

# **Zellularautomaten implementiert auf programmierbarer Grafikhardware**

Johannes Singler  
Lehrstuhl Informatik für Naturwissenschaftler und Ingenieure  
Universität Karlsruhe(TH)

10. September 2003

Zellularautomaten finden in den Naturwissenschaften Verwendung als Modell für verschiedene Prozesse. Man ist an einer möglichst performanten Implementierung der Simulation interessiert. Hier wird versucht zu zeigen, wie man mit Hilfe des Einsatzes moderner, programmierbarer Grafikhardware die Berechnung dieser Simulation beschleunigen kann.

# 1 Einführung

Zellularautomaten finden in den Naturwissenschaften Verwendung als Modell für verschiedene Prozesse. Man ist an einer möglichst performanten Implementierung der Simulation interessiert. Hier wird versucht zu zeigen, wie man mit Hilfe des Einsatzes moderner, programmierbarer Grafikhardware die Berechnung dieser Simulation beschleunigen kann.

In den letzten Jahren war die Entwicklungs-Geschwindigkeit bei Computerchips zur Berechnung von 3D-Grafik weitaus höher als die bei Standard-Prozessoren. Zuerst nur auf teurer Spezialhardware zu finden, die von CAD-Profis genutzt wurden, entwickelte sich zunehmend ein Konsumenten-Markt durch die Entwicklung von Spielen mit immer hochwertigerer Grafik. Effekt des Massenmarktes: Sehr leistungsfähige Grafikkarten wurden immer billiger, trotzdem verdienten die Hersteller mehr, womit auch gesteigerte Investitionen in die Entwicklung möglich waren. Dadurch potenzierte sich noch einmal die Entwicklungsgeschwindigkeit. Heutzutage sind Grafikkarten zu akzeptablen Preisen erhältlich, die Standard-Prozessoren in Bezug auf Komplexität und Transistorzahl weit in den Schatten stellen.

## 1.1 Grundbegriffe aus der 3D-Grafik-Programmierung

Ziel der 3D-Grafik-Programmierung ist es eigentlich, eine im 3-dimensionalen Raum gegebene Szene auf eine 2D-Oberfläche zu projizieren. Dabei werden Objekte im Raum üblicherweise durch ein Polygonnetz beschrieben. Dazu kommen Lichtquellen verschiedenen Typs (z. B. Umgebungslicht, punktförmiges Licht). Die Hardware zerlegt die Polygone vor der Verarbeitung in Dreiecke und zeichnet diese dann. Ein Dreieck kann entweder einfarbig sein, mit einem Farbverlauf zwischen den Ecken eingefärbt sein oder mit ein oder mehreren Texturen belegt werden. Texturen sind Bitmap-Bilder, die die „Maserung“ der Oberfläche definieren. Mehrere Texturen sind nur sinnvoll, wenn sie miteinander gemischt werden, z. B. eine Textur, in der vorberechnete Beleuchtungsinformation gespeichert ist (sog. Lightmap) und die Maserung selbst (die dann sinnvollerweise an mehreren Stellen verwendet wird, um Speicherplatz zu sparen).

Einen Eckpunkt eines jeden Polygons nennt man im Englischen auch Vertex. Dieser kann auch mehrfach verwendet werden, was sich natürlich gerade bei lückenlosen Polygonnetzen anbietet. Zu jedem Vertex existiert eine Datenstruktur, deren Aufbau je nach Verwendungszweck variabel ist (unter DirectX nennt sich diese Variabilität *Flexible Vertex Format*, *FVF*). Dieses Format muss vor Verwendung irgendeines Vertex deklariert und aktiviert werden. In jedem Fall enthält sie die 3D-Koordinaten des Punktes, denn diese sind zur Projektion auf die zweidimensionale Oberfläche unbedingt nötig. Weiterhin kann die Datenstruktur z. B. auch einen Normalenvektor beinhalten. Dieser steht senkrecht auf der idealisierten darzustellenden Oberfläche (nicht unbedingt senkrecht auf dem wirklichen Polygon) und wird zur Bestimmung der Beleuchtungsintensität verwendet. Einem Eckpunkt kann auch direkt eine Farbe zugewiesen werden, zwischen den Eckpunkten kann diese dann nach verschiedenen wählbaren Vorschriften interpoliert werden. Dies ist aber nicht mehr sinnvoll, wenn man, wie zuvor schon erwähnt, *Texturen* auf die Polygone projiziert. Dazu enthält jeder Vertex pro Textur ein sog. Texturkoordinaten-Paar. Dies gibt den Punkt innerhalb der Textur an, der auf diesen Vertex projiziert werden soll. Da sämtliche 3D-Daten vor dem eigentlichen Zeichnen in Dreiecke zerlegt werden, gibt es hier keine Geometrie-Probleme, denn ein Dreieck (innerhalb der Textur) kann immer auf ein Dreieck (im dreidimensionalen Raum) affin abgebildet werden. Wenn man Polygone mit mehr als drei Ecken definiert, muss die Applikation selbst sicher stellen, dass diese eben sind und die Textur-Koordinaten mit den Raumkoordinaten harmonisieren, so dass eine

affine Transformation möglich bleibt. Im anderen Fall kommt es auf Grund der nicht vorhersehbaren Aufteilung in Dreiecke zu willkürlichen Grafikfehlern.

Hier bei der Simulation von Zellularautomaten werden von den zahlreichen Features der verwendeten 3D-API jedoch nur relativ wenige genutzt. Die automatische Beleuchtungsberechnung aus Normalen in Verbindung mit verschiedenen Lichtquellen ist z. B. nicht sinnvoll. Anti-Aliasing wird auch nicht verwendet, könnte sich aber noch als nützlich erweisen. Die dritte Dimension wird auch nicht wirklich ausgenutzt, es gibt keine 3D-Objekte mit Rauminhalt, sondern nur plane Flächen. Auch die direkte Farbgebung für Polygone über die Vertices wird nicht benutzt, wichtig sind ja die Texturen, das notwendige Polygon dient nur als „Träger“.

### 1.1.1 Begriffserläuterungen

**Pixel** (von engl. picture element) Bildpunkt, kleinster Teil eines Pixelbilds, die Farbe ist normalerweise in den drei Komponenten Rot, Grün und Blau kodiert.

**Texel** (von engl. texture pixel) Bildpunkt einer Textur. Im Unterschied zum Pixel ist meistens auch noch eine Komponente für Transparenz (Alpha) enthalten. Diese eignet sich jedoch nicht zum Abspeichern von Daten, da sie nur indirekt den Rendering-Prozess beeinflusst: Wenn die Farbe des Pixels und der Alpha-Wert feststehen, wird die neue Farbe mit der schon im Speicher vorhandenen mittels des Alpha-Werts überblendet und dieser danach verworfen.

**Textur** (engl. texture) Textelbild, das normalerweise auf Polygone projiziert werden kann; zur Festlegung des Ausschnitts der Projektion dienen Texturkoordinaten.

**Vertex** Datenstruktur eines Polygon-Eckpunkts, enthält Informationen z. B. zur Position im Raum, Färbung (nicht ausschlaggebend bei Verwendung von Texturen), Beleuchtung (Umgebungslicht [ambient], diffuse Reflexion [diffuse] und Glanzlichter [specular]), Koordinaten für eine oder mehrere Texturen.

**Textur-Koordinate** Für jede Textur, die mit einem bestimmten Polygon verwendet werden soll, muss jeder zugehörige Vertex ein Koordinatenpaar des Texels enthalten, der auf ihn projiziert wird. Beim Zeichnen des Polygons wird der durch die Textur-Koordinaten definierte Textur-Ausschnitt entsprechend verzerrt. Dazu werden sämtliche vorhandenen Textur-Koordinaten entsprechend der Position des Pixels interpoliert.

**Texture-Filterung** Kann einem Pixel nicht exakt ein Texel zugeordnet werden, so bietet die Hardware an, die Farbe der nächstliegenden Pixel zu interpolieren, um eine bessere Bildqualität zu erhalten. Dies kann zur Einbeziehung der Nachbarschaft bei totalistischen Zellularautomaten ausgenutzt werden. Wird die Textur überhaupt nicht gefiltert, so spricht man von Point Sampling.

**Rendern** (von engl. to render) Abbilden der dreidimensionalen Szene auf eine Fläche inkl. sämtlicher dazu notwendiger Berechnungen wie z. B. Koordinatentransformation,

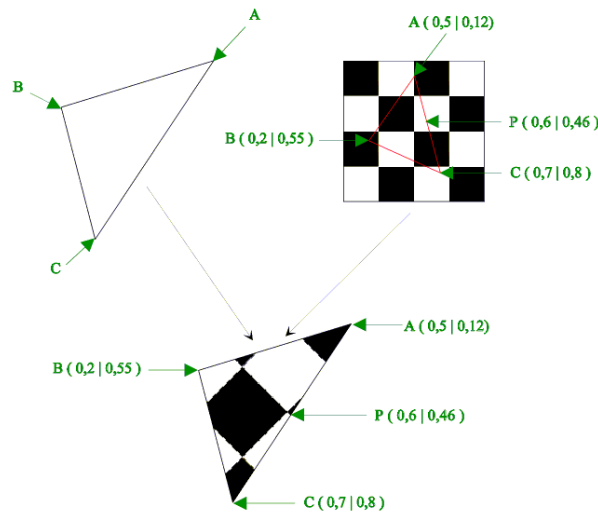


Abbildung 1: Textur-Projektion. Links oben ist das Dreieck mit den Eckpunkten; rechts die zugeordneten Texturkoordinaten in die Textur eingezeichnet; unten das Ergebnis. Punkt P liegt beispielsweise genau in der Mitte von A und C und veranschaulicht so die Interpolation von Texturkoordinaten. Die projizierte Textur ist (noch nicht) ganz korrekt.

Rastern (Berechnung der Farbe eines jeden Pixels).

Rendering-Durchgang (engl. rendering pass) Einmaliges Rendern einer Szene.

## 1.2 Vor- und Nachteile bei Berechnung auf der Grafikkarte

Moderne leistungsfähige Grafikkarten sind eigentlich optimiert auf die Berechnung von 3D-Grafiken. Dadurch ergeben sich gewisse Einschränkungen des sinnvollen Anwendungsbereichs, z. B. sind keine beliebig großen Auflösungen und Texturgrößen erlaubt. Zudem ist die Geschwindigkeit dem menschlichen Empfinden angepasst, d. h. der Erzeugung von größenordnungsmäßig 100 Bildern pro Sekunde.

Zellularautomaten gibt es mit unterschiedlicher Dimension. Grafikprozessoren eignen sich besonders gut für 2 Dimensionen (texturierte Polygone). Mehr Dimensionen sind nur sehr umständlich zu implementieren, bei weniger leidet die Geschwindigkeit unter Overhead.

Dennoch erhofft man sich Geschwindigkeitsvorteile in der Verwendung eines Grafikprozessors (engl. Graphics Processing Unit, GPU) für die Simulation aus folgenden Gründen:

- Vermeidung überflüssiger Datentransporte zwischen Prozessor und Grafikkarte. Da das Ergebnis der Simulation meist grafisch dargestellt werden soll, ergibt sich hier ein Vorteil, denn die Verbindung zwischen Prozessor, Arbeitsspeicher und Grafikkarte ist der Engpass.
- Hohe Parallelität der Berechnung auf der Grafikkarte. Moderne Grafikchips besitzen bis zu 8 Pipelines, die parallel Berechnungen ausführen können. Da gerade bei Zellularautomaten wenig Kommunikationsaufwand besteht, schlägt die Parallelisierung voll durch.

- Extreme Speicherbandbreite der Grafikkhardware. Der lokale Speicher der Grafikkarten (im Moment bis zu 256 MB) ist sehr breit (bis zu 512 Bit) angebunden, dazu noch mit der technisch höchsten erreichbaren Taktfrequenz (über 300 MHz). Dabei ergibt sich ein Vorteil in der Größenordnung von Faktor 5 (CPU: 4 GByte/s, GPU, 20 GByte/s).

Dem entgegen stehen auch einige Nachteile, die man in Kauf nehmen muss:

- Heutige GPUs laufen noch mit einer im Gegensatz zu CPUs langsamen Taktung (ca. 300 MHz gegen 3 GHz).
- Es sind keine „intelligenten Algorithmen“ ausführbar, denn die Berechnung eines Zustandsübergangs auf der Grafikkarte wird für alle Zellen gleichzeitig angestoßen und läuft dann autonom durch. Ein Eingreifen oder Abbrechen ist in diesem Zeitraum nicht möglich. Eine feinere Granulierung der Berechnung (Zerlegung in mehrere Durchläufe, von denen dann evtl. einer weggelassen werden kann) ist auch nicht besser, denn damit steigt der Overhead. Außerdem ist die Programmierung ein Extrembeispiel für SIMD (Single Instruction, Multiple Data). Für alle Pixel oder Vertices werden die gleichen Berechnungen angestellt, Einschränkungen und Fallunterscheidungen sind nur sehr schwierig zu machen. Weiterhin kann zwischen der Berechnung der einzelnen Zellen keinerlei Zustandsinformation gerettet werden. Somit sind z. B. Statistiken nicht zu realisieren. Insbesondere sollte der Aufwand für die Berechnung für alle Zellen bei jedem Durchlauf und für jede Konfiguration ungefähr gleich sein. Eine starke Datenabhängigkeit des Algorithmus (z. B. viele abzufangende Sonderfälle) können die Berechnung sehr verlangsamen bis ganz unmöglich machen.

## 2 Umsetzung für die Grafikkhardware

Moderne Grafikkarten bringen als Hauptvorteil eine mächtige Programmierbarkeit ihrer Einheiten, der sog. Shader, an. Durch die Software können diese Fähigkeiten über verschiedenen APIs angesprochen werden. Durchgesetzt haben sich DirectX von Microsoft und OpenGL als Hersteller übergreifender Standard. Da jedoch die Entwicklung in letzter Zeit von Microsoft in Verbindungen mit den Grafikkarten-Herstellern ATI und nVidia auf den lukrativen Spielmarkt konzentriert hat, besitzt DirectX inzwischen einen leichten Vorsprung, natürlich besonders bei den neuesten Features. Deshalb habe ich mich hier zur Verwendung von letzterem entschlossen. Von DirectX und dem hier vor allem interessanten Teil DirectGraphics ist seit Dezember 2002 die Version 9 verfügbar. Mit DirectX 8 und den Grafikkarten, die dessen Hardware-Beschleunigung voll unterstützten, hielt zum ersten Mal Programmierbarkeit Einzug (Vertex Shader 1.0–1.1, Pixel Shader 1.0–1.4). Deren Möglichkeiten wurden mit DirectX 9 deutlich erweitert und vereinfacht (Vertex Shader 2.0, Pixel Shader 2.0). Es existieren auch schon die Spezifikationen für Versionen 3.0 der beiden Shader-Typen, allerdings werden diese noch von keiner Hardware unterstützt und können daher nur wieder per Software vom Hauptprozessor in sehr schlechter Geschwindigkeit emuliert werden.

### 2.1 Vertex Shader

Vertex Shader sind Programme für die programmierbaren Einheiten des Grafikprozessors, die sich um die Verarbeitung der Vertices (Polygon-Eckpunkte) kümmern. Die Verarbeitungsgeschwindigkeit liegt typischerweise in der Größenordnung von 100 Millionen Vertices pro Se-

kunde. Es können Tausende von Instruktionen verarbeitet werden, auch sind Flusskontrolle und Schleifen möglich. Es gibt mindestens 12 temporäre Register und mindestens 256 Konstanten ( `c0`– `c255`), die vom Hauptprogramm aus gesetzt werden können. Das Mindeste, was ein Vertex Shader sinnvollerweise erledigen muss, ist die Transformation von 3D- nach 2D-Koordinaten. Dies geschieht durch eine Matrix-Multiplikation mit einer  $4 \times 4$ -Matrix. Der Vertex Shader befindet sich innerhalb der Pipeline vor den Pixel Shadern und gibt seine Ergebnisse an diese weiter. Das beinhaltet jedoch den Nachteil, dass die Ergebnisse nur temporär vorhanden sind und damit nicht direkt wieder abgespeichert werden können. Damit disqualifiziert sich diese Möglichkeit eigentlich zur iterativen Berechnung von zellulären Automaten. Daher werden nur ganze 4 Vertices zur Definition eines Rechtecks verwendet, und im Vertex Shader nur Hilfsgrößen wie leicht verschobene Textur-Koordinaten für die Adressierung der Nachbarschaft berechnet.

## 2.2 Pixel Shader

Pixel Shader werden für jeden zu zeichnenden Pixel aufgerufen. Ihre Ausgabe besteht daher nur aus einem 4-komponentigen Farbvektor. Dabei besteht sogar auch noch das Problem, dass die Alpha-Komponente nicht direkt wieder abgespeichert, sondern nur temporär zur Berechnung von Überblendungen verwendet und danach verworfen wird.

Als Ziel eines Rendering-Durchgangs, bei dem die Pixel Shader aufgerufen werden, kann nicht nur der Backbuffer zur direkten Darstellung auf dem Bildschirm sein, sondern auch eine oder mehrere Texturen. Das ermöglicht das direkte Abspeichern des Ergebnissen. Wenn man nun einen Texel aus der Ausgangs-Textur (Ausgangs-Zustand) auf genau einen Pixel abbildet, so kann man direkt eine Iteration des Zellularautomaten berechnen, in dem man einmal das Rechteck zeichnet. Dabei dürfen natürlich Ausgangs- und Ziel-Textur nicht die gleiche sein, was aber auch nicht schadet, da man bei Zellularautomaten sowieso synchron rechnen will. Die letzte endliche Reihenfolge der Berechnung der Pixel und damit der Zellen lässt sich gar nicht festlegen und hängt von der Grafikkarte und deren Treiber ab. Die naive Vorstellung, die Zellen würden bei einem rechteckigen Universum von links nach rechts und von oben nach unten berechnet, wird sich schon deswegen nicht erfüllen, weil ein Rechteck zur Berechnung der 3D-Grafik in zwei Dreiecke aufgeteilt werden muss, die auf jeden Fall nacheinander gezeichnet werden.

Pixel Shader rechnen intern ausschließlich mit Textur-Koordinaten und Farbwerten, und das alles in Fließkomma-Arithmetik. Die Ausgabe und damit Speicherung kann jedoch wahlweise auch in unterschiedlichen Ganzzahl-Formaten geschehen. Daraus ergibt sich automatisch eine Rundung und Quantisierung, die dem Konzept des Zellularautomaten entgegen kommt.

Als Eingabe akzeptiert ein Pixel Shader bis zu 32 Konstanten ( `c0`– `c31`), die während eines Rendering-Durchgangs gleich bleiben für alle Pixel, und bis zu 8 Textur-Koordinaten ( `t0`– `t8`), interpoliert aus den Ergebnissen des Vertex Shader und ebenfalls nur lesbar. In 12 Registern ( `r0`– `r11`) können Zwischenergebnisse gehalten werden und letztlich einmalig in die 4 Ausgaberegister ( `oC0`– `oC2`) geschrieben werden. Die Textur-Koordinaten sind ein- bis dreidimensional, alle Register enthalten die 3 Farbwerte sowie die Alpha-Komponente. Vorhandenen Instruktionen sind z. B. einfache wie `add`, `mul`, `mad` (multiply-add), aber auch komplexere wie `sin` und `exp`, bei welchen man sogar die einzuhaltende Genauigkeit vorgeben kann.

Flusskontrolle und Schleifen sind in Pixel Shadern bisher noch nicht möglich, aber es gibt zwei Behelfsmöglichkeiten, diskrete Entscheidungen zu treffen:

- Die bedingte Zuweisung (conditional move, ?-Operator):  
`cmp r0, r1, r2, r3` weist r0 komponentenweise entweder r2 oder r3 zu, je nach dem, ob die Komponente in `r1 < 0` oder `≥ 0` ist.
- Durch einen Texture Lookup mit einer zuvor berechneten Texture-Koordinate kann wie auf eine Tabelle zugegriffen und mit passender Initialisierung der Textur beliebige Funktionen in bis zu 3 Dimensionen berechnet werden. Dies eignet sich besonders auch zur Diskretisierung von Werten.

## 2.3 Umsetzung der Eigenschaften eines Zellularautomaten

- Das regelmäßige Gitter des zu simulierenden Zellularautomaten wird durch eine 1- oder 2-dimensionale Textur repräsentiert. In der Farbe des jeweiligen Texels ist der Zustand der entsprechenden Zelle kodiert, wobei die 3 Farbkanäle (rot, grün, blau) genutzt werden können.
- Die Genauigkeit der Speicherung der Farbkomponenten kann verschiedentlich eingestellt werden: von Fließkomma-Genauigkeit bis hinunter zu 256 oder auch nur 32 Ganzzahl-Stufen, auf die dann gerundet wird.
- Die Nachbarschaft wird definiert durch den Offset der Texturkoordinaten zur aktuellen Position, mit welchen dann auf die Zustände der Nachbarzellen zugegriffen werden kann.
- Die lokale Überföhrungsfunktion muss in einen für den Zellularautomaten charakteristischen Pixel Shader umgesetzt werden.
- Ein synchroner Zustandsübergang des gesamten Zellularautomaten geschieht durch einen Rendering-Durchgang: Es wird ein Polygon (meistens ein Quadrat) gezeichnet, dem die die Textur mit dem Ausgangszustand zugeordnet ist. Durch die Deklaration mehrerer Textur-Koordinaten pro Vertex (und die daraus folgende interpolierten Werte für jeden einzelnen Zielpixel) lassen sich die aktuell zu berechnende Zelle sowie ihre „Nachbarn“ (durch Verschiebung um wenige Pixelbreiten) im Sinne der Übergangsfunktion adressieren.  
Der Pixel Shader wird nun für jeden Pixel in der Ziel-Textur aufgerufen und berechnet damit den lokalen Folgezustand, denn die Textur-Koordinaten und die Projektion sind genau so gewählt, dass Ausgangs- und Zielpixel aufeinander fallen. Vor dem nächsten Berechnungsschritt müssen natürlich Ausgangs- und Zieltextur vertauscht werden.
- Die Randbedingungen lassen sich auch relativ flexibel und billig konfigurieren. Die Grafikkarten unterstützen normalerweise Texturkachelung (entspr. periodischen Randbedingungen), spezielle Randfarbe (entspr. festen Randbedingungen) oder sogar gespiegelte Fortsetzung.

### 2.3.1 Zufall

Für stochastische Zellularautomaten benötigt man Zufallszahlen. Da keinerlei Statusinformation zwischen den Pixel-Berechnungen weitergegeben werden kann, sind iterative Pseudozufallszahlen-Generatoren nicht möglich. Man muss sich mit einer „Zufallstextur“ behelfen, die im Hauptprogramm generiert wird. Es wäre jedoch weitaus zu langsam,

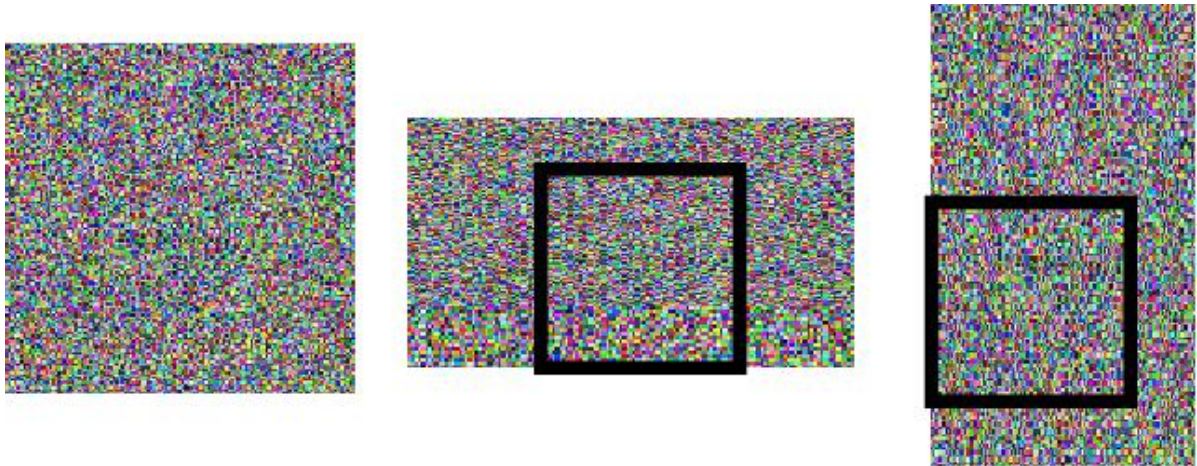


Abbildung 2: Eine Zufallstextur, und daraus verwendete jeweils zufällige Ausschnitte

diese für jeden Berechnungsschritt neu zu generieren. Daher verwendet man folgenden Trick, der allerdings Gefahr läuft, die Zufälligkeit und statistische Unabhängigkeit der Zahlen zu verringern: Dabei wird zu Anfang einmalig eine Zufallstextur generiert, die größer ist als das Zelluniversum. Wäre die Fläche kleiner, würde auf jeden Fall irgendwo zwei benachbarten Pixeln immer die gleiche Zahl zugewiesen werden, was nicht akzeptabel ist. Nun wird mit jedem Berechnungsschritt nur noch mittels weniger Zufallszahlen der Ausschnitt der Zufallstextur verschoben und gestaucht, der über den Zustandsspeicher gelegt werden soll (Veranschaulichung siehe Abbildung 2). Die ist mit hoher Geschwindigkeit möglich, da letztlich nur ein paar Textur-Koordinaten anders initialisiert werden müssen. Als Textur-Filterung wird „Point Sampling“ verwendet, um harte Sprünge (Flirren) und keine weichen Überblendungen zwischen den Zufallszahlen zu bekommen. In der Praxis funktioniert das Verfahren gut und lässt keine Artefakte erkennen, wenn man die Zufallstextur mit dem Prozessor angemessen initialisiert.

### 2.3.2 Verwendete Assembler-Befehle und -Direktiven

`vs.2.0` Deklariert die Version des verwendeten Vertex Shader (hier immer 2.0). Dadurch werden die erlaubten Befehle und Direktiven und damit die benötigten Fähigkeiten der Hardware festgelegt.

`decl_position vn` Deklariert die Raum-Koordinaten als erste Übergabeparameter, wird in `vn` abgelegt.

`decl_texcoordm vn` Deklariert Texturkoordinaten-Paare als weitere Übergabeparameter, werden in `vn` abgelegt. Die Deklarationen müssen mit der im Hauptprogramm deklarierten Struktur eines Vertex übereinstimmen.

`dp4 ra, rb, rc` Berechnet ein 4-dimensionales Skalarprodukt aus `rb` und `rc` und legt das Ergebnis in `r0` ab. Durch vier nacheinander geschaltete Operationen dieses Typs lässt sich eine  $4 \times 4$ -Matrix-Multiplikation durchführen, zu der die Matrix in transponierter Form vorliegen muss.

`oPos` Ausgaberegister des Vertex Shader für die transformierte Position ( $z = 0$ ).

`oTn` Ausgaberegister des Vertex Shader für die n-te Textur-Koordinate.

`ps.2.0` Deklariert die Version des verwendeten Pixel Shader (siehe oben).

`dcl_dd sn` Deklariert die Verwendung der Textureinheit `n` (engl. *texture stage*) für die Benutzung mit d-dimensionalen Texturen. Im Hauptprogramm können für jede Textureinheit Optionen wie Filterung und Randbedingungen festgelegt werden.

`dcl tn.xy` Deklariert die zweidimensionale Textur-Koordinate `tn`.

`def cn f` Setzt die Konstanten `cn` auf `f`. Das Hauptprogramm kann sich jedoch immer noch darüber hinwegsetzen und diesen Wert vor Ausführung ändern.

`texld rm, tn, sp` (von engl. *texture load*) Berechnet den Farbwert der Textur anliegend an Textureinheit `sp` an der (Textur-)Koordinate unter Berücksichtigung aller eingestellter Optionen `tn` und speichert das Ergebnis in `rm`.

`add ra, rb, rc` Addiert `rb` und `rc` komponentenweise und legt die Ergebnisse in `ra` ab.

`sub ra, rb, rc` Subtrahiert `rc` von `rb` komponentenweise und legt die Ergebnisse in `ra` ab.

`mul ra, rb, rc` Multipliziert `rb` mit `rc` und addiert danach `rd` jeweils komponentenweise und legt die Ergebnisse in `ra` ab.

`mad ra, rb, rc, rd` Multipliziert `rb` mit `rc` komponentenweise und legt die Ergebnisse in `ra` ab.

`mov ra, rb` Kopiert komponentenweise von `rb` nach `ra`. Durch nachgestellten Modifier für die Komponente kann auch ein Teil „ausgeschnitten“ bzw. vervielfältigt werden.

`frc ra, rb` Bildet komponentenweise den Nachkommateil von `rb` und speichert das Ergebnis in `ra`.

`abs ra, rb` Bildet komponentenweise den Absolutwert von `rb` und speichert das Ergebnis in `ra`.

`cmp ra, rb, rc, rd` Weist `ra` komponentenweise entweder `rc` oder `rd` zu, je nach dem, ob die Komponente in `rb`  $< 0$  oder  $\geq 0$  ist.

`oCn` Ausgaberegister für die Farbe der n-ten Ausgabeziels des Pixel Shader.

## 3 Beispiel-Implementierungen

### 3.1 Game of Life

Das *Game of Life* ist sozusagen das Standardbeispiel für Zellularautomaten. Es existierte auch bereits eine Implementierung für die Grafikkarte von nVidia mittels Pixel Shader. Die Überföhrungsfunktion ist jedoch so einfach, dass man mit spezieller Anwendung von Z- und Stencil-Buffer auch noch auf älteren Grafikkarten zum Ziel kommen kann. Die Variante von nVidia war auf Pixel Shader in der Version 1.3 zugeschnitten und benötigte daher 3 sequentielle

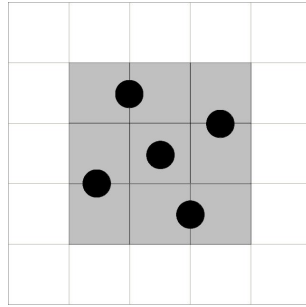


Abbildung 3: Textur-Koordinaten zur Aufsummierung der Nachbarschaft mittels bilinearer Filterung

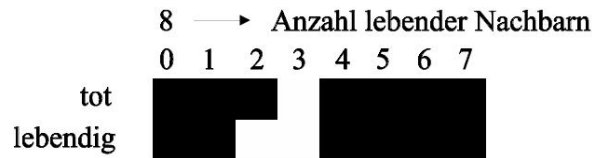


Abbildung 4: Hilfstextur zur Simulation des Game of Life

Durchläufe (Passes) für einen Zustandsübergang, was die Berechnung natürlich ziemlich langsam machte. Die hier vorgestellte Version verwendet den Funktionsumfang von Pixel Shader 2.0 und kommt daher mit einem Rendering-Pass pro Zustandsübergang aus.

### 3.1.1 Umsetzung

Die Lebendigkeit einer Zelle wird in allen drei Farbkanälen parallel gespeichert (1.0  $\hat{=}$  weiß  $\hat{=}$  lebendig; 0.0  $\hat{=}$  schwarz  $\hat{=}$  tot).

Die Nachbarschaft wird durch Texture Lookups mit verschobenen Koordinaten durchgeführt. Da nur die Summe der lebenden umgebenden Zellen interessiert, kann man einen Trick verwenden, um Aufwand zu sparen: Vier Abfrage-Koordinaten werden jeweils genau in der Mitte zweier Pixelzentren positioniert. Wir nun bilineare Filterung dazu aktiviert, bildet die Texturreinheit automatisch das gewichtete Mittel der umgebenden (im Allgemeinen vier) Pixel, was dann in dieser Konfiguration gleich der halben Summe der beiden Pixel ist. Die Lebendigkeit der aktuell betrachteten Zelle wird durch einen Texturzugriff genau in deren Mitte herausgefunden. Insgesamt benötigt man nur 5 Texturzugriffe, um genügend Informationen über die 9 Positionen zu erhalten, was Abbildung 3 noch einmal veranschaulicht.

Die letzt endliche Entscheidung ob die Zelle (weiter-)lebt, erfolgt wiederum durch einen Texturzugriff auf folgende speziell kodierte Regel-Textur der Dimension 2x8:

Dabei wird die Anzahl der lebenden Nachbarn durch acht geteilt und dieser Fließkommawert als x-Texturkoordinate verwendet. Der Status der eigenen Zelle (0.0 oder 1.0) wiederum wird als y-Koordinate verwendet. Die drei weißen Pixel in dieser Textur codieren genau die drei Möglichkeiten einer Zelle, geboren zu werden bzw. weiter zu leben.

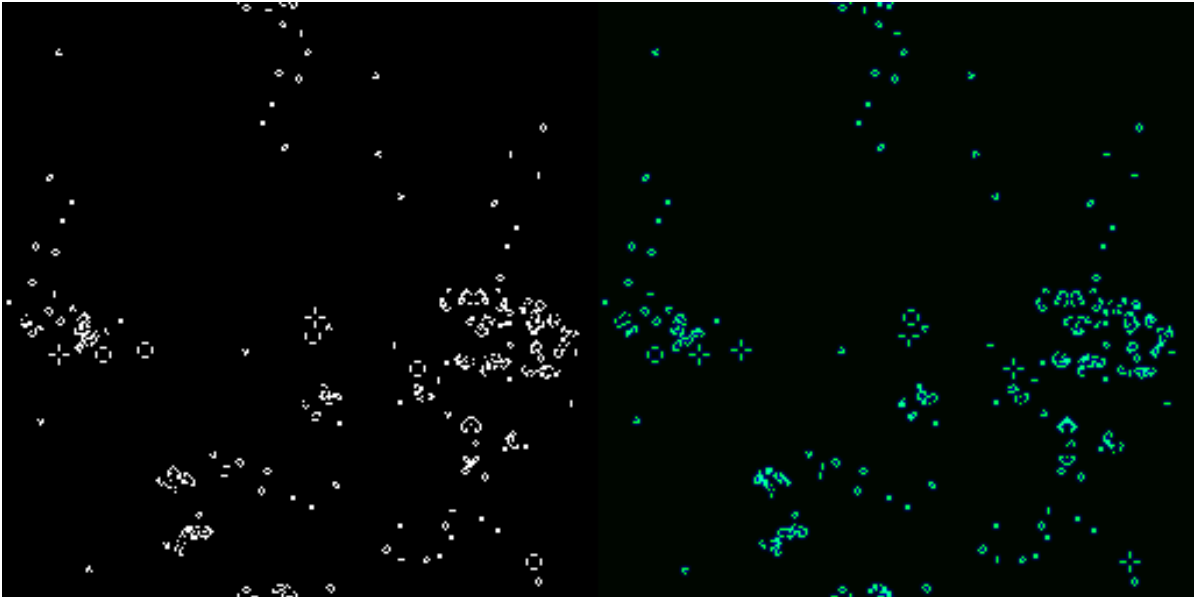


Abbildung 5: Beispiel-Konfiguration des Game of Life (256 × 256). Die linke Seite zeigt die lebenden Zellen, die rechte Seite im grünen Farbkanal die Lebendigkeit, im blauen die Anzahl der lebenden Nachbarn

### 3.1.2 Geschwindigkeitsvergleich

Die Simulation erreicht unter optimalen Bedingungen eine Geschwindigkeit von ca. 250 Millionen Zellupdates pro Sekunde. Dies ist etwas eine Größenordnung niedriger als die Berechnung auf dem Hauptprozessor mit ca. 2,25 Milliarden Zellupdates pro Sekunde. Dabei ist weniger die Grafikhardware langsam als eher der Prozessor sehr schnell, denn das Problem lässt sich durch Verwendung langer Bitvektoren sehr gut optimieren, während sich die Grafikkarte mit einem Overhead von 31 Bit pro Zelle herumschlagen muss. Die Fließkomma-Berechnung kommt zudem überhaupt nicht zum Tragen und stört eher.

Immerhin überträgt dieses Beispiel die gegebene Eigenschaft der Berechnungs-Universalität des Game of Life auf die verwendete Grafik-Hardware.

### 3.1.3 Der Pixel Shader zum Game of Life

```
//SimulateLife.psh
//
//Pixel shader for doing the logic of the Game of Life
//completely in one pass (Pixel Shader 2.0)

//Declare pixel shader version
ps.2.0

dcl_2d    s0    //cell state, point sampling
dcl_2d    s1    //cell state, bilinear filtering
dcl_2d    s5    //decision support texture, point sampling

dcl       t0.xy //current cell
dcl       t1.xy //4 texture coordinates to sample 8 neighbors
dcl       t2.xy //by means of bilinear filtering
dcl       t3.xy
```

```

dcl          t4.xy

//Multiplicative factor to isolate green
//  RGBA
def  c0, 0.0, 1.0, 0.0, 0.0

//Additive factor to bias the red-green slightly (to avoid problems with tight texture coordinates)
def  c1, 0.0234, 0.0234, 0.0, 0.0

//Weights for neighbor pixels
def  c2, 0.25, 0.0, 0.0, 0.0

//get colors from all 5 texture stages
texld r0, t0, s0          //current cell
texld r1, t1, s1
texld r2, t2, s1
texld r3, t3, s1
texld r4, t4, s1

mul r0, r0, c0           //select green only & add bias, but do not overflow 1.0
add_sat r0, r0, c1

//and add the average of the neighbors according to weights in constant mem:
//r0 = r0 + fac * r1 + fac * r2 + fac * r3 + fac * r4
mul r0.r, c2, r1.g
mad r0.r, c2, r2.g, r0
mad r0.r, c2, r3.g, r0
mad r0.r, c2, r4.g, r0

texld r5, r0, s5        //lookup decision texture

mov oc0, r5             //primary output = dead or alive

mov r0.b, r0.r          //change colors
mov r0.r, c1.a
mov oc1, r0             //secondary output = accumulated neighborhood aliveness

```

### 3.2 Reaktions-Diffusions-Prozess nach FitzHugh-Nagumo

$$\begin{aligned}
 p_{n+1} &= (a - p_n)(p_n - 1)p_n - i_n + D_p \nabla^2 p \\
 i_{n+1} &= e(bi_n - p_n) + D_i \nabla^2 i
 \end{aligned}$$

Die Simulation eines Reaktions-Diffusions-Prozesses nach FitzHugh-Nagumo stellt einen Grenzfall eines Zellularautomaten dar. Jede Zelle besitzt zwei reelle Zustandswerte, nämlich das elektrische Potential und die Ionenkonzentration. Mit diesen werden im Zustandsübergang stetige numerische Operationen durchgeführt. Jedoch wird das Ergebnis danach auf Stufen quantisiert (z. B. 100), womit man eine endliche Zustandsmenge erhält. Diese Quantisierung erfolgt stochastisch nach folgender Formel:

$$x \leftarrow \begin{cases} \lfloor x \rfloor & \text{mit } p = 1 - (x - \lfloor x \rfloor) \\ \lceil x \rceil & \text{mit } p = x - \lfloor x \rfloor \end{cases} \quad (1)$$

Die Startkonfiguration ist ein homogener Zustand mit nur kleinen zufälligen Abweichungen. Daraus ergibt sich eine zunächst chaotisch beginnende Reaktion, die irgendwann periodisch wird und beliebig lange weiterläuft.

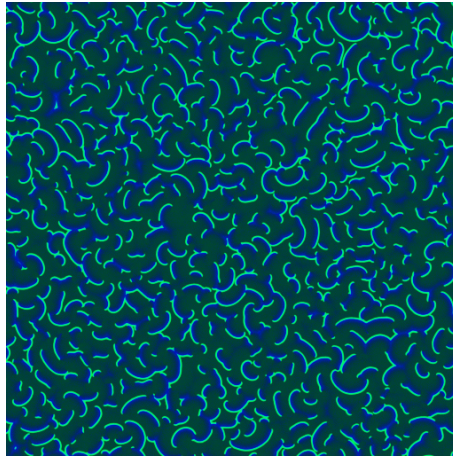


Abbildung 6: Beispiel-Konfiguration des Reaktion-Diffusions-Prozesses nach FitzHugh-Nagumo ( $2048 \times 2048$ ). Das elektrische Potential wird durch die blaue Komponente, die Ionenkonzentration durch die grüne Komponente dargestellt.

### 3.2.1 Geschwindigkeits-Vergleich

Die Simulation des Reaktions-Diffusions-Systems nach FitzHugh-Nagumo erreicht unter optimalen Bedingungen etwa 100 Millionen Zellupdates pro Sekunde. Dies liegt eine Größenordnung über dem Hauptprozessor, der mit optimiertem C-Quellcode auf ca. 10 MUpds/s schafft. Somit ergibt sich hier ein klarer Geschwindigkeitsvorteil für die Grafikkarte. Jedoch muss man auch einige Einschränkungen in Kauf nehmen:

- Weniger Flexibilität in der Ausdehnung: Die Grafikkarte erlaubt eine maximale Kantenlänge von Texturen von 2048. Es wäre im Prinzip möglich, durch Verwendung mehrerer Texturen und Synchronisation der Anschlussbereiche größere Zelluniversen zu berechnen, aber das wesentliche Geschwindigkeitseinbußen mit sich bringen. Zudem ist der Speicher auf der Grafikkarte kleiner und nicht auslagerbar. Die Speichergröße für eine Zelle kann man sich auch nicht frei auswählen, sondern muss mit den vorgegebenen Möglichkeiten für die Speicherung eines Pixels vorlieb nehmen. Durch den Overhead für die Initialisierung der Grafikkarte skaliert die Berechnungs-Geschwindigkeit der Grafikkarte bis zu einem Grenzwert. Die CPU verliert dagegen nur wenig Geschwindigkeit bei kleineren Universen.

Die Grafikkarte erwartet Textur-Kantenlängen als Zweier-Potenzen. Man könnte dieses Problem zwar damit umgehen, dass man nur einen Teil einer größeren Textur abbilden lässt, aber damit verschwendet man trotzdem Speicher und verliert die Möglichkeit der periodischen Randbedingungen.

- Weniger Flexibilität in der Quantisierung: Beim Abspeichern des Berechnungsergebnisses quantisiert der Grafikprozessor automatisch auf das gewünschte Format, z. B. 5 oder 8 Bit pro Farbe bzw. speichert direkt als Fließkommazahl. Beliebige Quantisierungsstufen wiederum sind jedoch nicht so einfach zu haben und schmälern die Performance, da man dann „von Hand“ runden muss.

### 3.2.2 Der Pixel Shader zu FitzHugh-Nagumo

```
//FitzHughNagumo.psh
//
//Pixel shader for calculating the FitzHugh-Nagumo reaction diffusion model
//
//g      f1 (electrical potential)
//b      f2 (ion concentration)

//Declare pixel shader version
ps.2.0

dcl_2d   s0      //cell state, point sampling
dcl_2d   s1      //cell state, bilinear filtering
dcl_2d   s6      //random texture, point sampling

dcl      t0.xy   //current cell
dcl      t1.xy   //4 texture coordinates to sample 8 neighbors
dcl      t2.xy   //by means of bilinear filtering
dcl      t3.xy
dcl      t4.xy
dcl      t6.xy   //random value

//      r   g   b   a
//      f1  f2

//Constants are set by main program, only examples here

//      a   b   e
//def    c0, 0.1, 1.0, 0.005, 0.0      various constants
//      D1  D2
//def    c1, 0.0, 0.111112, 1.0, 0.0    center
//def    c2, 0.0, 0.222222, 0.0, 0.0    straight/diagonal cells
//def    c3, 1.0, 0.003906, -0.001953, 0.0  rounding constants
//      1.0, 1.0/256, -0.5/256.0

//def    c4, 0.0, 1.4, 0.25, 0.0 rescaling from [0.0, 1.0], multiplicative component
//def    c5, 0.0, -0.4, -0.05, 0.0 rescaling from [0.0, 1.0], additive component
//def    c6, 0.0, 0.71428, 4.0, 0.0 rescaling to [0.0, 1.0], multiplicative component
//def    c7, 0.0, 0.285714, 0.2, 0.0 rescaling to [0.0, 1.0], additive component

texld   r0, t0, s0      //current cell
texld   r1, t1, s1      //neighborhood
texld   r2, t2, s1
texld   r3, t3, s1
texld   r4, t4, s1
texld   r6, t6, s6      //random texture

mul     r8, r0, c4      //rescale center from [0.0, 1.0] into r8
add     r8, r8, c5

add     r1, r1, r2
add     r1, r1, r3
add     r1, r1, r4      //r1 = sum straight/diagonal

mul     r1, c2, r1      //scale straight/diagonal

mad     r0, c1, r0, r1  //update center (with averaged value by diffusion)

mul     r0, r0, c4      //rescale updated center from [0.0, 1.0]
add     r0, r0, c5
//finished second part of equation here

//f1
sub     r4.g, c0.r, r8.g      //(a-f1)
sub     r7.g, r8.g, c3.r      //(f1-1)
mul     r4.g, r4.g, r7.g      //(a-f1)*(f1-1)
```

```

mad    r4.g, r4.g, r8.g, -r8.b    //(a-f1)*(f1-1)*f1-f2

//f2
mad    r4.b, c0.g, r8.g, -r8.b    //(b*f1-f2)
mul    r4.b, c0.b, r4.b          //e*(b*f1-f2)
//finished first part of equation here

add    r0, r0, r4                //add first part of equation

mul    r0, r0, c6                //rescale to [0.0, 1.0]
add    r0, r0, c7

mad    r6, r6, c3.g, c3.b        //(... + 0.5) / 256, [0.0, 1.0] => [-0.5/256, 0.5/256]
add    r0, r6, r0                //add for stochastic rounding

mov    oC0, r0                  //primary output: cell state, f1 green, f2 blue

mov    r1, c3.a                  //... = 0
mov    r1.g, r0.g

mov    oC1, r1                  //secondary output: f1 green only

mov    r2, c3.a                  //... = 0
mov    r2.b, r0.b

mov    oC2, r2                  //tertiary output: f2 blue only

```

### 3.3 Mandelbrot-Fraktal

Zur Demonstration der Leistungsfähigkeit der verwendeten Hardware wurde noch zusätzlich die Berechnung des Mandelbrot-Fraktals implementiert. Dies stellt keinen Zellularautomaten im eigentlichen Sinn mehr da, da die Nachbarschaft trivial ist und außerdem der Zustandsraum idealerweise kontinuierlich. Die Iteration wird in Abweichung zur normalerweise verwendeten Berechnungsmethode parallel für sämtliche Zellen (Punkte in der komplexen Zahlenebene) durchgeführt, statt für jede Zelle nacheinander. Die Iterationsformel lautet:

$$\begin{pmatrix} a \\ b \end{pmatrix} = z_{n+1} = z_n^2 + c = \begin{pmatrix} a^2 - b^2 + \overbrace{Re(c)}^{x\text{-Koordinate}} \\ 2ab + \underbrace{Im(c)}_{y\text{-Koordinate}} \end{pmatrix}$$

Der komplexe Zustand der Zelle wird in zwei Farbkanälen und mit voller Fließkommagenauigkeit (hier 15 Bit Mantisse) abgespeichert.

#### 3.3.1 Der Pixel Shader zum Mandelbrot-Fraktal

```

//Mandelbrot.psh
//
//Pixel shader for calculating iteratively the Mandelbrot set
//in multiple passes (Pixel Shader 2.0)
//
//Declare pixel shader version
ps.2.0

//r: real, g: imaginary, a: count

dcl_2d    s0    //cell iteration state

dcl      t0.xy
dcl      t7.xyzw    //cartesian coordinates

```

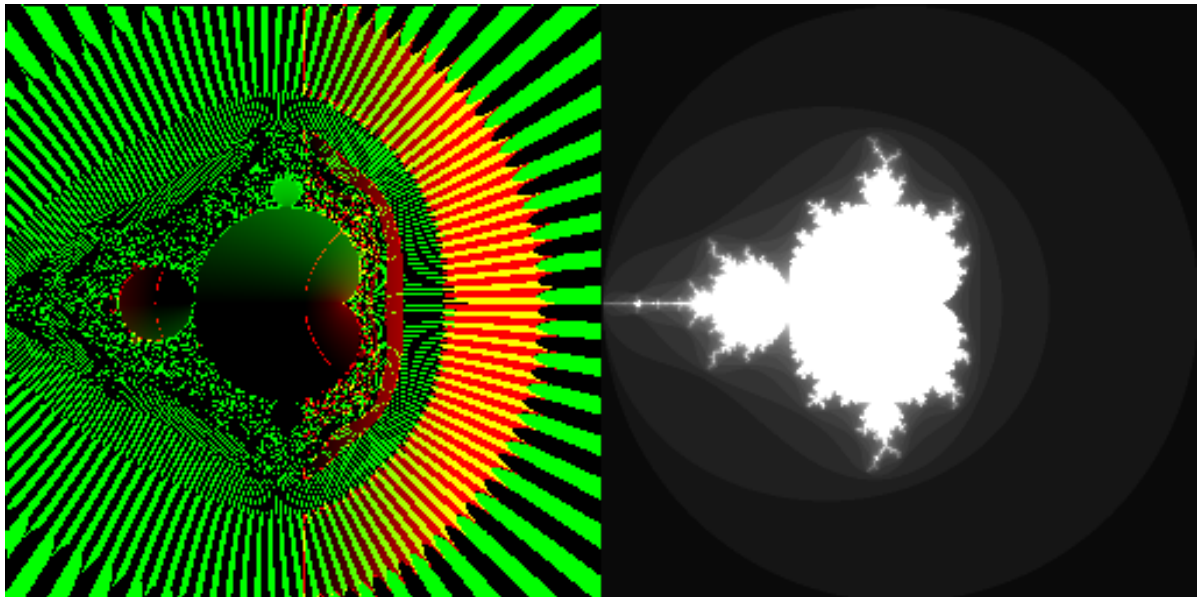


Abbildung 7: Das Mandelbrot-Fraktal, berechnet mit dem Pixel Shader. Der linke Teil zeigt die komplexen Werte der Iteration in den Komponenten grün und rot, die rechte Seite die Mandelbrot-Menge (weiß).

```

def      c0, 2.0, 4.0, 0.0, 0.1
def      c5, 0.0, 0.0, 0.0, 0.0
def      c6, 1.0, 1.0, 1.0, 0.04
def      c7, 0.5, 0.5, 0.5, 0.5

mov      r5, c5      //to avoid problems with using to many constants in one operation
mov      r6, c6

texld    r0, t0, s0      //load complex iteration state

mul      r1, r0, r0      //a^2, b^2
mul      r0.g, r0.r, r0.g //a*b
mul      r0.g, r0.g, c0.r //2*(a*b)
sub      r0.r, r1.r, r1.g //a^2-b^2
add      r0.rg, r0, t7    //+c

add      r1, r1.r, r1.g   //|a+ib|^2
sub      r1, r1, c0.g     //-s
cmp      r4, r1, r5, r6   //out of set, in set?

add      r0.a, r0.a, r4.a //count++, save

mov      oC0, r0         //primary output: complex iteration state red/green

mov      r2, c0.b        //... = 0
mov      r2, r0.a
mov      oC1, r2         //secondary output: stepped

abs      r0, r0
mov      oC2, r0         //tertiary output: absolute value of complex iteration state
mov      oC3, r4         //fourth output: Mandelbrot set
  
```

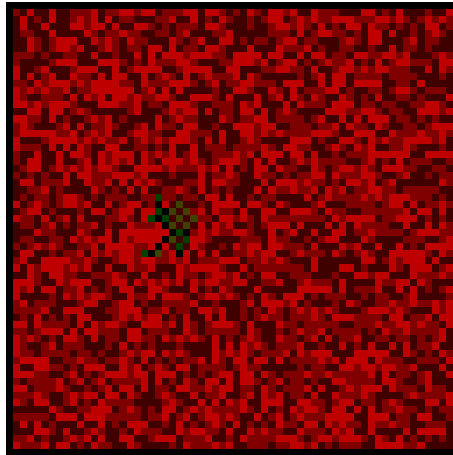


Abbildung 8: Beispiel für Sandbox. In der Mitte breitet sich gerade eine Lawine aus.

### 3.4 Simulation von Überlauf-Kaskaden mittels des Sandbox-Zellularautomaten

Der Sandbox-Zellularautomat simuliert einen Sandkasten. Jede Zelle kann 0 bis 3 Sandkörner enthalten. Kommt durch externe Einwirkung oder durch einen Iterationsschritt noch ein viertes dazu, so „läuft“ die Zelle über, die vier direkten Nachbarn erhalten ein weiteres Sandkorn, die aktuelle Zelle hat nachher kein Sandkorn mehr. Dieses Überlaufen kann sich in „kritischen“ Situation fortsetzen und bildet Kaskaden, die erst nach einiger Zeit aussterben. Das Universum wird mit einer zufälligen Gleichverteilung initialisiert. Der Benutzer kann durch einen Klick mit der rechten Maustaste einer Zelle ein Sandkorn hinzufügen und diese so zum Überlaufen bringen. *Sandbox* ist ein Beispiel für einen Zellularautomaten mit festen Randbedingungen. Am Rand des „Sandkasten“ laufen sich die Kaskaden spätestens tot.

Um den Folgezustand einer Zelle zu berechnen, muss sie eigentlich den Füllstand der vier direkt benachbarten Zellen betrachten und für jede einzeln entscheiden, ob diese gerade überläuft oder nicht. Dies widerspricht allerdings der Forderung, dass der Zellularautomat totalistisch sein soll. Außerdem würde die gleiche Entscheidung ineffizienter Weise gleich viermal getroffen. Deshalb wird folgendes Verfahren verwendet: Im roten Farbkanal wird die Anzahl der Sandkörner gespeichert. Der grüne Farbkanal speichert, ob die Zelle in der letzten Iteration übergelaufen ist (1.0) oder nicht (0.0). Über diese Flags wird von jeder Zelle einfach die Summe gebildet und damit ermittelt, wie viele Sandkörner dazukommen. Am Ende der Berechnung entscheidet die Zelle für sich, ob sie überläuft. Falls ja, setzt sie die grüne Farbkomponente auf 1.0 und erniedrigt die Zahl der enthaltenen Sandkörner um 4. In der zweiten Ansicht wird der grüne Anteil herausgefiltert, um für den Benutzer eine reine Darstellung zu gewinnen.

#### 3.4.1 Der Pixel Shader zu Sandbox

```
//Sandbox.psh
//Pixel shader for doing the logic of the Sandbox

//Declare pixel shader version
ps.2.0

dcl_2d      s0          //cell state, point sampling
```

```

dcl      t0.xy      //current cell
dcl      t1.xy      //4 texture coordinates to sample 4 neighbors
dcl      t2.xy
dcl      t3.xy
dcl      t4.xy

//Sandpile is in r component, overflow flag in g component
//def  c0, overflowValue, 0, 1, 0.0 (all transformed)

//get colors from 5 texture stages
texld r0, t0, s0      //this cell
texld r1, t1, s0      //straight neighbors
texld r2, t2, s0
texld r3, t3, s0
texld r4, t4, s0

//add up neighborhood
add      r0.r, r0.r, r1.g
add      r0.r, r0.r, r2.g
add      r0.r, r0.r, r3.g
add      r0.r, r0.r, r4.g

sub      r0.r, r0.r, c0.r //subtract overflowValue
cmp      r1, r0, c0.g, c0.r //r1 = (r0 > 0) 0:overflowValue
cmp      r2, r0, c0.b, c0.g //r2 = (r0 > 0) 1:0
add      r0.r, r0.r, r1.r //undo subtraction if necessary

mov      r0.g, c0.g      //set overflow flag to 0
mov      oC1, r0        //secondary output: without overflow flag

mov      r0.g, r2.r      //set overflow flag if necessary
mov      oC0, r0        //primary output: including overflow flag

```

### 3.5 Simulation von Logik-Gattern – Wire-World

Ein Zellularautomat namens *WireWorld* wird in [4] beschrieben und simuliert den Fluss von Elektronen durch Drähte und logische Verknüpfungen eben dieser durch verschiedentlich aufgebaute Gatter. Mit Hilfe diese Zellularautomaten lässt sich die Berechnungs-Universalität von Zellularautomaten noch einfacher beweisen als z. B. mit dem *Game of Life*.

Zur Simulation wird eine Hilfstextur verwendet. Zur eigentlichen Zustandsübergang dient nur die obere Hälfte. Mittels der unteren Hälfte der Textur wird die sekundäre Ausgabe erzeugt, die die originalen Farben der Beschreibung verwendet, sich aber nicht so gut zur Iteration eignen würde. Wie schon beim *Game of Life* wird auf der x-Koordinaten die Anzahl der roten Zellen in der Umgebung abgetragen, auf der y-Koordinate der aktuelle Zustand der Zelle. Um diese Summe effizient berechnen zu können, muss wiederum der zu akkumulierende Wert einen eigenen Farbkanal bekommen, in diesem Fall wird der rote genommen, um möglichst nahe am Original zu bleiben ( $r = 1.0$ , falls im roten Zustand, sonst  $r = 0.0$ ). In der blauen Farbkomponente wird der sonstige Zustand gespeichert.  $b = 0.0$  bedeutet hier, dass die Zelle Isolator ist (komplett schwarz). Ein Wert von  $b = 0.25$  zusammen mit  $r = 1$  kennzeichnet den roten Zustand einer Zellen („Kopf eines Elektrons“). Der „Schwanz des Elektrons“ wird kodiert mit  $b = 0.5$  und  $r = 0.0$ . Der letzte übrige Zustand ist  $b = 0.75$ ,  $r = 0.0$ , die Zelle ist also ein Leiter, bei dem sich aber gerade kein Elektron befindet.

#### 3.5.1 Der Pixel Shader zu WireWorld

```

//WireWorld.psh

//Pixel shader for doing the logic of the WireWorld

```

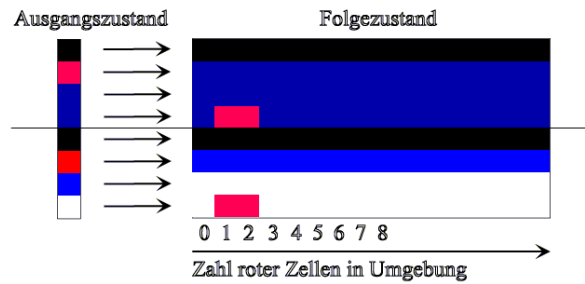


Abbildung 9: Die Hilfstextur für den WireWorld-Zellularautomaten.

```
//Declare pixel shader version
ps.2.0

dcl_2d    s0      //cell state, point sampling
dcl_2d    s1      //cell state, bilinear filtering
dcl_2d    s5      //decision support texture, point sampling

dcl       t0.xy   //current cell
dcl       t1.xy   //4 texture coordinates to sample 4 neighbors
dcl       t2.xy
dcl       t3.xy
dcl       t4.xy

//r: 0 = non-red, 1 = red
//b: 0 = black, non-conductor; 1/4 = red, electron head,
//   1/2 = blue, electron tail, 3/4 = white, conductor

//Multiplicative factor to isolate blue
//  RGBA
def  c0, 0.5, 0.5, 0.5, 0.5

//Additive factor to bias the red-green slightly (to avoid problems with tight texture coordinates)
def  c1, 0.01, 0.01, 0.0, 0.0

//Weights for neighbor pixels, 1/16 * 2
def  c2, 0.125, 0.0, 0.0, 0.0

//offset for nicer colors
def  c3, 0.0, 0.5, 0.0, 0.0

//get colors from 5 texture stages
texld r0, t0, s0      //this cell
texld r1, t1, s1      //straight neighbors
texld r2, t2, s1
texld r3, t3, s1
texld r4, t4, s1

mov r6, c1            //initialize with bias
mad r6.g, r0.b, c0, r6.g //mask out blue and put it in green

//and add the average of the neighbors according to weights in constant mem:
//r6 = fac * r1 + fac * r2 + fac * r3 + fac * r4
mad r6.r, c2, r1.r, r6
mad r6.r, c2, r2.r, r6
mad r6.r, c2, r3.r, r6
```

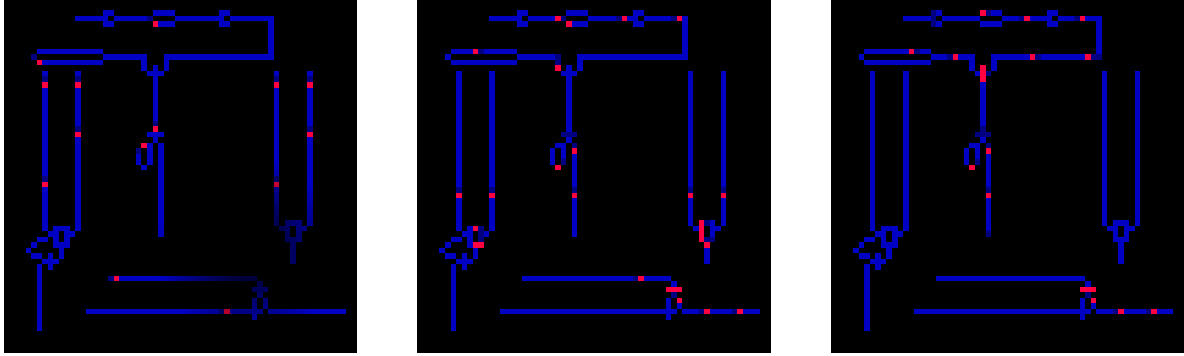


Abbildung 10: Links ein Beispiel-Ausgangszustand von WireWorld, in der Mitte nach einige Schritten, rechts im periodischen Endzustand

```

mad r6.r, c2, r4.r, r6
texld r5, r6, s5          //lookup decision texture

mov oC0, r5

add r6, r6, c3
texld r5, r6, s5

mov r6.gb, c3.r
mov oC1, r6              //secondary output = accumulated neighborhood redness

mov oC2, r5              //nicer colors

```

### 3.6 Simulation der Ausbreitung zweidimensionaler Wellen – WaveFront

Zur Simulation der Ausbreitung von Wellen in einem zweidimensionalen Medium lässt sich ein Zellularautomat verwenden, der in [4, S. 85f] beschrieben ist. Für die Approximation der zweiten räumlichen Ableitung wird folgende Formel verwendet:

$$\nabla^2 f \approx \frac{1}{4}(f[x-1, y] + f[x, y-1] + f[x+1, y] + f[x, y+1]) - f[x, y]$$

Für den Zustandsübergang wählen wir die zweite angegebene Möglichkeit:

$$Tf \approx 2f - T^-f + (\Delta t)\nabla^2 f$$

Dabei steht  $f[x, y]$  für die aktuelle Auslenkung am Punkt  $(x, y)$ ,  $Tf$  für den Wert in der nächsten Iteration und  $T^-f$  für den der vorhergehenden Iteration. Dieser wird also zusätzlich im grünen Farbkanal gespeichert, während die aktuelle Auslenkung den blauen Farbkanal beansprucht. Im roten Farbkanal wird der Typ der Zelle kodiert. Soll die Zelle als Erreger interagieren, so entspricht der eingetragene Wert  $n = \frac{\text{Schwingungen}}{\text{AnzahlIterationen}}$ . Im grünen Farbkanal wird dann die aktuelle Phase festgehalten, die auch vorbelegt sein kann. Im Spezialfall mit  $n = 0$  hat man eine stabile Wand. Da eine Wert  $n = 1$  sowieso keinen Sinn machen würde (eine komplette Schwingung pro Iteration), nimmt man diesen Wert als Kodierung für einen freien Schwinger.

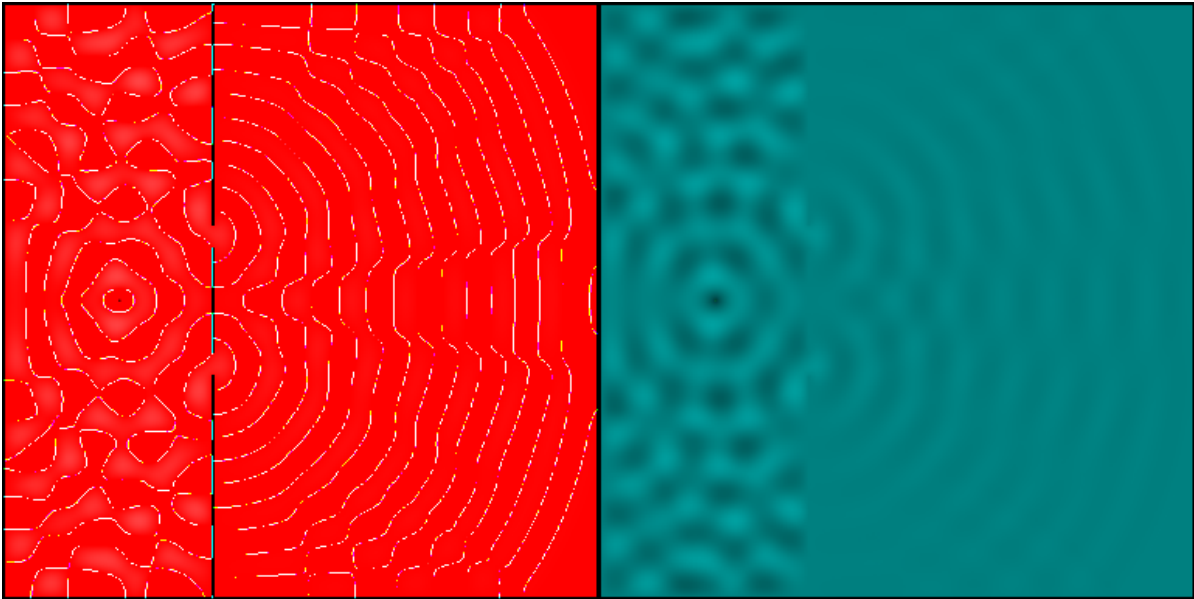


Abbildung 11: Ein Beispiellauf von *WaveFront* zur Simulation eines Doppelspalts.

Zur Speicherung der Werte haben sich 8-Bit-Ganzzahlen als zu grobkörnig erwiesen. Es ergeben sich keine kreisrunden Wellen mehr und es kommt zum Überlauf. Eine Genauigkeit von 16-Bit-Fließkomma (davon 10 Bit Mantisse) ist mindestens notwendig, doch hier „zerfließt“ die Welle sehr schnell und hat außerdem eine größere Wellenlänge als die der 32-Bit-Fließkomma-Version. Letztere ist dafür noch ein Stück langsamer. Zur Demonstration dieses Sachverhalts sind alle diese Versionen auswählbar.

### 3.6.1 Der Pixel Shader zu WaveFront

```
//WaveFront.psh
//
//Pixel shader for modelling travelling waves
//
//r:      0      wall
//      (0,1)  exciter with  $T \sim 1/r$  or wall ( $r = 0$ )
//      1      non-exciter
//g:      f(t-T) amplitude last step or current phase (exciter)
//b:      f(t)   amplitude

//Declare pixel shader version
ps.2.0

dcl_2d    s0      //cell state, point sampling
dcl_2d    s1      //cell state, bilinear filtering
dcl_2d    s5

dcl       t0.xy   //current cell
dcl       t1.xy   //4 texture coordinates to sample 8 neighbors
dcl       t2.xy   //by means of bilinear filtering
dcl       t3.xy
dcl       t4.xy

//def     c0, -1.0, 0.25, 0.1, 0.0159      //-1.0, 0.25, dFactor * deltat^2, T / 2*pi
def      c1, 0.5, 2.0, 0.0, 0.0          //0.5, 2.0, 0.0, 0.0
```

```

def      c2, 1.0, -1.0, 0.0, 0.0          //1.0, -1.0, 0.0, 0.0

//assume non-exciter

texld   r0, t0, s0                        //current cell
texld   r1, t1, s1                        //neighborhood
texld   r2, t2, s1
texld   r3, t3, s1
texld   r4, t4, s1

add     r1, r1, r2
add     r1, r1, r3
add     r1, r1, r4                        //r1 = sum straight/diagonal
mul     r1, c0.g, r1                      //scale straigh/diagonal
mad     r8.g, c0.r, r0.b, r1.b            //calculate new f'' [-1,1] = - 1 * middle + 1/4 * sum straight

mul     r5.b, c1.g, r0.b
sub     r5.b, r5.b, r0.g
mad     r5.b, c0.b, r8.g, r5.b

mov     r5.g, r0.b

mov     r5.r, r0.r
mov     r5.a, c2.a                        // ... = 0

//assume excitor or wall

mov     r6.r, r0.g                        //phase as texture coordinate
mov     r6.g, c2.a                        //... = 0
texld   r6, r6, s5
add     r6.g, r0.g, r0.r
frc     r6.g, r6.g                        //fmod(r6.g, 1.0f)
mov     r6.r, r0.r                        //still exciter
mov     r6.a, c2.a                        //... = 0

//final decision

sub     r7, r0.r, c2.r                    //... -1
cmp     r0, r7, r5, r6

mov     oC0, r0                           //primary output

mul     r1, r0, c1.r
add     r1, r1, c1.r
mov     r1.r, c2.a

mov     oC1, r1                           //secondary output: color shifted

```

### 3.7 Zellularautomaten zur Demonstration von Berechnungen (Randomize/Speed)

Zwei der auswählbaren Zellularautomaten haben keine Nachbarschaft und dienen nur zur Demonstration grundlegender Technik. *Randomize* bringt jede Zelle in jeder Iteration in einen zufälligen Zustand. Dies demonstriert die Einbringung von Zufallszahlen bei der Simulation stochastischer Zellularautomaten.

*Speed* implementiert die einfachste denkbare Überföhrungsfunktion: die Identität, der Zustand bleibt unverändert. Hiermit gewinnt man eine obere Schranke für die Berechnungsgeschwindigkeit des Programms.

### 3.8 Kommentierter Quelltext des gemeinsam verwendeten Vertex Shader

Für alle hier vorgestellten Anwendungsbeispiele wird ein gemeinsamer Vertex Shader verwendet, der wichtige Vorberechnungen erledigt. Nicht jede Anwendung würde alle davon benötigen, der Einfachheit halber werden aber immer alle durchgeführt. Die Ausführungszeit spielt keine Rolle, da pro Iteration nur 4 Vertices berechnet werden müssen.

```
//TexCoord4offset.vsh
//
//Vertex Shader for model-view-transformation and calculating some useful texture coordinates

vs.2.0
//Constants:
//c0-c3 - View+Projection matrix
//c4-c7 - Offsets

dcl_position v0
//following: 3 texture coordinates passed by the application
dcl_texcoord0 v1 //coordinate for current cell state texture
dcl_texcoord1 v2 //coordinate for random number texture
dcl_texcoord2 v3 //complex coordinate for Mandelbrot calculation

//Transform position
dp4 oPos.x, v0, c0 //standard 4x4 matrix multiplication for coordinate transformation
dp4 oPos.y, v0, c1
dp4 oPos.z, v0, c2
dp4 oPos.w, v0, c3

mov oT0, v1 //current cell coordinate remains unmodified
//offsets for neighborhood are added, maximum of 4

add oT1, c4, v1
add oT2, c5, v1
add oT3, c6, v1
add oT4, c7, v1
//pass unmodified

mov oT6, v2
mov oT7, v3
```

### 3.9 Benchmarks

Die Geschwindigkeitsmessungen fanden unter folgenden Bedingungen statt: Die Ausgabe des Ergebnisses auf den Bildschirm erfolgte während des Tests nicht oder nur in sehr großen Abständen. Ansonsten hätte sich ein unfairen Vorteil der Grafikkarte gegenüber ergeben, da die Grafikausgabe der CPU-Version in keinsten Weise optimiert ist.

Bei der Geschwindigkeitsmessung sind einige Punkte zu beachten:

- Die Messung sollte mit der Release- (GPU) bzw. mit der Topspeed-Version (CPU, Intel-Compiler notwendig) erfolgen.
- Die Version und Konfiguration der Grafikkarten-Treibers kann die Performance stark beeinflussen. Auf Dual-Monitor-Systemen läuft die Simulation u. U. nur dann mit voller Geschwindigkeit, wenn das Programmfenster sich auf dem primären Monitor befindet.

In folgender Tabelle finden sich die Ergebnisse der verschiedenen Geschwindigkeitsmessungen. Das Testsystem ist mit einem 2,53 GHz Pentium 4, 512 MB RAM sowie einer Grafikkarte mit ATI- Radeon-9700-Chip (Treiberversion CATALYST 3.6) ausgestattet. Alle Ergebnisse sind in Millionen Zellupdates pro Sekunde (MUpd/s) angegeben.

Zellularautomat	GPU 512×512	GPU 1024×1024	GPU 2048×2048	CPU 864×864	CPU 2048×2048
Game of Life	231	235	250	1750	-
FitzHugh-Nagumo	111	118	122	-	13
Mandelbrot	164	120	- <sup>1</sup>	-	-
Sandbox	120	124	- <sup>1</sup>	-	-
WireWorld	205	241	249	-	-
WaveFront32	92	104	- <sup>1</sup>	-	-
obere Schranke GPU	850	1340	1560	-	-

## 4 Technische Dokumentation

### 4.1 CellularAutomataGPU

#### 4.1.1 Verwendete Plattform

Die gewählte Betriebssystem-Plattform ist wie schon weiter oben begründet Microsoft Windows mit der 3D-Grafik-API DirectX in der Version 9.0b. Als Entwicklungsumgebung wurde Microsoft VisualStudio.net 2002 mit der Versionsverwaltung Microsoft Visual SourceSafe 6.0 verwendet. Das Projekt sollte sich auch einfach in das für die neue 2003er-Version notwendige Format konvertieren lassen, allerdings hat diese noch Probleme mit der Integration des Shader-Debuggings und wurde daher noch nicht verwendet.

Das Rahmenprogramm namens *CellularAutomataGPU* für die Berechnung auf der Grafikkarte wurde mit C# geschrieben. Diese Programmiersprache wird durch den Compiler in einen Bytecode umgewandelt, der erst direkt vor der Ausführung in Maschinencode umgesetzt wird. Diese Umsetzung kann allerdings von der Ausführungs-Geschwindigkeit her nicht mit hochoptimierenden C++-Compilern mithalten. Für die Berechnung auf der Grafikkarte spielt die „Langsamkeit“ von C# keine Rolle, da das Programm sowieso die meiste Zeit auf die Ausführung der Grafikkartenbefehle wartet.

#### 4.1.2 Systemvoraussetzungen

- DirectX-9-fähige Grafikkarte mit 128MB Speicher
- Windows XP
- DirectX 9.0b
- Managed DirectX
- .net Framework 1.0

#### 4.1.3 Fähigkeiten

Das Rahmenprogramm eignet sich bisher zur Implementierung von totalistischen Zellularautomaten (outer totalistic cellular automata) im zweidimensionalen Universum mit periodischen oder festen Randbedingungen. Eine Erweiterung des totalistischen Aspekts ist, dass die diagonalen Nachbarn unterschiedlich zur den direkten gewichtet werden können.



Abbildung 12: Screenshot der Oberfläche von CellularAutomataGPU.

#### 4.1.4 Bedienung des Programms

Das Bedienung des Programms spielt sich in einem Dialogfeld ab, das alle Einstellungsmöglichkeiten und die Ausgabe enthält.

Die fünf Buttons links oben steuern den Ablauf des Automaten. *Start* lässt die Simulation loslaufen, *Stop* hält sie an, wobei der Zustand erhalten bleibt. *Single Step* lässt die Simulation genau eine Iteration fortschreiten. Mit *Repaint* kann man evtl. auftretende Grafikfehler übermalen lassen. *Reset* schließlich bringt den Automaten in einen Ausgangszustand (evtl. nicht eindeutig, sondern zufällig) zurück.

Neben der Steuerung wird die Leistungsstatistik über die letzten 1000 Generationen angezeigt. Es dauert daher unter Umständen einige Sekunden, bis eine erste Anzeige erscheint. Die Einheiten sind Bilder pro Sekunde (frames per second, fps) und Millionen Zellberechnungen pro Sekunde (millions of updates per second).

Die Auswahlbox rechts oben schaltet zwischen allen verfügbaren Zellularautomaten um, der aktuelle Zustand geht dabei analog zu *Reset* verloren.

Hinter der Beschriftung *Size* lässt sich die Kantenlänge des Universums (immer quadratisch) in Zweierpotenzen auswählen. Unabhängig von dieser Größe wird die Darstellung immer auf die volle Größe der Anzeigefläche aufgezogen.

Hinter *Show every ... frame* lässt sich mit Eingabe einer ganzen Zahl und Klick auf den entsprechenden *Set*-Button die Anzahl der Generationen festlegen, nach der der aktuelle Zustand wieder auf den Bildschirm gezeichnet wird. Dies ist bei Benchmarks sinnvoll, da die Visualisierung auf dem Bildschirm doch etwas Zeit kostet. Wird aber z. B. nur jede 100. Generation

<sup>1</sup>Test nicht möglich auf Grund unzureichenden Speichers

angezeigt, so fällt das im Durchschnitt praktisch nicht mehr ins Gewicht.

Rechts und vertikal in der Mitte befindet sich der so genannte PixelInspector™. Ein Klick auf irgendeine Zelle im Universum führt zur Anzeige der aktuellen Zahlenwerte aller vier Farbkanaäle. Dabei steht in der unteren Zeile der direkt ausgelesene Wert, darüber der für den Automaten sinnvoll umgerechnete. Man kann die Werte auch von Hand verändern und mit einem Klick auf das entsprechende *Set* übernehmen lassen.

Neben der Beschriftung *Device* lässt sich der zu verwendende Typ des DirectX-Devices angeben. Bei „Hardware“ passiert das mit höchstmöglicher Geschwindigkeit auf der Grafikkarte, wogegen „Reference“ alles auf der CPU emuliert und deshalb extrem langsam ist. Vorteile von letzterem: Man benötigt keine DirectX-9-unterstützende Grafikkarte und außerdem ist es möglich, sich den Ablauf der Shader in einem speziellen Debugger anzuschauen.

Ein Zellularautomat kann evtl. mehrere Ausgaben hervorbringen (maximal 4). Der Anzeigebereich wird dann in Quadranten aufgeteilt. Mit der Auswahlbox rechts neben *# of planes* kann festgelegt werden, wie viele von diesen letztlich angezeigt werden sollen. Dabei wird immer von vorne durchgegangen.

Der Klick auf *Load Image...* öffnet eine Dateiauswahlbox, in der man eine BMP-Datei zum Laden in die Zustands-Textur (Plane 1) aussuchen kann.

Mit *Save Image of plane...* kann man letztlich Screenshots der Ausgaben bequem abspeichern. Unterhalb der ganzen Bedienungsschnittstelle befindet sich der Ausgabebereich.

#### 4.1.5 Modularität und Erweiterbarkeit

*CellularAutomataGPU* ist leicht um neue Zellularautomaten erweiterbar. In den meisten Fällen muss dabei nichts am eigentlichen Code geändert werden, sondern nur eine Konfiguration-Klasse des Typs *CAConfiguration* hinzugefügt werden, dass die zur verwendenden Pixel Shader und weitere Parameter festlegt. Es folgt eine Auflistung der zu initialisierenden Werte von *CAConfiguration* in der Reihenfolge gleich dem Konstruktoraufruf:

**name** (*string*) Der Name des Zellularautomaten zur Anzeige in der Auswahlbox.

**width** (*int*), **height** (*int*) Breite und Höhe des Zelluniversums in Zellen.

**numPlanes** (*int*) Anzahl der benötigten Ausgabe-Puffer (1–4). Der erste Puffer wird immer zur Weiterberechnung verwendet. Die anderen Puffer sind optional und können vom Benutzer auch deaktiviert werden. Sie dienen ausschließlich zur Anzeige des Zustands in anderer, meistens anschaulicherer Form.

**texelFormat** (*Direct3D.Format*) Gewünschtes Format der zur Speicherung des Zustands verwendeten Texel. Verwendet werden hier bisher *Format.A32B32G32R32F* (32-bit Fließkommazahl pro Farbkomponente) und *Format.A8R8G8B8* (8-bit Ganzzahl pro Farbkomponente).

**boundaryConditions** (*Direct3D.TextureAddress*) Bestimmt die Behandlung der Randbedingungen: *TextureAddress.Wrap* (periodische Randbedingungen), *TextureAddress.Clamp* (Rand wird fortgesetzt), *TextureAddress.Border* (Fortsetzung mit bestimmter Farbe, wurde von verwendeter Grafikkarte leider nicht unterstützt und daher nicht verwendet).

**border** (int) Breite des Streifens in Pixeln, der rundherum am Rand beim Zustandsübergang nicht berechnet wird und damit unverändert bleibt. Der Rand wird allerdings trotzdem als Nachbarschaft berücksichtigt und eignet sich daher zur Implementierung fester Randbedingungen.

**ratioStraightDiagonal** (Single) Anteil der direkten vier Nachbarn bei der Aufsummierung über die Nachbarschaft im Gegensatz zu den diagonalen Nachbarn. Z. B. bedeutet 1.0, dass nur die direkten Nachbarn (geradeaus) betrachtet werden; 0.0 bewirkt die ausschließliche Berücksichtigung der diagonalen nächsten Nachbarn, mit 0.5 ergibt sich eine gleichgewichtete Summe über alle 8 Nachbarn.

**initShader** (string) Zu verwendender Pixel Shader für die Initialisierung des Zelluniversum zu Beginn der Simulation. Anzugeben ist der Dateiname des Quelltextes ohne die Endung `.psh`, der sich im Verzeichnis `Pixel Shader` zwei Ebenen höher wie das Hauptprogramm befinden muss.

**initTexture** (string) Dateiname des Bitmaps zum Anfangszustand. Es wird dem **initShader** in Textur-Stage 0 zur Verfügung gestellt.

**initConstants** (Vector4[]) Vor der Ausführung zu setzenden Konstanten für den **initShader**.

**transShader** (string) Zu verwendender Pixel Shader für den Zustandsübergang. Konventionen siehe **initShader**.

**ruleTexture** (string) Dateiname einer Hilfstextur. Sie wird dem Pixel Shader in Textur-Stage 5 zur Verfügung gestellt. Wird ein leerer String übergeben, wird keine Hilfstextur geladen. Sonderfall: Wird „Sinus“ übergeben, so wird keine Textur geladen, sondern eine eindimensionale Hilfstextur mit einem sinusförmigen Grauverlauf einer kompletten Periode generiert.

**transConstants** (Vector4[]) Vor der Ausführung zu setzenden Konstanten für den **transShader**.

**fromPixelToValue** (RangeTransform[4]) Definiert für jede Farbkomponente eine Transformation, die vor der Anzeige des Wertes in der grafischen Oberfläche angewendet wird und so eine für den Benutzer anschaulichere Darstellung erlaubt. Das vordefinierte `identity` kann verwendet werden, wenn keine Umwandlung gewünscht bzw. sinnvoll ist.

#### 4.1.6 Beschreibung einiger Klassen

**CAConfiguration** Siehe oben.

**CellularAutomaton** Wichtigste Klasse. Wird durch **CAConfiguration** konfiguriert und kapselt die komplette Simulation und Anzeige des Zellularautomaten.

**Float16** Wrapper-Klasse für den 16-Bit-Fließkomma-Typ von DirectX.

**MainForm** Die Klasse des Hauptprogramms; zuständig für die GUI und das Multithreading.

**RangeTransform** Dient zur linearen Umrechnung von Werten zwischen zwei Intervallen, z. B. von  $[-1, 1]$  nach  $[0, 255]$ .

HiPerfTimer Implementiert einen hochauflösenden Zeitmesser auf Basis einiger Win32-Systemfunktionen.

#### 4.1.7 Bekannte Probleme

Es kann vorkommen, dass das Programm beim Schließen des Fensters nicht wirklich terminiert und den Prozessor voll auslastet. Der Zombie-Prozess muss dann mittels des Taskmanagers beendet werden.

Außerdem übersteht das Programm nicht immer den Verlust des DirectX-Devices, der z. B. bei Verwendung der schnellen Benutzerumschaltung auftritt.

#### 4.1.8 Verbesserungs-/Erweiterungsmöglichkeiten

- *Dreidimensionale Zellularautomaten* ließen sich theoretisch in dreidimensionalen Texturen speichern. Allerdings ist es wohl nicht möglich, diese direkt wieder als ein Ergebnis eines Rendering-Durchgangs auszugeben. Es könnte jedoch durch iteratives Rendern jeder einzelnen Schicht gelingen. Bisher wurde das Thema hier aber noch nicht weiter untersucht.
- Durch die Verwendung der *High Level Shader Language* ließen sich besser lesbare Programm für die Pixel Shader schreiben. Jedoch war diese zu Beginn des Projekts noch nicht verfügbar und wurde daher nicht berücksichtigt.

## 4.2 CellularAutomataCPU

Für die Vergleichsalgorithmen des Programms *CellularAutomataCPU*, die die Simulation der Zellularautomaten auf dem Prozessor durchführen, wurde der hochoptimierungs-fähige Intel-C++-Compiler in der Version 7.1 herangezogen. Dabei wurde die höchste Optimierungsstufe aktiviert. Das Programm benutzt die MFC als Klassenbibliothek und ergibt Win32-Binärcode.

*CellularAutomataCPU* wurde unter Verwendung der Microsoft Foundation Classes inkl. deren Document-View-Model erstellt.

### 4.2.1 Beschreibung einiger Klassen

`CCellularAutomataCPUApp` Die Klasse des Hauptprogramms.

`CCellularAutomataCPUDoc` Die Dokumentenklasse. Wichtigster Member ist der zu simulierende Zellularautomat `ca`.

`CCellularAutomataCPUView` Dient zur grafischen Ausgabe der Automaten.

`CCellularAutomaton` Abstrakte Basisklasse aller Zellularautomaten-Implementierungen, hier eben von `CGameOfLife`, `CFitzHughNagumo` und `CWaveFront`.

`CIterator<s>` Interface aller Iteratoren (Klassen, die die Berechnung der nächsten Generation übernehmen). Der Template-Parameter `s` definiert den Typ der Zustandsvariablen einer Zelle.

`C...Iterator` Diverse Iteratoren zu den verschiedenen Zellularautomaten.

`RangeTransform` Siehe oben.

#### 4.2.2 Systemvoraussetzungen

- Windows 98 SE
- Intel-C++-Compiler 7.1 (für die volle Geschwindigkeit)

#### Literatur

- [1] Kelly Dempski. Real-Time Rendering Tricks And Techniques in DirectX. Premier Press 2002.
- [2] Andreas Solymosi, Peter Solymosi. Effizient Programmieren mit C# und .net. Vieweg 2001.
- [3] Andreas Merkle. Theoretische und praktische Untersuchungen an stochastischen Zellularautomaten. Diplomarbeit 1997.
- [4] Thomas Worsch. Algorithmen in Zellularuatomaten. Skript zur Vorlesung. Universität Karlsruhe 2001.

# Inhaltsverzeichnis

<b>1</b>	<b>Einführung</b>	<b>3</b>
1.1	Grundbegriffe aus der 3D-Grafik-Programmierung . . . . .	3
1.1.1	Begriffserläuterungen . . . . .	4
1.2	Vor- und Nachteile bei Berechnung auf der Grafikhardware . . . . .	5
<b>2</b>	<b>Umsetzung für die Grafikhardware</b>	<b>6</b>
2.1	Vertex Shader . . . . .	6
2.2	Pixel Shader . . . . .	7
2.3	Umsetzung der Eigenschaften eines Zellularautomaten . . . . .	8
2.3.1	Zufall . . . . .	8
2.3.2	Verwendete Assembler-Befehle und -Direktiven . . . . .	9
<b>3</b>	<b>Beispiel-Implementierungen</b>	<b>10</b>
3.1	Game of Life . . . . .	10
3.1.1	Umsetzung . . . . .	11
3.1.2	Geschwindigkeitsvergleich . . . . .	12
3.1.3	Der Pixel Shader zum Game of Life . . . . .	12
3.2	Reaktions-Diffusions-Prozess nach FitzHugh-Nagumo . . . . .	13
3.2.1	Geschwindigkeits-Vergleich . . . . .	14
3.2.2	Der Pixel Shader zu FitzHugh-Nagumo . . . . .	15
3.3	Mandelbrot-Fraktal . . . . .	16
3.3.1	Der Pixel Shader zum Mandelbrot-Fraktal . . . . .	16
3.4	Simulation von Überlauf-Kaskaden mittels des Sandbox-Zellularautomaten . . . . .	18
3.4.1	Der Pixel Shader zu Sandbox . . . . .	18
3.5	Simulation von Logik-Gattern – Wire-World . . . . .	19
3.5.1	Der Pixel Shader zu WireWorld . . . . .	19
3.6	Simulation der Ausbreitung zweidimensionaler Wellen – WaveFront . . . . .	21
3.6.1	Der Pixel Shader zu WaveFront . . . . .	22
3.7	Zellularautomaten zur Demonstration von Berechnungen (Randomize/Speed) . . . . .	23
3.8	Kommentierter Quelltext des gemeinsam verwendeten Vertex Shader . . . . .	24
3.9	Benchmarks . . . . .	24
<b>4</b>	<b>Technische Dokumentation</b>	<b>25</b>
4.1	CellularAutomataGPU . . . . .	25
4.1.1	Verwendete Plattform . . . . .	25
4.1.2	Systemvoraussetzungen . . . . .	25
4.1.3	Fähigkeiten . . . . .	25
4.1.4	Bedienung des Programms . . . . .	26
4.1.5	Modularität und Erweiterbarkeit . . . . .	27
4.1.6	Beschreibung einiger Klassen . . . . .	28
4.1.7	Bekannte Probleme . . . . .	29
4.1.8	Verbesserungs-/Erweiterungsmöglichkeiten . . . . .	29
4.2	CellularAutomataCPU . . . . .	29
4.2.1	Beschreibung einiger Klassen . . . . .	29
4.2.2	Systemvoraussetzungen . . . . .	30