Friedrich-Schiller-Universität Jena
Physikalisch-Astronomische Fakultät
Max-Wien-Platz 1
07743 Jena

Belegarbeit Computational Physics I

Dozent: Prof. Pertsch
WS 2007/08

Thema: Implementierung eines einfachen zellulären Automaten mit Festlegung der Entwicklungsregel

Zusatz: Implementierung eines einfachen zellulären Automaten auf Modulo-Operator-Basis
Implementierung eines zweidimensionalen zellulären Automaten mit verschiedenen Nachbarschaftsabfragen

Eine Arbeit von:

Name:	Chemnitz, Mario
Adresse:	Friedenstraße 8, 07743 Jena
E-Mail:	mario.chemnitz@uni-jena.de
Studiengang:	Physik, Diplom
Matrikelnr.:	91255
Fachsemester:	3

Abgabedatum: 29.01.2007

Inhaltsverzeichnis

1 Aufgabe

Schreiben Sie ein Programm, welches die Entwicklung der Generation bei frei vom Benutzer wählbaren Fortpflanzungsregeln durchführt! Die Feldgrösse sollte dabei an die Anzahl der Generationen angepasst sein. Können Sie Beispiele für chaotisches oder fraktales Verhalten finden?

2 Einführung

2.1 Arbeitsweise eines Zellulären Automaten (ZA)

Zur Darstellung mathematisch dynamischer Systeme kann ein zellulärer bzw. zellularer Automat implementiert werden. Diese Art Automaten berechnen aus einer Startpopulation ($t = 0$) beliebig viele Folgegenerationen ($t = n$, n frei wählbar). Sie definieren sich durch ihre Dimensionalität, ihre Regel zur Entwicklung der Folgegeneration (Überführungsfunktion) und den Radius der Nachbarschaftsabfrage um eine Zelle.

2.1.1 Eindimensional

Einem eindimensionalen zellulären Automaten (1D ZA) werden eine einzelne Zeile Zellen definierter Länge (bspw. Datenarray oder Pixelzeile eines Bildes), die gewünschte Generationsanzahl und eine Entwicklungsregel übergeben. Der Zustand einer in ihr enthaltenen Zelle und deren links- und rechtsseitigen Nachbarn werden überprüft und nach der übergebenen Regel entschieden, ob die Folgezelle überlebt. Nachdem dies für alle Zellen der übergebenen Zeile gemacht und daraus die 1. Folgegeneration ermittelt wurde, wird die somit entstandene neue Zeile, also die Folgegeneration, genau nach demselben Schema bearbeiten und die nächste Generation berechnet. Dies geschiet solange bis die gewünschte Generationsanzahl erreicht ist.

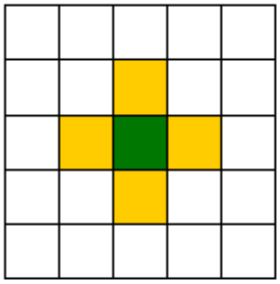
Bei dem Zustand einer Zelle wird im einfachsten Fall zwischen lebendig (bspw. schwarzes Feld/Pixel) und tot (bspw. weißes Feld/Pixel) unterschieden. Diese Zustandsdefinition kann natürlich in beliebiger Weise erweitert werden. So können beliebig viele Zwischenstufen zwischen tot und lebendig mit Hilfe von Graustufen oder Farben der Pixel definiert werden.

Die eigentliche Vielfalt der zellulären Automaten begründet sich mit der Vielzahl an anwendbaren Entwicklungsregeln. Der wohl einfachste Fall enthält eine Durchnummerierung der verschiedenen Zellzustände (bspw. 0 für eine „tote“ und 1 für eine „lebende“ Zelle), eine Aufsummierung über das oben benannte Zelltripel (also die Zelle an der Position x und deren Nachbarn $x-1$ und $x+1$) und eine Abfrage, basierend auf dem Modulo-Operator (für weitere Informationen s.Abschn.3.2). Ein statischeres System basiert auf einer festen Entwicklungsregel, da genau festgelegt ist, bei welcher Tripelkonstellation eine Folgezelle geboren wird oder nicht (s.Abschn.3.3).

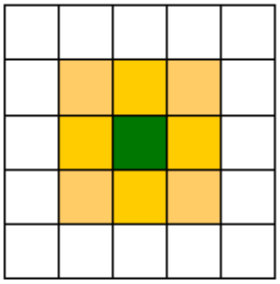
2.1.2 Zweidimensional

Bei einem zweidimensionalen zellulären Automaten (2D ZA) sind die Zellen nicht über eine Zeile, sondern über eine Fläche verteilt, die zweckmäßigerweise oftmals auch als „Lebensraum“

betitelt wird. Auch hier wird nacheinander die Nachbarschaft einer jeden Zelle erfasst und über bestimmte Entwicklungsregeln entschieden, welchen Zustand die jeweilige Zelle in der Folgegeneration besitzen soll. Der einfachste Fall wäre wieder ein Zweizustandssystem. 2D ZA's unterscheiden sich u.a. in der jeweils genutzen Nachbarschaft. Die beiden üblichsten Nachbarschaften sind die Moore- und die Neumann-Nachbarschaft:



Neumann- Nachbarschaft



Moore- Nachbarschaft

© Quelle (I)

Anders als beim 1D ZA erfolgen die Nachbarschaftsabfragen beim 2D ZA alle „gleichzeitig", da eine zeilenmäßige Bearbeitung logischerweise die Nachbarschaft der Folgezellen beeinflussen würde. Von daher sieht man bei einer zweidimensionalen Darstellung immer nur die aktuelle fertige Zellengeneration und nicht eine sich fließend verändernde Population. Die vorhergenden Generationen lassen sich ohne permanente Speicherung oder dreidimensionaler Darstellung nicht mehr zurückverfolgen.

2.1.3 Automatenklassen

Bei zellulären Automaten unterscheidet man zwischen verschiedenen Klassen. Manchmal kann ein und derselbe Algorithmus komplett andere Entwicklungseigenschaften hervorzeigen, nur aufgrund variierter Übergabeparamter. Daher rührt die Dynamik dieser Automatensysteme. Hier sind übersichtlich die vier ZA-Klassen dargestellt. Beispiele werden bei den Auswertungen der Ergebnisbilder gezeigt.

Klasse	Beschreibung
1	• Nahezu jede Startkonfiguration möglich • Enden nach kurzer Zeit in einem statischen Zustand • Informationen über den Anfangszustand gehen völlig verloren, da Zellzustände am Ende alle gleich
2	• Entwicklung nicht aus jeder Startkonfiguration möglich • Enden auch in einem statischen Zustand • Nicht alle Informationen über die Anfangskonfig. gehen verloren, da sich Blöcke gleicher Farbe ausbilden
3	• Unabhängig von der Startkonfiguration • Am häufigsten vertretene Klasse • Kein stabiler Endzustand $\Longrightarrow$ „chaotisches“ bis fraktales Verhalten • Kaum rückverfolgbar
4	• Verhalten stark abhängig von der Startkonfiguration • Entstehung von Mustern ähnlich der Klasse 1 und 2, aber auch von sog. „Gleitern“ oder „Oszillatoren“ • Kein stabiler, aber meist deterministischer Endzustand • Anfangszustand kann teilweise zurückverfolgt werden

2.2 Der C++-Builder

Für die Umsetzung der gestellten Aufgabe entschied ich mich für die Nutzung des Borland C++ Builders. Dies ist ein C++ Compiler zur Erstellung von Windows GUI Anwendungen. Ein C++ Builder Projekt besteht im Allgemeinen aus einem Formular (Blanco-Fenster), auf das die bekannten Komponenten der Windows-Anwendungen (bspw. Buttons, Images, Checkboxes etc.)) angeordnet werden. Den jeweiligen Komponenten können dann spezifische Ereignisse zugeordnet werden, bspw. ein OnClick Ereignis für einen Button, das genau dann ausgelöst wird, wenn dieser Button angeklickt wurde. Diese Ereignisprozeduren bilden die Grundlage des Programms. Der große Vorteil des Builders ist offensichtlich. Eine Windowsanwendung kann per Drag&Drop schnell zusammengestellt und zu einer fertigen .exe-Anwendung compiliert werden.

Als kurze Einführung gebe ich hier eine kleine Übersicht über die von mir verwendeten GUI Komponenten und den dazugehörigen Ereignissen:

GUI-Komponente	Beschreibung	genutzte Ereignisse	Ereignisauslöser
TButton	Konventioneller Knopf	OnClick	Anklicken des Knopfes
TEdit	Einzeiliges Eingabefeld	OnClick OnChange OnKeyDown	Anklicken des Feldes Eingabe im Feld Reaktion auf Knopfdruck der Tastatur
TCheckbox	Kontrollkästchen	OnClick	Anklicken des Kästchens (Häckchen setzen bzw. entfernen)
TRadioGroup	Gruppe aus Auswahlkästchen	OnClick	Anklicken eines Kästchens (Punkt setzen); Unterschied zur Checkbox: Es kann immer nur eine Radiobox aktiviert sein (kein multiple choice)
TImage	Zeichnungsfläche	OnClick	Anklicken der Zeichnungsfläche
TTimer	Getimte Ereignisaktivierung (Nach Ablauf eines systembedingten Zeitintervalls)	OnTimer	Aktivierung des Timers

Bemerkung: Der Zusatz `__fastcall` in den Funktionsdeklarationen ist eine Besonderheit des C++ Builders, wodurch Funktionsaufrufe anders behandelt werden.

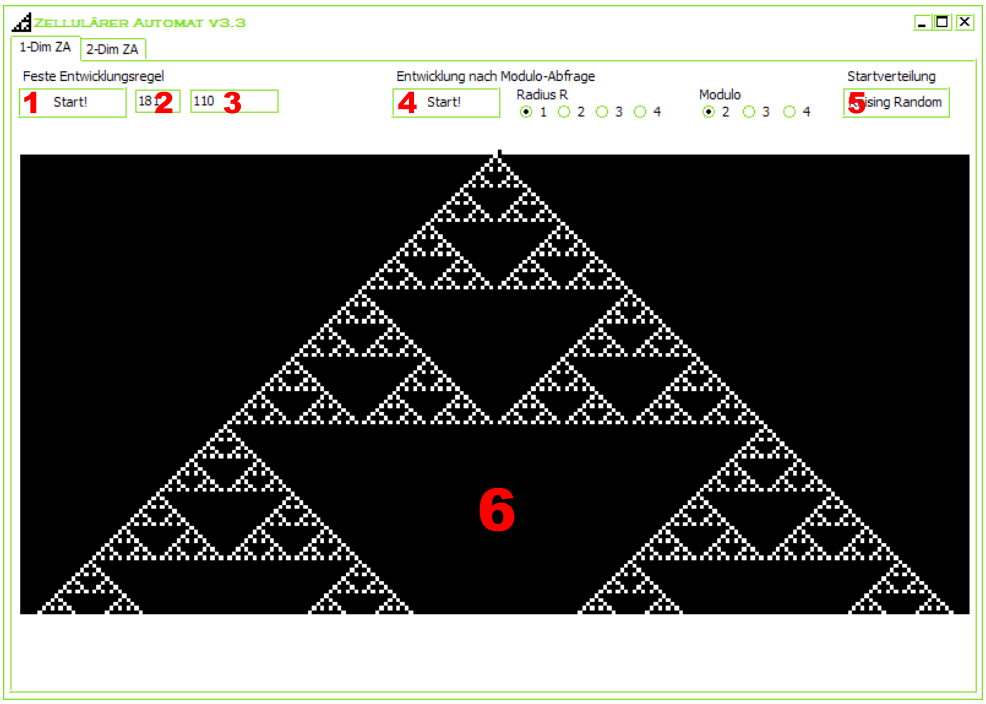
2.3 Entwicklung der Software

Diese Arbeit beschreibt eine eigens entwickelte Software zur Darstellung zellulärer Automaten. Zur Abschätzung des Zeit- und Arbeitsaufwandes, wurden die ersten beiden Versionen der Software in Objevtive Pascal (Delphi) geschrieben. Sie behandelten den 1D ZA auf Modulo-Basis und mit fester Regelübergabe (V 1.0) und den 2D ZA (V 2.0). Nachdem das Grundgerüst der Hauptroutinen stand, begann ich mit der übersetzung des Codes in C++, was sich wegen fehlender Vorkenntnisse als zeitaufwendiger herausstellte, als die eigentliche Implementation der einzelnen Algorithmen. Version 3.0 beinhaltete schlussendlich die übersetzten Algorithmen des 1D und 2D ZA. Leider stellte sich heraus, dass der Hauptalgorithmus nicht exakt der Aufgabenstellung entsprach und so schrieb ich die Routine der Zielstellung entsprechend um (V 3.1). Anschlieend wurden noch an geeigneten Stellen Randbedingungen eingebaut (V 3.2) und die Bearbeitung und grafische Darstellung des 2D ZA optimiert (V 3.3). Diese Version stellt nun eine benutzerfreundliche und eigenständige Applikation zur Veranschaulichung zellulärer Automaten dar.

3 Implementierung eines eindimensionalen ZA

3.1 Die Nutzeroberfläche

Zur Vereinfachung des Einstiegs in die Programmsteuerung und des Verständnisses des Programms wird an dieser Stelle eine kurze Übersicht über die Bedienoberfläche gegeben. In den folgenden Abschnitten wird an geeigneten Stellen auf die jeweiligen Bedienelemente verwiesen. Dies geschieht mittels Klammerausdrücken der Form (# Nummer des Bedienelements).



Beschreibung:

(1) - Startknopf des ZA mit fester Regelübergabe

(2) - Eingabefeld für die Regel

(3) - Eingabefeld der gewünschten Generationsanzahl, die berechnet werden soll

(4) - Startknopf des ZA basierend auf der Modulo-Abfrage

(5) - Knopf für die Generierung einer zufälligen Pixelzeile, wobei die Pixelanzahl in x-Richtung zunimmt (benötigt für den ZA auf Modulo-Ebene)

(6) - Zeichenbereich Image1

3.2 ZA auf Modulo-Ebene

3.2.1 Algorithmus

Zur Wahrung der Übersichtlichkeit des Programmcodes wurde der grobe Algorithmus des ZA im Folgenden als Struktogramm dargestellt. Mit der von mir gewählten Bezeichnung „Modulo-Ebene“ ist nicht weiter gemeint, als dass die möglichen Zustände einer Zelle durchnummeriert sind (hier weiß/Null für „tot“ und schwarz/Eins für „lebendig“) und aufsummiert werden, wenn die Nachbarschaft einer Zelle abgetastet wird. Die sich aus den Zuständen der Nachbarschaftszellen ergebende Summe wird durch einen vom Nutzer wählbaren Wert (im Struktogramm Zwei) mittels Modulo-Operator geteilt, wobei das Ergebnis eine ganze Zahl plus einem Rest ist.

Die Abfrage dieses Restwertes entscheidet den Zustand der Zelle, deren Nachbarschaft erfasst wurde, in der Folgegeneration.

Als Startsituation wird eine zufällig generierte Bildzeile mit schwarzen (bzw. lebenden) und weißen (bzw. toten) Pixeln (die eigentlichen Zellen) gegeben. Der Algorithmus durchläuft diese Zeile und generiert aus ihr nach der oben beschriebenen Entwicklungsregel die Folgegenerationen (bzw. die folgenden Bildzeilen).

1D ZA auf Modulo-Ebene

Globale Variablen:

Summe	{Integer-Variable: Zustandszähler}
R	{Integer-Input: Nachbarschaftsradius}
Zelle[Pos][Gen]	{Array der Breite Pos (für Position) und Höhe Gen (für Genereation) mit Zustandseinträgen (schwarz oder weiß für lebendig oder tot)}

Laufvariablen:

x	{Positionszähler}
zeit	{Generationszähler}
i	{Variable zur Nachbarschaftsabfrage}

Eingabe: $R \leftarrow$ Gewünschter Abfrageradius

Zähle $zeit$ von 0 bis $Gen - 1$, Schrittweite 1

Zähle x von R bis $Pos - R - 1$, Schrittweite 1

$Summe \leftarrow 0$

Zähle i von $-R$ bis R, Schrittweite 1

$Zelle$[x+i][zeit] = schwarz ?

WAHR: $Summe \leftarrow Summe + 1$ | FALSCH: $\varnothing$

$Summe$ MOD 2 = 0 ?

WAHR: $Zelle$[x][zeit+1] = schwarz | FALSCH: $Zelle$[x][zeit+1] = weiß

3.2.2 Implementierung

Nachdem der Algorithmus mittels Struktogramm veranschaulicht wurde, wird hier die eigene Implementierung in C++ erläutert. Es ist möglich, dass diese Umsetzung nicht den Musteralgorithmen aus den Lehrbüchern entspricht, falls überhaupt eine existiert.

Im Anschluss zu sehen, ist der Quellcode der Hauptroutine, der in einer Timer-Prozedur eines Timer-Elements des C++Builder's eingebettet ist. Diese wird durch das Drücken des entsprechenden Startknopfes (#4) aktiviert und nach einem bestimmten Zeitintervall erneut aufgerufen. Das Zeitintervall zählt dabei erst nach Abschluss der Routine runter, sodass es niemals zu einer Überschneidung zweier Abläufe der Hauptroutine kommen kann. Der Timer ersetzt somit die äußerste Schleife im Struktogramm. Durch erneuten Knopfdruck (#4) oder nach dem Erreichen des unteren Bildrandes von Image1 (#6), wird der Timer gestoppt. Für die weitere Erklärung des Programmablaufes sorgen die Kommentare im folgenden Quelltext:

```
void __fastcall TForm1::Timer2Timer(TObject *Sender)
{
   if (zeit < Image1->Height)  //Nutzung des unteren Bildrandes
                               //als Abbruchbedingung
   {
     int sum; //Einführung einer Zustandssumme

    //Durchlaufen der Zellzeile der Generation 'zeit'
     for (int y=R; y<(Image1->Width)-R;y++)
     {
       sum = 0;

       //Nachbarschaftsabfrage im wählbaren Radius R um die Zelle y
       for (int n=-R; n<R; n++)
         //Abfrage des Zustandes der Zelle bzw. des Bildpixels
         if (Image1->Canvas->Pixels[y+n][zeit]==clBlack)
           sum++; //Zelle lebt/Pixel schwarz, somit Inkrementierung der Zustandssumme

      //Zuweisung des neuen Zellzustandes
       if ((sum % Modulo == 0) && (sum != 0)) //Modulo-und Sicherheits-Abfrage
         Image1->Canvas->Pixels[y][zeit+1] = clBlack; //Folgezelle lebt
       else
         Image1->Canvas->Pixels[y][zeit+1] = clWhite; //Folgezelle stirbt
     }
   zeit++;  //Inkrementierung des Generationszählers
   }
   else
   {
     Timer2->Enabled = false; //Timer stoppt, wenn Abbruchbed. greift
     Button3->Caption = "Start!";
   }
}
//Wenn Timer weiterhin aktiv, erfolgt ein erneuter Aufruf der Funktion nach
//ca. einer Millisekunde (Timerintervall)
```

3.2.3 Auswertung der Ergebnisbilder

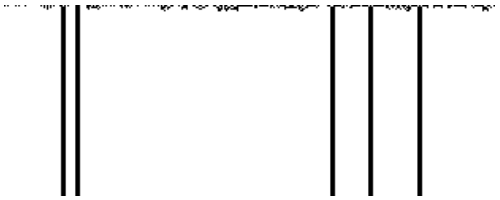
In diesem Abschnitt zeige ich anhand einiger Ergebisbilder, dass sich alle Automatenklassen mit diesem auf dem Modulo-Operator basierendem System erzeugen lassen. Dabei werden ausschließlich der Nachbarschaftsradius R und der Modulo-Wert variiert.

Klasse 1:
Anfängliche Strukturbildung aber ab einer bestimmten Grenzgeneration sind alle Folgezellen tot.

(a) R=4; MOD 4

(b) R=1; MOD 3

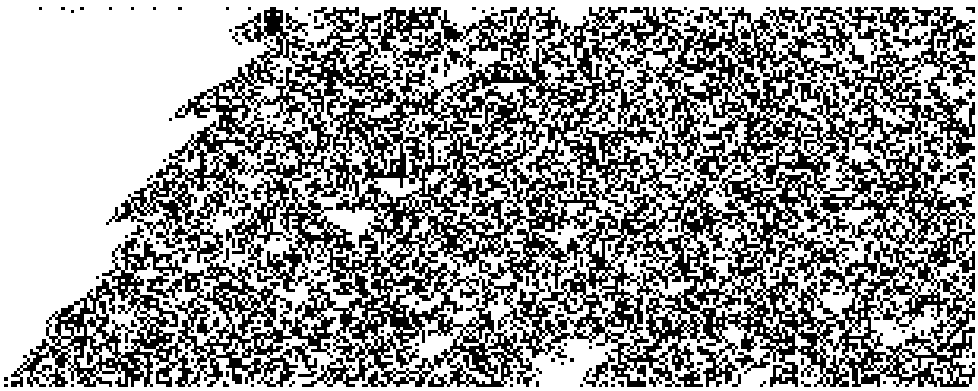
Klasse 2:
Ausbildung einzelner sich stetig fortsetzender Blöcke (bei R=3 seltener als bei R=2).



(c) R=2; MOD 3

(d) R=3; MOD 4

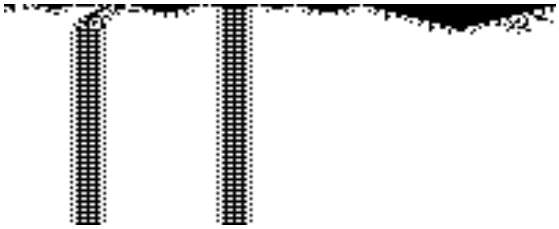
Klasse 3:
Chaotisches Zellwachstum und zusätzliche Ausbreitung in den „toten“ Außenraum.



(e) R=3; MOD 2

Klasse 4:
Oszillatoren (unendliche Fortsetzung)

(f) R=2; MOD 2



(g) R=4; MOD 3

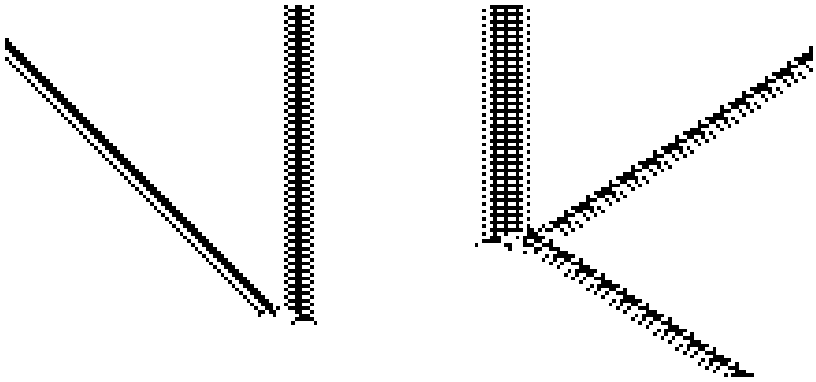
Gleiter (meist endliche Fortsetzung wegen Randgebietsdefinition)

(h) R=2; MOD 2

(i) R=4; MOD 3

Reaktionen:
Vereinzelt beobachtet man interessante Fälle, wie bswp. der Zusammenstoß zweier Gleiter. Bei Gleitern endet dieser meistens in deren Ausschlöschung, anders als bei einigen Fällen des Zusammenstoßes zwischen Gleiter und Oszillator.

(j) Vernichtung zweier Gleiter



(k) Vernichtung eines Gleiters an einem Oszillator

(l) Reflektion eines Gleiters an einem Oszillator

3.3 ZA mit Festlegung der Entwicklungsregel

3.3.1 Algorithmus

Zu Beginn werden vom Benutzer die Regelnummer (#2) und die gewünschte Generationsanzahl (#3) eingegeben. Nach Algorithmusstart durch Knopfdruck (#1) wird ein zur Generationsanzahl passender Entwicklungsraum (dynamisches Array) erstellt und eine lebende Startzelle in das Zentrum der ersten Arrayzeile gelegt. Eine zweidimensionale Schleife läuft nun, von der ersten beginnend, Zeile für Zeile ab. Aus den Nachbarschaftsinformationen einer jeden Zelle in der Zeile wird eine Folgezelle in der Folgezeile nach einer bestimmten Entwicklungsregel erstellt. So entsteht Stück für Stück die Folgegeneration in Form einer neuen Zeile. Zur Erfüllung der Aufgabenstellung muss die Größe des genutzten Arrays entsprechend der gewünschten Generationsanzahl anpassbar sein. Das Array besitzt also vom Eingabe-Parameter abhängige Dimension. Klar ist, dass die Matrix eine Tiefe entsprechend der zu entwickelnden Generationsanzahl haben muss, somit steht die Dimension einer Array-Richtung fest. Nun stellt sich jedoch noch die Frage, inwiefern die Breite des Arrays von der gewünschten Generationsanzahl abhängt. Hierfür nutzen wir den günstigen Umstand der freien Wahl der Anfangsbedingungen.

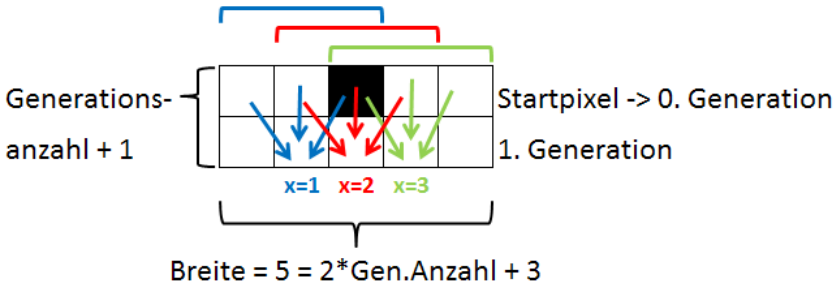


Abbildung 1: Schema zur Erläuterung der dynamischen Arraygrenzen

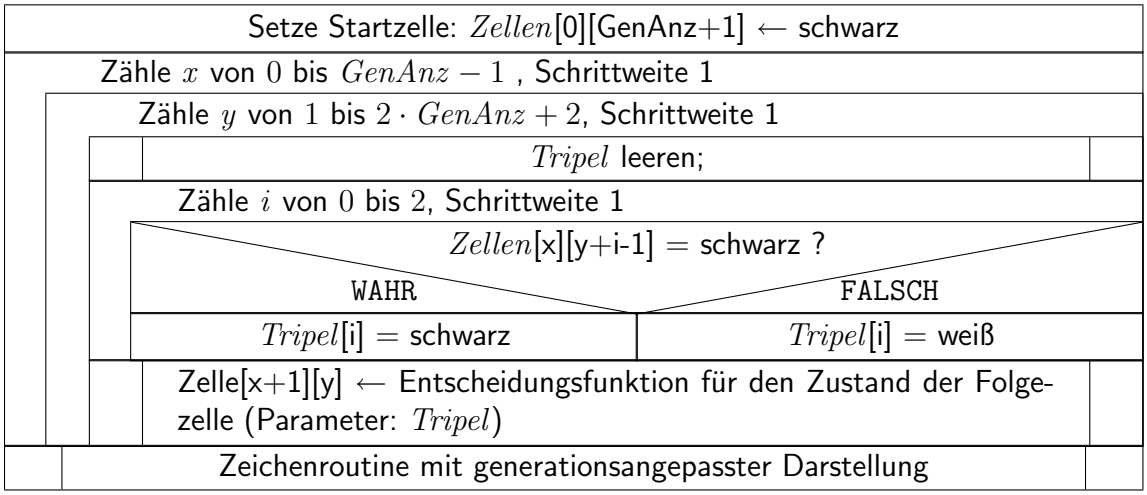
Geht man von einem einzigen Startpixel im Zentrum der ersten Arrayzeile (0. Generation; s.Abb.1) aus, sind für die Entwicklung der ersten Folgegeneration ausschließlich die drei Zelltripel

(Zellen bei $x-1$, x, $x+1$; in Abb.1 farbige Klammern) von Bedeutung, die den Startpixel enthalten. Anhand dieser Tripel kann die Folgegeneration hinreichend bestimmt werden, da alle anderen Zelltripel der Zeile nur leere Zellen enthalten würden. Somit sind deren Folgezellen eindeutig bestimmt und im eigentlichen Sinne für uns uninteressant. Drei aneinander liegende Tripel beanspruchen die Breite von fünf Zellen. Führt man diese Überlegungen fort, kommt man zu dem Schluss, dass mit jeder weiteren Generation zu den schon betrachteten Tripeln nur zwei Tripel hinzukommen und zwar links und rechts der jeweiligen Zeile (1. Generation → drei zu berechnende Folgezellen, drei Tripel aus der 0.Gen. benötigt; 2. Generation → fünf zu berechnende Folgezellen, fünf Tripel aus der 1.Gen. benötigt usw.). Somit kommen mit jeder zusätzlich betrachteten Generation zwei Zellen zur Breite hinzu. Durch Induktion lässt sich also zeigen, dass die effektive angepasste Breite des Arrays genau $2 \cdot Generationenanzahl + 3$ betragen muss.

Für den übersichtlichen Einstieg in die Hauptroutine, sei hier wieder ein Struktogramm gegeben:

1D ZA mit fester Regelübergabe

Globale Variablen:	
Regel	{Integer-Input: Regelnummer}
GenAnz	{Integer-Input: Generationsanzahl}
Zellen[ ][ ]	{Dynamisches Array der Generationsanzahl angepasster Höhe und Breite mit Zustandseinträgen (schwarz oder weiß für lebendig oder tot)}
Tripel[3]	{Array mit Zustandseinträgen entspricht Vektor aus drei Zellen}
Laufvariablen:	
x, y	{Positionszähler}
i	{Variable zur Vektorfüllung/Nachbarschaftsabfrage}
Eingabe: $Regel$ ← Gewünschte Regelnummer	
Eingabe: $GenAnz$ ← Gewünschte Generationsanzahl	
Erstellen des Arrays: $Zellen$[GenAnz][2*GenAnz+3]	
Zähle x von 0 bis $GenAnz - 1$, Schrittweite 1	
Zähle y von 0 bis $2 \cdot GenAnz + 2$, Schrittweite 1	
$Zellen$[x][y] ← weis	



Die „Entscheidungsfunktion für den Zustand der Folgezelle“ weist einer Folgezelle ihren Zustand zu und beinhaltet somit die Entwicklungsregel. Als Übergabeparamter dient hierbei ein Tripel (s.Abb.2), welches die Zustandsinformationen der gerade betrachteten Zelle und ihrer beiden Nachbarn beinhaltet.

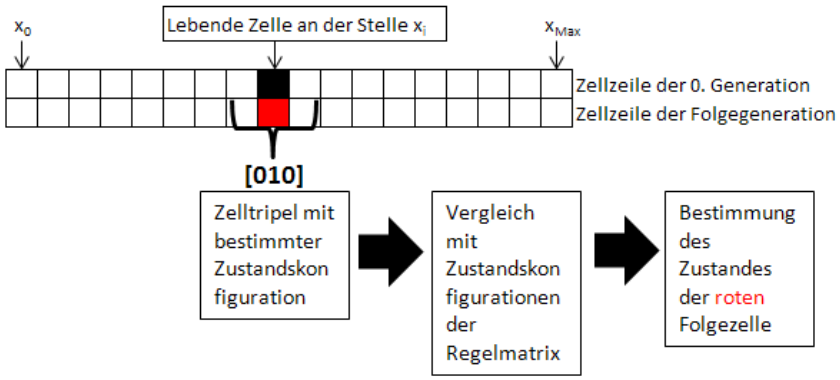


Abbildung 2: Veranschaulichung der Zellabfrage

Die Entwicklungsregel wird aus der eingegebene Regelnummer gewonnen. Diese Nummer wird in eine Binärdarstellung umgewandelt, wobei die Nullen für den Zustand tot und die Einsen für den Zustand lebendig stehen. Da man immer ein Zelltripel betrachtet, gibt es nur acht verschiedene Tripelkonfigurationen. Somit folgt, dass die Binärdarstellung der Regelnummer genau den Umfang eines Bytes besitzt, da zu jeder Tripelkonfiguration ja ein Zustand gehört, der sich aus der Regelnummer ergeben muss. Es gibt in dieser Automatenversion also 256 verschiedene Entwicklungsregeln. Dieses „Regelbyte“ wird in die vierte Zeile einer 4x8 Zustandsmatrix geschoben (s.Abb.3). Die ersten drei Einträge einer Spalte dieser Matrix enthalten mögliche Zustandskonfigurationen einer Zelle mit ihrem links- und rechtsseitigem Nachbarn, bspw. 010 für eine lebende mittlere Zelle und zwei tote Nachbarn. Mit dem vierten Eintrag, der wie beschrieben aus der Regelnummer gewonnen wird, wird einer Konfiguration der Zustand für die

darunter liegende Zelle zugewiesen, bspw. 010 1, d.h. wenn nur die mittlere Zelle eines Zellentripels lebt, lebt auch die Folgezelle. Das folgende Schema fast den genauen Aufbau der Regelmatrix nochmals zusammen:

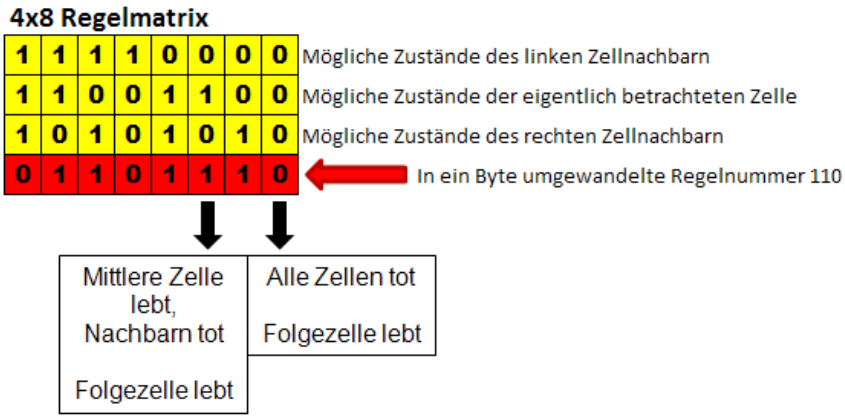


Abbildung 3: Aufbau der Regelmatrix

Interpretieren wir die Einsen und Nullen als schwarze und weiße Pixel, wird aus diesem Schema bspw. klar, dass ungerade Regelnummern, ab der ersten Entwicklungsgeneration (ausgehend von einem einzelnen Startpixel) schwarze Zeilen erzeugen, da unter dem letzten Konfigurationstripel (`[000]`) in diesem Fall eine Eins stehen wird und somit die Folgezellen eines weißen Zeilenbereichs schwarz werden. Bei Regelnummern, die kleiner als 128 sind, steht zusätzlich eine Null unter der ersten Konfiguration (`[111]`). Dadurch entsteht zumindest an den Randbereichen des Entwicklungsraumes ein mit der Generation gehender Wechsel zwischen schwarzen und weißen Zeilen. Gibt man stattdessen Regelnummern ein, die größer sind als 127, steht an dieser Stelle eine Eins, was bedeutet, dass auf schwarze Zellen wieder schwarze Zellen folgen und sich ein schwarzer Hintergrund ausbildet.

Weiterhin sieht man, dass es auf jeden Fall zu vielen Mustern ein Negativ (also ein farbinvertiertes Muster) geben wird, da ein „Regelbyte“ der Form `[00111100]` dieselbe Zellentwicklung mit sich bringen wird wie die Form `[11000011]`, nur mit dem Unterschied der Farbgebung. (Mehr dazu im Abschnitt 3.3.5.)

3.3.2 Implementierung der Hauptroutine

Dargestellt ist der Quellcode der Hauptroutine `void __fastcall TForm1::Button2Click (TObject *Sender)` (grau hinterlegt), die aufgesplitet wurde, um die wichtigen Unterroutinen (grauer Rahmen, weis hinterlegt) an den jeweils wichtigen Stelle zu erläutern. Die Hauptroutine ist diesmal kein Timer-Ereignis, sondern ein Button-Click-Ereignis welches einmalig durchläuft nachdem der dementsprechende Button (#1) angeklickt wurde.

```
{
   if(Edit1->Text != "" && Edit3->Text != "")  //Sicherheitsabfragen der Editfelder
                                               //(dürfen nicht leer sein)
   {
     GroupBox1->Caption = "Deterministischer Verlauf";
     Button2->Enabled = false; //Deaktivierung des Buttons

     regel = StrToInt(Edit1->Text); //Einlesen der Regelnummer
     regelzuweisung(regel); //Aufrufen einer Prozedur zur Erstellung der Regelmatrix
```

Die Unterprozedur `regelzuweisung` nutzt den vom Benutzer eingegeben Parameter `regel` um die beschriebene globale Regelmatrix zu modifizieren. Hierfür nutzt diese Routine die selbst geschriebene Funktion `String __fastcall IntToByte(double I, int laenge)` (Quellcode siehe Anhang A). Diese Funktion wandelt die übergebene Dezimalzahl `I` in ein Byte der Länge `laenge` um und steckt diese binäre Abfolge (Codierung) aus Übergabegründen in einen String. Selbstredend ist darauf zu achten, dass die umzuwandelnde Zahl sich als ein Byte der geforderten Länge ausdrücken lässt. Die Regelmatrix ist als `String regelmatrix[8]` deklariert, da es sich für die Implementierung anbot, mit Strings zu arbeiten.

```
void __fastcall regelzuweisung(int regel)  //Verändert die globale Regelmatrix
                                           //entsprechend der eingegebenen Regel
{
   String Byt;

   for (int m=0; m<8; m++)
     regelmatrix[m] = IntToByte(m,3); //Festlegung der möglichen Zelltripel-
                                      //konfigurationen (m = 0..7 -> [000]...[111])

   Byt = IntToByte(regel,8); //Umwandlung der Regelnummer in die bitweise Codierung
   for (int m=0; m<8; m++)
     regelmatrix[m] += Byt[8-m]; //Übergabe der Codierung
}
```

Nachdem die erste Benutzereingabe verarbeitet ist, folgt nun die Verarbeitung der eingegeben Generationsanzahl. Aus ihr wird ein dementsprechend dimensioniertes Array erstellt, welches den „Lebensraum“ der zu entwickelten Zellgenerationen darstellt. Die Grundlage für diese Array stellt eine Doppel-Zeigervariable `char **gen_sys` dar, der nun die entsprechenden Adressbereiche zugeordnet werden.

Die genutzte Funktion `StrToInt(String I)` gehört der C++ Builder Bibliotek an und sorgt für die Umwandlung eines aus Ziffern bestehenden Strings `I` in die dazugehörige Integer-Zahl. An dieser Stelle sei weiterhin erwähnt, dass jedes Eingabefeld des Programms von mir so modifiziert wurde, dass es nur Zifferneingaben und die Rückstelltaste annimmt. Ferner gibt es eine Eingabebeschränkung und zusätzliche Sicherheitsabfragen im gesamten Code, sodass bestimmte Zahlenbereiche nicht überschritten werden können, bspw. (#3) erlaubt nur Eingaben bis maximal vier Ziffern.

```
//Initialisierung der Generationsmatrix
     if(gen_changed) //Sicherheitsabfrage, dass Eingabe auch gettigt wurde
     {
       gen = StrToInt(Edit3->Text); //Einlesen der Generationsanzahl
       gen_changed = false;
     }
     gen_sys = new char*[gen]; //Erstellung eines Arrays der Breite gen bestehend aus
                               //Zeigerelementen char*
     for (int x=0;x<gen;x++)
     {
       gen_sys[x] = new char[2*gen+3]; //Jedem Zeigerelement dieses Arrays wird nun ein
                                       //weiteres Array der Breite 2*gen+3 zugewiesen
       for (int y=0;y<2*gen+3;y++)
         gen_sys[x][y] = 0; //Arrayfüllung -> Alle Zellen 'tot'
     }
     gen_sys[0][gen+1] = 1; //Setzen einer Ursprungszelle ins Zentrum der ersten Zeile

     BerechneGen(gen_sys,gen);    //Unterprozedur zur Berechnung der Folgegenerationen

     ZeichneGen(Image1,gen_sys,gen);     //Unterprozedur zu Grafischen Darstellung
```

In Anbetracht der Bedeutung und des Umfangs der beiden letzten Unterprozeduren wird an dieser Stelle kein Einschub gemacht. Die beiden Prozeduren werden dafür in den folgenden Abschnitten extra behandelt.

Nach der Darstellung der berechneten Zellmatrix `gen_sys` bleibt nun noch die Speicherfreigabe offen:

```
//Freigabe des Speichers der Generationenmatrix
     for (int x=0;x<gen;x++)
       delete[] gen_sys[x]; //Speicherfreigabe der 'Unterarrays' der Breite 2*gen+3

     delete[] gen_sys;        //Speicherfreigabe des Arrays der Breite gen
     gen_sys = 0;             //Zeiger zeigt nun ins nil
     Button2->Enabled = true; //Reaktivierung des Knopfes #1
   }
   else //Falls Sicherheitsabfrage zu Beginn greift...
   {
     GroupBox1->Caption = "Bitte Anfangswerte setzen!";
     Button2->Enabled = false;
   }
}
```

3.3.3 Berechnung der Folgegeneration

Dieser Abschnitt befasst sich mit der wichtigen Unterprozedur `void __fastcall BerechneGen(char** &gen_sys, int gen)`, welche für die Berechnung der Folgegeneration sorgt. Auf die Arbeitsweise dieser Routine wurde schon an früherer Stelle genauer eingegangen. An dieser Stelle von Interesse sind nun die Besonderheiten der praktischen Umsetzung.

Die Generationsmatrix gen_sys wird mittels call-by-reference Paramterübergabe bearbeitet. Dabei wird der Funktion statt der gesamten Matrix nur die Adresse der Matrix übergeben. Übergibt man eine Referenz (Typ &; nur in C++) statt einem Zeiger (Typ *) kann die Prozedur den Übergabeparameter verarbeiten, als wenn die tatsächliche Matrix übergeben worden wäre. Die Sprache C zwingt den Programmierer förmlich diesen Weg zu gehen, da Funktionen keine Matrizen zurückgeben können (mit Ausnahme von Strings). Jedoch spart diese Rechenmethode Zeit und Speicherplatz. Ferner wird die gewünschte Generationsanzahl gen übergeben.

Das Zelltripel, welches die Zustandskonfigurationen dreier Zellen beinhaltet, wurde hier als String deklariert, welche im eigentlichen Sinne Arrays vom Typ char sind. Dies vereinfacht den Vergleich mit den Einträgen der Regelmatrix. Beim Füllen des Tripels ist jedoch drauf zu achten, dass die Zellen in der richtigen Reihenfolge $(x-1 \ldots x+1)$ abgefragt werden, da jede neue Zustandsnummer nur an den schon existierenden String angehangen wird.

Damit keine ungewünschten Randeffekte entstehen, spielen Randbedingungen in allen zellulären Automaten eine große Rolle. Während das Problem bei dem ZA auf Modulo-Ebene umgangen wurde, indem die Laufvariable x die Ränder nie erreichte, sollte an dieser Stelle nicht darauf verzichtet werden, die Randgebiete mit einzuarbeiten. Dies erfolgt über Spiegelung des linken an den rechten Rand und umgekehrt. Man stelle sich also ein Blatt Papier vor, dessen Ränder zu einem Zylinder zusammengeklebt werden, sodass ein fortlaufender übergang entsteht. Die Notwendigkeit der Einbeziehung von Randbedingungen wird nochmal anhand eines Beispiels im Abschnitt 3.3.5 deutlich.

```
{
   for(int G = 0; G<gen; G++)  //Schrittweise Generationsentwicklung
     for(int x = 0; x < 2*gen+3; x++) //Zeilendurchlauf
     {
       tripel = "";
       if(x==0) //Randbedingung für den linken Rand
       {
         //Zur Einhaltung der richtigen Abtastreihenfolge zuerst rechter Rand (x-1)...
         if (gen_sys[G][2*gen+2]==1)
           tripel += "1";
         else
           tripel += "0";

         //...dann linker Rand (x, x+1)
         for(int i = 0; i<2; i++)
           if (gen_sys[G][i]==1)
             tripel += "1";
           else
             tripel += "0";
       }
```

```
      else if(x==2*gen+2) //Randbedingung für den rechten Rand
      {
        //Abtastreihenfolge -> wieder zuerst rechter Rand (x-1, x)...
        for(int i = 0; i<2; i++)
          if (gen_sys[G][x-1+i]==1)
            tripel += "1";
          else
            tripel += "0";

        //...dann linker Rand (x+1)
          if (gen_sys[G][0]==1)
            tripel += "1";
          else
            tripel += "0";
      }
      else //Berechnung im Zeileninneren (0<x<2*gen+2)
      {
        for(int i = 0; i<3; i++)    //Beachte Reihenfolge: x-1 -> x -> x+1
          if (gen_sys[G][x+i-1]==1) //Zelle lebt?
            tripel += "1";          //Ja: Hinzufügen einer Eins
          else
            tripel += "0";          //Nein: Hinzufügen einer Null
      }
      /* tripel enthält nun Kombinationen aus dem Bereich "000"..."111"!
      Zur Erinnerung: Die ersten drei Einträge der Regelmatrix-Strings enthalten
      die acht möglichen Tripelkonfiguration, der vierte den Zustand der Zelle
      der Folgegeneration. */

      for (int n=0; n<8; n++) //Abrasterung der möglichen acht Konfigurationen
        if (tripel == regelmatrix[n].SubString(0,3)) //String-Vergleich
          if (StrToInt(regelmatrix[n][4]) == 1) //Folgezustand aus Regel
            gen_sys[G+1][x] = 1;    //Zelle x der Folgegeneration G+1 lebt!
          else
            gen_sys[G+1][x] = 0;    //...ist tot!
    }
  }
```

3.3.4 Implementierung der Grafische Darstellung

Abschließend betrachten wir nun die Unterprozedur `void __fastcall ZeichneGen(TImage *Image, char** &gen_sys, int gen)` zur zweidimensionalen Darstellung. Der Prozedur werden zwei call-by-reference Parameter übergeben. `*Image` zeigt auf die Zeichenfläche, auf der gezeichnet werden soll. `&gen_sys` ist wiederum der Zeiger auf die nun schon gefüllte Zellmatrix, die dargestellt werden soll. Desweiteren wird die gewünschte Generationsanzahl `gen` übergeben.

Da die Größe des Zellarrays `gen_sys` von der Generationsanzahl `gen` abhängt, die Image-Größe jedoch statisch ist, muss ein geeignetes Zoomverhältnis `V` gefunden werden. Dieses sollte von der Arraygröße und somit von der Generationsanzahl abhängen. Schließlich müssen drei

Entwicklungsgenerationen auf dieselbe Leinwand passen wie 1500. Beim Zeichnen färben wir jeden Pixel separat. Wir durchlaufen also in x-Richtung Werte von 0 bis zur um Eins reduzierten Bildbreite `Image->Width-1`. Wir wissen, dass wenn der Maximalwert in x-Richtung erreicht ist, der gestreckte/gestauchte x-Parameter der Zellmatrix den Maximalwert $2 \cdot gen + 2$ haben sollte. Aus dieser überlegung lässt sich leicht, der gesuchte Zoomfaktor bestimmen (s.u.).

Hinzu kommt ein Streckungsfaktor `VG` (< 1), damit das gezeichnete Bild nicht in y-Richtung gezerrt wird, was aufgrund unterschiedlicher Höhe-Breite-Verhältnisse zwischen Zeichnungsfläche und Zellarray passieren würde. Dieser Faktor entspricht genau dem Höhe-Breite-Verhältnis des Zellarrays. Desweiteren wird eine weitere Unterprozedur `void __fastcall uebermalen(TImage *Image)` genutzt, um das evtl. schon genutze Image wieder zu leeren (Quellcode s.Anhang B).

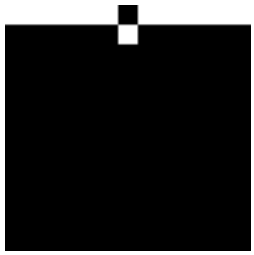
```
{
   float V = (float)(Image->Width-1)/(2*gen+2);  //Definition des Zoomfaktors
   float VG = (float)gen/(2*gen+3);              //Definition des Streckungsfaktors
   float X;           //Einführung neuer Matrixparameter
   float Y;
   uebermalen(Image); //Leeren der Leinwand

   for(int x = 0; x < Image->Width; x++) //Abrasterung der Leinwandpixel bis zur Bild-
     for(int y = 0; y < VG*Image->Width; y++) //breite und in angepasster y-Richtung
     {
       X = x/V ;  //Neue Parameter sind abhängig von den Laufvariablen der Leinwand
       Y = y/V ;

       if(gen_sys[(int)Y][(int)X] == 1)         //Zelle X der Generation Y lebt?
         Image->Canvas->Pixels[x][y] = clBlack; //Ja: Pixel schwarz.
       else
         Image->Canvas->Pixels[x][y] = clWhite; //Ja: Pixel weiß.
     }
}
```

3.3.5 Auswertung der Ergebnisbilder

Sogar bei dem 1D ZA mit fester Regelübergabe, lassen sich für jede Automatenklasse Beispiele finden. Da ich aus Zeitgründen, nur die Versuchsreihe für ein einzelnes Startpixel durchführen konnte, sehen wir im Folgenden leider nur die simplen Grundmuster und keine Interaktionen zwischen verschiedenen Ausbreitungen, ausgehend von verschiedenen Startpunkten. Einige vorhergehende Experimente mit älteren Programmversionen, zeigten jedoch, dass gerade bei Klasse 3 Automaten die Hinzunahme eines weiteren Punktes kaum einen großen Unterschied im Ergebnisbild bringt. Dies entspricht auch den Erwartungen, da die Entwicklung von Klasse 3 Automaten unabhängig von der Startkonfiguration sein muss.

Klasse 1:
Genau wie in Abschnitt 3.2.3 sehen die Bilder dieser Klasse relativ unspektakulär aus, aber auch hier erkennt man die diese Klasse definierende Statik des Endzustandes.

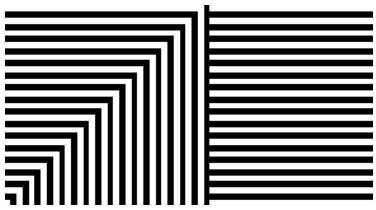
(a) Regel 249



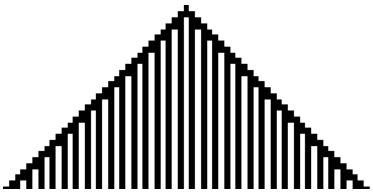
(b) Regel 251

Klasse 2:
Im eigentlichen Sinne ist nur die erste Abbildung ein zellulärer Automat der Klasse 2. Die beiden darauffolgenden Bilder sind Mischungen aus Klasse 2 und Klasse 3 Automaten, da sie mit einer räumlichen Musterausbreitung einhergehen. Die linke Seite der Regel 93 und das Zentrum der Regel 94 bilden einfarbigen Blöcke aus, wodurch sich die 2. Klasse definiert.

(c) Regel 100



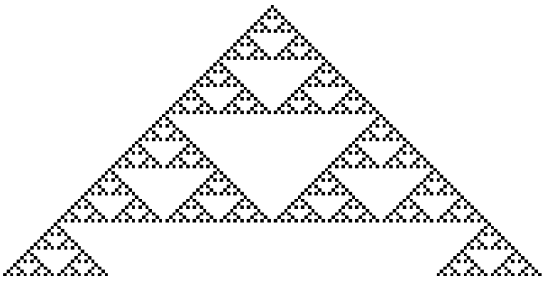
(d) Regel 93



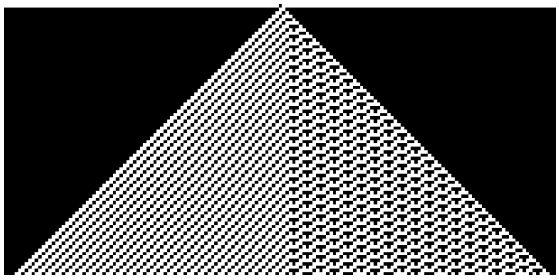
(e) Regel 94

Klasse 3:
Diese Klasse zeigt die Mustervielfalt des Automaten. Aufgrund der deterministen Entwicklung durch die spezifischen Regeln, kommt es nie zu einer vollständig chaotischen Fortpflanzung. Selbst die komplex-aussehensten Muster haben eine innere, meist fraktale Ordnung.

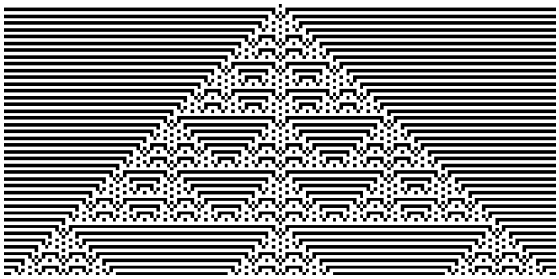
Man beachte besonders das Bild der Regel 18. Dieses hat die typische Form des fraktalen Sierpinski-Dreiecks, welches sich öfter in den Mustern des Automaten wiederspiegelt. Der charakteristische Effekt, dass sich gleiche Muster in unterschiedlichen Entwicklungen wiederspiegeln, ist sehr gut an den Bildern der Regeln 105 und 150 zu erkennen. Obwohl im linken Bild die Dreiecke aus Querstreifen bestehen, ähnelt es zweifellos sehr dem Rechten, in dem die Dreiecke vollständig weis sind. Den Bildern zu den Regeln 30 und 75 lässt sich eine teilweise chaotische Fortpflanzung zuschreiben, denn obwohl sich die jeweils linken Dreieckseiten regelmäig fortsetzen, scheint das Muster zur rechten Seite hin immer chaotischer zu werden. Es treten besondere Streifen- oder Dreiecksmuster zu dieser Seite hin scheinbar ungeordnet auf, was auf eine Art deterministisches Chaos hinweist.



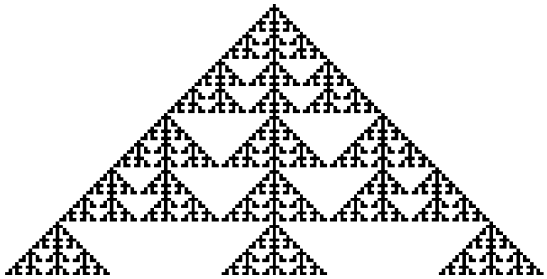
(f) Regel 18



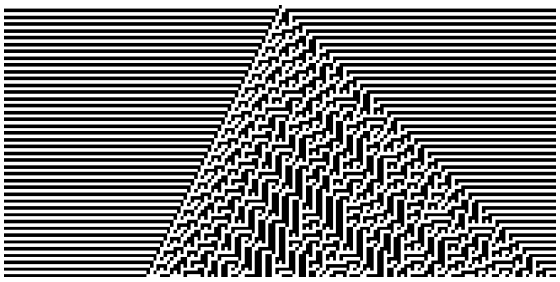
(g) Regel 131



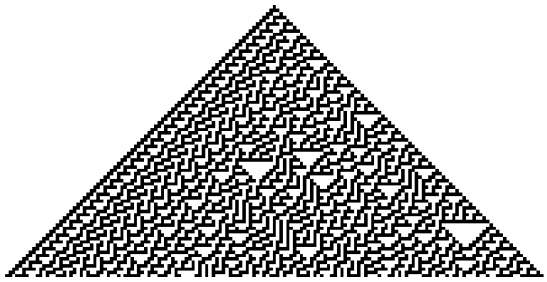
(h) Regel 105



(i) Regel 150



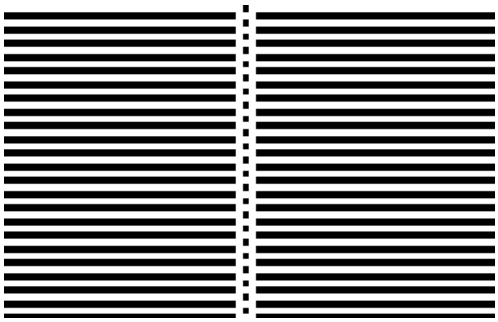
(j) Regel 75



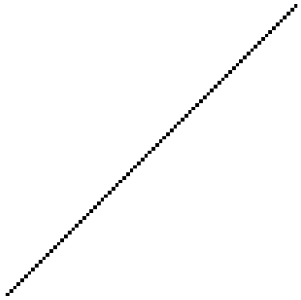
(k) Regel 30

Klasse 4:

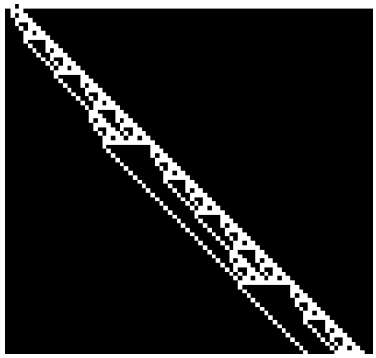
Auch das Vorkommen von Gleitern und Oszillatoren ist möglich, wie man anhand der folgenden Bildbeispiele erkennen kann. Bei dem Bild zur Regel 225 handelt es sich um einen Gleiter, der sich scheinbar chaotisch fortsetzt, aber genauer betrachtet, kommen auch in seiner Spur fraktale Elemente vor. Seine Fortpflanzung ist tatsächlich determiniert.



(l) Regel 1 - Oszillator



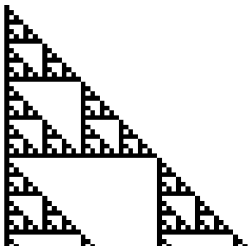
(m) Regel 2 - einfacher Gleiter



(n) Regel 225 - komplexer Gleiter

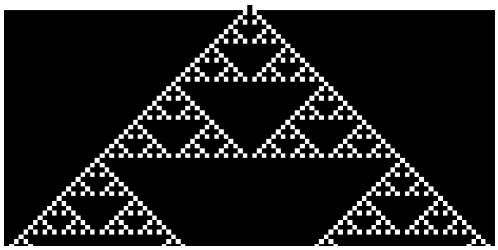
Farbinversion

Wie in Abschnitt 3.3.1 beschrieben, existieren genau farbinvertierte Muster zu einigen Entwicklungsregeln. Dies hängt teilweise mit der Symmetrie des „Regelbytes“ zusammen. Die bitweise Darstellung der Regelnummer, die symmetrisch bzgl. des Bytezentrums (zw. dem 4. und 5. Bit) ist, besitzt auf jeden Fall ein dazugehöriges Negativmuster. Dies wird an den folgenden Beispielen verdeutlicht. Das letzte Bildpaar ist kein Positiv-Negativ-Paar, im Gegensatz zu den beiden Bildpaaren zuvor, und das nur, weil das „Regelbyte“ nicht die nötige Symmetrie aufweist.

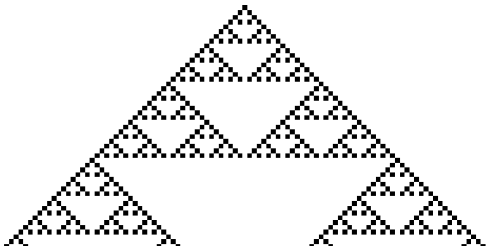


(o) Regel 60 bzw. 00111100

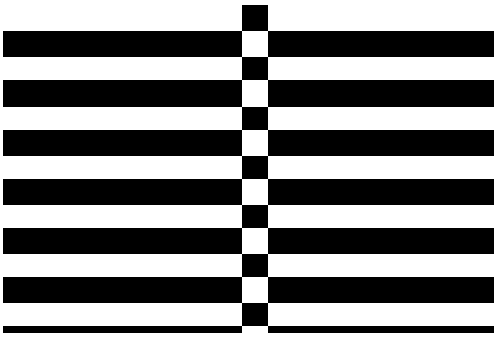
(p) Regel 195 bzw. 11000011



(q) Regel 90 bzw. 01011010



(r) Regel 165 bzw. 10100101

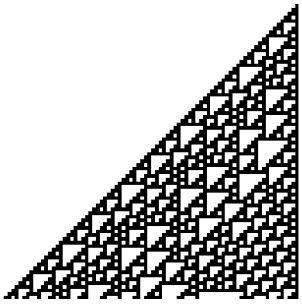


(s) Regel 51 bzw. 00110011

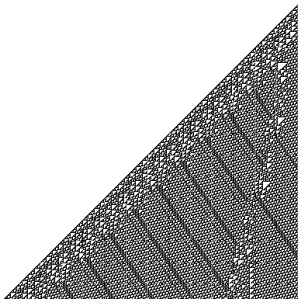
(t) Regel 204 bzw. 11001100

Größenverhältnisse

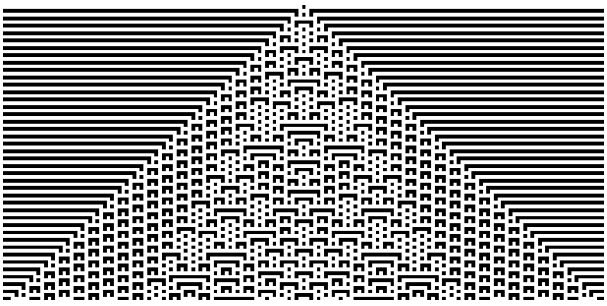
Die Entwicklung für höhere Generationsanzahlen bringt manchmal sehr interessante Gebilde mit sich. Sicherlich muss man differenzieren, welche Effekte sich durch den Zoom und welche sich tatsächlich aus dem Muster ergeben, aber es scheint sich manchmal eine Art makroskopische Struktur aus der mikroskopischen Zellentwicklung herauszubilden. Auch für dieses Phänomen seien hier zwei Beispiele gegeben:



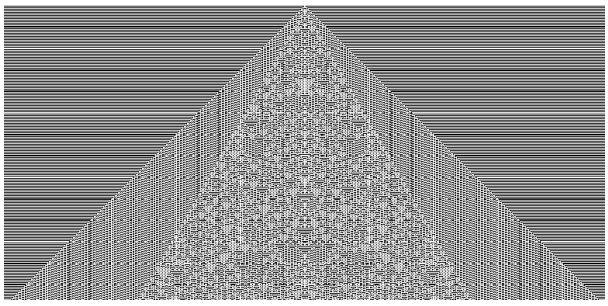
(u) Regel 110 - 80 Generationen



(v) Regel 110 - 400 Generationen



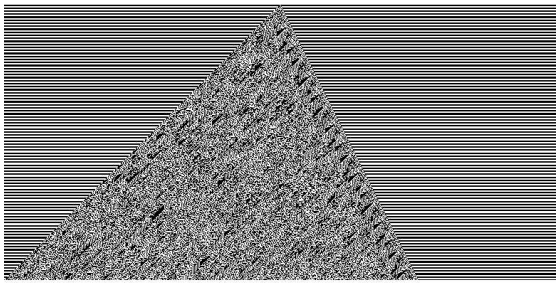
(w) Regel 73 - 80 Generationen



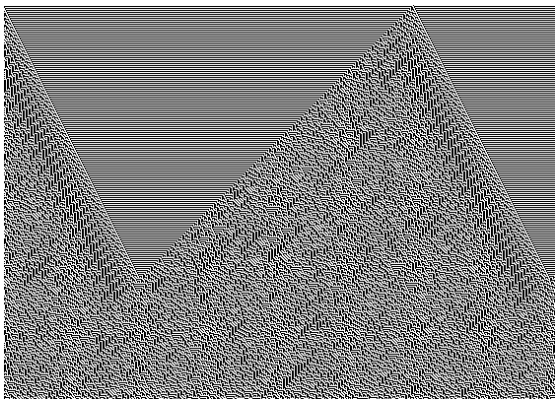
(x) Regel 73 - 400 Generationen

Randbedingungen

Zur Unterstreichung der vorhergehenden Erläuterungen über die Notwendigkeit von Randbedingungen in Abschnitt 3.3.3 werden hier zwei Ergebnisbilder gegenübergestellt, wobei das linke Bild mit den beschriebenen Randbedingungen entstanden ist. Das Zweite stammt aus einem älteren Algorithmus, in dem die Arrayhöhe noch nicht der Generation angepasst war und eine links- und rechtsseitige Ausbreitung oftmals an die Randbereiche hinein ragte. Die Randbereiche selber wurden von der x-Schleife in der Regel nicht erreicht, wodurch die äußersten Zellen immer weis blieben. Aber genau diese Zellen waren auch nötig für die Berechnung der Folgegeneration, weil man ja nicht mit jeder Generation immer mehr Randzellen weglassen konnte. Aus diesem Randgebiet ergeben sich die Effekte auf dem rechten Bild, welche die Hauptentwicklung (ausgehend vom Einzelpixel) beeinflussen.



(y) Regel 89 - ca. 600 Generationen; Angepasste Arraybreite und mit Randbedingungen



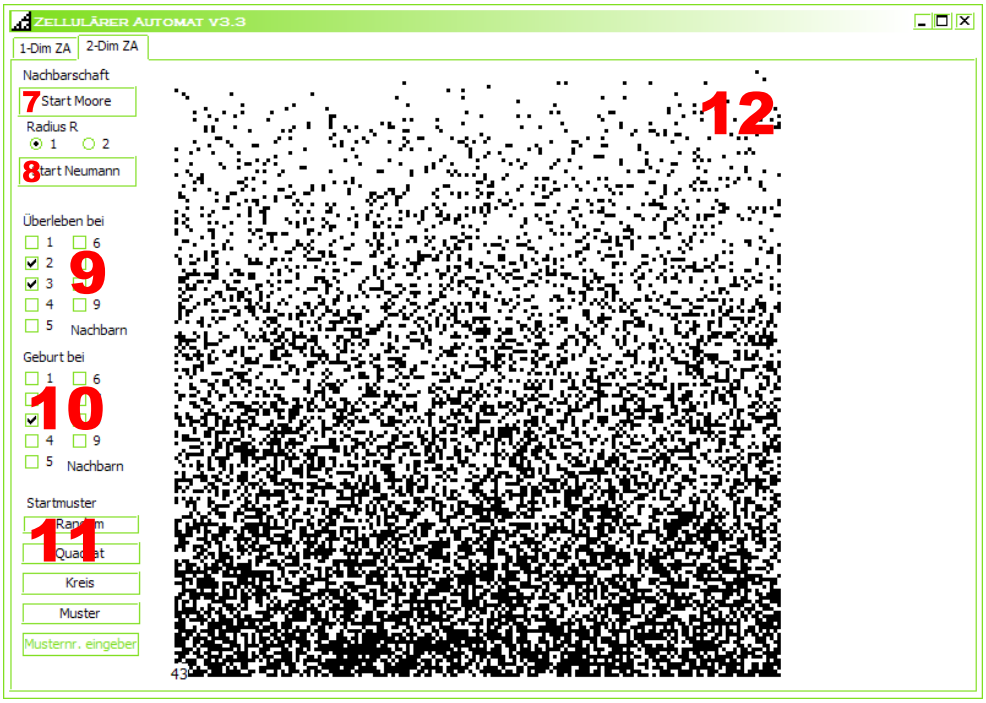
(z) Regel 89 - ca. 600 Generationen; Keine Angepasste Arraybreite und ohne Randbedingungen

4 Implementierung eines zweidimensionalen ZA

Dieser Programmteil wurde aus reinem Interesse an dem Themengebiet hinzugefügt und hat im eigentlichen Sinne nichts mit der Aufgabenstellung zu tun. Von daher gehe ich nicht so detailiert auf die Funktionsweise ein, wie ich es im Abschnitt 3.3 getan habe.

4.1 Die Nutzeroberfläche

Auch hier sei zur Einführung eine Übersicht über die Bedienoberfläche des zweiten Tab-Reiters der Software gegeben:



Beschreibung:

(7) - Startknopf des 2D ZA mit Moore'scher Nachbarschaftsabfrage

(8) - Startknopf des 2D ZA mit Neumann'scher Nachbarschaftsabfrage

(9) - Auswahlfeld zur Beeinflussung der Sterbemenge (auch während der Laufzeit)

(10) - Auswahlfeld zur Beeinflussung der Geburtenmenge (auch während der Laufzeit)

(11) - Diverse Knöpfe zur Erstellung verschiedener Startpopulationen

(12) - Zeichenfläche Image2 - „Lebensraum der Zellen"

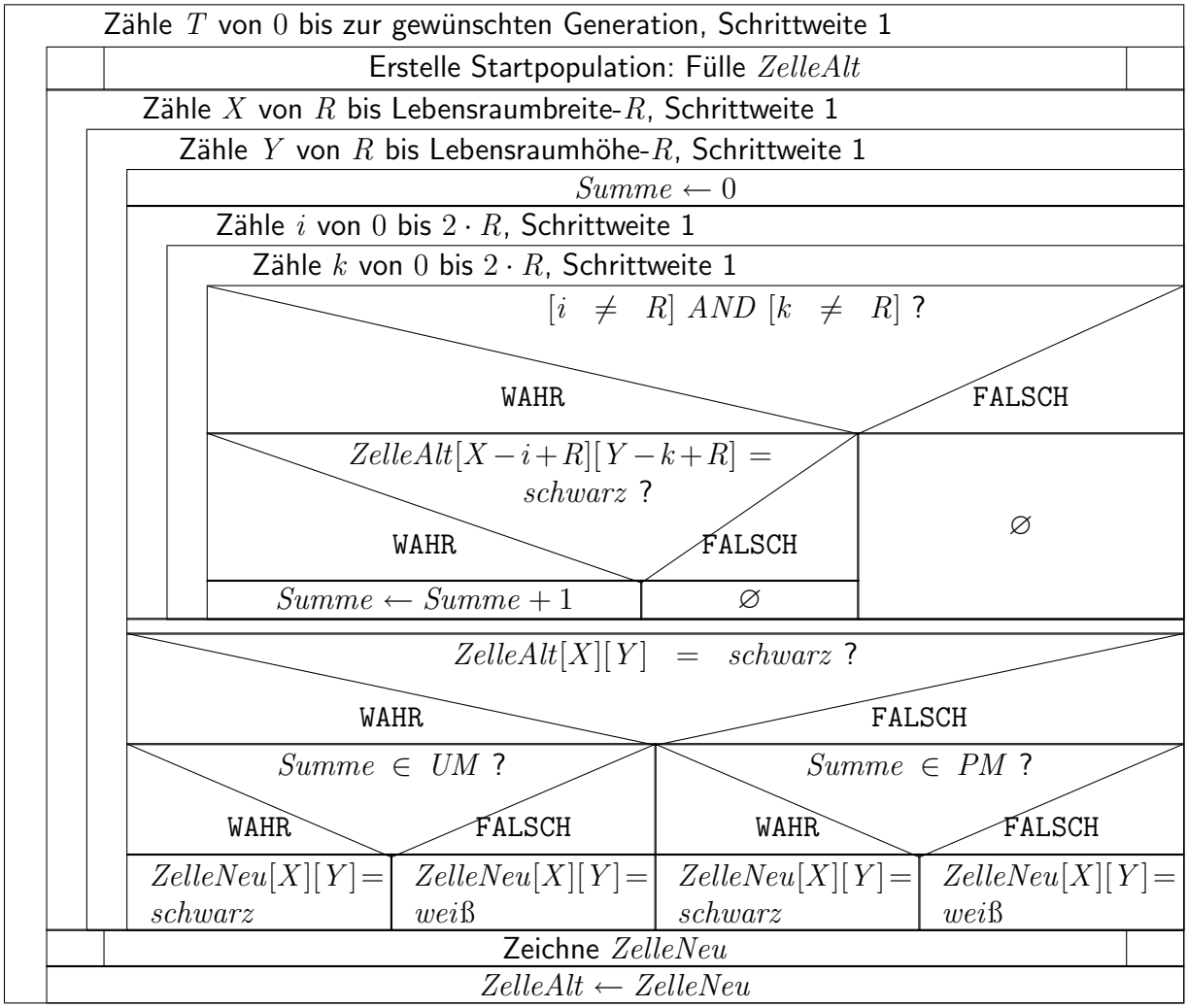
4.2 Algorithmus

Wie schon in Abschnitt 2.1.2 beschrieben dient die zweite Dimension nicht mehr zur Darstellung der Folgegenerationen, sondern als Aufenthaltsraum weiterer Zellen. Generationen werden nur noch bildweise und nicht mehr aneinander gereiht dargestellt. Die Arbeitsweise des zweidimensionale zellulären Automaten (2D ZA) basiert hier auf einem Zellmengenvergleich zwischen der Anzahl lebender Zellen in der Nachbarschaft einer betrachteten Zelle und dem Inhalt zweier Mengen, der Überlebensmenge und der Geburtenmenge. Ist bspw. die betrachtete Zelle tot, so lebt sie in der Folgegeneration genau dann, wenn die Anzahl ihrer Nachbarn in der Geburtenmenge vorkommt, ähnlich verhält es sich mit schon lebenden Zelle und der Überlebensmenge.

Für den einfachen Einstieg in die Implementierung ist hier wieder ein Struktogramm dargestellt. Dieses beschreibt jedoch nur die Abfrage der Moore'schen Nachbarschaft.

2D ZA mit Kontroll-Mengen

Globale Variablen:	
Summe	{Integer-Variable}
R	{Integer-Input: Radius der Moore-Nachbarschaft}
UM	{Überlebensmenge - Enthält mögliche Anzahlen (Integer zwischen 1 und 9) von Nachbarzellen für das Überleben der Zelle im Zentrum der Nachbarschaft}
PM	{Paarungsmenge - Enthält mögliche Anzahlen (Integer zwischen 1 und 9) von Nachbarzellen für die Geburt einer Zelle im Zentrum der Nachbarschaft}
ZelleAlt[x][y], ZelleNeu[x][y]	{Arrays der Breite x (für x-Richtung) und Höhe y (für y-Richtung) mit Einträgen vom Typ char → Zelllebensraum}
Integer-Laufvariablen:	
X, Y	{Positionszähler}
T	{Generationszähler}
i, k	{Variablen zur Nachbarschaftsabfrage}



4.3 Implementierung

Die hier dargestellte Routine `void __fastcall TForm1::Timer3Timer(TObject *Sender)` (grau hinterlegt) ist wieder eine Timerprozedure, die auf wiederholte Aufrufe von dem Timer3 reagiert und durch die Startknöpfe (#7) und (#8) ausgelöst wird. Diese Routine enthält die Abfrage der Moore'schen und Neumann'schen Nachbarschaft. Mit der Variablen `R2D` wird der Radius der Moore'schen Nachbarschaft bezeichnet. Er kann durch die Radioboxen unter dem Startknopf (#7) geändert werden. Die Neumann'sche Nachbarschaft besitzt in dieser Version den festen Radius Eins. Wiederum wurden zur eleganten geschachtelten Umsetzung des Programmcodes eigene Unterroutinen genutzt, welche an den geeigneten Stellen wieder zwischengeschoben (weis hinterlegt) oder kurz kommentiert werden.

```
{
   int sum;

   if(zeit<600) //Obergrenze für Zähler -> Verhinderung einer Dauerschleife
   {
     if(moore)     //Moore'sche Nachbarschaft ohne Randbedingungen
       for(int x=R2D; x<W-R2D; x++)    //xy-Abtastung des Lebensraums
         for(int y=R2D; y<H-R2D; y++)  //unter Ausschluss der Ränder
         {
           sum = 0;

           for(int n=-R2D; n<R2D+1; n++)  //Nachbarschaftsabfrage mit dem Radius R2D
             for(int m=-R2D; m<R2D+1; m++)
               if((m!=0) || (n!=0))        //Ausschluss der zentralen Zelle
                 if(BildAlt[x+n][y+m] == 1) //Wenn eine Nachbarzelle lebt...
                   sum++;

           BildNeu[x][y] = schicksal(sum,BildAlt[x][y]);  //Zuweisung des Zustandes der
                                                          //Folgezelle
         }
```

`BildNeu` ist der neue Lebensraum der Zellen, der nach und nach aus den Daten des alten Lebensraums `BildAlt` generiert wird und zwar nach den Vorschriften, die in der Funktion `schicksal` stecken. Sind alle Zellen der Folgegeneration berechnet, wird der alte gegen den neuen Lebensraum ausgetauscht und die Prozedur beginnt von vorn. Die beiden Lebensräume sind vom Typ TBildmatrix, dessen Definition lautet: `typedef char TBildmatrix[W][H];` wobei hier $H = W = 167$ gilt. Es handelt sich also um gleichdimensionierte Matrizen mit Einträgen vom Typ char.

Der Funktion `schicksal` werden zwei Paramter übergeben, zum Einen die Anzahl der lebenden Nachbarzellen `sum` und zum Anderen der Zustand `farbe` der betrachteten Zelle selbst. Die Funktion arbeitet mit den zwei Mengen `MengeU` (Überlebensmenge) und `MengeG` (Geburtenmenge). Da mir kein Mengenbegriff für C++ bekannt ist, mussten Mengen in Form von Arrays erstellt werden. Beide Mengen sind nun vom Typ TMenge, welcher wie folgt definiert wurde: `typedef int TMenge[9]`. Die beiden Arrays werden nach Programmstart als Nullmengen initialisiert. Daraufhin werden sie mit den Zahlenwerten gefüllt, deren entsprechende Checkboxen in den Gruppierungen (#9) und (#10) aktiviert sind. Diese Checkboxen sind während der Laufzeit (also auch während Timeraktivität) variierbar, was zur Folge hat, dass auch die beiden Mengen zur Laufzeit anpassbar sind. Das Checkboxsystem enthält die Ziffer neun. Diese Zahl spielt natrlich bei der Moore'schen Nachbarschaftsabfrage mit dem Radius Eins kein Rolle.

Zum Vergleich, ob die Nachbarzellenanzahl in einer der beiden Mengen enthalten ist, musste wiederum eine Funktion geschrieben werden, die diesen Vergleich übernimmt. Den Quellcode dieser Funktion `WertInMenge` findet man im Anhang C.

```
char __fastcall schicksal(int sum, char farbe)
{
   if(farbe == 1)                    //Betrachtete Zelle lebt.
     if (WertInMenge(MengeU, sum))   //Wenn sum in MengeU vorkommt...
       return(1);    //dann überlebt die betrachtet Zelle in der Folgegeneration
     else return(0); //sonst stirb die Folgezelle.
   else                              //Betrachtete Zelle ist tot.
     if (WertInMenge(MengeG, sum))   //Wenn sum in MengeG vorkommt...
       return(1);    //dann wird eine Folgezelle geboren
     else return(0); //sonst passiert nichts.
}
```

Nachdem die Erstellung der Folgegeneration mittels eine Moore'schen Nachbarschaft erläutert wurde, komme ich nun zu der Neumann'schen Nachbarschaft. In diese Nachbarschaftsabfrage beziehe ich die Ränder des Lebensraum-Arrays mit ein, indem ich einfach wieder die Randgebiete fließend aneinander lege und meine quadratische Zeichenfläche sozusagen in die jeweilige Richtung zu einem Zylinder forme. Bei der Verwendung des Programms fällt jedoch auf, dass diese Randwertbetrachtung nicht fehlerfrei sein kann (immernoch Pixelansammlungen an den Randbereichen). Leider habe ich es bisher nicht geschafft diesen Fehler ausfindig zu machen.

Die Entscheidung über die Zellzustände der Folgegeneration verläuft ähnlich wie bei dem 1D ZA auf Modulo-Basis.

```
if(!moore) //Start Neumann
  for(int x=0; x<W; x++)   //xy-Abtastung des Lebensraums
    for(int y=0; y<H; y++)
    {
      switch(x)       //Randabfrage in x-Richtung
      {
        case 0:       //Am linken Rand
          switch(y)   //Randabfrage in y-Richtung
          {
            case 0:   //Am oberen Rand
              sum = BildAlt[W-1][y]+BildAlt[x+1][y]+BildAlt[x][H-1]+BildAlt[x][y+1];
            case H-1: //Am unteren Rand
              sum = BildAlt[W-1][y]+BildAlt[x+1][y]+BildAlt[x][y-1]+BildAlt[x][0];
            default:  //Im Inneren 0<y<H-1
              sum = BildAlt[W-1][y]+BildAlt[x+1][y]+BildAlt[x][y-1]+BildAlt[x][y+1];
```

```
            case W-1:     //Am rechten Rand
              switch(y)   //Randabfrage in y-Richtung
              {
                case 0:
                  sum = BildAlt[x-1][y]+BildAlt[0][y]+BildAlt[x][H-1]+BildAlt[x][y+1];
                case H-1:
                  sum = BildAlt[x-1][y]+BildAlt[0][y]+BildAlt[x][y-1]+BildAlt[x][0];
                default:
                  sum = BildAlt[x-1][y]+BildAlt[0][y]+BildAlt[x][y-1]+BildAlt[x][y+1];
              }

            default:     //Im Inneren 0<x<W-1
              switch(y)  //Randabfrage in y-Richtung
              {
                case 0:
                  sum = BildAlt[x-1][y]+BildAlt[x+1][y]+BildAlt[x][H-1]+BildAlt[x][y+1];
                case H-1:
                  sum = BildAlt[x-1][y]+BildAlt[x+1][y]+BildAlt[x][y-1]+BildAlt[x][0];
                default: //Normale Aufsummierung der Nachbarzellzustände ohne Ränder
                  sum = BildAlt[x-1][y]+BildAlt[x+1][y]+BildAlt[x][y-1]+BildAlt[x][y+1];
              }
          }
           if ((sum % 2 == 0) && (sum != 0)) //Wenn gerade Anzahl an Nachbarzellen...
             BildNeu[x][y] = 1; //dann lebt die Folgezelle
           else
             BildNeu[x][y] = 0; //sonst stirbt die Folgezelle
        }
```

Der folgende Code gilt wieder für beide Nachbarschaften und er wird durchgeführt, wenn der neue Lebensraum `BildNeu` nach einer Methode fertig berechnet wurde. Hier erfolgt unter anderem die grafische Ausgabe des neu berechneten Lebensraums. Das Array `BildNeu` wird mittels einer Grafikroutine, die der aus Abschnitt 3.3.4 nahe kommt (s. Anhang D), auf der Leinwand (#12) dargestellt, deren Pixelbreite und -höhe der dreifachen Arraybreite und -höhe entspricht (Zoomeffekt).

```
    for(int x=0; x<W; x++)
       for(int y=0; y<H; y++)
       {
         BildAlt[x][y] = BildNeu[x][y] ; // Überschreibung des alten Lebensraums
       }

    MatrixToBild(Image2, BildNeu); //Grafische Ausgabe des neuen Lebensraums auf Image2
    zeit++;  //Erhöhung des Generationszählers zeit
             //-> wird auch auf Image2 ausgegeben
  }
  else
  {
    Timer3->Enabled = false; //Timerdeaktivierung bei 600 entwickelten Generationen
    Button6->Caption = "Start Moore";
  }
}
```

5 Diskussion

Wie die Aufgabenstellung es verlangte, ist es mir gelungen, einen eindimensionalen zellulären Automaten mit fester Regelübergabe und generationsangepasster grafischen Ausgabe zu implementieren. Vom Benutzer frei wählbar sind die Regelnummer (Wert zwischen 0 und 255) und die zu berechnende Generation, ausgehend von einem festen Startpixel. Der Algorithmus wurde anhand von Grafiken, Struktogramm und Quellcode anschaulich erläutert. Unter der allgemeinen Betrachtung des Klassensystems der zellulären Automaten konnte ich auch fraktale und teilweise chaotische Entwicklungen aufzeigen. Auf die Bedeutung von Randbedingungen an den Lebendraumgrenzen und auf Farbinversionen bin ich zusätzlich eingegangen.

Verbesserungfähig ist auf jeden Fall die Geschwindigkeit des Systems. Geht man weit über die Berechnung von 500 Generationen hinaus, kann das ganze Verfahren sehr zeitaufwendig werden. Eine Zeitersparnis würde eine angepasste Zeilenabtastung mit sich bringen. Betrachtet man nur gerade Regelnummern, weis man, dass auf Zelltripel der Konfiguration [000] wieder nur eine weiße Zelle folgt. Somit sind diese Bereiche an sich uninteressant und man kann, statt die gesamte Zeile zu durchlaufen und Zelle für Zelle abzufragen, gleich zu dem interessanten Bereich springen und die wichtigen Entwicklungen betrachten. Die Schleifengrenzen wären dann von der temporären Generation und von der gesamten Arraybreite abhängig.

Als zusätzliche Erweiterung könnte man einen Ausbau des Startzustandes vornehmen, über ein einzelnes Startpixel hinweg. Dies würde auf jeden Fall eine Verbreiterung des Zellarrays mit sich führen. Da das daraus resultierende Ergebnis jedoch nicht allzu spektakulär sein sollte, wäre es vielleicht interessanter, in die laufende Entwicklung Punkte einzustreuen, sodass das „fertige“ deterministische Muster an diversen Punkten gestört und evtl. sogar teilweise unterbrochen wird. Die Untersuchung dieser Störfaktoren könnte nochmals eine vom Benutzer bedingbare Parameterschar hervorbringen, wodurch das Verhalten des Automaten nochmal grundlegend verändert werden und die Vielseitigkeit des Systems enorm gesteigert werden könnte.

Zusätzlich entschied ich mich für die Implementierung eines eindimensionalen zellulären Au-

tomaten auf Modulo-Operator-Basis und eines zweidimensionalen zellulären Automaten, welcher die Moore'sche und die Neumann'sche Nachbarschaftsabfrage beherrscht.

Aufgrund des großen Umfangs, habe ich mich dazu entschieden, die Ergebnisse des 2D ZA nicht mit in diese Arbeit einzuarbeiten. Es sei an dieser Stelle aber gesagt, dass ich viele der Strukturen (staitische sowie dynamische), die unter der Quelle (III) nachzulesen sind, auch in meinem System für die spezifischen Einstellungen beobachten konnte.

Gerade an dem zweidimensionalen System können noch eine Vielzahl an Verbesserungen vorgenommen werden. So versuche ich seit einiger Zeit auch für die Moore'sche Nachbarschaft Randbedingungen einzuarbeiten, was mir vom Ansatz her auch schon gelungen sein sollte. Jedoch zeigt sich hier genau dasselbe Problem wie bei der Neumann'schen Nachbarschaft. Trotz Randbedingungen sammeln sich Zellen an den Kanten an, die dort, logisch betrachtet, mit einbezogenen Randbedingung gar nicht mehr existieren dürften.

Eine weitere Verbesserungsmöglichkeit, wäre die Ausweitung der grafischen Darstellung des zweidimensionalen Automaten auf drei Dimensionen, wobei die Zeitschritte die dritte Dimension aufspannen (äquivalent zu den eindimensionalen Automaten, deren zweite Dimension auch die zeitliche Abfolge darstellt). Sicherlich würde sich dafür sehr interessante Gebilder ergeben (wie bspw. bei einer Klasse 1 Konfiguration $\longrightarrow$ Kegel bzw. Spitzen die in die Zeitrichtung ragen).

Generell gilt für alle in dieser Arbeit beschriebenen Automaten eine Erweiterungsmöglichkeit, nämlich die Ausweitung von Zwei-Zustandssystemen auf Viel-Zustandssysteme. Grafisch lassen sich diese bswp. durch Verwendung verschiedener Farben der Zellen realisieren. Numerisch müsste man von der eleganten bitweisen Darstellung der Zustandskonfigurationen und Entwicklungsregeln abkommen, was u.a. eine Regelzuweisung unwahrscheinlich verkomplizieren würde, aber die Möglichkeit besteht.

6 Quellen

(I) `http://www.vlin.de/material/ZAutomaten.pdf`

(II) `http://de.wikipedia.org/wiki/Zellul%C3%A4rer_Automat`

(III) `http://de.wikipedia.org/wiki/Conways_Spiel_des_Lebens`

(IV) `http://www.usf.uni-osnabrueck.de/~mhinsch/zellu/zellu.html#Kla1`

7 Anhang

A Quellcode der Funktion IntToByte

```
String __fastcall IntToByte(double I, int laenge)
{
    String Byt = "";
    double Arg;

    for (int n=laenge; n>0; n--)
    {
      Arg = exp((n-1)*log(2));
      if (Arg <= I)
      {
        Byt += "1";
        I -= Arg;
      }
      else
        Byt += "0";
    }
       return(Byt);
}
```

B Quellcode der Prozedur uebermalen

```
void __fastcall uebermalen(TImage *Image)
{
  Image->Canvas->Brush->Color = clWhite;
  Image->Canvas->Pen->Color = clWhite;
  Image->Canvas->Rectangle(0,0,(Image->Width),(Image->Height));

  if(Image == Form1->Image2)
  for(int x = 0; x<W ; x++)
    for(int y = 0; y < W; y++)
      BildAlt[x][y]= 0;
}
```

C Quellcode WertInMenge

```
bool __fastcall WertInMenge(TMenge Menge, int Wert)
{
  if(Wert<10 && Wert!=0)
    return(Menge[Wert-1]==Wert);
```

```
    else
      return(false);
  }
```

D Quellcode der grafischen Ausgabe des 2D ZA

```
  void __fastcall MatrixToBild(TImage *Bild, TBildmatrix &Matrix)
  {
    int X;
    int Y;

    for(int x = 0; x < Bild->Width; x++)
      for(int y = 0; y < Bild->Height; y++)
      {
        X = x/3 ;
        Y = y/3 ;
        if(X>=W)       //Sicherheitsabfragen damit
          X=W-1;       //Matrixgrenzen nicht berschritten
        if(Y>=H)           //werden!
          Y=H-1;

        if(Matrix[X][Y] == 1)
          Bild->Canvas->Pixels[x][y] = clBlack;
        else
          Bild->Canvas->Pixels[x][y] = clWhite;
      }
  }
```

E Quellcode der Variablen-Initialisierung bei Programmstart

```
     void __fastcall TForm1::FormCreate(TObject *Sender)
  {
  //Startwerte des ersten Tabs
    Button2->Enabled = false;
    Timer2->Enabled = false;
    R = RadioGroup1->ItemIndex+1;
    Modulo = RadioGroup3->ItemIndex+2;
    for (int m=0; m<8; m++)
      regelmatrix[m] = "";


  //Startwerte des zweiten Tabs
    CheckBox2->Checked = true;
```

```
    CheckBox3->Checked = true;
    CheckBox12->Checked = true;
    Timer3->Enabled = false;
    for(int n=0;n<9;n++)
    {
      MengeU[n] = 0;
      MengeG[n] = 0;
    }
    MengeHinzu(MengeU, 2);
    MengeHinzu(MengeU, 3);
    MengeHinzu(MengeG, 3);
    Image2->Height = H2;
    Image2->Width = W2;
    RadioGroup2->ItemIndex = 0;
    R2D = RadioGroup2->ItemIndex+1;

    Button8->OnClick(Button8);
  }
```