwendig um Pixelkoordinaten in Zellenkoordinaten umzurechnen. Das Ergebnis wird außerdem darauf überprüft, ob es größer als Null ist. Derselben Rechnung wird auch die kommende y-Koordinate unterzogen. Ist sichergestellt, dass die Koordinaten innerhalb des Feldes liegen, wird die entsprechende Zelle auf den neuen Zustand (die „Farbe“) gesetzt. Zusätzlich wird der Zustand auch in das Backup-Array geschrieben.

Nach dem `MouseListener` 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 ist 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.
3	Dieser Zustand wird nicht angezeigt. Zellen mit diesem Zustand werden kurz darauf in den Zustand 2 versetzt. Dieser Zwischenzustand ist nur notwendig, da sonst einzelne Zellen in einem Zyklus mehrmals verändert werden können.

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 diesem `if` Statement wird der Zellenzustand in das Backup-Array übertragen.

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.

Die Methode render()

```
public void render(GameContainer gc, Graphics g) throws SlickException {
     if (this.disp == true)
     {
        int i,j;
        for (i = 0; i < this.width; i++)
        {
           for (j = 0; j < this.height; j++)
           {
              switch (this.field[i][j])
              {
              case 1:
                 gc.getGraphics().setColor(new Color(0,150,0));
                 gc.getGraphics().fillRect(i * this.scale,j *
this.scale,this.scale,this.scale);
                 break;
              case 2:
                 gc.getGraphics().setColor(Color.red);
                 gc.getGraphics().fillRect(i * this.scale,j *
this.scale,this.scale,this.scale);
                 break;
              case 0:
                 gc.getGraphics().setColor(Color.white);
                 gc.getGraphics().fillRect(i * this.scale,j *
this.scale,this.scale,this.scale);
                 break;
              }
           }
        }
     }
  }
```

Zunächst wird überprüft, ob das Feld überhaupt gezeichnet wird. Ist dies der Fall, 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 gezeichnet. Dazu wird die Zeichenfarbe auf die dem Zustand entsprechende Farbe gesetzt. Danach wird ein Rechteck gezeichnet und gefüllt, welches den Koordinaten multipliziert mit dem Skalierungsfaktor entspricht.

Der Algorithmus/ Die Methode update()

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

➔ Der hier behandelte Code steht im Anhang „Die Methode update()“.

Anfangs wird überprüft ob die Simulation bereits begonnen hat, oder ob sie sich noch im Zeichenmodus befindet. Ist dies nicht der Fall, wird der Zyklenzähler um 1 hochgezählt. Danach werden die schon öfters verwendeten Zählvariablen i und j, die für den Durchlaufzähler wichtige Variable `nothingCount` und die zu statistischen Zwecken benutzte Variable `burningCount` 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 3 zugeordnet.

Nachdem die Schleife entsprechend dem Parameter `spread` oft durchgelaufen ist, wird die Ausgangszelle auf den 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.

Nach dieser Schleife wird eine ähnliche `for` Schleife durchlaufen, die alle Zellen mit dem Zustand 3 auf den Zustand 2 setzt. Dieser Umweg ist notwendig, da sich sonst Zellen die auf den Zustand 2 gesetzt wurden, zwei Schleifendurchläufe später schon wieder ausbreiten könnten. In dieser Schleife wird auch der Zähler für die aktuell in Flammen stehenden Zellen aktualisiert. Zusätzlich wird der Zähler für die unbearbeiteten Felder aktualisiert. Dieser ist dazu da festzustellen, wann die Simulation zu Ende ist, also keine brennenden Zellen mehr vorhanden sind.

Danach werden Informationen zur aktuell durchlaufenen Anzahl an Zyklen und den aktuell in Flammen stehenden Zellen in die Konsole ausgegeben.

Zuletzt kommt noch das if Statement welches für die Durchläufe zuständig ist. Ist der Zähler für unbearbeitete Durchläufe so groß wie die Gesamtzahl der Zellen, so ist die Simulation zu Ende. In dem darauffolgenden if Statement wird überprüft, ob die Simulation noch weiterläuft. Ist dies der Fall, so wird der Zyklenzähler auf 0 gesetzt. Nachdem der Durchlaufzähler um 1 verringert wurde, wird überprüft, ob er immer noch größer als 0 ist. Ist er kleiner oder gleich 0, wird das Programm in den Beendigungsmodus versetzt. Ansonsten wird das anfangs gespeicherte Feld geladen und die Simulation beginnt erneut. Hier wird die Methode `cloneArray()` benutzt. Diese ist notwendig, da Java Arrays über den Operator „=“ nicht kopiert, sondern nur der Pointer übergibt.

Beispiel:

```
int[] array1 = new int[1], array2 = new int[1];
array1[0] = 10;
array2[0] = 20;
array2 = array1;
array2[0] = 100;
System.out.println(array1[0]);
```

⇨ In der Konsole steht 100

Man sieht, dass nur der Pointer übertragen wurde, nicht aber der Inhalt des Arrays. Für 1-dimensionale Array gibt es dafür die Methode `.clone()`. Diese funktioniert aber nicht bei 2-dimensionalen Arrays. Daher ist eine eigene Methode erforderlich;

```
public int[][] cloneArray(int[][] array)
{
   int rows = array.length;
   int[][] newarray = (int[][]) array.clone();
   for (int row = 0; row < rows; row++)
   {
      newarray[row] = (int[]) array[row].clone();
   }
   return newarray;
}
```

Die Anzahl der Arrays in den Arrays (die Größe der zweiten Dimension) wird in die Variable `rows` gespeichert. Dann wird ein neues Array erstellt und mit der groben Struktur (der Größe der beiden Dimensionen) gefüllt. In der `for` Schleife wird dann in jedes Feld das entsprechende Array kopiert. Somit hat man am Ende eine Kopie des Array im Speicher. Ein Pointer darauf wird zurückgegeben.

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

Kommt das `if` Statement am Anfang der Klassendefinition jedoch zu dem Ergebnis, dass sich das Programm im Beendigungsmodus befindet, so wird lediglich überprüft ob die Taste Enter gedrückt ist. Ist dies der Fall, wird das Programm beendet.

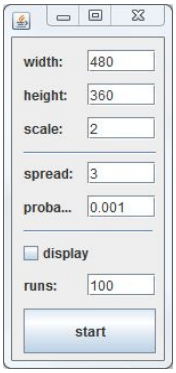
Versuche & Analysen

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

Versuche

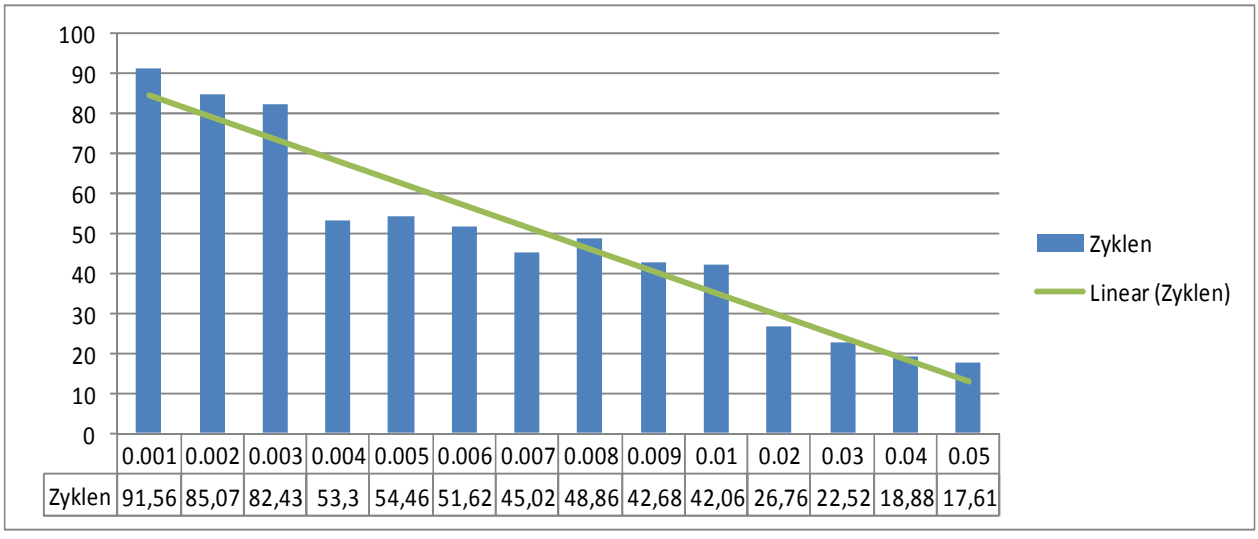
Durchschnittliche Anzahl an Zyklen

In einem ersten Versuch werde ich die durchschnittliche Anzahl der benötigten Zyklen in 100 Durchläufen mit den Standarteinstellungen ermitteln.



Die Ergebnisse sind rechts in einer Tabelle eingetragen. Der Durchschnitt der Ergebnisse ist 91,56. Im Vergleich dazu habe ich die Wahrscheinlichkeit auf 0,005 erhöht. Das Ergebnis: Durchschnittlich 54,46 Zyklen. Erhöhe ich die Wahrscheinlichkeit um das 5-fache, halbiert sich die Anzahl der Zyklen. In einem weiteren Versuch verzehnfache ich die Wahrscheinlichkeit, setze sie also auf 0,01. Das Ergebnis ist 46,06. Offensichtlich erhöht sich die Anzahl der Zyklen in Relation zur Wahrscheinlichkeit nicht linear. Also habe ich weitere Versuche mit den Wahrscheinlichkeiten 0,002 - 0,009 und 0,02 - 0,05 gemacht. Die Ergebnisse sind in diesem Diagramm dargestellt.

114	81	91	87	96
96	109	113	89	93
87	90	84	106	82
83	82	98	97	84
83	89	82	94	103
94	136	95	92	93
89	116	99	90	83
85	79	119	86	92
86	91	83	84	110
84	83	86	86	87
89	86	87	104	82
90	93	85	97	85
94	99	92	81	88
91	84	92	84	94
109	84	92	77	94
82	97	77	87	104
79	94	96	80	92
91	83	105	79	79
104	82	102	99	83
83	91	82	94	117

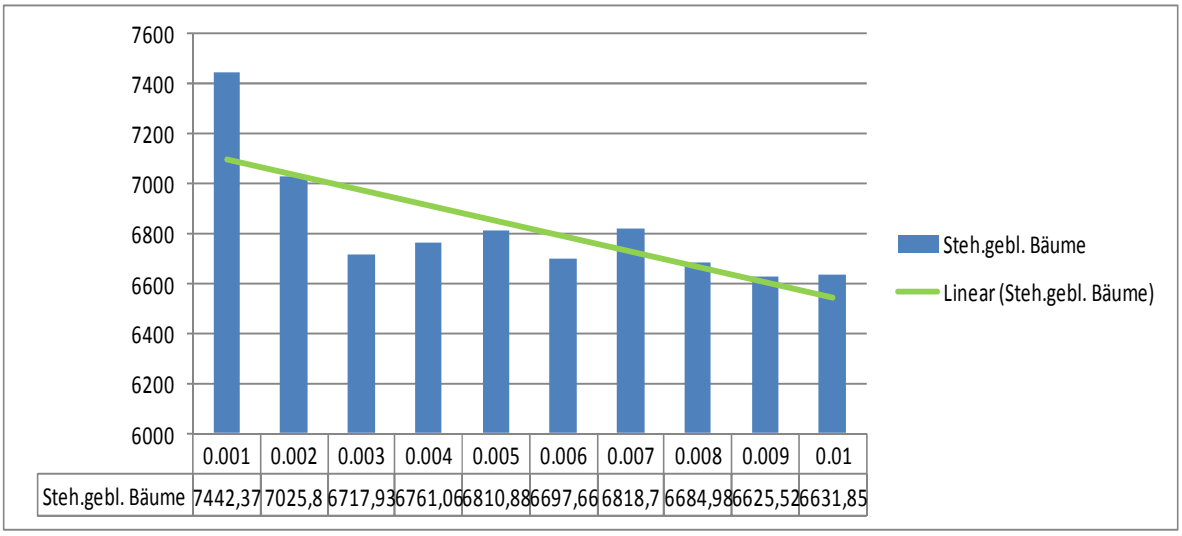


	0.001	0.002	0.003	0.004	0.005	0.006	0.007	0.008	0.009	0.01	0.02	0.03	0.04	0.05
Zyklen	91,56	85,07	82,43	53,3	54,46	51,62	45,02	48,86	42,68	42,06	26,76	22,52	18,88	17,61

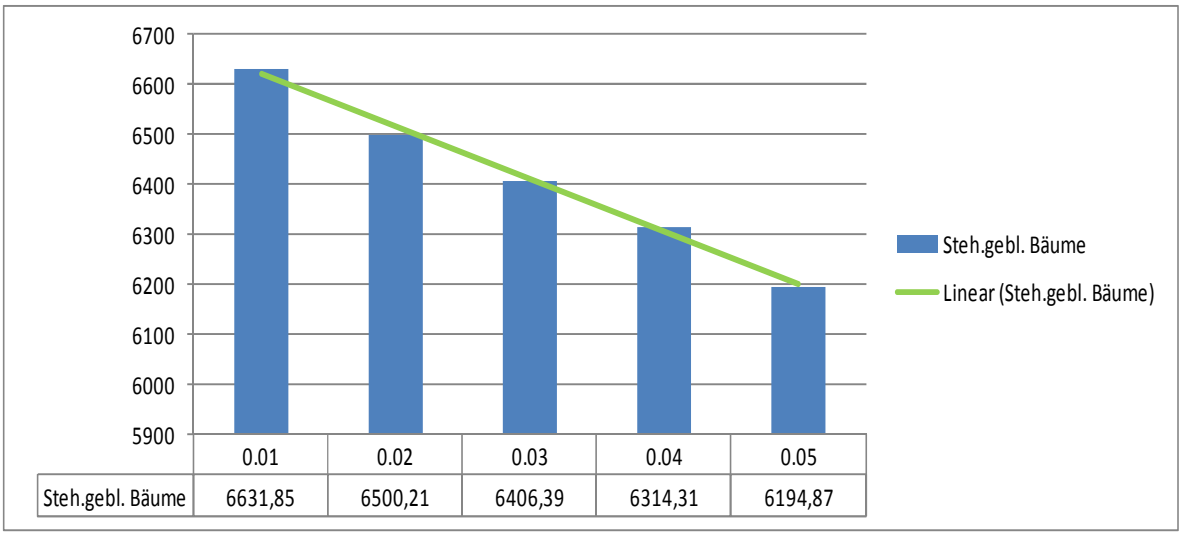
Durch die lineare Regression wird deutlich, dass sich die Ergebnisse linear verhalten. An einem Punkt ändert sich die Skalierung, dementsprechend läuft die Gerade nicht gut durch sie durch. Die Zyklen sind aber trotzdem proportional zu der Wahrscheinlichkeit.

Durchschnittlich stehengebliebene Bäume

In diesem nächsten Versuch werde ich die durchschnittliche Anzahl der stehengebliebenen Bäume untersuchen. Dazu lasse ich die Simulation mit den Standarteinstellungen und verschiedenen Wahrscheinlichkeiten 100-mal laufen. Die Ergebnisse sind in diesen Diagrammen dargestellt.



	0.001	0.002	0.003	0.004	0.005	0.006	0.007	0.008	0.009	0.01
Steh.gebl. Bäume	7442,37	7025,8	6717,93	6761,06	6810,88	6697,66	6818,7	6684,98	6625,52	6631,85



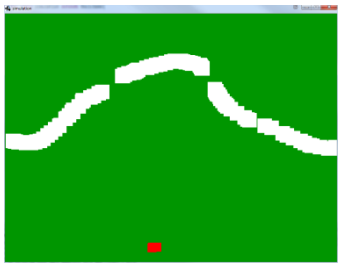
	0.01	0.02	0.03	0.04	0.05
Steh.gebl. Bäume	6631,85	6500,21	6406,39	6314,31	6194,87

Das erste Diagramm von 0,001 – 0.01 verläuft nicht wirklich linear. In Ansätzen ist es aber zu erkennen. Das zweite Diagramm von 0,01 – 0,05 verläuft linear. Es ist also anzunehmen, dass die Anzahl der stehenden Bäume mit steigender Wahrscheinlichkeit linear absinkt.

Schneisen

In den folgenden Versuchen werde ich in das Feld Schneisen mit Öffnungen einbauen und die Ausbreitung des Feuers beobachten.

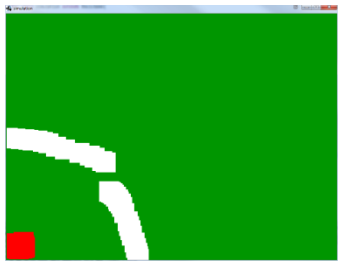
Versuch 1 – Großer Startbereich, Zwei Öffnungen:



Das Feuer breitet sich zuerst in alle Richtungen aus. Dann kommt das Feuer aber an der rechten Öffnung zum Erliegen. Im weiteren Verlauf erlischt das Feuer auch auf der rechten Seite. Auf der linken Seite findet es allerdings die Öffnung und breitet sich weiter nach links aus. Dann erlischt das Feuer total und die rechte obere Seite bleibt komplett unberührt.

➔ Bilder, die den Verlauf illustrieren, finden sich im Anhang „Versuch 1“

Versuch 2 – Kleiner Startbereich, Eine Öffnung:



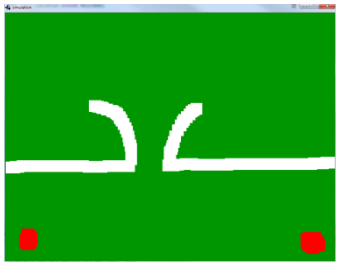
Das Feuer startet in der linken unteren Ecke umgeben durch eine Schneise mit einer Öffnung. Das Feuer schafft es allerdings nicht diese Öffnung zu erreichen. Es erlischt vorher, da zwei Feuer von zwei Seiten gegeneinander laufen.

➔ Bilder, die den Verlauf illustrieren, finden sich im Anhang „Versuch 2“

Gegenfeuer

In den folgenden Versuchen untersuche ich das Verhalten von Feuern die auf begrenztem Raum gegeneinander laufen.

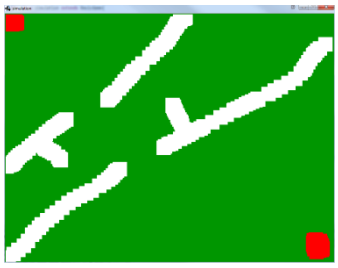
Versuch 3 – Zwei Feuer, Mittlerer Startbereich, Eine Öffnung



Das Feuer startet in der linken und rechten unteren Ecke. Obwohl weiter entfernt, erreicht das rechte Feuer die Öffnung zuerst. Allerdings läuft ein Großteil des rechten Feuers gegen das andere sodass sich beide vernichten und das Feuer nicht durch die Öffnung kommt.

➔ Bilder, die den Verlauf illustrieren, finden sich im Anhang „Versuch 3“.

Versuch 4 – Zwei Feuer, Mittlerer Startbereich, vier Öffnungen



Die beiden Feuer breiten sich zunächst ähnlich aus und bilden einen breiter werdenden Trichter. Wenn das Feuer links oben allerdings die Öffnung trifft, ist das gesamte linke Feuer links von der Öffnung oder in der Öffnung. Unten rechts sieht es ähnlich aus. In dem Moment, in dem das Feuer die Öffnung erreicht, kommt allerdings das Gegenfeuer von oben und eliminiert das von unten kommende Feuer. Dieses füllt jetzt noch den Bereich rechts und links der Schneise. Aber der Bereich der getestet wurde (zwischen den zwei Schneisen) blieb, außer das kleine Stück links, unangetastet.

➔ Bilder, die den Verlauf illustrieren, finden sich im Anhang „Versuch 4“.

Analyse

In diesem Teil werde ich die Ergebnisse aus dem Teil „Versuche“ analysieren und/ oder interpretieren.

Durchschnittliche Anzahl an Zyklen

Die Ergebnisse dieses Versuches sind insoweit wichtig, da sie zeigen, dass das Programm richtig funktioniert. Würde die Zahl der Zyklen nicht linear sinken wenn die Wahrscheinlichkeit steigt, wäre entweder ein Fehler in meinem Code oder ein Fehler in der Zufallsgenerierung von Java. Beides ist aber nicht der Fall.

Durchschnittlich stehengebliebene Bäume

Dieser Versuch beweist ebenfalls, dass das Programm richtig funktioniert. Zudem beweist sie das Vorhandensein von Ordnung in diesem Chaos. Auch wenn die exakte Anzahl der brennenden Bäume und die Ausbreitungsrichtung zufällig generiert werden, lässt sich so vorhersagen, bei welcher Wahrscheinlichkeit eine wie große Anzahl Bäume stehen bleibt. Dies gilt allerdings nur auf einem freien Feld, wie die Versuche 1 – 4 zeigen. Ist das Feld nicht frei und zufällig generiert, ist die Zahl der stehenbleibenden Bäume von anderen Faktoren (wie der Wahrscheinlichkeit, dass sich die Feuerwände treffen) abhängig. Dieser Versuch beweist außerdem, dass Systeme, die nach einer festen Ordnung richten, auch absolut vorhersagbar sind.

Schneisen

Versuch 1 – Großer Startbereich, Zwei Öffnungen:

Hier zeigt sich, dass Schneisen, auch wenn sie Löcher haben, das Feuer effektiv aufhalten können. Denn das Feuer schafft es nur durch eine der zwei Öffnungen und breitet sich dahinter auch nur noch nach links aus. Dass das Feuer überhaupt einen Weg durch die Schneise gefunden hat, liegt in erster Linie an dem großen Startbereich. Dies wird in Versuch 2 deutlicher.

Versuch 2 – Kleiner Startbereich, Eine Öffnung:

Hier zeigt sich, dass das Feuer keine Chance hat, sich auf kleinem Bereich auszubreiten. Da es sich nach links und rechts ausbreitet und mittig aufeinander zuläuft, löscht es sich selbst. Dieser Effekt ist auch bei Versuch 1 zu beobachten gewesen. An der rechten Öffnung liefen zwei Feuerwände gegeneinander und haben sich gegenseitig gelöscht.

Gegenfeuer

Versuch 3 – Zwei Feuer, Mittlerer Startbereich, Eine Öffnung

Obwohl die Effekte eines Gegenfeuers schon in Versuch 1 und 2 aufgetreten sind, wird hier nochmal spezifisch darauf getestet. Es ist erkennbar, dass sich die beiden Feuer vernichten bevor sie die große Öffnung treffen.

Versuch 4 – Zwei Feuer, Mittlerer Startbereich, vier Öffnungen

Erstaunlicher ist dieser Versuch. So ist unbestreitbar, dass das Feuer von unten rechts garantiert auf den rechten Bereich des Grabens übergegriffen hätte. Da das Gegenfeuer aber schneller war, ist der Bereich unangetastet geblieben. Dies zeigt deutlich, dass sowohl Schneisen als auch Gegenfeuer effektive Methoden zum Aufhalten des Feuers sind.

Interpretation & Schluss

Insgesamt lassen sich zwei Erkenntnisse aus diesen Versuchen ziehen. Zum einen, dass das System auf freiem Feld absolut berechenbar und vorhersehbar ist. So kann man genau bestimmen, wie viele Bäume wahrscheinlich beim nächsten Durchlauf stehen bleiben. Die zweite Erkenntnis ist, dass sowohl Schneisen als auch Gegenfeuer – wenn richtig eingesetzt – sehr effektiv zur Manipulation des Feuers eingesetzt werden können. Diese Ergebnisse lassen sich aber nicht ohne Probleme ins reale Leben übertragen. Die Natur läuft Stephen Wolfram nach zwar nach ähnlichen Systemen, aber der hier vorgestellte Zelluläre Automat ist bei weitem nicht komplex genug um Vorgänge der Natur realistisch abzubilden. Als Facharbeitsthema war er allerdings gut geeignet. Die Programmierung war überaus interessant. Das Schreiben und die Versuche selbst waren nicht immer einfach zu realisieren, aber letzten endlich hat sich der Aufwand gelohnt. Zu der Programmierung füge ich noch hinzu, dass ich möglicherweise nach Abgabe der Arbeit noch etwas daran arbeite. Deswegen hier noch der Ausblick auf weitere Entwicklungen und die Information, dass die Version 0.3 diejenige war mit der ich hier gearbeitet habe. Für einige Versuche habe ich allerdings etwas Code ergänzt oder entfernt (z.B. um weniger/mehr Ausgaben zu haben). Zu dem Programm könnte man noch ein Ausgabefenster hinzufügen oder die Ausgaben einstellbar machen, ebenso variable Wahrscheinlichkeiten einführen (man stellt ein 0.001 bis 0.005 und das Programm läuft dann für jede Wahrscheinlichkeit eine bestimmte Anzahl von Zyklen lang).

Quellen, Links & Hilfsmittel

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/ (23.04.12)

Hilfsmittel

Greenshot – Ein Open-Source Screenshot Programm.
(getgreenshot.org)

lwjgl – Eine Open-Source Bibliothek für 3D-Spieleentwicklung mit OpenGL in Java
(www.lwjgl.org)

slick – Eine Bibliothek um die Realisierung von 2D Spielen in lwjgl zu vereinfachen
(slick.cokeandcode.com)

eclipse – Ein Open-Source IDE für Java.
(www.eclipse.org)

Erklärung

Ich erkläre, dass ich die Facharbeit ohne fremde Hilfe angefertigt und nur die im Literatur- und Quellenverzeichnis angeführten Quellen und Hilfsmittel benutzt habe.

_____	_____
Ort, Datum	Unterschrift