Proseminar „Zellularautomaten“

Betreuer: Thomas Worsch

Beitrag von
Johannes Singler
zum Thema

Effiziente Implementierung von Zellularautomaten
Praktisches Beispiel: „Game of Life“

8. Februar 2002

Inhaltsverzeichnis

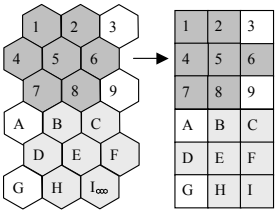
1. 1. Einführung

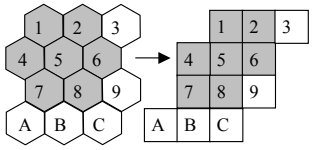
Ein Zellularautomat ist ein Berechnungsmodell, das sowohl in der Informatik selbst als auch in anderen Naturwissenschaften benutzt wird. Zellularautomaten sind im Gegensatz zu heutigen Rechnerarchitekturen allerdings massiv parallel aufgebaut, denn jede Zelle rechnet für sich in einem Zeitschritt. Es gibt zwar spezielle Hardware, die für die Simulation von Zellularautomaten konzipiert und darauf optimiert ist, doch im Normalfall wird man sich damit zufrieden geben müssen, den Automaten auf einem Rechner mit herkömmlicher Von-Neumann-Architektur zu simulieren.
Diese Arbeit soll die dabei notwendigen Design-Entscheidungen beleuchten und Möglichkeiten der effizienten Implementierung von Zellularautomaten beschreiben. Am Beispiel des sehr bekannten „Game of Life“ habe ich einige der noch anzusprechenden Aspekte konkret implementiert und auf ihre Laufzeit und Speicherverbrauch getestet.

2. Eigenschaften von Zellularautomaten und deren geschickte Implementierung

2.1 Raster

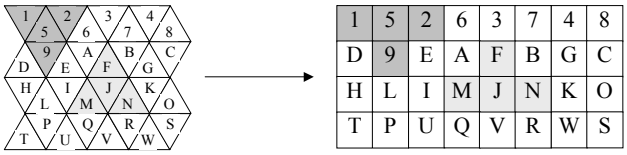
Die vielen Zellen eines Zellularautomaten sind regelmäßig in einem Raster angeordnet. Nur darüber kann man dann auch die Nachbarschaft definieren, deren Zustand in den Zustandsübergang einer Zelle mit eingeht. Es gibt jedoch viele verschiedene Möglichkeiten dieses Rasters. Für jede Zelle muss deren Zustand gespeichert werden.
Im 1-dimensionalen Raum gibt es nur einen Fall: Alle Zellen sind linear nebeneinander angeordnet. Das lässt sich sehr einfach und effizient als 1-dimensionales Array der Zustände implementieren. Alternativ könnte auch man eine verkettete Liste verwenden, denn die Zelle „sieht“ nur begrenzt weit und damit wäre die Zugriffszeit auf die Nachbarschaft konstant. Letztlich stellt dieser Fall also kein Problem dar.
Sind die Zellen in 2 Dimensionen angeordnet, ergeben sich schon mehrere Möglichkeiten eines Rasters. Das naheliegendste ist hier sicherlich das quadratische, bei dem die Zellen in Reihen und Spalten angeordnet sind. Hier bietet sich entsprechende dem 1-dimensionalen Fall wieder eine Speicherung der Zustände in einem 2-dimensionalen Array an.
Schwieriger wird es schon bei Rastern aus Dreiecken bzw. Sechsecken, die in der praktischen Anwendung durchaus Verwendung finden. Am elegantesten ist es, diese Raster auch wieder auf ein quadratisches abzubilden und dann wie oben zu speichern.
Hierzu kann man bei hexagonalen Raster folgendermaßen vorgehen z. B. auf die zwei folgenden Arten vorgehen:





Die erste Variante bietet dabei den Vorteil, dass die Nachbarschaft immer die gleiche Form hat, bei der anderen wechselt sie mit jeder Zeile (siehe Schattierung). Das erfordert einen weitere Fallunterscheidung.

Beim Dreieck-Raster bietet sich folgendes „Mapping“ an:



Im 3-dimensionalen Raum gibt es viele Möglichkeiten regelmäßiger Anordnung, z. B. ein Raster aus Würfeln. In diesem Fall kann man wieder ein 3-dimensionales Array benutzen.
Die bisher genannten Vorschläge erlauben es, die Position eines bestimmten Nachbarn der Zelle als Index im Array zu „berechnen“ und dann dessen Zustand auszulesen. Eine generell anderen Möglichkeit wäre aber noch, jeder Zelle in ihrem willkürlichen Speicherplatz Zeiger auf ihre Nachbarn mitzugeben und diese nur einmal beim Simulationsstart zu initialisieren. Dies funktioniert in beliebigen Dimensionen. Dies dürfte aber den Speicherverbrauch enorm erhöhen, da Anzahl der Zustände gegenüber der Referenz der Nachbarschaft meistens klein ist. Man gewinnt höchstens den Vorteil, das man keinen zusammenhängenden Speicher benötigt und dann besser aufteilen kann.
Über die interne Darstellung eines Zustands ist bisher noch nichts gesagt worden. Kann eine Zelle zum Beispiel n Zustände haben, lässt sich das mit einem Aufwand von $\lfloor \log_2 n \rfloor$ Bits speichern. Oft kann es jedoch sein, dass dieser Platz kleiner ist als die minimale Wortgröße des verwendeten Computers. In diesem Fall ist zu überlegen, mehrere „nebeneinander liegende“ Zellen zu einer zusammenzufassen und so den Verschnitt zu minimieren. Für die performante Berechnung der Überführungsfunktion kann es allerdings im Widerspruch dazu besser sein, wenn die zu lesenden Daten an „geraden“ Adressen liegen (Alignment). Dies resultiert aus Hardware-Interna und Cache-Management. Hier muss man einen Kompromiss aus geringeren Speicherverbrauch und höherer Berechnungsgeschwindigkeit finden.

2.2 Nachbarschaften

Für jeden Zelle muss die Nachbarschaft gleich definiert sein. Aus ihrem eigenen und den Zuständen der benachbarten Zellen berechnet sich der Zustandsübergang. Dabei gilt im Normalfall das Prinzip der Kurzsichtigkeit, d. h. die Größe der Nachbarschaft ist klein im Vergleich zum Universum. Oft genutzte Beispiele sind die Von-Neumann- und die Moore-Nachbarschaft. Aufgrund ihrer Regelmäßigkeit lassen diese sich gut mit der Speicherung im Array implementieren.
Bei der Berechnung des Folgezustands ist noch von Bedeutung, ob zwischen den Zellen in der Nachbarschaft unterschieden wird, oder ob nur über Zustände kumuliert wird. Im Extremfall muss auch der Zustand der untersuchten Zelle nicht von denen der Nachbarschaft unterschieden werden. Dies eröffnet weitere Optimierungsmöglichkeiten.

2.3 Begrenzungen

Das Universum für einen Zellularautomaten und damit die Menge der dazu gehörigen Zellen ist im klassischen Fall unendlich. Das kann jedoch auf einer Maschine mit nur endlich viel Speicher im Allgemeinen nicht simuliert werden, außer, der „aktive“ Bereich ist endlich. Daher muss man sich eine Sonderbehandlung der Grenzen überlegen.
Ein periodisches Rasters kommt einem unendlichen Raum noch am nächsten. Der rechte Nachbarn der Zelle ganz rechts ist die Zelle ganz links, der obere Nachbar der obersten die unterste Zelle usw. . Damit wird der Raum im 1-dimensionalen Fall zu einem Ring gekrümmt, im 2-dimensionalen zu einem Torus. Bei der Implementierung ist zu überlegen, die Ränder jeweils zweimal zu speichern, um Abfragen auf Sonderfälle zu vermeiden. Nach dem

Zustandsübergang müssen dann allerdings alle redundanten Daten wieder in Übereinstimmung gebracht werden.
Um geschlossene Systeme zu simulieren, bietet es sich an, Grenzen mit konstanten Zuständen einzuführen. Diese machen einfach den Zustandsübergang nicht mit und sind sonst für die Berechnung der anderen Zellen transparent.
Auch reflexive Grenzen für gewisse Anwendungen sinnvoll. Das Universum wird am Rand gespiegelt.
Darüber hinaus können natürlich auch noch alle diese Arten von Grenzbehandlung kombiniert werden oder sogar für die Ränder und evtl. auch Ecken spezielle Zustandsübergangsfunktionen definiert werden.

2.4 Berechnung der Übergangsfunktion

Die Übergangsfunktion des Zustand einer Zelle ist die stärkste Charakteristik eines bestimmten Zellularautomaten. Obwohl der Ablauf durch dieses Regel schon eindeutig festgelegt ist, kann man im Allgemeinen einen beliebig später nachfolgenden Zustand nicht berechnen, ohne den Zellularautomaten Schritt für Schritt zu simulieren.
Die Übergangsfunktion kann z. B. in einer Tabelle festgelegt sein, in der links der augenblickliche Zustand der Zelle und die Zustände in der Nachbarschaft stehen und rechts der Folgezustand eingetragen ist. Diese Tabelle kann auch Wildcards enthalten, die von jedem Zustand erfüllt werden. Durch Festlegen einer Priorität unter den Tabelleneinträgen kann die Fallunterscheidung früher abgebrochen und damit Zeit gespart werden (vgl. Routing-Tabellen).
Ein Spezialfall ist die Klasse der „Outer Totalistic CA“. Bei diesen Zellularautomaten hängt der Folgezustand nur vom eigenen und der Anzahl gewisser Zustände in der Nachbarschaft ab, d. h., die Zellen in der Nachbarschaft brauchen nicht unterschieden zu werden.
Um eine effizientere Berechnung vieler Schritte zu implementieren, kann man auch versuchen, mehrere Zellen zu einer zusammenzufassen, und deren Übergang dann gemeinsam zu suchen. Dies bietet sich vor allem bei einer geringen Anzahl Zustände an oder bei einer geeigneten Überführungsfunktion an, denn eine Überführungstabelle wächst natürlich exponentiell.
Andererseits kann man auch das Ergebnis über mehrere Schritte zu einer Regel zusammenfassen, falls die Zwischenergebnisse nicht interessieren.

3. Praktische Anwendung auf den Zellularautomaten „Game of Life“

3.1 Spezifikation des „Game of Life“

Das „Game of Life” ist ein von J.H. Conway im Jahre 1970 eingeführter Zellularautomat. Dessen Zellen sind in einem zweidimensionalen quadratischen Gitter angeordnet und besitzen jeweils nur zwei Zustände: entweder es herrscht „Leben“ oder die Zelle ist „tot“. Damit lässt sich der Zustand in nur einem Bit codieren. Die Nachbarschaft ist definiert als Moore-Nachbarschaft der Größe 1, d. h. alle 8 direkt umliegenden Zellen. Darüber hinaus ist das „Game of Life“ auch ein Beispiel für einen „Outer Totalistic CA“, denn die Überführungsfunktion hängt bloß von der *Anzahl* der lebenden Zellen in der Nachbarschaft ab.
Eine Zelle lebt im nächsten Zeitschritt genau dann, wenn eine der folgenden Bedingungen erfüllt ist:

- In der Nachbarschaft gibt es genau 3 lebendige Zellen.

- In der Nachbarschaft gibt es genau 4 lebendige Zellen und die betrachtete Zelle lebt auch bereits.

In allen anderen Fällen stirbt die Zelle. Obwohl dieser klassische Zellularautomat eine sehr einfache Überführungsfunktion und nur eine minimale Anzahl von Zuständen pro Zelle hat, lässt sich doch z. B. zeigen, dass er berechnungs-universell ist.

3.2 Konkrete Implementierung

Ich habe die Simulation des „Game of Life“ mit verschiedenen Algorithmen und Optimierungen implementiert und deren Geschwindigkeit und Speicherverbrauch experimentell getestet.
Das Programm ist in C++ geschrieben für Win-32 geschrieben. In einem umgebenden Framework sind die 5 verschiedenen Algorithmen integriert, die jeweils nur die Errechnung des nächsten Schritts übernehmen. Bei den Benchmarks wird natürlich die grafische Ausgabe nicht mitgezählt, denn das Zeichnen dauert viel länger als die eigentliche Übergangs-Berechnung und würde dadurch das Ergebnisse stark verzerren.
Im Idealfall spielt man das „Game of Life“ in einem unendlich großen Universum. Da dies auf endlichen Maschinen aber nicht simulierbar ist, wurde hier ein periodischer Raum verwendet.
Ein ziemlich allgemeingültiges Prinzip habe ich bei den ersten vier Varianten verfolgt: Da von dem Zustand einer Zelle die Folgezustände mehrerer Zellen abhängen, darf dieser nicht gleich beim Berechnen überschrieben werden. Vielleicht wird der ursprüngliche Wert noch einmal in diesem Schritt verwendet, denn im Prinzip erfolgt der Zustandsübergang simultan. Daher müssen die Folgezustände erst in einem Zwischenspeicher abgelegt werden. Es macht nun aber keinen Sinn, diese Zwischenergebnisse nach Abschluss eines Übergangs wieder zurück an die Ausgangsposition zu kopieren. Lieber vertauscht man nach jedem Schritt die Rollen von Speicher und Zwischenspeicher und spart sich so diesen Vorgang.

3.2.1 Naive Implementierung (Simple)

Der 1. Algorithmus dient als Referenz einer naiven Implementierung. Die Zustände der Zellen werden in einem 2-dimensionalen Array gespeichert. Dabei wird für jede Zelle ein ganzes Datenwort (4 Byte) verwendet, was natürlich zu großem Speicher-Verschnitt und -verschwendung führt.
Bei der Berechnung des Folgezustands wird für jede Zelle per verschachtelter Schleifen die Anzahl der lebenden Zellen in der Moore-Nachbarschaft gezählt. Aus diesem wird mittels Fallunterscheidung der Folgezustand gewonnen.

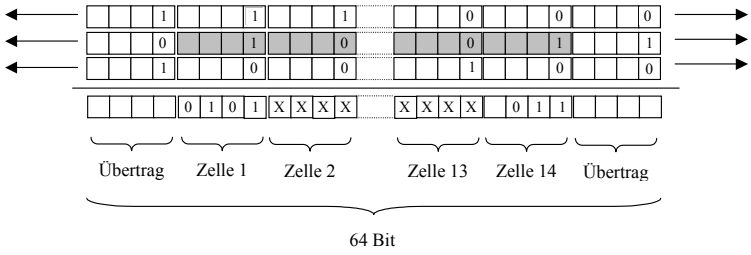
3.2.2 Mitspeicherung Anzahl lebender benachbarter Zellen (Good)

Eine höhere Berechnungs-Geschwindigkeit wird im 2. Algorithmus erzielt, indem man für jede Zelle nicht nur deren Zustand, sondern auch die aktuelle Anzahl lebender Nachbarn mitspeichert. Diese Zahl muss natürlich mit dem Zustandsübergang aktualisiert werden. Andererseits sind für die Entscheidung des Nachfolgezustands nur noch zwei statt neun Lesezugriffe auf den Speicher notwendig. Ändert sich jedoch der Zustand der Zelle, sind dann aber 9 Schreibzugriffe von Nöten. Dieses Verfahren wird also umso besser, je weniger sich im Universum noch ändert. Er nutzt vor allem die Ununterscheidbarkeit der Zellen in der Nachbarschaft aus.

3.2.3 Mehrere Zellen in einem Langwort (BitPacksVector)

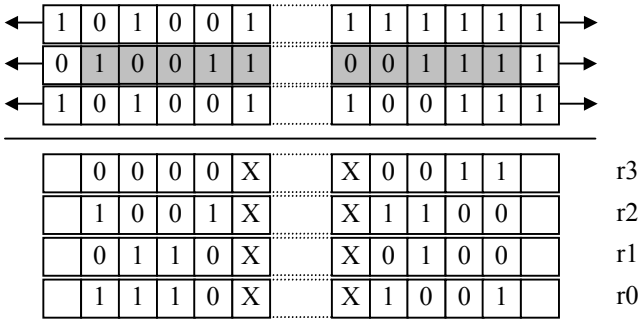
Der Speicherung eines Zellen-Zustands benötigt eigentlich nur 1 Bit, eine Anzahl lebender Zellen in der Nachbarschaft ist kleiner gleich 9, kann also mit 4 Bit repräsentiert werden. Die Idee in Algorithmus 3 ist, mehrere Zellen einer Zeile als jeweils 4 Bit nebeneinander in einem

Langwort zu speichern (z. B. 16 Stück in 64 Bit), eine 1 für *lebendig* oder eine 0 für *tot*. Die Langwörter selbst sind wieder in einem 2-dimensionalen Array strukturiert. Zur Berechnung addiert man nun jeweils 3 übereinanderliegende Langwörter, wobei man jedes jeweils einmal zuvor entweder um 4 Bit nach links schiebt, stehen lässt oder um 4 Bit nach rechts schiebt. Damit ergeben sich insgesamt 9 Summanden. Da die Summen nur bis höchstens 9 geht, entsteht innerhalb der 4-Bit-Päckchen jeweils kein Überlauf und die Zahlen werden so unabhängig voneinander addiert. Für die mittlere Zeile kann nun über einen booleschen Ausdruck der neue Zustand der Zelle berechnet werden. Einen Haken hat die Sache noch: Die beiden äußersten Zellen werden aufgrund der nicht ganz vorhandenen Nachbarschaft nicht korrekt berechnen, sie sind Verschnitt. In ein 64-Langwort passen so effektiv nur noch 14 Zellen. Außerdem müssen nach jedem Schritt diese Außenpositionen synchronisiert werden. Trotz dieses Overheads ergibt die starke Parallelisierung eine sehr verbesserte Performance gegenüber Algorithmus 2.



3.2.4 Speicherung des Zustands in einem einzelnen Bit (BitVector)

Noch weiter getrieben werden kann der Grad der Parallelisierung durch folgenden Ansatz: Wieder werden die Zustände in Langworten zusammengefasst, in diesem Fall allerdings nur noch mit einem Bit pro Zelle (somit z. B. 64 Zellen pro Langwort). Nun kann man aber nicht mehr in situ die Anzahl der lebenden Zellen in der Nachbarschaft zusammenaddieren. Zur Lösung dieses Problems werden 4 Langwörter temporär verwendet. Die jeweils gleichwertigen Bits ergeben viele virtuelle 4-Bit-Register, in denen die Summe gespeichert werden kann. Die Addition erfolgt über boolesche Ausdrücke entsprechende einem Dualzähler. Das 4. Bit kann man sich sogar sparen, denn ein Überlauf von 8 bzw. 9 nach 0 bzw. 1 ändert nichts am Endergebnis. Die Behandlung der Randstellen erfolgt entsprechend Algorithmus 3.
Die beschriebene Optimierung erhöht die Parallelität und damit effektiv auch die Ausführungsgeschwindigkeit noch einmal ungefähr um den Faktor 4.



3.2.5 Keine Berechnung inaktiver Regionen (Smart)

Die Geschwindigkeit der vorgestellten Algorithmen 1, 3 und 4 ist völlig unabhängig von der Konfiguration des Universums. Algorithmus 5 verfolgt einen intelligenteren Ansatz, der nur

wirklich benötigte Neuberechnungen durchführt. Denn in Regionen, in denen sich im vorigen Schritt keine Veränderung ergeben hat, wird sich höchstens abgesehen vom Rand (da Kurzsichtigkeit von einem Feld Weite) auch in diesem Schritt nichts tun. Sie brauchen also nicht neu berechnet zu werden. Die Erfahrung zeigt, dass sich beim „Game of Life“ praktisch immer nach einer gewissen Zeit stabile Muster bilden und es überhaupt nur einen ziemlich geringen Prozentsatz noch lebender Zellen gibt.
Dies nutzt das Berechnungsverfahren aus. Die Teilmenge des Universums, die im nächsten Schritt höchstens berechnet muss, wird in einer Menge von Rechtecken gespeichert. Schon bei Eingabe der Anfangskonfiguration wird diese Menge aufgebaut. Für eine einzelne lebende Zelle ist diese Region genau ihre Moore-Nachbarschaft. Überlappen sich zwei Rechtecke aus der Menge, werden diese zu einem großen zusammengefasst, falls damit weniger Zellen berechnet werden müssen (im Prinzip ist Doppelberechnung möglich). Dabei bekommt allerdings das Zusammenlegen von Rechtecken einen kleinen Bonus, denn dadurch spart man bei der Verwaltung der zu Menge der Rechtecke.
Bei der Übergangs-Berechnung werden nun nur noch die Inhalte der Rechtecke berechnet, dies allerdings nur mit dem naiven Algorithmus. Ändert sich der Zustand einer Zelle am Rand, wird das Rechteck um eine Position in diese Richtung vergrößert. Zudem wird zu jedem Rechteck die Anzahl der enthaltenen lebenden Zellen mitgeschrieben. Fällt der Füllungsgrad unter 10%, wird das Rechteck aufgeteilt. Jede lebende Zelle erhält ein eigenes Rechteck, und dann wird wieder zusammengefasst, falls sinnvoll. Wäre dieser letzter Schritt nicht, würden die Rechtecke nie mehr kleiner werden.
Die Performance von Algorithmus 5 hängt stark davon ab, wie viel Änderungspotential es noch gibt. Bei einem nahezu leeren oder mit stabilen Mustern gefüllten Universum ist er schneller als alle anderen, bei zufälliger Konfiguration bricht er extrem ein und ist mit Abstand die langsamste Variante.
Die Verwaltung und Konsistenzhaltung der Rechtecke benötigt relativ viel Rechenzeit und im schlimmsten Fall auch sehr viel Speicherplatz.

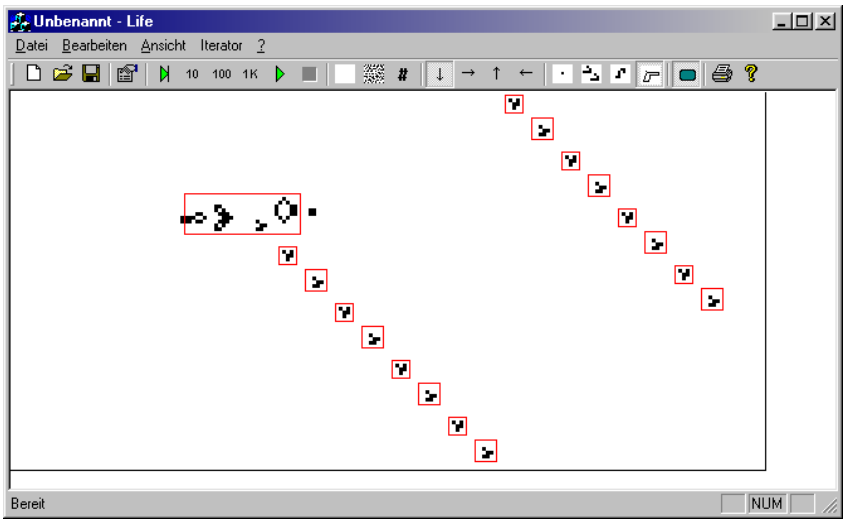
3.3. Benchmark-Ergebnisse

Die Benchmarks wurden auf einem PC mit Intel-Pentium-III-Prozessor (733 MHz) und 384 MB Arbeitsspeicher unter Windows 98 SE durchgeführt. Die Geschwindigkeit ist in der Einheit „Millionen Zustandsübergänge · Anzahl Zellen pro Sekunde“ (MUpds) angegeben, der Speicherverbrauch in KByte.
Die Tests erfolgten jeweils auf einem großen und einem kleinen quadratischen Feld. Dabei entsteht bei Größe 868 bei Algorithmus 3 und 4 kein Verschnitt, d. h. sie laufen in optimaler Konfiguration. Bei den Algorithmus 2, deren Geschwindigkeit von der Konfiguration abhängt, wurden verschiedene getestet: leeres Feld/ bistabile Konfiguration/ nur eine „Gosper’s Gun“ zu Anfang/ zufällige Konfiguration (50% lebende Zellen). Dabei wurden jeweils die ersten 100 Schritte simuliert.

	Kleines Feld (100×100)		Großes Feld (868×868)	
Algorithmus	Geschwindigkeit	Speicher	Geschwindigkeit	Speicher
1 (Simple)	1,5	78	1,4	5886
2 (Good)	34,6/29,4/32,8/14,3	19	28,3/25,0/28,0/12,6	1471
3 (BitPacks)	55,3	12	51,0	840
4 (BitVector)	165,7	3	243,0	189
5 (Smart)	6500/11,0/30,2/1,1	546	502000/–/755/–[1]	41000

[1] – : Test wurde nicht in angemessener Zeit beendet

Wie nicht anders zu erwarten war, hängt die Performance von Algorithmus 5 sehr stark von der Konfiguration ab. Bei (fast) leerem Feld bzw. lokaler Eingrenzung ist er schneller als alle anderen, spart allerdings nicht mit Speicher, da er für den schlimmsten Fall gerüstet sein muss. Bei zufälliger oder stabiler, aber großflächiger Konfiguration, wird er auf dem großen Feld gar nicht mehr fertig. Algorithmus 4 ist die Konfiguration egal, und mit einer für ihn günstigen Breite schafft er ein Zellupdate in durchschnittlich nur 3 Takten.



Screenshot des Programms mit einer Gosper's Gun, Rechtecke des Algorithmus 5 (Smart)

Die Ausarbeitung, die Folien zum Vortrag sowie das lauffähige Programm inklusive Quelltext sind unter folgende Web-Adresse zu finden:

http://ww.jsingler.de/proseminar_index.html

Verwendete Literatur:

- Jörg R. Weimar - Simulation with Cellular Automata, Braunschweig 1996 http://www.tu-bs.de/institute/WiR/weimar/ZAscript/ZAscript.html
- M. Delorme, J. Mazoyer (Herausgeber) – Cellular Automata – A Parallel Model Kluwer Academic Publishers 1999