Zellomat3D

Studie Algorithmen

Projekt:	3D Cellular Automata Simulator – Diplomarbeit – SS/2005	
Auftraggeber:	Hochschule Rapperswil HSR	
Betreuer:	Eduard Glatz – Prof. Dipl. Ing. ETH	eglatz@hsr.ch
Mitarbeiter:	Michael Florin	loop@loop.li
	Andreas Weinmann	a.weinmann@gmx.ch
Ablage:	StudieAlgorithmen - 05052005.doc	

Inhaltsverzeichnis

1. EINFÜHRUNG 4

ZWECK 4
GÜLTIGKEITSBEREICH 4

2. BESTIMMUNG DER ZELLZUSTÄNDE 4

PROBLEMATIK 4
LÖSUNGSANSÄTZE DER ZELLBERECHNUNG 4

3. LÖSUNGANSATZ MIT 3D-ARRAY 5

BESCHREIBUNG 5
SPEICHERVERBRAUCH VON ARRAY 5
ZUGRIFF AUF DIE NACHBARN EINER ZELLE IM ARRAY 5
AUSWAHL DER ZU BERECHNENDEN ZELLEN (ZYKLUSSTEUERUNG) IM ARRAY 5

4. LÖSUNGANSATZ MIT VERKETTETEN LISTEN 6

BESCHREIBUNG 6
SPEICHERVERBRAUCH BEI VERKETTETER LISTE 6
ZUGRIFF AUF DIE NACHBARN EINER ZELLE BEI VERKETTETER LISTE 6
AUSWAHL DER ZU BERECHNENDEN ZELLEN (ZYKLUSSTEUERUNG) BEI VERKETTETER LISTE 6

5. LÖSUNGANSATZ MIT OKTALBAUM 7

BESCHREIBUNG 7
SPEICHERVERBRAUCH MIT OKTALBAUM 7
ZUGRIFF AUF DIE NACHBARN EINER ZELLE MIT OKTALBAUM 7
AUSWAHL DER ZU BERECHNENDEN ZELLEN (ZYKLUSSTEUERUNG) MIT OKTALBAUM 8

6. LÖSUNGANSATZ MIT TABLE TREE 9

BESCHREIBUNG 9
SPEICHERBRAUCH MIT TABLE TREE 10
ZUGRIFF AUF DIE NACHBARN EINER ZELLE MIT TABLE TREE 10
AUSWAHL DER ZU BERECHNENDEN ZELLEN (ZYKLUSSTEUERUNG) MIT TABLE TREE 10

7. AUSWERTUNG LÖSUNGSANSÄTZE (ZELLBERECHNUNG) 11

THEORETISCHE AUSWERTUNG 11
RESULTATE DER PROTOTYPEN 12

8. THEORETISCHE BETRACHTUNG DES TABLE TREE 13

THEORETISCHE BETRACHTUNGEN DER STÄRKEN UND SCHWÄCHEN 13
„WORST CASE“ 13
THEORETISCHER „BEST CASE“ 13
PRAKTISCHER „BEST CASE 13
DURCHSCHNITTLICHER PERFORMANCEGEWINN 14

9. VARIANTE DES TABLE TREE 15

GRÖSSERE ANZAHL DER KINDER 15

10. OPTIMIERUNG EINES TABLE TREE 16

REDUKTION DER NACHBARSCHAFTS- ABFRAGEN BEI DEN KNOTEN 16
KEIN „CHANGED“ IN DEN BLÄTTERN 16
MASKIEREN DER „CHANGED“ 16
KINDER BASIERTE NACHBARSCHAFTS-PRÜFUNG 16
UPDATERATE 17

11. PROTOTYPENAUSWERTUNG VERSCHIEDENER TABLE TREE 18

ALLGEMEIN 18

VERGLEICH OHNE UND MIT „CHANGED“ IN DEN BLÄTTERN 18
RESULTAT VERGLEICH OHNE UND MIT „CHANGED“ IN DEN BLÄTTERN 18
REDUKTION DER NACHBARSCHAFTS- ABFRAGEN BEI DEN KNOTEN 19
RESULTAT VERGLEICH OHNE UND MIT „DELEGATES“ 19
MASKIEREN DER „CHANGED“ 20
RESULTAT VON MASKIERUNG 20
KINDERBASIERTE NACHBARSCHAFTS-PRÜFUNG 20
RESULTAT VON KINDER-BASIERTE NACHBARSCHAFTS-PRÜFUNG 20
FAZIT 20
RESULTAT VON UPDATERATE 21

12. DATENAUFBEREITUNG 22

PROBLEMATIK 22
LÖSUNG 22
OBERFLÄCHE INNEN 22
OBERFLÄCHE AUSSEN 23
VORGEHEN 23

13. GENERIERUNG DER OBERFLÄCHE DURCH ITERIERUNG 24

HINWEIS 24
ERKLÄRUNG 24
FAZIT 25

14. GENERIERUNG DER OBERFLÄCHE IM TABLETREE 26

HINWEIS 26
ERKLÄRUNG 26

15. FRAMEGENERIERUNG 27

PROBLEMATIK 27
LÖSUNG 27

1. Einführung

Zweck

Dieses Dokument dient dazu, dem Leser die verschiedenen Algorithmen, die während der Diplomarbeit entwickelt wurden, zu erklären. Die Gründe für die Entwicklung werden erläutert und der Einsatzbereich sowie die Tauglichkeit für die finale Applikation eruiert.

Gültigkeitsbereich

Dieses Dokument gilt für die Diplomarbeit "Zellomat3D", welche im SS/2005 an der Hochschule Rapperswil HSR durchgeführt wurde.

2. Bestimmung der Zellzustände

Problematik

Der Zellberechnungsalgorithmus muss die in der Analyse gefundenen Ansprüche an die Zyklussteuerung und den einfachen Zugriff auf die Nachbarn berücksichtigen. Ausserdem darf der Algorithmus nicht zu viel Speicherbedarf haben.

Lösungsansätze der Zellberechnung

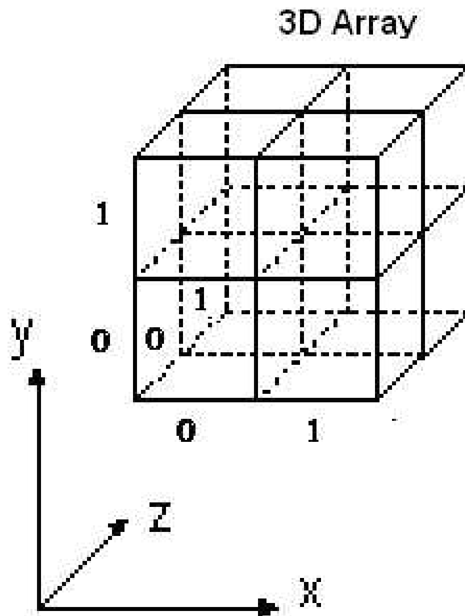
In diesem Teil werden vier grundsächlich verschiedene Algorithmen der Zellberechnung analysiert. Auf Grund dieser Analyse wird entschieden, aus welchen Möglichkeiten die Prototypen implementiert werden. Diese Prototypen dienen dann zum endgültigen Entscheid, welcher Algorithmus in der Applikation verwendet wird.

Die vier Lösungsansätze der Berechnung sind:

1. **Dreidimensionales Array**
2. **Verkettete Listen**
3. **Oktalbaum**
4. **Table Tree (Oktalbaum mit Array's)**

3. Lösunganansatz mit 3D-Array

Beschreibung



In einem 3D-Array werden alle Zellen so gespeichert, dass die Werte der Indizes den X-, Y- und Z-Koordinaten der Darstellung entsprechen.

Speicherverbrauch von Array

Die Datenhaltung durch ein Array braucht ein sehr geringes Speichervolumen. Die Position der Zellen wird aus dem Speicherort im Array abgeleitet und muss deshalb nicht gespeichert werden.

Zugriff auf die Nachbarn einer Zelle im Array

Durch eine Addition oder Subtraktion eines Indexwertes einer Zelle können die Nachbarzellen einfach bestimmt werden.

Auswahl der zu berechnenden Zellen (Zyklussteuerung) im Array

In einem Array müssen eine Zelle sowie alle ihre Nachbarn angesprungen werden, bevor man bestimmen kann, ob sie gerechnet werden muss. Es ist am effizientesten, man berechnet in jedem Zyklus alle Zellen.

4. Lösungansatz mit verketteten Listen

Beschreibung

Die Zustände aller Zellen werden in einem 3D-Array gespeichert. Zusätzlich werden die Koordinaten aller Zellen, die im anstehenden Zyklus berechnet werden müssen, in einer verketteten Liste gespeichert.

Speicherverbrauch bei verketteter Liste

Der Speicherverbrauch ist höher als bei einem Array. Es müssen noch zusätzlich alle Koordinaten der zu berechnenden Zellen in der Liste gespeichert werden.

Zugriff auf die Nachbarn einer Zelle bei verketteter Liste

Die Zugriffe auf die Nachbarn einer Zelle erfolgt im dreidimensionalen Array der gespeicherten Zustände. Es gelten dieselben Prinzipien wie bei der Datenhaltung mit einem 3D-Array.

Auswahl der zu berechnenden Zellen (Zyklussteuerung) bei verketteter Liste

In jedem Zyklus müssen die Koordinaten der Zellen, die im nächsten Zyklus zu berechnen sind, in einer verketteten Liste gespeichert werden. Dies kann nur erreicht werden, indem die Koordinaten aller Zellen, die sich im laufenden Zyklus ändern, mit allen ihren Nachbarn der verketteten Liste hinzugefügt werden. Nachdem das Ende des laufenden Zyklus erreicht ist, müssen die redundanten Einträge in der Liste wieder entfernt werden, um eine Reduktion der zu berechnenden Zellen zu ereichen. Denn eine Zelle kann sonst bis zu 27-mal in der Liste enthalten sein.

5. Lösungansatz mit Oktalbaum

Beschreibung

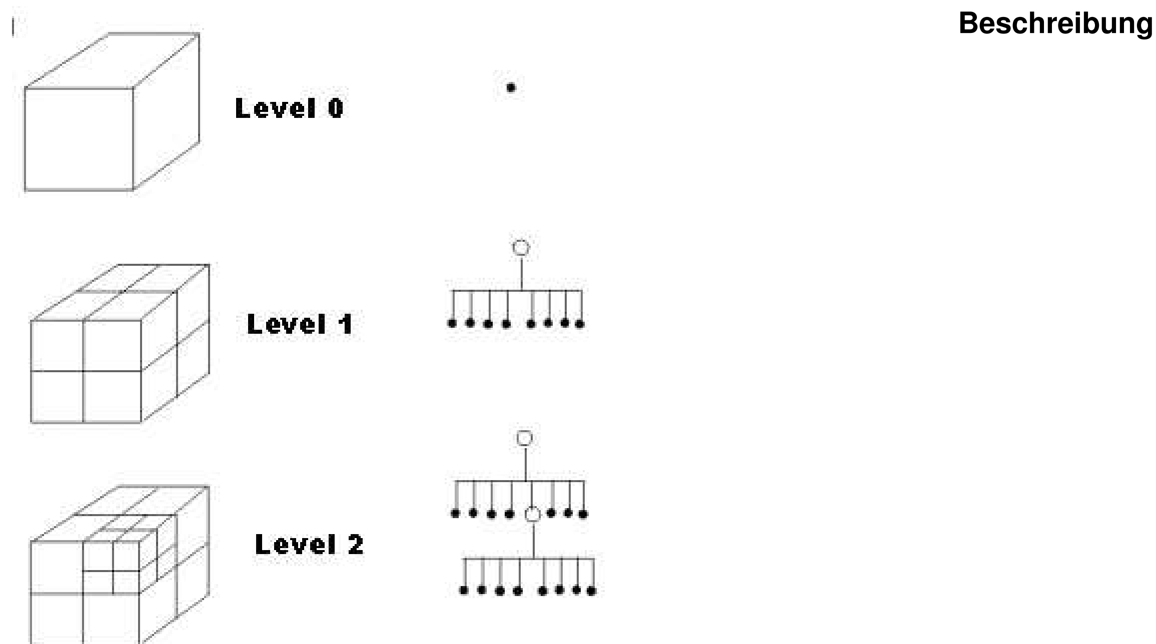


Abbildung 1: http://cfd.enginsoft.it/software/icemcfd/octree.html

Ein Oktalbaum ist eine hierarchische Struktur, die einen dreidimensionalen Raum beschreibt. Bei allen Würfeln des Raumes werden mit jedem Level, das hinzugefügt wird, alle Kanten halbiert. Aus jedem Würfel entstehen so acht kleine Würfel. Jeder Vaterknoten repräsentiert den gleichen Raum wie alle seine Kinder. Die Abbildung 1 illustriert den Aufbau eines Oktalbaums.
Alle Zellen werden in den Blättern des Oktalbaums gespeichert.

Speicherverbrauch mit Oktalbaum

Man muss zusätzlich zu den Zellen auch alle Knoten des Baumes speichern und jeder Knoten muss die Referenzen zu seinen Kindern besitzen, um im Baum vom Wurzelknoten zu den Blättern zu gelangen. Somit ist der Speicherbedarf einer Datenhaltung mit Oktalbaum sehr gross.

Zugriff auf die Nachbarn einer Zelle mit Oktalbaum

Das Auffinden der Nachbarn ist sehr aufwändig. Man kann nur über den Vater zu den Nachbarn gelangen. Es sei denn, jedes Blatt besitze noch die Referenzen seiner Nachbarn. Dies würde aber den Arbeitsspeicher auch zu schnell aufbrauchen

Auswahl der zu berechnenden Zellen (Zyklussteuerung) mit Oktalbaum

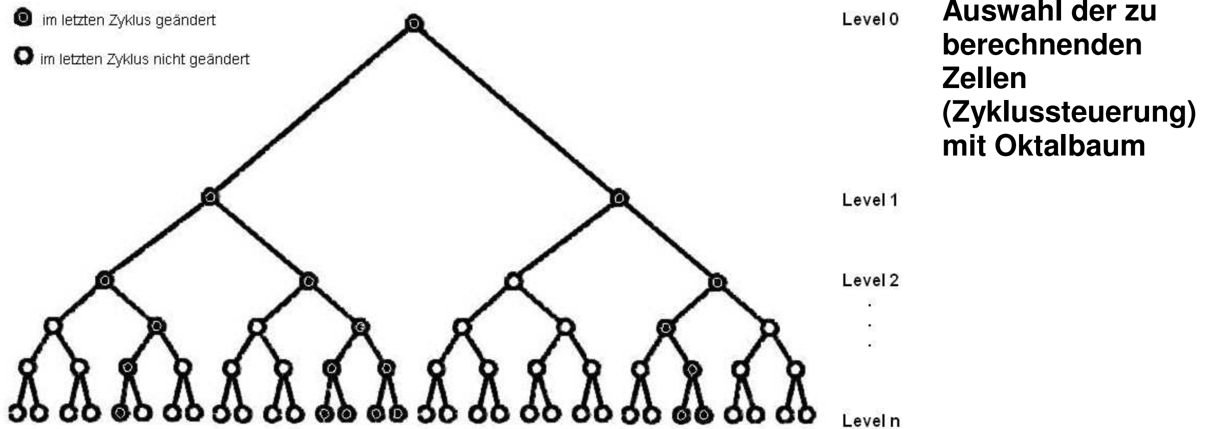


Abbildung 2: Vorteil einer hierarchischen Struktur für Zyklussteuerung

In einer solchen Baumstruktur können alle Zellen in den Blättern gespeichert werden. In den Knoten wird gespeichert, ob sich eines seiner Kinder im letzten Zyklus geändert hat (Abbildung 2). So können schon frühzeitig ganze Bereiche von der Neuberechnung ausgeschlossen werden.
Wenn zum Beispiel keines der Kinder eines Knoten im letzten Zyklus geändert hat und sich zugleich auch keines der Kinder seiner Nachbarkonten geändert hat, muss der Ast, in dem man sich befindet, in diesem Zyklus nicht berechnet werden.

6. Lösungansatz mit Table Tree

Beschreibung

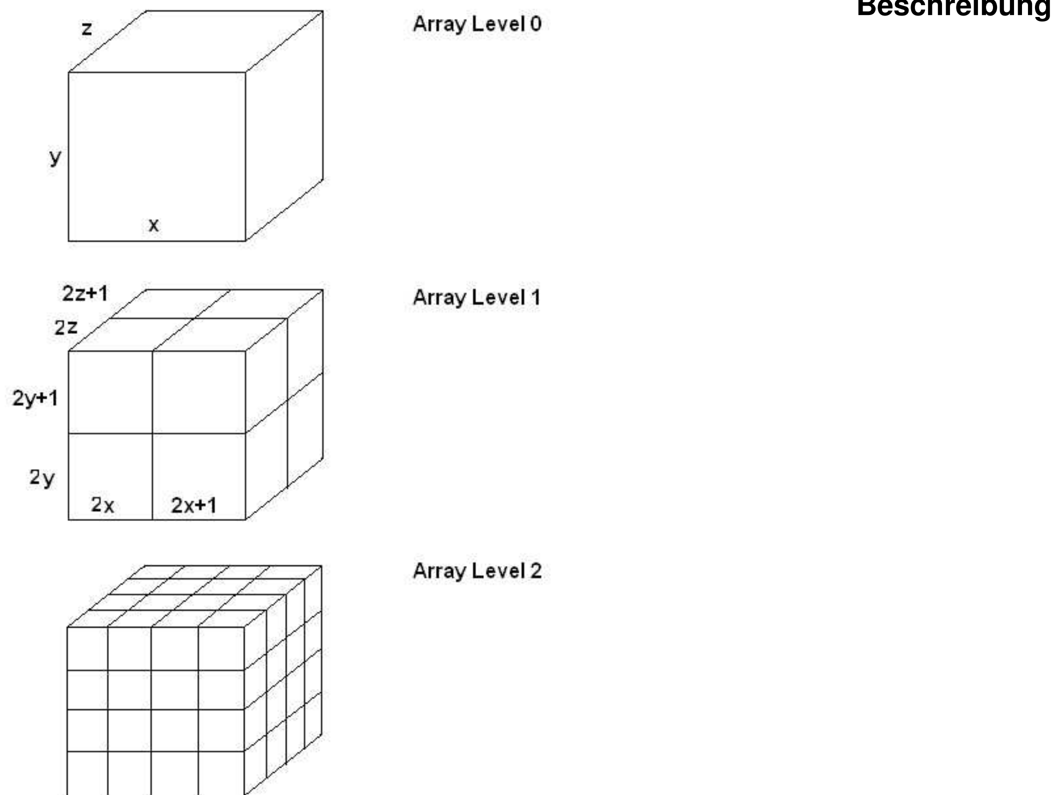


Abbildung 3: Datenstruktur eines ZellTrees

Nach reiflicher Überlegung kam die Idee einer Kreuzung eines Arrays mit einem Oktalbaum. Es entstand der Table Tree.
Ein Table Tree ist eine Art Oktalbaum. Die Baumstruktur wird aber nicht mit Knoten realisiert. Für jedes Level des Baumes wird ein dreidimensionales Array mit passender Grösse erstellt. Alle diese Levelarrays können auch in einem übergreifenden Array zusammengefasst werden. Es können mit einfachsten mathematischen Funktionen die Kinder eines „Knotens“ gefunden werden (siehe obige Abbildung).
Die Zellen werden wie bei einem Oktalbaum in den Blättern gespeichert.

Speicherbrauch mit Table Tree

Bei einem Table Tree müssen, wie bei einem Oktalbaum, zusätzlich zu den Zellen in den Blättern auch noch die Knoten gespeichert werden. Dafür entfallen aber die speicheraufwändigen Referenzen auf die Kinder. Sein Bedarf an Arbeitspeicher ist grösser als der eines Arrays, aber bei weitem nicht so gross wie der eines Oktalbaums.

Zugriff auf die Nachbarn einer Zelle mit Table Tree

Die Bestimmung der Nachbarzellen sowie der eigenen Position verläuft exakt gleich wie in einem simplen Array.

Auswahl der zu berechnenden Zellen (Zyklussteuerung) mit Table Tree

Die Auswahl der zu berechnenden Zellen erfolgt nach dem exakt gleichen Prinzip wie beim Oktalbaum.

7. Auswertung Lösungsansätze (Zellberechnung)

Theoretische Auswertung

Da die Datenhaltung mit einem dreidimensionalen Array sehr einfach und schnell zu implementieren ist, haben wir entschieden, einen entsprechenden Prototypen zu schreiben. Dieser Prototyp wird nicht konkurrenzfähig sein, er dient aber als Referenz für die Korrektheit sowie als Vergleich für die Performanz der anderen Prototypen. Er eignet sich sehr gut dafür, weil die Berechnungszeit für einen Zyklus nicht stark vom Zustand der Zellen abhängt.

Bei der Lösungsidee mit der verketteten Liste ist das grosse Problem, die redundanten Zellen wieder aus der Liste zu entfernen. Der Aufwand, der dafür getrieben werden muss, steigt mit zunehmender Grösse beträchtlich an (n log(n)). Es ist aber schwierig, die Grenzen dieses Algorithmus abzuschätzen. Wir haben uns deshalb entschlossen, auch einen Prototypen für diesen Algorithmus zu schreiben.

Für die Datenhaltung mittels eines Oktalbaums schreiben wir keinen Prototypen. Der komplizierte Zugriff auf die Nachbarn und der sehr grosse Speicherverbrauch sind zu grosse Nachteile.

Ein Table Tree vereinigt die gute Unterstützung der Zyklussteuerung des Oktalbaums mit einer einfachen Positionsbestimmung der Zellen und einem schnellen Zugriff auf die Nachbarzellen wie in einem Array. Wir können deshalb nicht darauf verzichten, einen Prototyp für diese Idee zu entwickeln.

Wir werden somit drei Prototypen entwickeln, um das beste Grundprinzip der Datenhaltung für die Rohdatenberechnung herauszufinden. Diese drei Prototypen sind:

- **Prototyp 3D-Array: Stupido**
- **Prototyp verkettete Liste: Sortido**
- **Prototyp Table Tree: TableTree**

Resultate der Prototypen

		SimpelCA				
Prototypen	**Grösse**	**Durchschnitliche Zeit pro Zyklus**	**Anzahl Zyklen**	**Gesamtzeit**	**Zeit Initialzykluss**	**gebrauchtes RAM**
Stupido	32*32*32 Zellen	0.121882353	17	2.072		985.24KB
	64*64*64 Zellen	0.968424242	33	31.958		7.542MB
	128*128*128 Zellen	7.909892308	65	514.143		60.024MB
Sortido	32*32*32 Zellen	24.56429412	17	417.593	0.003	1.951MB
TableTree mit Backup Array mit Changed im Leaf	32*32*32 Zellen	0.062470588	17	1.062	0.196	160KB
	64*64*64 Zellen	0.257969697	33	8.513	1.466	1.093MB
	128*128*128 Zellen	1.043815385	65	67.848	11.155	8.593MB

Wie zu erwarten war, ist der Table Tree die leistungsfähigste Datenhaltung. Seine einzige Schwäche ist die Dauer, wenn der ganze Lebensraum der Zellen berechnet werden muss. In diesem Fall braucht er etwa 1.5-mal so lange wie die Datenhaltung mit einem simplen 3D-Array.
Die Idee mit der verketteten Liste ist weit abgeschlagen. Die durchschnittliche Zeit von 24 Sekunden bei einem Lebensraum von 32 Zellen Kantenlänge ist nicht tragbar.
Aus diesen Messungen ist klar ersichtlich, dass der Table Tree die Datenhaltung ist, welche weiterzuverfolgen ist.

8. Theoretische Betrachtung des Table Tree

Da der Table Tree am besten von allen drei ersten Prototypen abgeschlossen hat, wird er noch theoretisch analysiert.

Theoretische Betrachtungen der Stärken und Schwächen

In der Theorie sollen die Stärken und Schwächen einer Datenhaltung mit dem Table Tree ausgelotet werden. Dazu werden Best- und Worst-Case-Betrachtungen angestellt.

„Worst Case“

Im schlechtesten Fall, wenn alle Zellen berechnet werden müssen, ist der zu betreibende Aufwand grösser, als wenn man einfach alle Zellen berechnet. Es würde zusätzlich

$$\sum_{j=0}^{n-1} 8^j$$

mal geprüft, ob sich ein Kind des Knotens im letzten Zyklus geändert hat, wobei 2^n die Kantenlänge des Zellwürfels ist. Das Verhältnis zwischen der Anzahl der Nachbarschaftsprüfungen des Table-Tree-Algorithmus und des Array-Algorithmus in diesem Fall beträgt:

$$\left(\sum_{j=0}^{n} 8^j\right) / 8^n <= 8/7,$$

wie gezeigt werden kann.[1]

Theoretischer „Best Case“

Der beste Fall wäre, wenn ein stabiler Zustand ereicht ist und keine der Zellen berechnet werden müsste. In diesem Fall würde nur der Knoten im Level 0 geprüft. In der Praxis kann der Zustand des Knotens im Level 0 nach jedem berechneten Zyklus zurückgegeben werden. Dieser Zustand dient als Abbruchkriterium.

Praktischer „Best Case

Im besten Fall, wenn genau eine Zelle ihren Zustand im letzten Zyklus verändert hat, werden

(Anzahl Zellen – 1)

Zellenberechnungen gespart. Aber das Finden der zu berechnenden Zelle benötigt

(n-1)*64*26+ 8*26

[1] Geometrische Reihe

Prüfungen, ob sich ein Kind des Knotens im letzten Zyklus geändert hat. Das Verhältnis zwischen der Anzahl der Nachbarschaftsprüfungen des Table-Tree-Algorithmus und des Array-Algorithmus in diesem Fall beträgt:

$$\mathbf{(n*2^6+8)/8^n.}$$

Für n=6 ergibt dies 1/799 und 1/5349 für n=7.

Durchschnittlicher Performancegewinn

Wenn man in Betracht zieht, dass die Berechnung einer Zelle bei komplexeren ZA mehr Zeit benötigt als das Prüfen eines Knotens, kann aufgrund des Vergleichs des Worst- mit dem Best-Case ein erheblicher durchschnittlicher Performancegewinn gegenüber dem Berechnen aller Zellen erwartet werden, insbesondere, wenn sich nicht allzu viele Zellen geändert haben.

9. Variante des Table Tree

Grössere Anzahl der Kinder

Es ist auch vorstellbar, dass ein Würfel mit jedem zusätzlichen Level in mehr als acht kleine Würfel aufgeteilt wird. Es könnten auch 3^3, 4^3 oder mehr anstelle von 2^3, wie in einem Oktaltabletree, sein. Es ist schwer zu bestimmen, welche Unterteilung am sinnvollsten ist. Folgende Punkte sprechen unserer Meinung nach für den Oktaltabletree:
Bei einem Oktaltabletree können am meisten Zellen sehr hoch im Baum ausgeschlossen werden.
Setzt man einen ZA, der alle Nachbarn benötig, um einen Zustand zu berechnen, und einen unendlichen Raum voraus, so müssen in einem Oktaltabletree alle Kinder eines Knotens, die den Wert „geändert" aufweisen, weiter betrachtet werden. In einem Baum mit 3^3=27 Kindern müssten in 8 Fällen (Ecken) eigentlich nur 8 der Kinder betrachtet werden und in 12 Fällen (Kanten) eigentlich nur 12 der Kinder. Nur in einem Fall (Zentrum) müssten alle 27 Kinder auch wirklich betrachtet werden, es würden jedoch immer alle betrachtet.
Darum ist der Oktaltabletree sehr wahrscheinlich die bessere Lösung als Bäume mit mehr Ästen.

10. Optimierung eines Table Tree

Reduktion der Nachbarschafts-abfragen bei den Knoten

Nicht jeder ZA betrachtet für die Berechnung alle Nachbarn. Für solche ZA müssen in den Knoten auch nicht alle Nachbarn geprüft werden. Dadurch ist die Selektion besser, als wenn trotzdem alle Nachbarn betrachten werden.
Die Nachbarschaftsbeziehungen, die in einem ZA verwendet werden, können aber nicht eins zu eins auf die Knoten übertragen werden. In den Knoten muss man berücksichtigen, dass die Prüfung in einem Level n stattfindet. Das Ziel dieser Prüfung ist es, die Kinder im Level (n+1) zu bestimmen, die berechnet werden müssen. Wird im ZA der Nachbar einer Ecke betrachtet, müssen in den Knoten der Nachbar dieser Ecke, die Nachbarn aller drei angrenzenden Kanten sowie allen drei angrenzenden Flächen betrachtet werden. Wird im ZA der Nachbar einer Kante betrachtet, müssen in den Knoten der Nachbar dieser Kante und die Nachbarn an beiden angrenzenden Flächen betrachtet werden.

Kein „changed“ in den Blättern

In den Blättern des Baumes muss nicht unbedingt gespeichert und geprüft werden, ob das Blatt im letzten Zyklus geändert wurde. Es reicht, wenn dieser Wert den Baum hoch gegeben wird. Eine solche Prüfung würde sehr wahrscheinlich nur zu einer Verlangsamung führen und viel Speicher verbrauchen.

Maskieren der „changed“

In einem Knoten wird nicht nur gespeichert, ob eines seiner Kinder geändert hat, sondern darüber hinaus auch, welches geändert hat. Dies kann durch Bit-Maskierung in einem Byte einfach realisiert werden. Dadurch wird eine Reduktion der weiter zu verfolgenden Äste ermöglicht, da Kinder der Nachbarschafts-zelle, welche auf der zu dieser Zelle abgewandten Seite liegen, zu keinem der Kinder benachbart sind und somit keine Kinderberechnungsanforderung implizieren.

Kinder basierte Nachbarschafts-Prüfung

Wenn keines der Kinder eines Knotens im Baum den Wert „geändert“ aufweist, müssen auch noch die Nachbarn geprüft werden. Für diese Nachbarschaftsprüfung gibt es zwei unterschiedliche Methoden:
Bei der ersten Methode werden die Nachbarn des Knotens geprüft, um so zu bestimmen, welche Kinderäste weiterverfolgt werden müssen.
Bei der zweiten Methode können aber auch, statt die Nachbarn eines Knotens, direkt die Nachbarn eines Kindes geprüft werden, um zu entscheiden, ob dieser Kinderast weiterverfolgt werden muss.
Durch die zweite Methode kann man exakt bestimmen welche Unteräste weiter zu verfolgen sind. Bei der ersten Art werden in vielen Fällen zu viele Kinderäste weiterverfolgt. Die zweite Art

benötigt aber bis zu achtmal mehr Nachbarschaftsabfragen als die erste.
Welches die bessere Lösung ist, kann nur durch Implementation von Prototypen herausgefunden werden.

Updaterate

Ein zusätzlicher Performancegewinn kann erzielt werden, wenn es Zustände von Zellen gibt, die stabiler sind als andere. Eine solche stabile Zelle bräuchte nicht jedes Mal berechnet zu werden, wenn eine unstabile Zelle berechnet werden muss. Um dies zu realisieren, wird neben dem Zustand zusätzlich auch ein Faktor gesetzt, der bestimmt, nach wie vielen Zyklen die Zelle erst wieder neu berechnet werden muss. Jeder Knoten speichert das Minimum der Updateraten aller Kinder. Das Auslassen einer Berechnung eines Astes aufgrund der Updaterate bedingt ein Gedächtnis (der Länge der maximal möglichen Updaterate) der „Changed“ Bits.

Alle diese Optimierungsmöglichkeiten werden schrittweise implementiert. Nach jedem Schritt wird geprüft, ob auch wirklich eine Performancesteigerung erzielt worden ist.

11. Prototypenauswertung verschiedener Table Tree

Allgemein

Alle Prototypen sind mit zwei unterschiedlichen ZA verglichen worden. Der SimpelCA betrachtet alle Nachbarn, der WurmCa nur einen bestimmten Nachbarn.

Vergleich ohne und mit „changed“ in den Blättern

Der Prototyp „TableTree mit Backup Array mit ‚changed' im Leaf“ speichert in den Zellen, ob diese im letzten Zyklus geändert worden sind. Er prüft diesen Wert auch, bevor er die Zelle neu berechnet.
Der Prototyp „TableTree mit Backup Array ohne ‚changed' im Leaf“ speichert und prüft in einer Zelle nicht, ob diese im letzten Zyklus geändert hat. Wird die unterste Ebene der Knoten erreicht, werden von dort aus immer alle ihrer Kinder durchgerechnet.

		SimpelCA				
Prototypen	**Grösse**	**Durchschnitliche Zeit pro Zyklus**	**Anzahl Zyklen**	**Gesamtzeit**	**Zeit InItialzykluss**	**gebrauchtes RAM**
TableTree mit Backup Array ohne Changed im Leaf	32*32*32 Zellen	0.040823529	17	0.694	125	317.8KB
	64*64*64 Zellen	0.167212121	33	5.518	0.893	611.9KB
	128*128*128 Zellen	0.682246154	65	44.346	6.453	4.597MB
TableTree mit Backup Array mit Changed im Leaf	32*32*32 Zellen	0.062470588	17	1.062	0.196	160KB
	64*64*64 Zellen	0.257969697	33	8.513	1.466	1.093MB
	128*128*128 Zellen	1.043815385	65	67.848	11.155	8.593MB

		WurmCa				
Prototypen	**Grösse**	**Durchschnitliche Zeit pro Zyklus**	**Anzahl Zyklen**	**Gesamtzeit**	**Zeit InItialzykluss**	**gebrauchtes RAM**
TableTree mit Backup Array ohne Changed im Leaf	32*32*32 Zellen	0.00475	40	0.19	0.113	211.2KB
	64*64*64 Zellen	24.2	40	968	0.84	607KB
	128*128*128 Zellen	0.168775	40	6.751	6.244	4.591MB
TableTree mit Backup Array mit Changed im Leaf	32*32*32 Zellen	0.006025	40	0.241	0.16	159KB
	64*64*64 Zellen	0.03475	40	1.39	1.214	1.092MB
	128*128*128 Zellen	0.262875	40	10.515	9.698	8.953MB

Resultat Vergleich ohne und mit „changed“ in den Blättern

Die Performance des Prototyps ohne „changed“ ist bei beiden ZA erheblich besser. Er braucht ausserdem viel weniger Speicher. Er ist klar der Sieger dieses Tests.

Reduktion der Nachbarschafts-abfragen bei den Knoten

Der Prototyp „TableTree mit Backup Array ohne ‚changed' im leaf mit delegates" betrachtet auch in den Knoten nur die relevanten Nachbarzellen. Er hat zusätzlich noch die ZA als separate Klassen, was zusätzlich ein klein wenig Performance verbraucht.

		SimpelCA			
Prototypen	**Grösse**	**Durchschnitliche Zeit pro Zyklus**	**Anzahl Zyklen**	**Gesamtzeit**	**Zeit InItialzykluss**
TableTree mit Backup Array ohne Changed im Leaf	32*32*32 Zellen	0.040823529	17	0.694	0.125
	64*64*64 Zellen	0.167212121	33	5.518	0.893
	128*128*128 Zellen	0.682246154	65	44.346	6.453
TableTree mit Backup Array ohne Changed im Leaf mit delegates	32*32*32 Zellen	0.040941176	17	0.696	0.106
	64*64*64 Zellen	0.175939394	33	5.806	0.852
	128*128*128 Zellen	0.735384615	65	47.8	6.683

		WurmCa			
Prototypen	**Grösse**	**Durchschnitliche Zeit pro Zyklus**	**Anzahl Zyklen**	**Gesamtzeit**	**Zeit InItialzykluss**
TableTree mit Backup Array ohne Changed im Leaf	32*32*32 Zellen	0.004175	40	0.167	0.113
	64*64*64 Zellen	0.022875	40	0.915	0.84
	128*128*128 Zellen	0.1684	40	6.736	6.244
TableTree mit Backup Array ohne Changed im Leaf mit delegates	32*32*32 Zellen	0.0028	40	0.112	0.095
	64*64*64 Zellen	0.020475	40	0.819	0.761
	128*128*128 Zellen	0.1581	40	6.324	5.873

Resultat Vergleich ohne und mit „Delegates"

Der Prototyp mit „Delegates" ist im Verhältnis nur unwesentlich langsamer, wenn alle Nachbarn einer Zelle verwendet werden. Werden aber nicht alle Nachbarzellen benötigt, ist ein Zyklus bis zu 10-mal schneller.

Maskieren der „changed“

Der Prototyp „TableTree mit Backup Array ohne ‚changed’ im leaf mit delegates und masking“ speichert in allen Knoten, welche Kinder ein „changed“ aufweisen. Er wählt auch dementsprechend, aus welche Kinder er weiterverfolgt.

		SimpelCA			
Prototypen	Grösse	Durchschnitliche Zeit pro Zyklus	Anzahl Zyklen	Gesamtzeit	Zeit InItialzykluss
TableTree mit Backup Array ohne Changed im Leaf mit delegates und Masking	32*32*32 Zellen	0.031941176	17	0.543	0.106
	64*64*64 Zellen	0.135757576	33	4.48	0.818
	128*128*128 Zellen	0.563169231	65	36.606	6.566
TableTree mit Backup Array ohne Changed im Leaf mit delegates	32*32*32 Zellen	0.040941176	17	0.696	0.106
	64*64*64 Zellen	0.175939394	33	5.806	0.852
	128*128*128 Zellen	0.735384615	65	47.8	6.683

		WurmCa			
Prototypen	Grösse	Durchschnitliche Zeit pro Zyklus	Anzahl Zyklen	Gesamtzeit	Zeit InItialzykluss
TableTree mit Backup Array ohne Changed im Leaf mit delegates und Masking	32*32*32 Zellen	0.002925	40	0.117	0.103
	64*64*64 Zellen	0.020975	40	0.839	0.772
	128*128*128 Zellen	0.159025	40	6.361	5.923
TableTree mit Backup Array ohne Changed im Leaf mit delegates	32*32*32 Zellen	0.0028	40	0.112	0.095
	64*64*64 Zellen	0.020475	40	0.819	0.761
	128*128*128 Zellen	0.1581	40	6.324	5.873

Resultat von Maskierung

Die Messungen zeigen deutlich, dass das Maskieren eine deutliche Verbesserung darstellt.

Kinderbasierte Nachbarschafts-Prüfung

Die kinderbasierte Nachbarschaftsprüfung wurde nur für einen ZA entwickelt, der nur die Nachbarn an den angrenzenden Flächen der Zelle betrachtet, und jene an den Ecken und Kanten ausser Betracht lässt.

		CaFaces			
Prototypen	Grösse	Durchschnitliche Zeit pro Zyklus	Anzahl Zyklen	Gesamtzeit	Zeit InItialzykluss
TableTree mit Backup Array ohne Changed im Leaf mit delegates und Masking	32*32*32 Zellen	0.015510204	49	0.76	0.097
	64*64*64 Zellen	0.064463918	97	6.253	0.751
	128*128*128 Zellen	0.261274611	193	50.426	6.061
TableTree mit Backup Array ohne Changed im Leaf mit Delegates und Masking und Childbased	32*32*32 Zellen	0.029275	40	1.171	0.129
	64*64*64 Zellen	0.096175258	97	9.329	0.966
	128*128*128 Zellen	0.385746114	193	74.449	7.701

Resultat von kinder-basierte Nachbarschafts-prüfung

Es zeigt sich aber, dass der zusätzliche Aufwand den Gewinn deutlich übertrifft.

Fazit

In der Applikation wird die Datenhaltung auf dem Prototyp „TableTree mit Backup Array ohne ‚changed’ im leaf mit delegates und masking“ beruhen.

Resultat von Updaterate

Die Implementation der Updaterate wurde nicht eingehend gestestet, da diese Verbesserung zu diesem Zeitpunkt als nicht prioritär angesehen wurde. Es hat sich jedoch gezeigt, dass der zusätzliche Berechnungsaufwand minime Performanceeinbussen zur Folge hat.

12. Datenaufbereitung

Problematik

Die grafische Datenmenge muss aufgrund der Hardwarelimiten massiv reduziert werden, um eine flüssige Ausgabe der Daten auf dem Bildschirm zu ermöglichen.

Lösung

Die hier erklärten Algorithmen beziehen sich auf die 2. Optimierungsmöglichkeit der Datenaufbereitung, beschrieben im Dokument „Analyse – 06042005.doc“
Konkret bedeutet das, dass man so etwas wie ein „Tuch“ über alle zusammenhängenden Bereiche von Zellen des gleichen Zustandes legt und nur die generierte Oberfläche visualisiert. Dieses Verfahren ermöglicht es, die Rohdatenmenge beträchtlich zu reduzieren.

Oberfläche innen

Hier ersichtlich ist eine Ansicht, die man sieht, wenn man sich im Innern eines Zellhaufens befindet. Man schaut an die Innenflächen eines Volumens, das aus Zellen mit den gleichen Zuständen besteht. Die Farbe der Flächen bestimmt sich durch die angrenzenden Zellen, die einen anderen Zustand als jene innerhalb des Volumens besitzen. Im untenstehenden Bild sind Zellen ausserhalb des Volumens mit dem Zustand 0 vorhanden, was bedeutet, dass sie durchsichtig sind, (bzw. Luft,) die mit der Standartfarbe Himmelblau gezeichnet werden.

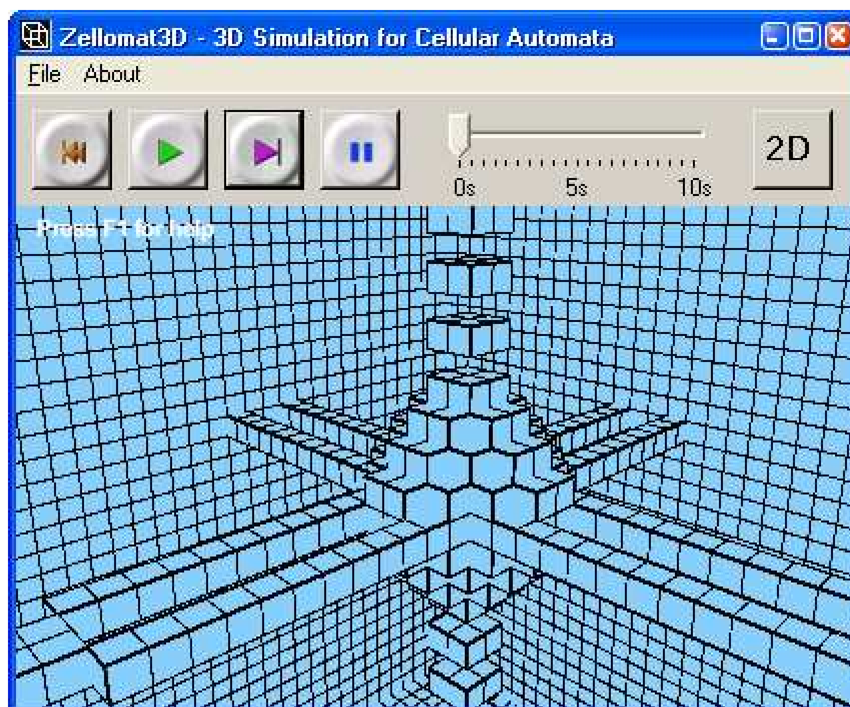


Abbildung 4: Blick innerhalb eines Volumens

Oberfläche aussen

Im nachfolgenden Bild ist klar ersichtlich, wie sich das Volumen gleichartiger Zellen für den Benutzer präsentiert. Hier befindet sich der Betrachter ausserhalb eines Zellhaufens und sieht somit die Farben, die den Zuständen innerhalb des Volumens entsprechen.

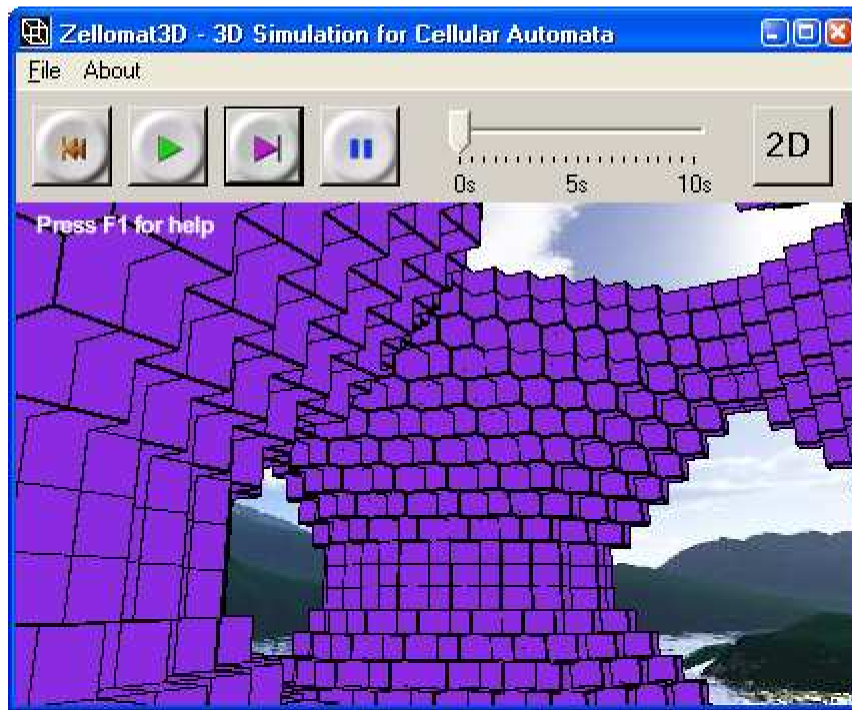


Abbildung 5: Blick von ausserhalb eines Volumens

Vorgehen

Insgesamt wurden zwei unterschiedliche Prototypen zu dieser Problemlösung realisiert und ausgetestet. Nachfolgend werden diese zwei Algorithmen vorgestellt und ihre Funktionsweise erklärt.

13. Generierung der Oberfläche durch Iterierung

Hinweis

Der entwickelte Prototyp mit diesem Algorithmus findet sich unter Prototypen/Zusammenspiel/ 22. (Prototype DataPreparation) auf der Projekt-CD.

Erklärung

Alle Zellen in der Datenhaltung, einem 3D-Array, besitzen einen definierten Zustand. Folglich kann die Oberfläche aus der Datenhaltung eindeutig bestimmt und generiert werden. Für die Oberfläche benötigt man für jede Aussenfläche die Position der vier Eckpunkte in Weltkoordinaten. Jede Zelle hat sechs quadratische Seitenflächen, nämlich eine Vorder- und eine Hinterseite in jeder der drei Raumrichtungen.

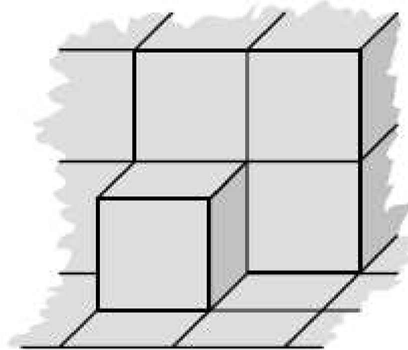


Abbildung 6: Struktur der Zellenanordnung

Da bei unserer Darstellung die Seitenflächen immer gleich gross sind, kann die Position aller acht Eckpunkte einer Zelle aus dem Index seines Elements im Array und der gewünschten Kantenlänge einer Zelle berechnet werden. Hinsichtlich der Oberfläche besteht die Aufgabe jetzt darin, alle Aussenflächen der Zellen zu bestimmen. Dazu werden in drei Schritten jeweils alle Elemente der Datenhaltung geprüft. Bei jedem dieser Schritte werden nur diejenigen Aussenflächen gesucht, die sich aus den Vorder- und Hinterseiten der Zellen in einer der drei Raumrichtungen X, Y und Z ergeben und folglich alle parallel sind. Deshalb werden nacheinander alle Reihen von Zellen, die in der entsprechenden Richtung hintereinander liegen, geprüft. Abbildung 7 zeigt das Beispiel einer solchen Reihe in X-Richtung.

Für eine Reihe wird nacheinander von jedem Element mit Hilfe des Indexes der Wert ausgelesen. Wenn eine Zelle mit Zustand 7 auf eine mit ebenfalls Zustand 7 folgt, werden keine Seitenflächen generiert.
Wenn jetzt aber Unterschiede der Zustände zweier Nachbarzellen gefunden werden, generiert der Algorithmus jeweils die beiden Flächen, die sich zwischen den unterschiedlichen Zellen befinden. Und zwar so, dass die Farbe der jeweils anderen Zelle auf die Innenfläche der einen Zelle gezeichnet wird.

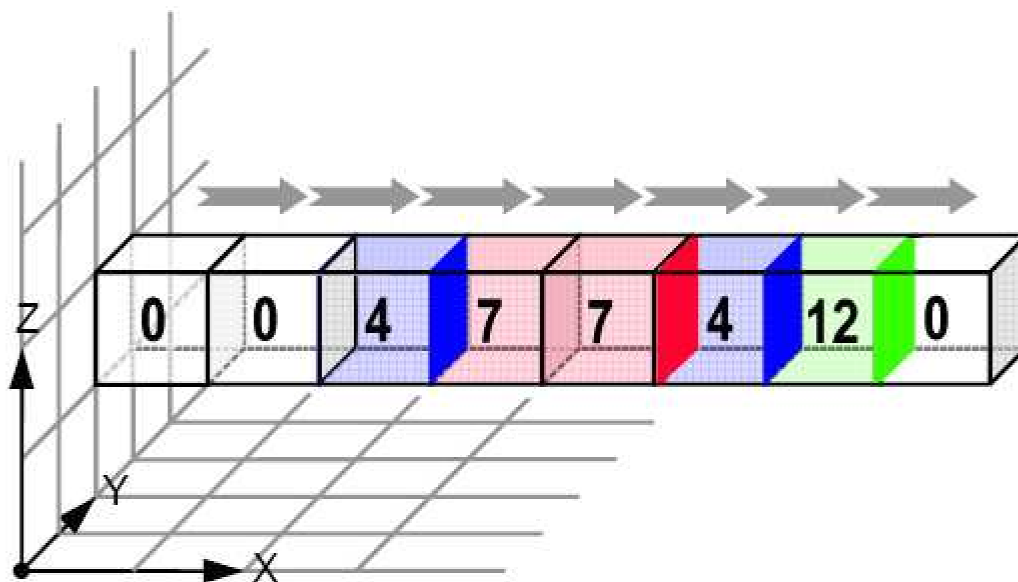


Abbildung 7: Durchsuchen der Datenhaltung nach Seitenflächen

Jede gefundene Seitenfläche wird über die Koordinaten der vier Eckpunkte zu der Oberfläche hinzugefügt und mit zwei Dreiecken modelliert. Durch die Reihenfolge der Eckpunkte wird die Orientierung der Fläche nach aussen festgelegt, was für das Rendering von Bedeutung ist. Auf diese Weise entsteht eine zusammenhängende Oberfläche.

Fazit

Technisch funktioniert der entwickelte Algorithmus einwandfrei, nur die Rechenzeit, um das 3D-Array mit den Zellen und deren Zuständen in jeder Achse einmal durchzuiterieren, dauert eine dritte Potenz länger als die eigentliche Rohdatenberechnung des Zyklus. Dies ist inakzeptabel und darum wurde ein neuer Algorithmus entwickelt.

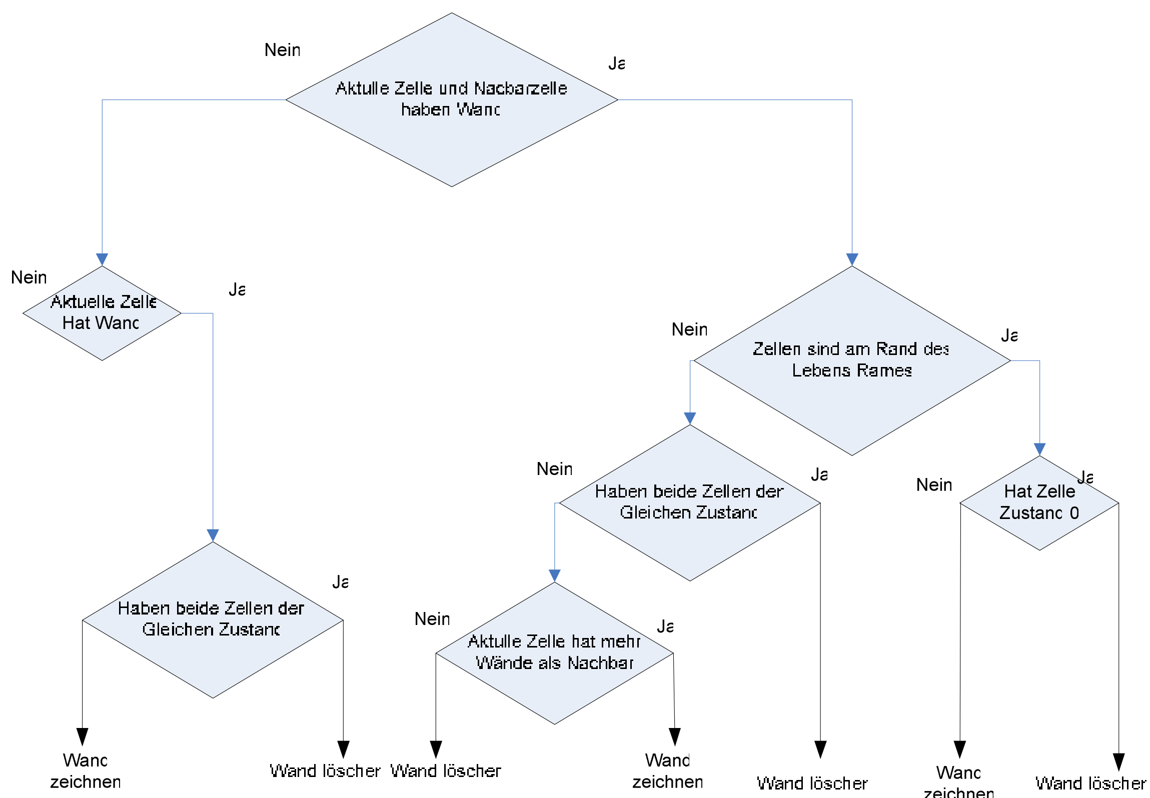
14. Generierung der Oberfläche im Tabletree

Hinweis

Dieser Algorithmus wurde in der Finalen Version der Applikation Zellomat3D verwendet.

Erklärung

Eigentlich sind die Informationen über „wer welche Nachbarn hat" schon einmal erzeugt worden. Anstelle einer Neuerzeugung dieser Informationen, wie im vorhergehenden Algorithmus, könnten diese Informationen für die Datenaufbereitung weitergenutzt werden.
Es werden bei allen Zellen, die sich durch die Berechnung verändert haben, alle Wände gesetzt. Auch dieser Wert (Border) wird im Baum nach oben propagiert. Ist die Berechnungsphase der Zustände abgeschlossen, werden allen Borderwerten folgend die Zellen mit Wänden gesucht. Für jede Zelle, die eine Wand besetzt, wird eine Bereinigungsmethode aufgerufen. Die Bereinigung der Wände erfolgt wie die folgende Grafik erläutert.

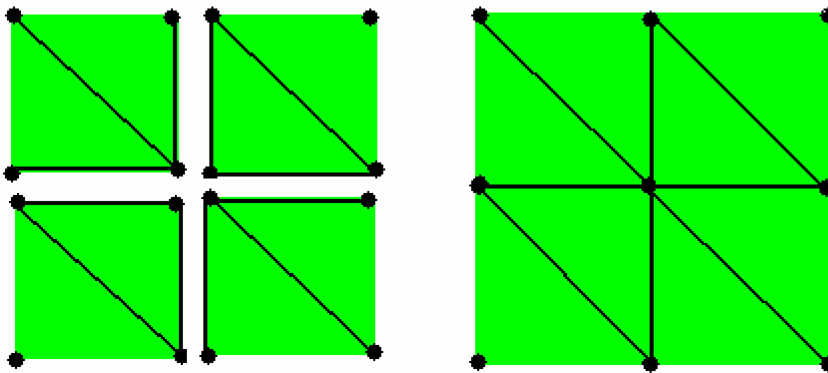


Die Wände, die gezeichnet werden müssen, werden der Datenaufbereitung übergeben.

15. Framegenerierung

Problematik

In unserer Applikation müssen für jede Zelle sechs Seitenflächen dargestellt werden, welche je durch vier Punkte bestimmt sind. Diese grosse Anzahl von Punkten übersteigt sehr schnell die Leistungsfähigkeit des Grafikprozessors, da er bei Veränderung der Betrachtungsposition jeden einzelnen Punkt geometrisch transformieren muss.
Es reicht nicht, eine Oberfläche zu erzeugen. Darüber hinaus muss darauf geachtet werden, zu ihrer Darstellung möglichst wenige Punkte zu verwenden. Das heisst konkret, dass die Eckpunkte der Flächen mehrfach genutzt werden sollten.



Lösung

Die Grafikkarte benötigt eine Liste aller darzustellenden Punkte. Da die Positionsinformation in Listenformat verloren ist, benötigt das Auffinden eines Punktes einen Sortieralgorithmus, welcher für die zu erwartende Punktanzahl aus Performancegründen nicht praktikabel ist. Deshalb werden die Referenzen der Punkte in einem Array zwischengespeichert, welches die geometrische Position und die Farbe im den 4ten Indexwerten kodiert. Dadurch kann die Existenz eines Farbpunktes durch einfachen Arrayzugriff eruiert werden.
Durch geschickte Wahl der möglichen Farbpunktposition wird eine optimale Ausnutzung gewährleistet.