Darstellung von Differentialgleichungen mit einem zellulären Automaten

Detlev Ziereisen, Florian Lüthi, i10b

12. Juni 2012

Inhaltsverzeichnis

1 Einleitung

Diese Projektarbeit (bestehend aus diesem Dokument sowie der zugehörigen Software `Tusk`) hat zum Ziel, partielle Differentialgleichungen aus dem Umfeld der physikalischen Simulationen mit einem zellulären Automaten zu berechnen sowie darzustellen. Es soll gezeigt werden, dass dies unter den konzeptionellen Restriktionen der Theorie der zellulären Automaten möglich ist, insbesondere soll die Annahme Konrad Zuses, dass die Naturgesetze diskreten Regeln folgten und das gesamte Geschehen im Universum das Ergebnis der Arbeit eines gigantischen zellulären Automaten sei [22, 24], zumindest für den Scope dieser Arbeit bestätigt werden. Ausserdem soll gezeigt werden, dass die intuitiv erwarteten Resultate in sinnvoller Rechenzeit dargestellt werden können.

Diese Projektarbeit wurde im Rahmen des Kurses Softwareprojekt 2 an der Hochschule für Technik Zürich durchgeführt. Der Quellcode sowie auch dieses Dokument sind Online auf `http://www.github.com/foyan/Tusk` zu finden.

Das vorliegende Dokument ist so gegliedert, dass im folgenden Kapitel die nötigen theoretischen Grundsteine über zelluläre Automaten, numerische Lösung von Differentialgleichungen sowie die Verknüpfung dieser beiden Aspekte gelegt werden. Daraufhin werden ausgewählte Bereiche der Implementation kommentiert, und zu guter Letzt folgt ein Résumé, das sich mit der Beantwortung oder Nichtbeantwortbarkeit der eingangs gestellten Fragen und Thesen beschäftigt.

Der Autoren Dank geht (in alphabetischer Reihenfolge der Vornamen) an Albert Heuberger für geballte mathematische Kompetenz gepaart mit unendlicher Geduld, an Lukas Eppler für die ständige Begleitung, sowie an Philippe Nahlik für die tolle Kursführung.

Teil I
Theorie

2 Zelluläre Automaten

> Ein zellulärer Automat ist eine regelmäßige Annordnung von Zellen. Jede Zelle kann eine endliche Zahl von Werten / Zuständen annehmen und hat eine begrenzte Zahl von Nachbarzellen, die sie beeinflussen können. Das Muster des gesamten zellulären Automaten ändert sich in einzelnen Schritten, die

durch eine Reihe von Übergangsregeln bestimmt werden, die für alle Zellen gelten.[2]

Also:

Definition 1 (Zellulärer Automat)**.** Ein zellulärer Automat ist durch folgende Eigenschaften festgelegt:

- einen Zellularraum R,
- eine endliche Nachbarschaft N, wobei $\forall r \in R\,(N_r \subset R)$,
- eine Zustandsmenge Q,
- eine Überführungsfunktion $\delta : Q^{|N|+1} \mapsto Q$.

Die Zustandsübergänge erfolgen für alle Zellen nach derselben Überführungsfunktion und gleichzeitig. Die Zellzustände können wie die Zeitschritte diskret sein. [22]

Bemerkung. Aus dieser Definition folgt unmittelbar, dass der neue Zustand einer Zelle nur vom momentanen Zustand dieser Zelle sowie den Zuständen der Nachbarzellen abhängig sein kann.

Für zweidimensional organisierte R sind zwei Arten von Nachbarschaft üblich:

Von-Neumann-Nachbarschaft Die Nachbarschaft besteht jeweils aus den vier geographisch nächsten Nachbarzellen: $N_{i,j} = \{R_{i,j-1}, R_{i,j+1}, R_{i-1,j}, R_{i+1,j}\}$

Moore-Nachbarschaft Die Nachbarschaft besteht jeweils aus allen Zellen der Von-Neumann-Nachbarschaft sowie zusätzlich der diagonalen Nachbarzellen:

$N_{i,j} = \{R_{i,j-1}, R_{i,j+1}, R_{i-1,j}, R_{i+1,j}, R_{i-1,j-1}, R_{i-1,j+1}, R_{i-1,j+1}, R_{i+1,j+1}\}$ [12]

Beispiel (Wolfram's eindimensionales Universum)**.** Stephen Wolfram[1] definiert in [23] und in etlichen Arbeiten aus der Mitte der 1980er-Jahre einen parametrierbaren zellulären Automaten, der nur aus einer einzigen Raumdimension besteht[20]. In seiner einfachsten Ausprägung ist die Nachbarschaft N_i definiert als $\{R_{i-1}, R_{i+1}\}$, und jede Zelle kann genau zwei Zustände annehmen (tot und lebendig bzw. 0 und 1). Damit ist

$$\delta : \{0,1\}^{|\{R_{i-1},R_{i+1}\}|+1} \mapsto \{0,1\} = \{0,1\}^3 \mapsto \{0,1\}\,,$$

ergo existieren 8 mögliche Zustandsänderungen. Ein Beispiel:

Man stellt nun fest, dass bei 8 möglichen Zustandsänderungen für die Anzahl der möglichen Konfigurationen gilt:

$$|K| = |Q|^{|\,\mathrm{dom}(\delta)|} = 2^8 = 256.$$

alte $R_{i-1}R_iR_{i+1}$	111	110	101	100	011	010	001	000
neues R_i	0	1	1	0	1	1	1	0

Tabelle 1: Wolfram-Konfiguration 110

111	110	101	100	011	010	001	000	Zahl
0	0	0	0	0	0	0	0	$00000000_b = 0_d$
0	0	0	0	0	0	0	1	$00000001_b = 1_d$
0	0	0	0	0	0	1	0	$00000010_b = 2_d$
					$\vdots$			
0	1	1	0	1	1	1	0	$01101110_b = 110_d$
					$\vdots$			
1	1	1	1	1	1	1	1	$11111111_b = 255_d$

Tabelle 2: Alle Wolfram-Konfigurationen

Jeder dieser Konfigurationen kann eine natürliche Zahl $\{0, 1, 2, \ldots, 255\}$ zugeordnet werden, indem die neuen R_i wie oben tabelliert und pro Zeile als binäre Zahl aufgefasst werden [3]:

Dadurch ist es möglich, sämtliche eindimensionalen Wolfram-Universen durch eine einzige Zahl zu identifizieren.

Als besonders spannend hat sich die in Tabelle 1 dargestellte Konfiguration 110 erwiesen, weil sie die Eigenschaft hat, ein Turing-vollständiges System zu sein [3, 23]. Dadurch ist sie die Konfiguration einer universellen Turingmaschine, die mit nur 2 Zuständen und 5 Symbolen umgesetzt werden kann – somit hat die Wolfram-Konfiguration 110 als Turingmaschine einen Umfang von $2 \cdot 5 = 10$ und zählt damit zu den kleinsten bis dato bekannten Turingmaschinen [3].

Beispiel (Game of Life)**.** Das von Conway[2] 1970 entworfene Game of Life ist eine bis heute populäre Umsetzung der Automatentheorie und insbesondere der Idee der zellulären Automaten [14].

Der ursprüngliche Entwurf befindet sich in einem zweidimensionalen R unter Verwendung der Moore-Nachbarschaft. Die Zellen können zwei mögliche Zustände $\{q_{\text{lebend}}, q_{\text{tot}}\}$

[1]Stephen Wolfram (* 29. August 1959), britischer Physiker und Mathematiker, Schöpfer der Software Mathematica sowie der Suchmaschine Wolfram Alpha [20]

[2]John Horton Conway (* 26. Dezember 1937), englischer Mathematiker[16]

annehmen, und die Übergangsfunktion ist definiert [5] als:

$$\delta(r,N) = \begin{cases} q_{\text{lebend}} & (r = q_{\text{tot}} \wedge \varsigma(q_{\text{lebend}}, N) = 3) \\ q_{\text{tot}} & (r = q_{\text{lebend}} \wedge \varsigma(q_{\text{lebend}}, N) < 2) \\ q_{\text{lebend}} & (r = q_{\text{lebend}} \wedge 2 \leq \varsigma(q_{\text{lebend}}, N) \leq 3) \\ q_{\text{tot}} & (r = q_{\text{lebend}} \wedge \varsigma(q_{\text{lebend}}, N) > 3) \\ q_{\text{tot}} & (\text{sonst}) \end{cases}$$

unter Zuhilfenahme der Statuszählfunktion

$$\varsigma(q,N) = \sum_{i=1}^{8} \begin{cases} 1 & (N_i = q) \\ 0 & (N_i \neq q) \end{cases}$$

3 Vom zellulären Automaten zur Differentialgleichung

Die Berechnung und Darstellung physikalischer Begebenheiten (allgemein ausgedrückt durch folgende partielle Differentialgleichung, mit $u : u(\vec{x}, t)$)

$$k_n \frac{\partial^n u}{\partial t^n} + k_{n-1} \frac{\partial^{n-1} u}{\partial t^{n-1}} + \cdots + k_1 \frac{\partial u}{\partial t} + k_0 = \frac{\partial^n u}{\partial \vec{x}^n} + \frac{\partial^{n-1}}{\partial \vec{x}^{n-1}} + \cdots + \frac{\partial u}{\partial \vec{x}}$$

mit zellulären Automaten kann durchgeführt werden, indem folgendes getan wird:

- R entspricht einer sinnvollen (groben) Diskretisierung der örtlichen Variablen $\vec{x}$ in einer, zwei oder drei Dimensionen
- Jede Zelle in R ist ein Tupel (Q, D) mit Q als einer Menge von berechnungsfernen Zustandsinformationen und den Differentialen nach der Zeit

$$D = \left(u, \frac{\partial u}{t}, \frac{\partial^2 u}{\partial t^2}, \cdots, \frac{\partial^n u}{t^n} \right) \in \mathbb{R}^n$$

- Eine neue Generation entspricht jeweils der fortgelaufenen Zeit ∂t, welche sehr fein diskretisiert werden muss
- In der Übergangsfunktion δ steckt die eigentliche Differentialgleichung. In der Regel verändert sie nur die Elemente von D. Sollte die Differentialgleichung Terme mit verschiedenen Ordnungen enthalten, wird die Gleichung unter Zuhilfenahme entsprechender Hilfsgleichungen $\lambda_1, \lambda_2, \ldots, \lambda_n$ in ein äquivalentes System gewöhnlicher Differentialgleichungen umgeformt.

4 Numerische Lösung von Differentialgleichungen

Damit die numerische Lösung von Differentialgleichungen gelingt, sind vor allem zwei Fertigkeiten vonnöten: Die numerische Differenzierung sowie die numerische Integration. Wir wollen beide Gebiete kurz streifen, und zwar unter den für uns nützlichen Gesichtspunkten bezüglich der Wahl der Variablen sowie der vorliegenden impliziten Funktionen (beziehungsweise diskreten Werten als Zellinhalte des zellulären Automaten).

4.1 Differenzierung nach dem Ort

Wir starten mit den gegebenen Werten $u_{\vec{x}}$ (den Zuständen der Zellen aus R). Daraus erhalten wir die diskrete Funktion

$$u : R \mapsto \mathbb{R}, u(\vec{x}) = u_{\vec{x}}$$

welche durch komponentenweises Einsetzen des Differenzenquotienten $\frac{\Delta u}{\Delta \vec{x}}$ folgendermassen differenzierbar ist (unter Zuhilfenahme des Einheitsvektors $\vec{e}_i$ für jede Komponente von $\vec{x}$ sowie der Erkenntnis, dass durch die diskrete Ausgangsfunktion $\Delta\vec{x}$ komponentenweise 1 ist; die Interpretation, wo genau sich die Nachbarzelle $\vec{x} \pm n \cdot \vec{e}$ befindet, überlassen wir der für den zellulären Automaten gültigen Nachbarschaftsfunktion):

$$\left(\frac{\Delta}{\Delta\vec{x}}u\right)_{\vec{x}} = u_{\vec{x}} - \sum_{i=1}^{\dim(\vec{x})} u_{\vec{x}-\vec{e}_i} \tag{1}$$

Diese Differenzen können natürlich wiederum differenziert werden:

$$\left(\frac{\Delta}{\Delta\vec{x}}\left(\frac{\Delta}{\Delta\vec{x}}u\right)_{\vec{x}}\right)_{\vec{x}} = \left(\frac{\Delta}{\Delta\vec{x}}u\right)_{\vec{x}} - \sum_{i=1}^{\dim(\vec{x})}\left(\frac{\Delta}{\Delta\vec{x}}u\right)_{\vec{x}-\vec{e}_i}$$

Durch Einsetzen von 1 in die obige Gleichung ergibt sich dann für die Differenz zweiter Ordnung:

$$\begin{aligned}
\left(\frac{\Delta^2}{\Delta\vec{x}^2}u\right)_{\vec{x}} &= u_{\vec{x}} - \sum_{i=1}^{\dim(\vec{x})} u_{\vec{x}-\vec{e}_i} - \sum_{i=1}^{\dim(\vec{x})}\left(u_{\vec{x^*}} - \sum_{i=1}^{\dim(\vec{x})} u_{\vec{x^*}-\vec{e}_i}\right)_{\vec{x^*}:=\vec{x}-\vec{e}_i} \\
&= u_{\vec{x}} - \sum_{i=1}^{\dim(\vec{x})} u_{\vec{x}-\vec{e}_i} - \sum_{i=1}^{\dim(\vec{x})} u_{\vec{x}-\vec{e}_i} + \sum_{i=1}^{\dim(\vec{x})} u_{\vec{x}-2\vec{e}_i} \\
&= u_{\vec{x}} - 2\cdot\sum_{i=1}^{\dim(\vec{x})} u_{\vec{x}-\vec{e}_i} + \sum_{i=1}^{\dim(\vec{x})} u_{\vec{x}-2\vec{e}_i}
\end{aligned}$$

Bemerkung (Indizes)**.** Da der Definitionsbereich der besprochenen Funktion diskret ist, verringert sich die Kardinalität des Definitionsbereichs der Differenzen mit fortschreitender Ordnung jeweils um 1 pro Dimension. Geographisch gesprochen, liegt die Differenz zweier Zellen ja eigentlich auf der gemeinsamen Kante dieser zwei Zellen. Daher ist es in der Praxis sinnvoll, die Differenzen gerader Ordnungen (bei deren Berechnung wie eben gesehen 3 Zellen pro Dimension involviert sind) jeweils in die Zelle in der Mitte zu schieben. Im Falle der zweiten Ordnung ergibt sich damit:

$$\left(\frac{\Delta^2}{\Delta\vec{x}^2}u\right)_{\vec{x}} = \sum_{i=1}^{\dim(\vec{x})} u_{\vec{x}-\vec{e}_i} - 2\cdot u_{\vec{x}} + \sum_{i=1}^{\dim(\vec{x})} u_{\vec{x}+\vec{e}_i} \tag{2}$$

4.2 Integration nach der Zeit mit Einschrittverfahren

4.2.1 Forderung

Wir starten mit den gegebenen Werten $\left(\frac{\Delta}{\Delta\vec{x}}u\right)_{\vec{x}}$ (den vorgängig berechneten Differenzen zwischen den Zellen). Wir definieren die allgemeine Integrationsfunktion

$$\mathbf{int} : \mathbb{R}^2 \mapsto \mathbb{R}$$

für welche jeweils gelten muss:

$$\mathbf{int}\left(\left(\frac{\Delta}{\Delta\vec{x}}u\right)_{\vec{x},t}, h\right) \approx \int_t^{t+h} u_{\vec{x}}\mathrm{d}t$$

Da die $u_{\vec{x}}$ zu jedem Zeitpunkt t (also pro Zeitschritt h beziehungsweise $\mathrm{d}t$) jeweils von ihren Nachbarzellen abhängen, ist es vonnöten, sämtliche Zellen für einen einzigen Zeitschritt simultan durchzurechnen.

4.2.2 Die impliziten Runge-Kutta-Verfahren

Da uns hier nur Einschrittverfahren interessieren, schauen wir uns im Detail die Klasse der Runge-Kutta-Verfahren[3][4] an, welche auf dem expliziten Euler-Verfahren[5] basieren und durch Butcher-Tableaux[6] allgemein definiert werden können.

Sämtliche Runge-Kutta-Verfahren können folgendermassen dargestellt werden:

$$y_{n+1} = y_n + h \cdot \sum_{j=1}^{s} \gamma_j \cdot k_j$$

wobei h die gewählte Schrittweite, γ_j die charakteristischen Koeffizienten des gewählten Verfahrens sowie k_j die Auswertungen der zu integrierenden Funktion f an bestimmten Stützstellen repräsentieren. Für k_j gilt dann:

$$k_j = f\left(t_n + h \cdot \alpha_j, y^n + h \cdot \sum_{i=1}^{m} \beta_{j,i} \cdot k_i\right)$$

wobei α_j sowie $\beta_{j,i}$ wiederum charakteristische Koeffizienten sind [7, 19], welche zusammen mit den γ_j in einem Butcher-Tableau folgendermassen angeordnet werden können:

$$\left[\begin{array}{c|c} a & B \\ \hline & c \end{array}\right] = \left[\begin{array}{c|ccc} \alpha_1 & \beta_{1,1} & \beta_{1,2} & \cdots & \beta_{1,m} \\ \alpha_2 & \beta_{2,1} & \beta_{2,2} & \cdots & \beta_{2,m} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \alpha_m & \beta_{m,1} & \beta_{m,2} & \cdots & \beta_{m,m} \\ \hline & \gamma_1 & \gamma_2 & \cdots & \gamma_m \end{array}\right]$$

[3] Carl David Tolmé Runge (1856–1927), deutscher Mathematiker [13]
[4] Martin Wilhelm Kutta (1867–1944), deutscher Mathematiker [18]
[5] Leonhard Euler (1707–1783), Schweizer Mathematiker und Physiker [17]
[6] John Charles Butcher (*1933), neuseeländischer Mathematiker [15]

Runge-Kutta-Verfahren sind im Allgemeinen implizit, weil zur Bestimmung der k_j sowie y^{n+1} ein (mindestens lineares) $(m+1) \times (m+1)$-Gleichungssystem gelöst werden muss; dies verursacht natürlich entsprechende Kalamitäten. Solchartige Verfahren sind aber sehr stabil.[11]

4.2.3 Die expliziten Runge-Kutta-Verfahren

Ist es nun möglich, Koeffizienten zu finden, deren Matrix B strikte nilpotente untere Dreiecksgestalt hat, so spricht man von einem expliziten Verfahren, weil sich das Gleichungssystem einfach durch Rückwärtseinsetzen lösen lässt (sogar wenn es nichtlinear ist):

$$\left[\begin{array}{c|c} a & B \\ \hline & c \end{array}\right] = \left[\begin{array}{c|cccc} \alpha_1 & 0 & & & \\ \alpha_2 & \beta_{2,1} & 0 & & \\ \vdots & \vdots & \vdots & \ddots & \\ \alpha_m & \beta_{m,1} & \beta_{m,2} & \cdots & 0 \\ \hline & \gamma_1 & \gamma_2 & \cdots & \gamma_m \end{array}\right]$$

Für ausgewählte explizite Verfahren lassen sich nun sowohl Butcher-Tableau als auch die geforderte Integrationsfunktion **int** folgendermassen aufstellen:

Euler

$$\begin{aligned} \left[\begin{array}{c|c} a & B \\ \hline & c \end{array}\right] &= \left[\begin{array}{c|c} 0 & 0 \\ \hline & 1 \end{array}\right] \\ \Rightarrow \mathbf{int}\left(\left(\frac{\Delta}{\Delta\vec{x}}u\right)_{\vec{x},t}, h\right) &= u_{\vec{x},t} + h \cdot \left(\frac{\Delta}{\Delta\vec{x}}u\right)_{\vec{x},t} \end{aligned}$$

Runge-Kutta 2. Ordnung

$$\begin{aligned} \left[\begin{array}{c|c} a & B \\ \hline & c \end{array}\right] &= \left[\begin{array}{c|cc} 0 & 0 & \\ \frac{1}{2} & \frac{1}{2} & 0 \\ \hline & 0 & 1 \end{array}\right] \\ \Rightarrow \mathbf{int}\left(\left(\frac{\Delta}{\Delta\vec{x}}u\right)_{\vec{x},t}, h\right) &= u_{\vec{x},t} + h \cdot k_2 \\ &= u_{\vec{x},t} + h \cdot \mathbf{int}\left(\left(\frac{\Delta}{\Delta\vec{x}}u\right)_{\vec{x},t+\frac{1}{2}h}\right) \end{aligned}$$

Runge-Kutta 4. Ordnung

$$\left[\begin{array}{c|c} a & B \\ \hline & c \end{array}\right] = \left[\begin{array}{c|cccc} 0 & 0 & & & \\ \frac{1}{2} & \frac{1}{2} & 0 & & \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 & \\ 1 & 0 & 0 & 1 & 0 \\ \hline & \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6} \end{array}\right]$$

4.3 Die homogene Wellengleichung

Am Beispiel der homogenen Wellengleichung soll nun die Umsetzung einer Differentialgleichung in einem zellulären Automaten gezeigt werden. Die homogene Wellengleichung ist eine partielle Differentialgleichung zweiter Ordnung und lautet [21]:

$$\frac{1}{c^2}\frac{\partial^2 u}{\partial t^2} - \sum_{i=1}^{n}\left(\frac{\partial^2 u}{\partial x_i^2}\right) = 0$$

oder ein bisschen umgeformt ($\frac{1}{c^2} = \frac{1}{k}$) und die Einzelkomponenten $x_1, x_2, \ldots, x_n$ als Vektor geschrieben:

$$\frac{\partial^2 u}{\partial \vec{x}^2} = k \cdot \frac{\partial^2 u}{\partial t^2}$$

Nach diesem Schema soll zweimal nach dem Ort differenziert und zweimal nach der Zeit integriert werden:

$$\begin{array}{rl} u_i & u_i(t+\Delta t) = u_i(t) + \dfrac{\partial u_i(t+\Delta t)}{\partial t}\Delta t \\ \downarrow & \uparrow \\ \dfrac{\partial u_i}{\partial \vec{x}} = u_i - \sum u_{i-1} & \dfrac{\partial u_i(t+\Delta t)}{\partial t} = \dfrac{\partial u_i(t)}{\partial t} + \dfrac{\partial^2 u}{\partial t^2}\Delta t \\ \downarrow & \uparrow \\ \dfrac{\partial^2 u_i}{\partial \vec{x}^2} = -2u_i + \sum u_{i-1} + \sum u_{i+1} \underbrace{\longrightarrow}_{\frac{\partial^2 u}{\partial t^2} = k \cdot \frac{\partial^2 u}{\partial \vec{x}^2}} & \dfrac{\partial^2 u}{\partial t^2} \end{array}$$

Als Nachbarschaft wird die Von-Neumann-Nachbarschaft gewählt, weil bei der Moore-Nachbarschaft die Eckzellen nicht linear unabhängig wären. Ergo ergibt sich folgende Übergangsfunktion δ, abhängig von der Zelle r sowie den Nachbarzellen n, e, s, w (Norden, Osten, Süden, Westen für ein zweidimensionales Zellfeld):

$$\begin{aligned} \delta(r,n,e,s,w) &= \begin{bmatrix} \delta_{\frac{\Delta r}{\Delta t}}(r,n,e,s,w) \\ \delta_{\frac{\Delta^2 r}{\Delta t^2}}(r,n,e,s,w) \end{bmatrix} \\ &= \begin{bmatrix} r_{\frac{\Delta r}{\Delta t}} + \delta_{\frac{\Delta^2 r}{\Delta t^2}}(r,n,e,s,w) \cdot \Delta t \\ \frac{\Delta^2 r}{\Delta \vec{x}^2} \cdot k \end{bmatrix} \\ &= \begin{bmatrix} r_{\frac{\Delta r}{\Delta t}} + \delta_{\frac{\Delta^2 r}{\Delta t^2}}(r,n,e,s,w) \cdot \Delta t \\ (-2r + \frac{1}{2}(n+e+s+w)) \cdot k \end{bmatrix} \end{aligned}$$

sowie unter der Annahme, dass die Richtung der Nachbarzellen bekannt sind (d.h. jeweils der Vorgänger beziehungsweise der Nachfolger innerhalb einer Dimension bekannt

sind und deswegen N auch ausgedrückt werden kann als:)

$$N^* = \begin{pmatrix} \overbrace{n_{1,1} \in N}^{\text{1. Dimension}} & \overbrace{n_{2,1} \in N}^{\text{2. Dimension}} & \cdots & \overbrace{n_{\frac{1}{2}|N|,1} \in N}^{\frac{1}{2}|N|\text{te Dimension}} & \text{Vorgänger} \\ n_{1,2} \in N & n_{2,2} \in N & \cdots & n_{\frac{1}{2}|N|,2} \in N & \text{Nachfolger} \end{pmatrix}$$

kann die Übergangsfunktion δ für die Wellengleichung folgendermassen formuliert werden:

$$\delta(r, N^*) =$$

Teil II
Bemerkungen zur Implementation

5 Wahl des Umfelds

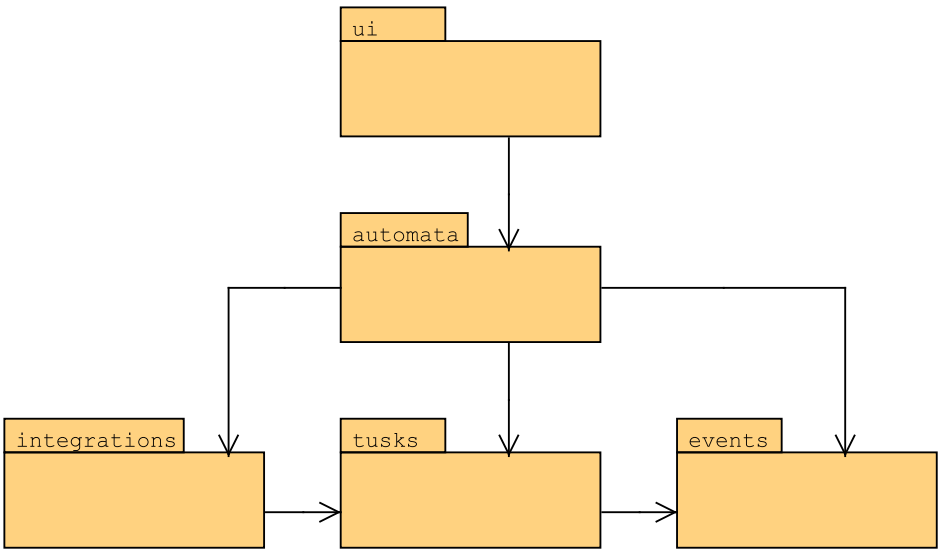
Die Wahl des Umfelds fiel auf eine Kombination aus JavaScript und dem HTML5-`canvas`-Element für die grafische Ausgabe. Die hauptsächlichen Gründe dafür sind die einfache Umsetzung sowie die sehr gute Portabilität, vorallem auch auf portable Devices mit Touch-Bedienung. Dadurch kommt der primäre Use Case der Software (Bewegung einer Flüssigkeit mittels dem Pointing Input) besonders gut zur Geltung, weil der Input eben mit dem Finger vorgenommen werden kann.

Bedenken wurden vorgängig und während der ersten Implementierungsphase vor allem in Bezug auf die Perfomance geäussert, konnten aber im Laufe des Projekts zerstreut werden, auch weil gezeigt werden konnte, dass Optimierungen möglich sind (paralleles Berechnen, Umstellung von Vektorgrafik-Operationen auf Bitmap-Manipulationen).

Als Alternativen waren das Game-Framework XNA sowie die 3D-LED-Installation in der Haupthalle des Zürcher Hauptbahnhofs in Betracht gezogen worden.

6 Architektur

Eine Übersicht über die grobe Architektur sei hier als Package-Diagramm skizziert:



7 Testing

Unit Testing mit JavaScript ist nicht ganz einfach – aber möglich und natürlich sinnvoll. Wenn die Unit Tests in einem tatsächlichen Browser auszuführen sind, weil sie beispielsweise mit dem Document Object Model interagieren, führt das zwangsläufig zu Mühsamitäten, vor allem bezüglich Continuous Integration.

Da allerdings das UI nur einen kleinen Teil unseres Programms ausmacht, schien es ratsam, vor allem die anderen Bereiche zu testen. Daraus ergibt sich die Möglichkeit, auf eine Implementation von JavaScript in einem nicht-interaktiven Kontext zu setzen. Die Wahl fiel auf Node.js, eine auf WebKit aufbauende Server-Implementation von JavaScript [10], sowie das Test-Framework Mocha [8]. Da das produktive Programm eine reine Browser-Geschichte ist, stehen während des produktiven Betriebs weder Build-Tools noch Server zur Verfügung, alle JavaScript-Files werden vom verwendenden HTML-Dokument über profane `<script>`-Tags eingebunden. Um das sich daraus ergebende Dilemma zwischen Test- und Produktionsumgebung zu lösen, werden die Komponenten über Node-Module wo nötig zusammengestöpselt, aber nur dann, wenn Node tatsächlich zur Verfügung steht (der Ausführer also ein Test Runner ist):

```
if (typeof(module) != "undefined") {
  module.exports = CellularAutomata;
  var Cell = require('../../src/automata/Cell.js');
  var Swimmer = require('../../src/automata/Swimmer.js');
}
```

Ein Unit Test (unter Verwendung der Should-Assertion Library) sieht dann beispielsweise folgendermassen aus:

```
describe('CellularAutomata', function() {
```

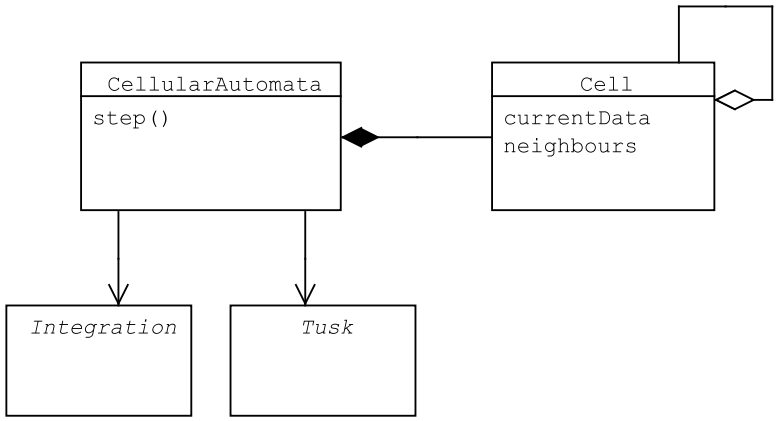
```
  it('should step without Tusk', function() {
    var auto = new CellularAutomata();
    auto.initCells();
    var iterations = auto.iterations;
    auto.step();
    auto.iterations.should.equal(iterations+1);
  });

});
```

Die Summe der Tests kann dann auf der Kommandozeile (oder durch einen CI-Server) aufgerufen werden mit:

```
~/node node_modules/mocha/bin/mocha
```

8 Automata

Die Klasse `CellularAutomata` implementiert zusammen mit ihren Hilfsklassen den zellulären Automaten:



`CellularAutomata` definiert zuallererst einmal eine coole Hilfsmethode, welche über alle Zellen des Automaten iteriert und eine beliebige Funktion ausführt:

```
this.forEachCell = function(fn) {
  for (var x = 0; x < this.cols; x++) {
    for (var y = 0; y < this.rows; y++) {
      var cell = this.model[x][y];
      fn(cell);
    }
  }
};
```

Diese Methode wird dann beispielsweise während der Initialisierung der Zellen folgendermassen verwendet[7]:

```
this.wireNeighbourCells = function() {
  if (this.tusk != null) {
    this.forEachCell(
      (function(automata) {
        return function(cell) {
          cell.neighbours = automata.tusk.getNeighbours(cell, automata.
              model);
        };
      })(this)
    );
  }
}
```

Des weiteren führt `CellularAutomata` die Zelltransitionen durch:

```
this.step = function() {
  if (this.tusk != null) {
    this.fireEvents();
    this.integration.integrate(this);
    this.travelSwimmers();
  }
  this.iterations++;
}
```

Spannend hierbei zu sehen ist die Aufteilung der Verantwortlichkeiten. `CellularAutomata` kennt weder die Übergangsfunktion δ, die Nachbarschaftsfunktion[8] noch die Integrationsfunktion **int**, sondern verwendet via Strategy-Pattern die ihr zugeteilten Implementationen (beispielsweise die Wellengleichung mit dem Euler-Verfahren).

Es ist sogar so, dass der eigentliche Zellinhalt (der ja nur von der Übergangsfunktion δ verwendet wird) unbekannt ist. Jeder Tusk besitzt entsprechend eine Factory-Methode, um einen solchen Zellinhalt herzustellen. Die Unterschiede in den Datenstrukturen sind frappant. Das Game Of Life begnügt sich mit:

```
this.createCellData = function() {
  return {status: 0};
```

[7] Ein Wort zu der auf den ersten Blick unnötig komplizierten Definition der von `this.forEachCell()` auszuführenden anonymen Funktion: In JavaScript wird der Scope von Variablen durch die Funktion und nicht den Block begrenzt, darum gilt als Best Practice zur expliziten Scope-Gebung im Allgemeinen das Wrappen in einer anonymen Funktion und unmittelbarer Ausführung derselben [4]. Da `this` schlussendlich auch nur eine Variable in einem Funktionsscope ist, wäre `this.tusk` keine sinnvolle Referenz, weil `this` nicht bedingungslos (wie in anderen funktionalen Programmiersprachen üblich) eine Instanz von `CellularAutomata` (den Definitionsscope von `wireNeighbourCells()`), sondern der Caller ist (falls das Drücken eines Buttons zu einem Aufruf von `wireNeighbourCells()` führte, wäre `this` das Browser-`document`).

[8] Wie im Listing weiter oben ersichtlich, werden sämtliche Nachbarzellen sämtlicher Zellen während der Initialisierung aufgefunden und verknüpft. Da sich gemäss der Definition des Zellulären Automaten die Nachbarschaftsbeziehungen zwischen Zellen nie ändern, ergo von Anfang an feststehen, ist dies eine zulässige Performance-Optimierung; ansonsten müsste auf das Feld der Zellen bei jedem Iterationsschritt mindestens $|R| \cdot |N|$ mal zugegriffen werden (wobei R der ganze Zellraum und N die Menge der Nachbarzellen einer Zelle ist).

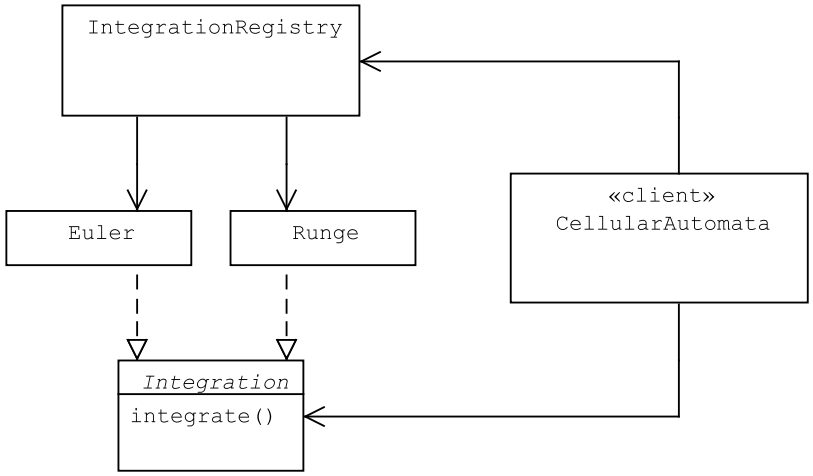
```
}
```

wogegen die Wellengleichung einige Daten mehr benötigt:

```
this.createCellData = function() {
  return {
    u: 0,
    udx: 0,
    udxdx: 0,
    udtdt: 0,
    udt: 0,
    vx: 0,
    vy: 0
  };
};
```

9 Integration

Die Wahl der Integrationsmethode ist wiederum eine Anwendung des Strategy-Patterns, wie in der Übersicht ersichtlich:



`integrate()` iteriert dann typischerweise über alle Zellen des Automaten und führt die Integration unter ein- oder mehrmaligem Aufruf der Übergangsfunktion δ des Tusk durch. Hier ist das Beispiel für das explizite Euler-Verfahren dargestellt:

```
this.integrate = function(automata) {
  var dt = 1 / automata.tusk.slices;
  for (var t = 0; t < automata.tusk.slices; t++) {

    automata.forEachCell(
      function(cell) {
        var differentials = automata.tusk.calcDifferentials(cell, dt,
            function(cell2) {
          return cell2.currentData;
```

```
        });
        cell.nextData = automata.tusk.applyDifferentials(cell, dt,
            differentials, function(cell2) {
          return cell2.currentData;
        })
      }
    );

    automata.forEachCell(
      function(cell) {
        cell.currentData = cell.nextData;
      }
    );

  }
}
```

10 Tusk

11 Grafische Ausgabe

Die grafische Ausgabe des zellulären Automaten findet in einem `canvas`-Element statt, in das mittels JavaScript-Code gezeichnet wird. Die dafür zuständigen Klassen heissen `...CanvasPainter`, wovon verschiedene Implementationen verfügbar sind. Es wird davon ausgegangen, dass jeder `...CanvasPainter` folgende Methoden anbietet:

`begin()` Beginnt den Zeichnungszyklus.

`updateCellView(cell)` Zeichnet eine einzelne Zelle (`cell`) neu.

`drawDuck(duck, cell)` Zeichnet eine Ente (`duck`) in eine einzelne Zelle (`cell`).

`end()` Beendet den Zeichnungszyklus.

Die Methoden `begin()` und `end()` sind für Implementationen gedacht, die sich nur sinnvoll verhalten wenn sie den gesamten `canvas` neu bezeichnen können und deswegen nach einem atomaren Aufruf von `updateCellView()` einen inkonsistenten Zustand aufweisen.

In den folgenden Abschnitten sind die verfügbaren Implementationen grob erklärt. Es existiert ausserdem eine `Noop`-Implementation, die aber trivial ist und verwendet werden kann, um die grafische Ausgabe komplett abzuschalten.

11.1 `VectorCanvasPainter`

Diese Klasse bezeichnet den `canvas` durch Vektorgrafik-Operationen:

```
this.updateCellView = function(cell) {
  var du = cell.currentGradients; // get values
  var x = cell.x * scaleWidth; // calculate x position in pixels
  var y = cell.y * scaleHeight; //  calculate y position in pixels
  var baseColor = this.getBaseColor(); // get base painter color
  // let Tusk calculate actual cell color, and format it as 'rgb(r,g,b)'
  var color = getFormattedColor(du[this.getUIndex()], baseColor.r,
      baseColor.g, baseColor.b);
  this.context.fillStyle = color;
  // draw and fill a rectangle.
  this.context.fillRect(x, y, scaleWidth, scaleHeight);
};
```

11.2 PixelCanvasPainter

Diese Klasse bezeichnet den `canvas` durch direkte Pixel-Manipulationen. Grundsätzlich funktionert das folgendermassen [9]:

```
var myImageData = context.createImageData(cssWidth, cssHeight);
// ... do manipulation here ...
context.putImageData(myImageData, 0, 0);
```

Es muss allerdings gesagt werden, dass die Methode `putImageData()` in allen verfügbaren Implementationen (Gecko, WebKit usw.) einigermassen langsam ist. Dies wird vor allem damit erklärt, dass das zu manipulierende Pixel-Array (`myImageData.data` in unserem Beispiel) ein Array von Integern ist, das von `putImageData()` jeweils noch durchlaufen und auf Gültigkeit (Werte $\in [0:255]$) geprüft werden muss.

Andrew J. Baker zeigt nun in [1] einen Weg, durch den die Pixelmanipulationen drastisch beschleunigt werden können (solange die verwendete Implementation den Typ `Uint8ClampedArray` unterstützt, wie vom HTML5-Standard [6] eigentlich vorgesehen).

Dazu ist es erst einmal nötig, in `begin()` die Pixel-Daten vorzubereiten:

```
this.begin = function() {
  this.imageData = this.context.createImageData(WIDTH, HEIGHT);
  this.buf = new ArrayBuffer(this.imageData.data.length);
  this.buf8 = new Uint8ClampedArray(this.buf);
  this.data = new Uint32Array(this.buf);
};
```

Dadurch ist es möglich, die in `this.data` enthaltenen Daten als ganze Pixel zu bearbeiten (und nicht den Alpha-, Rot-, Grün- und Blau-Kanal einzeln):

```
this.updateCellView = function(cell) {
  /* [...] */
  for (var ix = x; ix < x + scaleWidth; ix++) {
    for (var iy = y; iy < y + scaleHeight; iy++) {
      var p = (iy * WIDTH + ix);
      this.data[p] =
        (255   << 24) | // alpha
        (color.b << 16) |
        (color.g <<  8) |
        color.r;
```

```
    }
  }
  /* [...] */
}
```

Abschliessend werden dann die manipulierten Pixel wieder zurückgeschrieben:

```
this.end = function() {
  this.imageData.data.set(this.buf8);
  this.context.putImageData(this.imageData, 0, 0);
};
```

Teil III

Schlussfolgerung

Literaturverzeichnis

[1] Andrew J. Baker. Faster Canvas Pixel Manipulation with Typed Arrays. http://hacks.mozilla.org/2011/12/faster-canvas-pixel-manipulation-with-typed-arrays/, 2011. [Online; Stand 17. Mai 2012].

[2] Hans-Georg Beckmann. Zelluläre Automaten. http://www.vlin.de/material/ZAutomaten.pdf, 2003. [Online; Stand 28. April 2012].

[3] André Betz. *Das eindimensionale Universum. Einführung in ein informationstheoretisches Weltbild*. 1 edition, 2003.

[4] Rob Gravelle. Understanding JavaScript Closures. http://www.webreference.com/programming/javascript/rg36/index.html, 2012. [Online; Stand 20. Mai 2012].

[5] Christian Hafner. The Game of Life, vom Spiel zur Wissenschaft. http://alphard.ethz.ch/Hafner/PPS/PPS2001/Life/Life2.htm#Game, 2001. [Online; Stand 10. Mai 2012].

[6] Ian Hickson. HTML Canvas 2D Context – Editor's Draft 7 March 2012. http://dev.w3.org/html5/2dcontext/, 2012. [Online; Stand 17. Mai 2012].

[7] Dr. Ralph Massjung. Numerik gewöhnlicher Differentialgleichungen, Lektion 11. http://elearning.zhaw.ch/moodle/mod/resource/view.php?id=276205, 2012. [Online; Stand 2. Juni 2012].

[8] Mocha. Mocha. http://visionmedia.github.com/mocha/, 2012. [Online; Stand 11. Juni 2012].

[9] Mozilla Developer Network. Pixel manipulation with canvas. https://developer.mozilla.org/En/HTML/Canvas/Pixel_manipulation_with_canvas, 2012. [Online; Stand 17. Mai 2012].

[10] Node. Node.js. http://www.node.js, 2012. [Online; Stand 11. Juni 2012].

[11] Peter Deuflhard und Folkmar Bornemann. *Numerische Mathematik II: Anfangs- Und Randwertprobleme Gewohnlicher Differentialgleichungen 2., Erweiterte Und Überarbeitete Auflage*. Walter De Gruyter, 4 2002.

[12] Stefan Baur und Markus Hanselmann. Zelluläre Automaten. http://www5.in.tum.de/FA/_2005/K5/Schurr/11_CA.PDF, 2005. [Online; Stand 28. April 2012].

[13] Wikipedia. Carl Runge — Wikipedia, Die freie Enzyklopädie. http://de.wikipedia.org/w/index.php?title=Carl_Runge&oldid=100925604, 2012. [Online; Stand 2. Juni 2012].

[14] Wikipedia. Conways Spiel des Lebens — Wikipedia, Die freie Enzyklopädie. http://de.wikipedia.org/w/index.php?title=Conways_Spiel_des_Lebens&oldid=100624377, 2012. [Online; Stand 28. April 2012].

[15] Wikipedia. John C. Butcher — Wikipedia, Die freie Enzyklopädie. http://de.wikipedia.org/w/index.php?title=John_C._Butcher&oldid=102152136, 2012. [Online; Stand 2. Juni 2012].

[16] Wikipedia. John Horton Conway — Wikipedia, Die freie Enzyklopädie. http://de.wikipedia.org/w/index.php?title=John_Horton_Conway&oldid=101731222, 2012. [Online; Stand 28. April 2012].

[17] Wikipedia. Leonhard Euler — Wikipedia, Die freie Enzyklopädie. http://de.wikipedia.org/w/index.php?title=Leonhard_Euler&oldid=103781219, 2012. [Online; Stand 2. Juni 2012].

[18] Wikipedia. Martin Wilhelm Kutta — Wikipedia, Die freie Enzyklopädie. http://de.wikipedia.org/w/index.php?title=Martin_Wilhelm_Kutta&oldid=99262803, 2012. [Online; Stand 2. Juni 2012].

[19] Wikipedia. Runge-Kutta-Verfahren — Wikipedia, Die freie Enzyklopädie. http://de.wikipedia.org/w/index.php?title=Runge-Kutta-Verfahren&oldid=103843085, 2012. [Online; Stand 2. Juni 2012].

[20] Wikipedia. Stephen Wolfram — Wikipedia, Die freie Enzyklopädie. http://de.wikipedia.org/w/index.php?title=Stephen_Wolfram&oldid=99357821, 2012. [Online; Stand 28. April 2012].

[21] Wikipedia. Wellengleichung — Wikipedia, Die freie Enzyklopädie. http://de.wikipedia.org/w/index.php?title=Wellengleichung&oldid=101675074, 2012. [Online; Stand 10. Juni 2012].

[22] Wikipedia. Zellulärer Automat — Wikipedia, Die freie Enzyklopädie. http://de.wikipedia.org/w/index.php?title=Zellul%C3%A4rer_Automat&oldid=100860810, 2012. [Online; Stand 28. April 2012].

[23] Stephen Wolfram. *A New Kind of Science*. Wolfram Media, 1 edition, 5 2002.

[24] Konrad Zuse. *Rechnender Raum*. 1969.