Humboldt-Universität zu Berlin
Kultur-, Sozial-, und Bildungswissenschaftliche Fakultät
Institut für Musikwissenschaft und Medienwissenschaft
Masterstudiengang Medienwissenschaft
Wintersemester 2014/15
Seminar „So spielt das Leben. Zelluläre Automaten in Theorie und Praxis."
Dozent: Dr. Stefan Höltgen

Schriftliche Hausarbeit/Abschlussprüfung im Modul III
„Zeitbasierte Medien und zeitkritische Medienprozesse"

Wireworld – Eine Welt voller Signale

Ein Versuch über einen zeitkritischen Begriff der Medientheorie mittels zellulärer Automaten

Johannes Maibaum

6. April 2015

Matrikelnummer: 533368
3. Fachsemester
Paul-Grasse-Str. 12, 10409 Berlin
`jmaibaum@gmail.com`

Inhaltsverzeichnis

1 Einleitung

In dieser Arbeit wird versucht, anhand des zellulären Automaten „Wireworld" einen Begriff näher zu untersuchen, der für die Medientheorie – wie sie von Wolfgang Ernst an der Berliner Humboldt-Universität seit 2003 theoretisch in Vorlesungen und Seminaren, sowie praktisch-experimentell im am Lehrstuhl eingerichteten *Signallabor,* ausgearbeitet wird – von zentraler Bedeutung ist: das *Signal.* Ernst möchte mit diesem Begriff gezielt auf den Unterschied zwischen technischen Signalen, wie sie die Kybernetik definiert, und semantischen Zeichen, wie sie die Semiotik untersucht, hinweisen. Er kritisiert so die häufige Gleichsetzung von Signalen und Zeichen, von technischer Information und semantischer Bedeutung in der zeitgenössischen Medientheorie, beispielsweise durch unbedachte Angaben von Begriffsentsprechungen, wie sie z. B. Michal Eckardt vornimmt: „Den eingeführten semiotischen Termini Zeichen, Zeichenvorrat und Bedeutung wird in der Informationstheorie mit den Begriffen Signal, Signalvorrat (Kode) und Nachricht entsprochen."[1] Ernst sieht darin die Gefahr, über den semantischen Bedeutungen des Inhalts von Medien das eigentliche Medium zu vergessen und proklamiert somit in Anlehnung an McLuhan: „Medienwissenschaft ist gerade dort interessant, wo sie nicht inhaltistisch auf Massenmedienanalyse beschränkt ist, sondern Signalmassen konfrontiert."[2] Sein Signalbegriff fokussiert demnach die in den Medien am Werk befindliche physikalische Materialität und schärfe so den

> medienarchäologische[n] Blick, der die Wahrnehmung des Scanners selbst zum Archäologen eines Bild-Wissens macht, das menschlichen, (be)deutungsfixierten Augen entgeht und gerade die Leere, die Verständnislosigkeit, die „Blödigkeit der Signifikanten" (Lacans *alphabêtise)* zur Chance erklärt und damit auf andere denk- und sichtbare Zusammenhänge lenkt [...].[3]

Im Fall dieser Arbeit soll der medienarchäologische Blick erneut auf den Unterschied von physikalischem Signal und bedeutungstragendem Zeichen, sowie den perspektivischen Unterschied zwischen der Interpretation von Signalen und Zeichen im Computer und im Menschen gelenkt werden. Dies geschieht in zwei Etappen, einer praktischen in Abschnitt 2, wo zunächst der zelluläre Automat Wireworld vorgestellt wird. Dieser nimmt im vorliegenden Experiment die Perspektive der Maschine ein. Die zweite Etappe folgt in Abschnitt 3. Hier wird zunächst Ernst Signalbegriff aus der Kybernetik Georg Klaus' hergeleitet und anschließend die Unterschiede zum semiotischen Begriff des Zeichens nach Umberto Eco herausgear-

---

[1] Michael Eckardt. *Medientheorie vor der Medientheorie. Überlegungen im Anschluß an Georg Klaus.* Berlin: Trafo, 2005, S. 30.

[2] Wolfgang Ernst. „Signal *versus* Zeichen? Zeit, Medium, Maschine". In: *Kybernetik und Interdisziplinarität in den Wissenschaften. Georg Klaus zum 90. Geburtstag*. Hrsg. von Klaus Fuchs-Kittowski und Siegfried Piotrowski. Berlin: Trafo, 2002, S. 323–332, hier S. 331.

[3] Ebd., S. 330.

beitet. Abschließend wird der Frage nachgegangen, wo in der Simulation einer in Wireworld implementierten Zellschaltung, die zwei binär codierte Zahlen addiert, Signale und wo Zeichen, bzw. Symbole am Werk sind.

1.1 Zur Theorie zellulärer Automaten

Das allgemeine Konzept zellulärer Automaten wird dem ungarisch-US-amerikanischen Mathematiker John von Neumann zugeschrieben, wobei dieser selbst angibt, es gegen Ende der 1940er Jahre nach Vorschlägen und Hinweisen seines Freunds und Kollegen Stanisław Ulam ausgearbeitet zu haben.[4] Von Neumanns *Theory of Self-Reproducing Automata* wird von Arthur Burks – ebenso wie von Neumann Mitglied der Forschungsgruppe, die in den 1940er Jahren den ENIAC konstruierte – als eine allgemeine systematische Theorie beschrieben, die von Neumann aufgrund der „important similarities between computers and natural organisms, and […] the heuristic advantages in comparing such different but related systems“ entwickelt habe.[5] Als basales Charakteristikum lebender Organismen hielt von Neumann fest: „they can produce other organisms like themselves.“[6] Die primäre Fragestellung, die er mit seiner Theorie beantworten wollte, lautete folglich: „what kind of logical organization is sufficient for an automaton to control itself in such a manner that it reproduces itself?“[7] Dazu entwarf er zunächst ein fiktives Modelluniversum, indem ein unerschöpflicher Vorrat an elementaren Bauteilen, „elementary parts“,[8] existierte, die ihrerseits jedoch nur ein paar wenigen unterschiedlichen Typen angehörten. Jeder Bauteiltyp könnte bestimmte Teilfunktionen innerhalb eines selbstreproduzierenden Automaten übernehmen. Es gäbe demnach Elemente, die ausschließlich Informationen verarbeiteten, indem sie logische Funktionen durchführten, „starre“ Teile, aus denen der Rahmen, das Gerüst des Automaten aufgebaut wäre, Elemente zur Herstellung oder Trennung von Verbindungen zwischen anderen Teilen sowie Muskelzellen, die den Automaten oder Teile davon in Bewegung versetzten.[9] Der eigentliche Automat wäre aus hunderten dieser Bauteile aufgebaut und würde sich zur Reproduktion durch das Universum bewegen: „its essential activity is to pick up parts and put

---

[4] Vgl. Arthur W. Burks. „Von Neumann’s Self-Reproducing Automata“. In: *Essays on Cellular Automata*. Hrsg. von dems. Urbana/IL und London: University of Illinois Press, 1970, S. 3–64, hier S. 7; John von Neumann. *Theory of Self-Reproducing Automata*. Hrsg. von Arthur W. Burks. Urbana/IL und London: University of Illinois Press, 1966, S. 94.

[5] Arthur W. Burks. „Editor’s Introduction“. In: John von Neumann. *Theory of Self-Reproducing Automata*. Hrsg. von Arthur W. Burks. Urbana/IL und London: University of Illinois Press, 1966, S. 1–28, hier S. 18.

[6] Von Neumann, *Theory of Self-Reproducing Automata*, S. 78.

[7] Burks, „Von Neumann’s Self-Reproducing Automata“, S. 4.

[8] Von Neumann, *Theory of Self-Reproducing Automata*, S. 75.

[9] Vgl. ebd., S. 80–82.

them together, or, if aggregates of parts are found, to take them apart."[10] Die Selbstreproduktion wäre geglückt, wenn es dieser Maschine gelänge, eine vollständige Kopie seiner selbst zusammenzusetzen und diese anschließend ihrerseits beginnen würde, weitere Automaten herzustellen.[11]

Da von Neumann die selbstreproduzierenden Automaten nicht nur theoretisch beschreiben, sondern auch ihre praktische Operation demonstrieren wollte, generalisierte er das soeben beschriebene Modell, welches Burks als „kinematic automaton system" bezeichnete,[12] in einigen wichtigen Punkten. Aus dem kinematischen Universum wurde ein n-dimensionales Gitternetz, dessen Koordinatenpunkte fortan als Zellen bezeichnet wurden. Jede Zelle kann sich in einem bestimmten Zustand befinden, der aus einer endlichen Menge von Zuständen ausgewählt wird. Die unterschiedlichen Zustände nehmen die Rolle ein, die die elementaren Bauteile im kinematischen Modell inne hatten. Anstatt sich also frei im Universum zu bewegen, Elemente aufzusammeln und zu verbinden, arbeitet dieser neue, *zelluläre* Automat durch die Veränderung der Zustände der Zellen im zellulären Universum. Diese Veränderungen geschehen ausschließlich zu diskreten Zeitpunkten – „$t = 0, 1, 2, 3, \ldots$",[13] später *Generationen* genannt – und nach klar festgelegten Regeln. Burks fasst dieses allgemeine System eines zellulären Automaten wie folgt zusammen:

> A *cellular automaton system* [...] is specified by giving a finite list of states for each cell, a distinguished state (called the "blank state"), and a rule which gives the state of a cell at time $t + 1$ as a function of its own state and the state of its neighbors at time t. We will call the list of states for a cell together with the rule governing the state transition of the cell a *transition function.*[14]

Mit diesem generalisierten System lässt sich demnach nicht nur von Neumanns selbstreproduzierender Automat,[15] sondern jeder beliebige Automat beschreiben.[16] Gegeben sein muss lediglich die *Liste aller möglichen Zellzustände,* eine *Ausgangskonfiguration* sowie ein *Regelsatz,* der die Übergangsfunktion der Zellzustände von einer Generation zur nächsten beschreibt. Wie im obigen Zitat von Burks beschrieben, wählt die Übergangsfunktion den nächsten Zustand einer Zelle auf Basis ihres aktuellen Zustands sowie des Zustände ihrer Nachbarzellen aus. Welche Zellen als Nachbarzellen in Betracht kommen, regelt die *Nachbarschaftsrelation* des zellulären Automaten. Theoretisch sind hierbei unendlich viele Mög-

---

[10] Von Neumann, *Theory of Self-Reproducing Automata*, S. 75.

[11] Die dazu nötigen Schritte werden in ebd., S. 83–87, beschrieben.

[12] Burks, „Von Neumann's Self-Reproducing Automata", S. 4.

[13] Ebd., S. 7.

[14] Ebd.

[15] Burks tut dies ausführlich in ebd., Abschnitte 3–11, S. 8–52.

[16] Eine Übersicht verschiedener zellulärer Automaten liefern z. B. Tommaso Toffoli und Norman Margolus. *Cellular Automata Machines. A New Environment for Modeling.* Cambridge/MA und London: MIT Press, 1987.

lichkeiten denkbar – von einer einzigen, z. B. der linken Nachbarzelle im Gitternetz, bis zu allen im Universum vorhandenen Zellen. Die meisten praktisch realsierten zellulären Automaten operieren jedoch nach einer von zwei Nachbarschaftsrelationen: der *Von-Neumann-Nachbarschaft* oder der *Moore-Nachbarschaft.* Diese gehen von den unmittelbar an eine Zelle angrenzenden Zellen aus, wie sie von Neumann anhand der Koordinaten einer Zelle im Gitternetz, i und j, formal benennt: „The nearest neighbors of (i, j) are the four points $(i \pm 1, j)$, $(i, j \pm 1)$. The next nearest neighbors are the four points $(i \pm 1, j \pm 1)$.“[17] Die *Von-Neumann-Nachbarschaft* geht nur von den zuerst genannten *allernächsten* Nachbarzellen aus,[18] also den vier horizontal und vertikal an die Zelle angrenzenden. Die *Moore-Nachbarschaft,* benannt nach dem Mathematiker und Informatiker Edward Moore, schließt auch die vier diagonal angrenzenden Zellen mit ein, sodass hier alle acht umliegenden Zellen als Nachbarzellen gelten.[19]

Von Neumanns Generalisierungen haben zusätzlich einen praktischen Vorteil: Sie lassen sich leicht auf einem Digitalcomputer implementieren. Dies liegt darin begründet, dass sich von Neumann eng am Konzept der Turingmaschine und frühen Rechenmaschinen wie dem ENIAC orientierte, den Maschinen, mit denen er sich auskannte: „We will stick to automata which we know completely because we made them, either actual artificial automata or paper automata described completely by some finite set of logical axioms.“[20] Einen zellulären Automaten auf einem Computer zu simulieren ist zudem häufig notwendig, wenn man sein Verhalten untersuchen möchte:

> [M]any interesting questions about cellular systems cannot be answered analytically. This is especially true for cellular systems which are adequate for modeling natural systems. Here the sheer complexity of the situation and/or the indefinite nature of the problem forces one to simulate the system by computing many histories of it.[21]

Eine praktische Implementierung von Wireworld ist daher ebenso Teil dieser Arbeit. Der zelluläre Automat startet, wenn die auf der beiliegenden CD zu findende Datei `wireworld`

---

[17] Von Neumann, *Theory of Self-Reproducing Automata*, S. 133.

[18] „We will choose the former, since it leads to a simpler set of tools.“ Ebd.

[19] Vgl. Edward F. Moore. „Machine Models of Self-Reproduction“. In: *Mathematical Problems in the Biological Sciences.* Hrsg. von Richard Bellman. Proceedings of Symposia in Applied Mathematics 14. Providence/RI: American Mathematical Society, 1962, S. 17–33, hier S. 21. Moores Artikel zeigt, dass auch vor der Publikation von von Neumanns *Theory of Self-Reproducing Automata* bereits theoretisch zu einer allgemeinen *Automatentheorie* geforscht wurde. Der Autor beweist hier mathematisch, welche Bedingungen im Regelsatz eines zellulären Automaten gegeben sein müssen, damit es sogenannte *Garten-von-Eden*-Konfigurationen geben kann. So werden Zellanordnungen bezeichnet, die *niemals* aus anderen Zellkonfigurationen entstehen können, also höchstens als Ausgangskonfiguration vorliegen können. Moores Theorem 2 lautet: „For a tesselation structure for which there exist erasable configurations, there exist Garden-of-Eden configurations.“ Ebd., S. 26.

[20] Von Neumann, *Theory of Self-Reproducing Automata*, S. 79.

[21] Burks, „Von Neumann’s Self-Reproducing Automata“, S. 54.

`.html` in einem Webbrowser aufgerufen wird.[22] Im abschließenden Abschnitt 4 wird der entwickelte Programmcode vollständig abgedruckt und kommentiert.

2 Der zelluläre Automat Wireworld

Bei Wireworld handelt es sich um den Regelsatz für einen zweidimensionalen zellulären Automaten, der zur Simulation von logischen Operationen geeignet ist, wie sie auch in elektronischen Schaltkreisen implementiert werden können. Im Gegensatz zu von Neumanns selbstreproduzierenden Automaten mit seinen 29 Zellzuständen und zum bekannten *Game of Life,* das Ende der 1960er Jahre vom Mathematiker John Conway entwickelt wurde und mit nur zwei Zuständen, *lebend* und *tot*, auskommt,[23] sind in Wireworld vier Zellzustände vorgesehen:

1. *leere* Zellen, der „blank state“ nach Burks,[24]
2. *Draht*-Zellen, in gewisser Weise ebenfalls *leere* Zellen, die jedoch den sich darauf bewegenden *Elektronen* als Grundlage dienen,
3. *Elektronenköpfe,* die den Anfang eines Elektrons darstellen und
4. *Elektronenenden,* die stets nach einem *Elektronenkopf* folgen und zur eindeutigen Festlegung der Bewegungsrichtung eines Elektrons dienen.

Erdacht wurde dieser Regelsatz durch den kanadischen Computerwissenschaftler Brian Silverman[25] und im Jahr 1987 in der Software *The Phantom Fish Tank* für den Apple-II und MS-DOS veröffentlicht.[26] Das Programm erlaubte es, beliebige zelluläre Automaten auszuführen und ihre Entwicklung am heimischen Computermonitor grafisch zu beobachten:

---

[22] Getestet wurde das Programm in Mozilla Firefox, Version 37.0.1, und Google Chrome, Version 41.0.2272.118.

[23] Zu *Game of Life* siehe den Artikel von Martin Gardner. „The fantastic combinations of John Conway’s new solitaire game ‘life’“. In: *Scientific American* 223.4 (1970), S. 120–123.

[24] S. o., das Zitat in Abschnitt 1.1 auf S. 3.

[25] Silverman arbeitete in den 1980er Jahren für Logo Computer Systems Inc. (kurz: LCSI), die Firma des Mitentwicklers der Programmiersprache Logo, Seymour Papert, vgl. Logo Foundation. *What is Logo?* 2011. URL: `http://el.media.mit.edu/logo-foundation/logo/index.html` (besucht am 06. 04. 2015). In jüngster Zeit ist Silverman auch bekannt für seine Arbeit am Projekt „Visual 6502“, das er gemeinsam mit seinem Bruder Barry Silverman und Greg James gestartet hat, vgl. Nikhil Swaminathan. „Digging into Technology’s Past. ‘Digital Archaeologists’ excavate the microprocessor that ushered in the home computing revolution“. In: *Archaeology* 64.4 (Juli–Aug. 2011). URL: `http://archive.archaeology.org/1107/features/mos_technology_6502_computer_chip_cpu.html` (besucht am 06. 04. 2015).

[26] Der Spielsoftware lag ein 84 Seiten umfassendes und von Silverman geschriebenes Handbuch bei: Brian Silverman. *The Phantom Fish Tank. An Ecology of Mind.* 2. Aufl. [Erstausgabe 1987]. Montreal: Logo Computer Systems Inc., 1988.

> *The Phantom Fish Tank* allows for more states than simply living or dead. There are 16 life states, numbered 0 to 15; the top eight are considered "alive." […] In *The Phantom Fish Tank* you create the rules of the universe by defining a "transition table" which specifies, for each given life state and all nine possible numbers of live neighbors (zero through eight) what life state that cell shall become.[27]

Wohl um den Heimcomputernutzer ein wenig an die Hand zu nehmen, brachte das Programm einige fertige Regelsätze mit. Denn wie der Rezensent bemerkte: „One of the first things you notice is that most universes are *not* interesting." Neben *Game of Life* waren dies weitere Automaten „that do remarkable things, from creating a ripple [tank, J. M.] to emulating logic circuits."[28] Die Gedankenkette, die Silverman – zumindest der narrativen Dramaturgie in *The Phantom Fish Tank* nach – zur Entwicklung der „Wire Rules" brachte, lohnt hier kurz erwähnt zu werden.

Was Silverman besonders faszinierte, war die Tatsache, dass *Game of Life* und viele weitere Regelsätze zellulärer Automaten turingvollständig sind, was bedeutet, dass mit ihnen prinzipiell neue Computer entworfen werden können.[29] Für Silverman hieß dies zunächst, dass die zellulären Automaten in *The Phantom Fish Tank* die vermeintlich strikte Grenze zwischen Hard- und Software verwischen:

> *The Phantom Fish Tank* provides the perfect starting point for a discussion of this somewhat arbitrary boundary. In the earliest computers, everything was hard. The programs were hard-wired into the computer. Changing a program involved rewiring the computer. […] In present day computers, there is a division between hardware and software. The processor and its memory are implemented in hardware.
>
> The instructions that tell the computer what to do are implemented in software. They can be changed without changing the physical structure of the computer. Future computers built from cellular automata may see a new shift in the boundary between hardware and software. Processor and memory will all be part of the stored program along with the data and the instructions.[30]

Um einen Computer in *Game of Life* zu entwerfen, muss die *Emergenz* der Spielregeln gezielt ausgenutzt werden. Mit der Emergenz sind die komplexen Effekte gemeint, die nicht in den Regeln des zellulären Universums direkt ausformuliert sind, aber „aus einfachen lokalen Wechselwirkungen von Elementen" entstehen und dann ihrerseits „erstaunliche Eigenschaften, wie z. B. Selbstreproduktion" aufweisen können.[31] Das wohl bekannteste emergente

---

[27] Tim Erickson. „Game Review. The Phantom Fish Tank by Brian Silverman." In: *Simulation & Games* 19.2 (Juni 1988), S. 225–228, hier S. 226.

[28] Beide Zitate: Ebd.

[29] Zur Turingvollständigkeit von zellulären Automaten vgl. z. B. Klaus Mainzer. *Die Berechnung der Welt. Von der Weltformel zu Big Data.* München: C. H. Beck, 2014, S. 99 ff.

[30] Silverman, *The Phantom Fish Tank*, S. 4.

[31] Mainzer, *Die Berechnung der Welt*, S. 99.

„Objekt“[32] in *Game of Life* ist der Gleiter: „Gliders move slowly across the screen traversing one square diagonally every four generations.“[33] Silverman führt weiter aus: „[T]here are configurations of cells that can be used to represent numbers and other configurations that can represent computational elements like adders and multipliers.“[34] Mithilfe von Gleiterströmen, wie sie in *Game of Life* beispielsweise die bekannte *glider gun* von Bill Gosper erzeugt,[35] lassen sich demnach Rechenelemente verbinden und gleichzeitig eine binäre Informationscodierung implementieren:

> Information is usually represented as a collection of “bits,” where each bit can be in either one of two states. The states are usually called either true and false, or 1 and 0, or on and off.
>
> […] A glider missing from the stream can be interpreted as a 0. With this interpretation, a sequence of 1’s and 0’s can be transferred from one place to another.[36]

Das Problem dabei: „The simplest computational elements and the representation of numbers turn out to be much too large and complicated to be shown here.“[37] Silverman meint hier nicht die Fläche der mit seinen Erläuterungen bedruckten Buchseiten, sondern die Hardware-Bedingungen der Software, welcher das gedruckte Buch beiliegt. *The Phantom Fish Tank*, entwickelt für einen 8-bit Homecomputer sowie die ersten 16-bit Personal Computer, trifft auf technologiebedingte Schranken: „Unfortunately, the computer screen has only a 40 × 40 matrix, severely limiting the amount of computation that can be seen at once.“[38] Anstatt zu resignieren, geht er diese Hürde jedoch kreativ an: „To build more complicated circuits, rules that allow for building more gates in a smaller space are necessary.“[39] Silvermans neue „Wire Rules“ folgen direkt aus diesem Entwurfsziel. Kleinstmögliche Zellstrukturen, die die Funktion von Logikgattern übernehmen können, benötigen auch kleinstmögliche Gleiter:

> How big is the smallest possible glider? If a glider, traveling through a space with homogenous rules, were only one cell long, because of its symmetry it couldn’t remember in which direction it was going. So the glider must be at least two cells long with a distinct “head” and “tail.” […] If […] the glider were traveling along a track it could be only one cell wide. The track would provide a

---

[32] Obwohl *The Phantom Fish Tank* als Spielsoftware vornehmlich an ein jüngeres Publikum gerichtet ist, behandelt der Autor auch klassische philsophische Fragestellungen. Zu den „Objekten“ in einer Zellanordnung schreibt er: „What are these things? They don’t actually exist as ‘objects.’ There is no glue or force holding them together. They are just collections of cells being born and dying at the right times in such a way as to appear object-like.“ Silverman, *The Phantom Fish Tank*, S. 17.

[33] Ebd., S. 16.

[34] Ebd., S. 20.

[35] Vgl. ebd., S. 18.

[36] Ebd., S. 24.

[37] Ebd., S. 20.

[38] Ebd., S. 30.

[39] Ebd.

> path along which the glider could travel, so there would be no need to have a two-cell front.[40]

In einen implementierbaren Regelsatz gegossen, lesen sich diese Überlegungen auf Basis dreier verschiedener Zelltypen und der Moore-Nachbarschaftsrelation wie folgt:

> 1. Wire cells with 1 or 2 live neighbors (electron heads) turn into electron heads.
> 2. Electron heads turn into electron tails.
> 3. Electron tails turn into wire cells.[41]

Der vierte, „leere" Zelltyp benötigt keine eigene Regel. Er bleibt unveränderlich leer, da Elektronen nur aus schon vorhandenen Drahtzellen entstehen können und zwei Generationen später ihrerseits wieder zu Drahtzellen werden.

Es war jedoch weniger *The Phantom Fish Tank,* das Silvermans zellulären Automaten bekannt machte. Ganz ähnlich wie bei Conways *Game of Life* hatte ein Artikel im *Scientific American* hieran einen viel größeren Anteil. In der 1990er Januarausgabe schrieb Alexander Dewdney in seiner Kolumne „Computer Recreations" über *The Phantom Fish Tank* und gab dem Regelsatz den heute bekannten Namen: „My own favorite cellular automaton in Silverman's package allows one to build a simple computer within a two-dimensional cellular space. I call it wireworld."[42] Alle heutigen Internetquellen zu Wireworld berufen sich auf diesen Artikel von Dewdney,[43] der die oben genannten Regeln als vollständigen Regelsatz deklarierte. Entsprechend weisen alle umlaufenden Implementierungen von Wireworld auch nur die oben genannten Zelltypen auf.[44] Dies ist insofern interessant, da Silverman seine „Wire Rules" in *The Phantom Fish Tank* noch weiter ausbaute und um drei zusätzliche

---

[40] Silverman, *The Phantom Fish Tank*, S. 30 f.

[41] Ebd., S. 31. Die Regelanwendung im eigens implementierten Wireworld-Programm findet sich in der Datei `wireworld.js`, s. u., Abschnitt 4.2 ab S. 42, Zeilen 135–185.

[42] Alexander K. Dewdney. „Computer Recreations. The cellular automata programs that create wireworld, rugworld and other diversions". In: *Scientific American* 262.1 (Jan. 1990), S. 146–149, hier S. 146.

[43] Vgl. David Moore und Mark Owen. *The Wireworld Computer.* 7. Nov. 2014. URL: `http://www.quinapalus.com/wi-index.html` (besucht am 06. 04. 2015), Abschnitt „Introduction"; Karl Scherer. *Wireworld. The biggest and best collection of examples of logical elements built with this amazing cellular automaton.* Sep. 2003. URL: `http://karlscherer.com/Wireworld.html` (besucht am 06. 04. 2015); Nyles Heise. *World Wide WireWorld.* URL: `http://www.heise.ws/wireworld.html` (besucht am 06. 04. 2015); Matthieu Walraet. *Wireworld – Un monde cablé.* URL: `http://matthieu.walraet.net/automate/automate.html` (besucht am 21. 03. 2015).

[44] Als Beispiel sei hier nur auf das aktuell auf allen drei üblichen Desktop-Betriebssystemen (Microsoft Windows, Apple Mac OSX und GNU/Linux) laufende Zellularautomatenprogramm *Golly* verwiesen, das auch den Wireworld-Regelsatz enthält: Andrew Trevorrow u. a. *Golly. An open source, cross-platform application for exploring the Game of Life and other cellular automata.* Dez. 2013. URL: `http://golly.sourceforge.net` (besucht am 06. 04. 2015). *Golly* bringt eine Reihe Wireworld-Beispielschaltungen mit, u. a. den vollständigen, programmierbaren Computer von Moore und Owen.

Zellzustände ergänzte: *vacuum cells, photon heads* und *photon tails.*[45] Seine Beweggründe dazu folgen dem oben genannten Ziel, logische Schaltungen so klein wie möglich zu halten. Er erläutert: „In moving from the Life Game to the Logic World[46] to the Wire Rules, the rules became more complicated and the structures became smaller. Diodes and gates can be made even smaller if the rules are allowed to become more complex."[47] Die Vakuumzellen sind als Pendants zu den Drahtzellen zu verstehen, sie bieten den Photonen – als Gleiterzellen analog zu den Elektronen – eine Bewegungsbahn. Ein Unterschied zu den vorigen drei Zelltypen besteht nur im Regelsatz der Vakuumzellen: sie benötigen zwei oder drei „lebende" Nachbarzellen; anstatt ein oder zwei Nachbarn wie die Drahtzellen, um sich in einen Photonenkopf zu verwandeln.[48] Silverman erläutert diesen Zusatz:

> The addition of vacuum cells makes it possible to build simpler diodes. It takes two live cells to excite a vacuum cell into turning into a photon head. An electron travelling the "right" way through the new diode will start to split into two before reaching the vaccum cell. The two cells can turn the vacuum cell into a photon head. That photon head is a live neighbor for the following [wire] cell so the electron can continue along its merry way.
>
> On the other hand, if an electron flows the "wrong" way through the diode, it gets stopped, since the lone electron head cannot turn the vacuum cell into a photon head. The smaller diodes makes it possible to build smaller gates.[49]

Es ist nicht ganz klar, warum Dewdney Silvermans Vakuum- und Photonenzellen in seinem Artikel verschweigt, da er ebenfalls Fragen nach kleinstmöglichen Zellkonstruktionen stellt, jedoch hauptsächlich bezogen auf die kleinstmöglichen Abstände zwischen zwei aufeinanderfolgenden Elektronen in einem mit Wireworld entwofenen Computer:

> If one attempts to design a complete computer, one would certainly have to decide on a timing convention for internal signals. [...] The individual electron signals must be spaced out in time. A constant number of clock cycles will seperate consecutive signals along any wire inside our computer. If the constant number of clock cycles is T, then for every T cycles every component of the computer will be receiving either a single electron (1) or no electron (0). The question is, How small a value of T can we get away with?[50]

---

[45] Vgl. Silverman, *The Phantom Fish Tank*, S. 37.

[46] Die „Logic World" ist ein weiterer Regelsatz, den Silverman auf seinem Weg von *Game of Life* zu den „Wire Rules" einführt. Auch damit lassen sich Logikschaltungen aus Gleiterströmen aufbauen, jedoch sind die Gleiter hier noch um einiges größer als in den „Wire Rules". Vgl. ebd., S. 20–29.

[47] Ebd., S. 37.

[48] „Lebende" Zellen sind in *The Phantom Fish Tank* solche Zellen, die in der Regelanwendung als Nachbarn anderer Zellen beachtet werden, vgl. a. das Zitat aus Tim Ericksons „Game Review" (s. o., S. 6). Im Falle der „Wire Rules" werden Elektronen- und Photonenköpfe als „lebende" Zellen angesehen, vgl. Silverman, *The Phantom Fish Tank*, S. 37. Wenn im Folgenden von „lebenden" oder „lebendigen" Nachbarzellen die Rede ist, wird diese Terminologie übernommen.

[49] Ebd., S. 38 f.

[50] Dewdney, „Computer Recreations", S. 147.

Das *T*, das Dewdney schließlich vorschlägt, richtet sich nach seiner Flip-Flop-Schaltung zur Speicherung eines Elektrons: „The smallest memory loop I could manage takes 13 ticks."[51] Zum Sieger seines ausgeschriebenen Wettbewerbs zur Suche nach Speicherschaltungen mit kleineren Zirkulationszeiten erklärt er dann in einer folgenden Ausgabe jedoch Nyles Heise, dessen „most efficient memory element has a cycle time of only four ticks."[52] Mittlerweile sind Speicherschaltungen sowie Logikgatter für nahezu alle möglichen Elektronenabstände gefunden, auch für den kleinstmöglichen Abstand von nur drei Zellen. Kleinere Abstände sind aufgrund der Regeln nicht möglich, da ein Elektronenkopf nur aus einer Drahtzelle entstehen kann. Deswegen muss vor einem Elektronenkopf und dem dazugehörigen Ende immer mindestens eine Drahtzelle liegen. Zur Klassifizierung von Wireworld-Schaltungen nach dem Elektronenabstand hat sich die Bezeichnung „*n*-tick logic" durchgesetzt, man spricht also von „3-tick logic", „4-tick logic", usw.[53]

Dewdneys reduzierte Beschreibung des Regelsatzes im *Scientific American* hatte zur Folge, dass alle im Internet veröffentlichten Wireworld-Experimente ausschließlich mit Draht- und Elektronenzellen auskommen. Für die eigene Implementierung, die dieser Arbeit beiliegt, wurde ebenfalls Dewdneys reduzierter Wireworld-Regelsatz verwendet und nicht Silvermans etwas komplexere „Wire Rules".[54] Im Folgenden werden einige beispielhafte Schaltungen vorgestellt, die mit Wireworld realisiert werden können, auch, um das bisher gesagte verständlicher zu machen.

2.1 Die Grundelemente: Abzweigung und Sperre

Es wurde bereits gesagt, dass sich Wireworld-Elektronen – als Zellenpaar aus einem Elektronenkopf und einem Elektronenende – wie die bekannten Gleiter aus *Game of Life* bewegen können, wenn ihnen eine Leiterbahn aus Drahtzellen vorgegeben ist. Ein Elektron bewegt sich darauf pro Generation um eine Zelle vorwärts. Die dem Automaten zugrundeliegende Moore-Nachbarschaft sorgt dafür, dass Elektronenbahnen aus Drahtzellen sowohl horizontal, vertikal als auch diagonal angelegt werden können (siehe Abbildung 1).

Aufgrund der ersten Regel, die besagt, dass eine Drahtzelle zu einem Elektronenkopf wird, sobald einer ihrer Nachbarn „lebendig", d. h. ein Elektronenkopf ist, lassen sich Elektronen an

---

[51] Dewdney, „Computer Recreations", S. 148.

[52] Alexander K. Dewdney. „Mathematical Recreations. An odd journey along even roads leads to home in Golygon City". In: *Scientific American* 263.1 (Juli 1990), S. 118–121, hier S. 121. Heise hat seinen Brief an Dewdney, indem er ihm seine Ergebnisse mitteilt, auf seiner Website veröffentlicht: Nyles Heise. *Letter to A. K. Dewdney.* 4. Feb. 1990. URL: `http://www.heise.ws/lettertodewdney.html` (besucht am 06. 04. 2015).

[53] Vgl. z. B. Scherer, *Wireworld*.

[54] Aus diesem Grund wird im Folgenden Dewdneys Namensgebung Wireworld gefolgt.

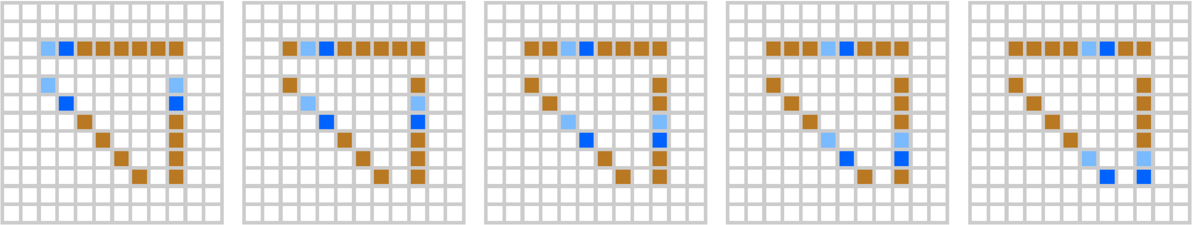

Abbildung 1: Die Bewegung von Wireworld-Elektronen auf unterschiedlichen Leiterbahnen aus Drahtzellen in fünf aufeinander folgenden Generationen. Elektronenköpfe sind dunkelblau, Elektronenenden hellblau und Drahtzellen ‚kupferfarben‘. Auf der beiligenden CD im Ordner `examples` findet sich eine entsprechende Beispielschaltung in der Datei `01-electron-movement.txt`. Sie kann durch *drag and drop* in das zelluläre Spielfeld geladen werden.

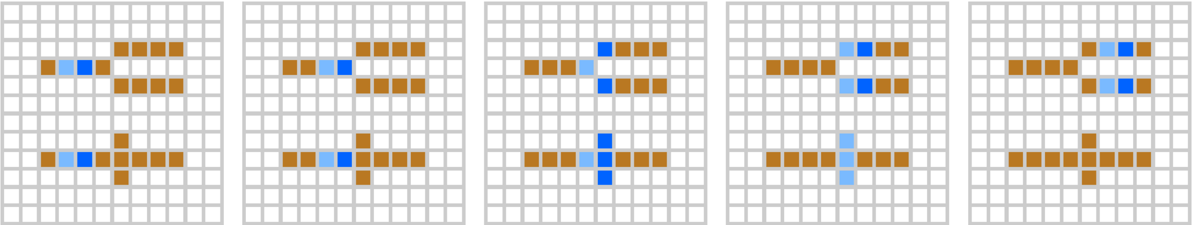

Abbildung 2: An Drahtzellenabzweigungen werden Elektronen vervielfältigt, die drei senkrecht angeordneten Drahtzellen darunter bilden eine Sperre. Sie blockieren ein ankommendes Elektron. (`02-branch-block.txt`).

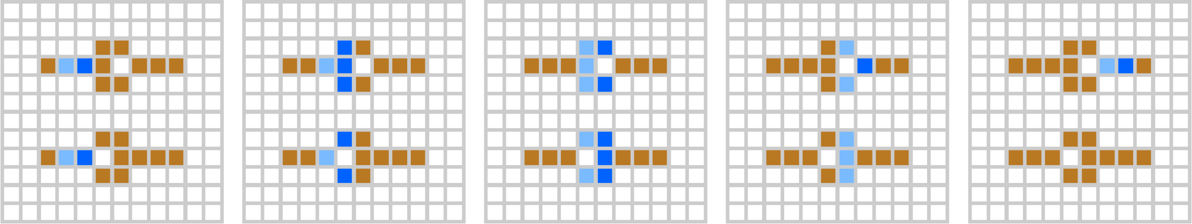

Abbildung 3: Die Wireworld-„Diode“, oben in ‚Durchlassrichtung‘ und unten in ‚Sperrrichtung‘ (`03-diodes.txt`).

Drahtzellenabzweigungen sehr einfach vervielfältigen. Da in der gleichen Regel jedoch für den Wechsel von Drahtzelle zu Elektronenkopfzelle auch eine obere Grenze von maximal zwei Nachbarn angegeben ist, verbleiben Drahtzellen mit drei oder mehr Elektronenkopf-Nachbarzellen in ihrem Zustand. Es ergibt sich eine Art Sperre, die ein Elektron nicht passieren kann (siehe Abbildung 2). Tatsächlich sind diese beiden Zellstrukturen – die Abzweigung und die Sperre – *die grundlegenden Elemente,* aus denen alle größeren Zellstrukturen, die sich emergent wie Logikschaltungen verhalten, aufgebaut sind. Ihr Verhalten lässt sich noch unmittelbar aus dem Regelsatz erschließen und stellt daher die Schwelle dar, jenseits derer sich in Wireworld komplexes, emergentes Verhalten einstellt. Dies soll im Folgenden anhand weiterer Beispielschaltungen gezeigt werden.

2.2 Die „Diode“

Kombiniert man eine Drahtabzweigung mit einer ‚Sperrschicht‘ aus drei zusammenhängenden Drahtzellen, ergibt sich eine Zellstruktur, die in der Wireworld-Terminologie als „Diode“ bezeichnet wird.[55] Sie hat, ähnlich wie ihr elektronisches Pendant, die emergente Eigenschaft, je nach der Richtung aus der ein Elektron eintrifft, dieses entweder passieren zu lassen oder zu blockieren. Den Fall in ‚Durchlassrichtung‘ beschreibt Silverman wie folgt:

> If an electron hits the three-cell side of a diode first, those three cells, having one neighbor each, turn into electron heads. In the next generation, the two cells turn into electron heads since they have two neighbors each. Finally the single wire cell turns into an electron head and the electron leaves the diode.[56]

Im umgekehrten Fall, also in ‚Sperrrichtung‘, trifft das Elektron zuerst auf die Abzweigung aus zwei Drahtzellen, die entsprechend zu Elektronenköpfen werden. In der nächsten Generation wird die ‚Sperrschicht‘ aus den drei senkrechten Drahtzellen aktiviert. Dies führt

---

[55] Silverman, *The Phantom Fish Tank*, S. 33.
[56] Ebd., S. 34.

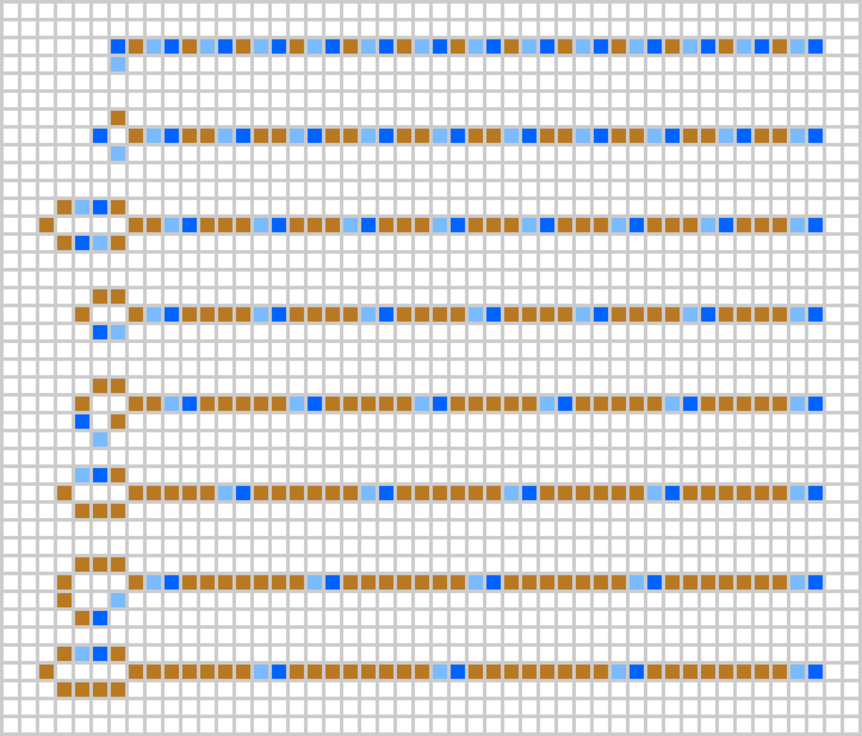

Abbildung 4: „Taktschleifen“ von $n = 3$ bis $n = 10$ (`04-clocks.txt`).

dazu, dass die einzelne Drahtzelle am Ausgang nun drei Elektronenkopf-Nachbarn hat, sodass sie ihrerseits im Drahtzustand verbleiben muss, wodurch das Elektron blockiert wird (siehe Abbildung 3).

2.3 Die „Taktschleife“

Das klassische *Game-of-Life*-Problem einer „Gleiterkanone“ ist in Wireworld aufgrund von Regel Nr. 1 geradezu trivial. Jede ringförmige Anordnung aus Drahtzellen, in der mindestens ein Elektron zirkuliert, erfüllt die Anforderung, sofern irgendwo eine Drahtabzweigung angebracht ist. Anders als in *Game of Life* bezeichnet man Gleiterkanonen in Wireworld jedoch ihrer hauptsächlichen Verwendung entsprechend als „Clock loops“, Taktgeneratoren oder wörtlich *Taktschleifen.*[57] Die mittels dieser Schleifen konstruierten Taktgeneratoren werden nach dem Generationenabstand zweier aufeinander folgender Elektronenköpfe bezeichnet, angegeben in *ticks* oder *cycles.*[58]

---

[57] Heise, *World Wide WireWorld*, Unterseite „4-tick Logic“. Eine Auswahl an Taktschleifen mit kleinen Elektronenabständen zeigt Abbildung 4. Schleifen für ungerade Taktfrequenzen lassen sich einfacher aufbauen, wenn man einen Ring mit doppelter Länge erstellt und darin zwei Elektronen zirkulieren lässt. Für $n = 5$ ist dies die einzige Möglichkeit, da sich kein Ring aus fünf Drahtzellen konstruieren lässt, vgl. Scherer, *Wireworld*.

[58] Vgl. Heise, *Letter to A. K. Dewdney*.

A	B	$A \lor B$	$A \dot{\lor} B$	$A \land \neg B$	$A \land B$
0	0	0	0	0	0
0	1	1	1	0	0
1	0	1	1	1	0
1	1	1	0	0	1

Tabelle 1: Duale Wahrheitswerte der logischen Funktionen OR ($\lor$), EOR ($\dot{\lor}$), AND-NOT ($\land\neg$) und AND ($\land$).

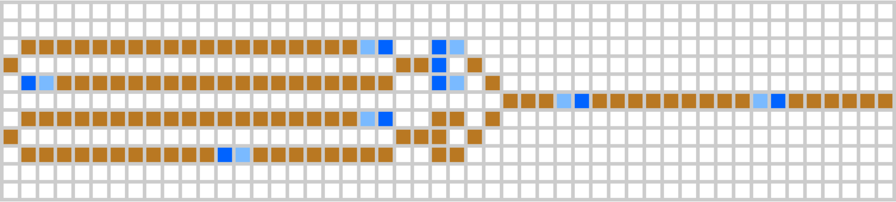

Abbildung 5: Ein OR-Gatter nach Brian Silverman mit „Dioden“ zum Schutz der Eingangsdrähte. In der dargestellten Generation sind die Ergebniselektronen der Ausdrücke $0 \lor 1$ und $1 \lor 0$ bereits auf dem Ausgangsdraht erschienen, die Elektronen für den Ausdruck $1 \lor 1$ verlassen in der nächsten Generation den Taktgenerator. An der ‚Sperrschicht‘ der „Diode“ von Eingang B ist noch das reflektierte Elektron der letzten Operation zu sehen, welches in der folgenden Generation verschwunden sein wird (`05-or-gates.txt`). ***Anmerkung:*** *In dieser und in allen folgenden Logikschaltungen wird der obere Eingang des Gatters stets mit B, der untere mit A bezeichnet!*

Durch Abzweigungen, Sperren und Taktgeneratoren lassen sich Strukturen aufbauen, mit deren emergentem Verhalten alle logischen Grundfunktionen, wie *Und, Oder, Exklusiv-Oder* und *Nicht* erzeugt werden können. Im Folgenden werden diese Funktionen mit den üblichen englischen Kurzformen AND, OR, EOR und NOT angegeben. Kombiniert man diese logischen Schaltungen dann erneut, lässt sich in letzter Konsequenz eine turingvollstände Maschine, also ein programmierbarer Computer konstruieren. Alle dazu nötigen Schritte führen Moore und Owen aus und vor.[59] Hier soll das Ziel lediglich die Konstruktion eines seriellen Addierers aus den genannten Logikgattern sein (siehe Tabelle 1).

2.4 Das OR-Gatter

Das einfachste logische Gatter in Wireworld ist das OR-Gatter. Eigentlich ermöglicht schon die einfache Abzweigung, die oben in Abbildung 2 dargestellt ist, eine OR-Funktion. Lässt man, nach der Grafik oben, von rechts gleichzeitig zwei Elektronen kommen, so verlässt ein

---

[59] Vgl. Moore und Owen, *The Wireworld Computer*.

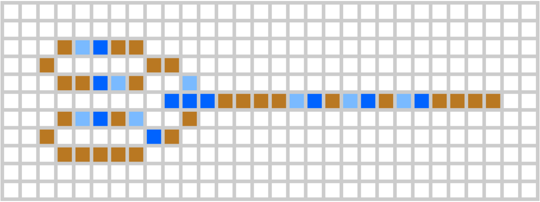

Abbildung 6: Das OR-Gatter nach Nyles Heise, hier dargestellt im minimalen Elektronentakt von $n = 3$ (`05-or-gates.txt`).

Elektron die linke Seite: $1 \vee 1 \implies 1$. Ein Problem ergibt sich allerdings, wenn nur ein Elektron in die Schaltung läuft, denn an einer Abzweigung entsteht grundsätzlich auf allen freien Drahtzellen in der nächsten Generation ein Elektronenkopf, sodass in diesem Fall der Ausgang der Schaltung nicht mehr eindeutig bestimmt werden könnte. Oder wie Silverman es formuliert „When only one electron enters, it will not only go out the bottom, [in Silvermans Schaltungsgrafik zeigt der Ausgang nach unten, J. M.] it will also go out the other side and possibly corrupt the other input stream."[60] Silvermans Lösung nutzt zwei der oben eingeführten „Dioden", um die Eingangsdrähte vor einem Elektron aus der falschen Richtung zu schützen (siehe Abbildung 5).[61] Auch Dewdney bildet dieses OR-Gatter in seinem Artikel ab.[62] Leider hat diese Schaltungsvariante, auch wenn sie mit Strukturen entworfen wurde, die realen elektronischen Bauteilen wie Dioden nachempfunden wurden, einen schwerwiegenden Nachteil: Sie funktioniert nur mit relativ großen Elektronenabständen.[63]

Die Lösung dieses Problems ließ jedoch nicht lange auf sich warten. Nyles Heise beschreibt schon in seinem Leserbrief zu Dewdneys Kolumne eine wesentlich kompaktere und obendrein multifunktional einsetzbare Struktur aus nur vier Drahtzellen, die in Form eines Pluszeichens angeordnet sind: „When proper input and output wires are added we arrive at an OR gate, and [sic] AND gate, an inverter, and the set and reset to the latch. […] The salient feature of this element[] is that there are never any reverse-traveling electrons, hence diodes are not required."[64] Betrachtet man Heises Entdeckung genauer, kann man feststellen, dass es sich prinzipiell lediglich um eine einzelne Elektronensperre aus drei Drahtzellen handelt, die mit einem zusätzlichen dritten Anschlussdraht versehen wurde, welcher den Ausgang des Gatters darstellt. Aktiviert ein einzelnes ankommendes Elektron die drei Zellen der Sperre, so verhindern diese, dass ein Elektronenkopf auf dem zweiten Eingangsdraht entsteht, da dieser

---

60 Silverman, *The Phantom Fish Tank*, S. 33.

61 Vgl. ebd., S. 35.

62 Vgl. Dewdney, „Computer Recreations", S. 146.

63 Die in Abbildung 5 verwendeten Taktschleifen emittieren alle 11 Generationen ein Elektron entweder auf dem oberen (B) oder auf dem unteren Eingangsdraht (A).

64 Heise, *Letter to A. K. Dewdney*.

nun drei Elektronenkopf-Nachbarn besitzt. Die einzelne Drahtzelle auf dem Ausgangsdraht hingegen hat nur einen Nachbarn unter den Elektronenköpfen in der Sperrschicht, sodass hier in der folgenden Generation ein neuer Elektronenkopf entsteht (siehe Abbildung 6).[65]

Heises Suche nach der kleinstmöglichen Zellstruktur für ein OR-Gatter zeigt demnach, dass die Analogien, die bei der Beschreibung von Wireworld-Zellstrukturen zur physikalischen Welt der Elektronik hergestellt werden – z. B. die „Dioden“ – ihre Grenzen haben; in diesem Fall sogar *zeitkritische:* Während Silvermans diodengeschütztes OR-Gatter nur bei minimal $n = 9$ noch funktioniert, proklamiert Heise zu seinem OR-Gatter, das ausschließlich mit einer Variante der grundlegenderen, *automatentypischen* Drahtzellensperre funktioniert: „The OR gate works at any speed.“[66] Besonders Dewdney war erstaunt über dieses Ende der Analogien zwischen kontinuierlicher Physik und diskretem Automatenuniversum als er seine Leserpost mit neuen Wireworld-Schaltungen abschließend kommentierte: „The layouts sent in by all readers looked like nothing on earth, so alien is the notion of cellular computing to the continuous physics employed by actual computing microcomponents.“[67]

Heises stellt noch einen weiteren praktischen Punkt klar, der für nahezu alle Zellstrukturen in Wireworld gilt: Es kommt bei dem Entwurf einer Schaltung, die eine bestimmte Funktion erfüllen soll, darauf an, sich im Voraus Gedanken über die zu verwendende Elektronenfrequenz zu machen, da diese, sowohl auf die benötigten Zellstrukturen wie auch auf ihre resultierende Größe, Auswirkungen hat. Dabei bedeutet ein kleinerer Elektronenabstand keineswegs automatisch kleinere Schaltungen, wie er in seinem Brief an Dewdney festhält: „8-cycle outperforms 4-cycle in the size of AND gates, XORs, and latches. [...] And of course it is better than slower logic in the size of elements and in wiring.“[68] Da Moore und Owen zeigen, dass die Schaltungen, die Heise für einen Elektronenabstand von acht Generationen angibt, auch bei $n = 6$ funktionieren,[69] wird im Folgenden für alle logischen Schaltungen ebenfalls ein Elektronenabstand von $n = 6$ verwendet.[70]

---

65 Die Abbildung zeigt genau die hier beschriebenen Situation. Die drei Sperrzellen sind gerade Elektronenköpfe, das einzelne Elektronenende darüber zeigt an, dass das Signal aus der Eingangsleitung B stammt. In der folgenden Generation wird nur auf dem Ausgangsdraht ein neuer Elektronenkopf entstehen, das Ergebnis des logischen Ausdrucks $0 \vee 1 \Longrightarrow 1$.

66 Heise, *Letter to A. K. Dewdney*. Die Angabe zu Silvermans OR-Gatter wurde experimentell ermittelt. Mittels der Schaltungen in `05-or-gates.txt` kann dieser Wert überprüft werden: Die oberste Anordnung entspricht Abbildung 5 ($n = 11$), die zweite funktioniert bei $n = 9$ gerade noch, wohingegen die dritte Schaltung ($n = 8$) für den Ausdruck $1 \wedge 0$ kein korrektes Ergebnis anzeigt, da hier das reflektierte Elektron des vorherigen Signals nicht rechtzeitig von der Diode blockiert wird.

67 Dewdney, „Mathematical Recreations“, S. 121.

68 Heise, *Letter to A. K. Dewdney*.

69 Vgl. Moore und Owen, *The Wireworld Computer*, Abschnitt „Signals“.

70 Alle logischen Schaltungen sind in der Beispieldatei `06-logic-6-tick.txt` enthalten.

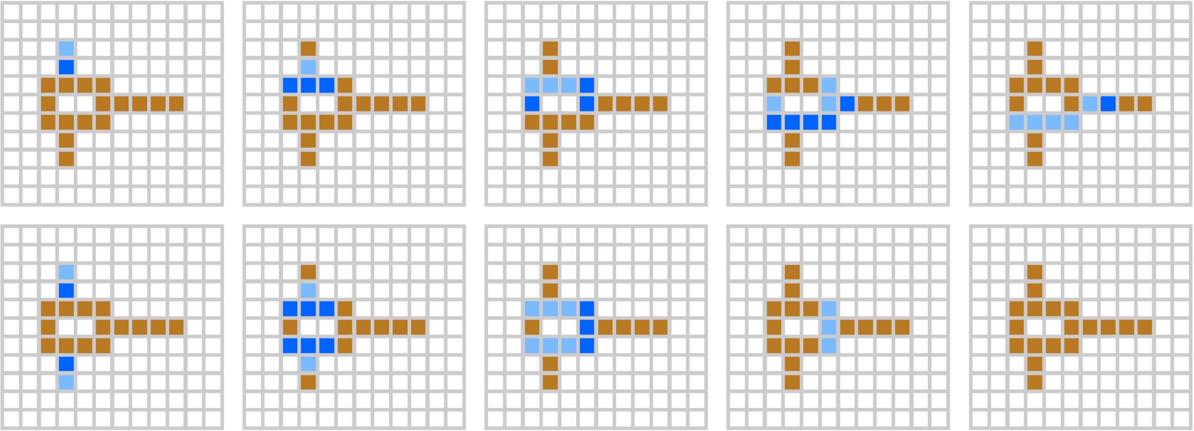

Abbildung 7: Das EOR-Gatter nach Nyles Heise. Oben: Der Durchgang eines einzelnen Elektrons durch das Gatter (entspricht $0 \dot{\vee} 1 \implies 1$). Dies gilt genauso, wenn ein einzelnes Elektron vom unteren Eingang (A) ankommt ($1 \dot{\vee} 0 \implies 1$). Unten: Wenn zwei Elektronen gleichzeitig ankommen, entspricht dies $1 \dot{\vee} 1 \implies 0$. Ein EOR-Gatter findet sich auch in `06-logic-6-tick.txt`.

2.5 Das EOR-Gatter

Ein EOR-Gatter, manchmal auch XOR genannt, unterscheidet sich von den Ergebnissen der logischen Funktion her nur in einem Fall vom ‚normalen' OR: Für den Ausdruck $1 \dot{\vee} 1$ lautet das Ergebnis 0 – anstelle von 1 beim OR. Optisch weist ein EOR-Gatter nach Heise in Wireworld jedoch große Unterschiede zum OR-Gatter auf. Es wird aus einem rechteckigen Rahmen aus drei mal vier Drahtzellen gebildet, wobei die Eingänge sich an den vier Zellen langen Seiten befinden. Der Ausgang ist an derjenigen drei Zellen langen Seite angebracht, die von den Eingängen eine Drahtzelle weiter entfernt ist. Es liegt also erneut eine Struktur vor, die ausschließlich aus dem Grundelement der Sperre aufgebaut ist. Wenn ein einzelnes Elektron an einem der Eingänge ankommt, so kann es das Gatter durchqueren, da in der dritten Generation nur zwei Zellen der am Ausgang liegenden Elektronensperre aktiviert werden. Kommen jedoch an beiden Eingängen gleichzeitig Elektronen an, so werden in der dritten Generation alle drei Sperrzellen zu Elektronenköpfen und die Drahtzelle am Ausgang kann ihren Zustand nicht wechseln (siehe Abbildung 7).

2.6 Auf der Suche nach dem AND: Das AND-NOT-Gatter

Etwas schwieriger als OR und EOR gestaltet sich der Entwurf einer Zellanordnung, welche die logische Funktion eines AND-Gatters übernehmen kann. Doch mit einem Trick lässt sich auch hierfür die von Heises OR-Gatter bekannte plusförmige Zellstruktur verwenden. Dazu wird der Ausgang an die Unterseite der Plusform verlegt. Nach den oben eingeführten

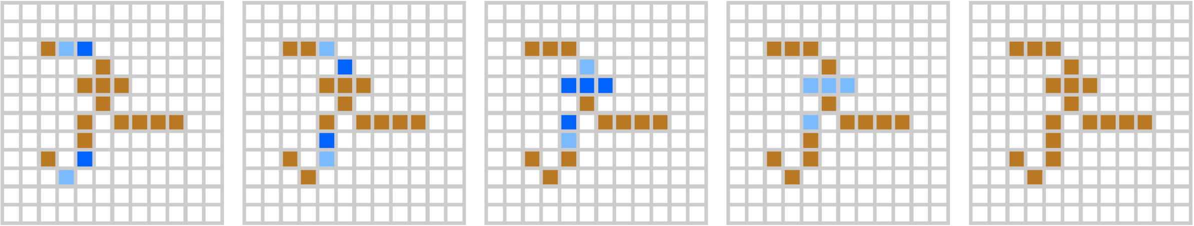

Abbildung 8: Das AND-NOT-Gatter nach Nyles Heise, David Moore und Mark Owen. Gezeigt wird der Fall $1 \land \neg 1 \Rightarrow 0$. Siehe auch das AND-NOT-Gatter in `06-logic-6-tick.txt`.

Grundformen kann man diese Schaltung vom *B*-Eingang aus gesehen als Sperre bezeichnen. Am *A*-Eingang ist nun zusätzlich noch eine Abzweigung angebracht, die den Ausgang darstellt. So gesehen ergibt sich auch das Verhalten dieser Schaltung unmittelbar: Der obere Eingang ist konsequent blockiert, und nur Elektronen aus dem unteren Eingang gelangen zum Ausgang. Wie gelangt man nun von dieser Struktur zu einem AND-Gatter?

Der Trick beruht darauf, das Eintreffen des Elektronenkopfs am unteren Eingang (*A*) *um genau zwei Generationen zu verzögern.* Dies gelingt, indem man die Drahtzellen an diesem Eingang in einer Art Bogen anordnet. Auf diese Weise wird ein Elektronenkopf am Eingang *A* blockiert, wenn auch am oberen Eingang (*B*) ein Elektron eintrifft. Genau diesen Fall zeigt Abbildung 8: In der dritten Generation ist die Sperre am Eingang *B* bereits aktiv, sodass die untere Drahtzelle der Plusstruktur, dem Eingang *A*, zusammen mit dem dort erst jetzt eintreffenden Elektron vier Elektronenkopf-Nachbarn hat, sich also ihrerseits nicht in einen Elektronenkopf verwandeln kann. In der Liste aller bei zwei Eingängen möglichen Kombinationen erreicht ein Elektron also nur in jenem Fall den Ausgang des Gatters, bei dem an *A* ein Elektron ankommt, an *B* jedoch nicht. Diese Zellstruktur aus einer Sperre, einer Abzweigung als Ausgang und einer Verzögerung des Eingangs *A* um zwei Generationen erfüllt also die logische Funktion $A \land \neg B$,[71] weshalb sie auch als AND-NOT-Gatter bezeichnet wird.[72]

2.7 Das NOT-Gatter

Ein AND-NOT-Gatter lässt sich in ein NOT-Gatter umwandeln, indem der Eingang *A* durch einen Taktgenerator der gewünschten Elektronenfrequenz ersetzt wird. Durch ein Elektron zum richtigen Zeitpunk am dann einzig verbleibenden Eingang *B* lässt sich der Elektronenstrom gezielt unterbrechen, was als als logische Funktion $\neg B$ verstanden werden kann.

---

[71] Siehe oben, S. 14, Tabelle 1.

[72] Vgl. Heise, *Letter to A. K. Dewdney*; Moore und Owen, *The Wireworld Computer*, Abschnitt „The AND-NOT gate“.

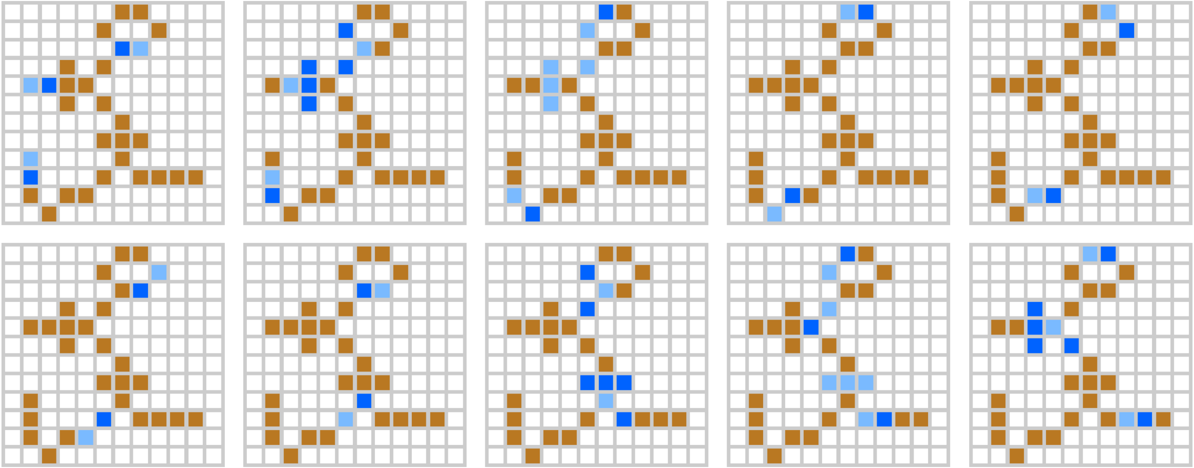

Abbildung 9: Ein vollständiges AND-Gatter, kombiniert aus einem AND-NOT-Gatter und einem NOT-Gatter. In Generation zwei hat gerade ein Elektron den 6-tick-Taktgenerator des oberen NOT-Gatters verlassen, welches nun durch das Elektron an Eingang *B*, das gerade die drei Sperrzellen aktiviert hat, blockiert wird. Es kann also den oberen Eingang des unteren AND-NOT-Gatters nicht mehr erreichen, sodass das Elektron, das zu Beginn an Eingang *A* eintraf, ungehindert zum Ausgang gelangt. Dies entspricht $1 \wedge 1 \implies 1$. Ein solches AND-Gatter ist auch in `06-logic-6-tick.txt` enthalten.

2.8 Das AND-Gatter

Schlussendlich lässt sich ein AND-Gatter erzeugen, wenn entsprechend der logischen Äquivalenz $A \wedge B \iff A \wedge \neg(\neg B)$ ein AND-NOT- mit einem NOT-Gatter kombiniert wird. In einer solchen Zellanordnung erscheint nur dann ein Elektron auf dem Ausgangsdraht, wenn vorher an beiden Eingängen je ein Elektron eingetroffen ist (siehe Abbildung 9). Damit sind für den Elektronentakt von $n = 6$ nun für alle genannten logischen Funktionen entsprechende Wireworld-Zellanordnungen gefunden. In der Beispieldatei `06-logic-6-tick.txt` sind die beschriebenen Gatter in der Reihenfolge der Unterabschnitte dieses Kapitels zu finden. Sie sind so zueinander angeordnet, dass sich ihre Ergebnisse miteinander vergleichen lassen. Als Lesehilfe für den Betrachter sind dazu oberhalb der Ausgangsdrähte einzelne Drahtzellen angebracht. Lässt man die Simulation laufen, werden in genau 24 Generationen nacheinander alle vier möglichen Kombinationen abgearbeitet. Stoppt man die Simulation zu diesem Zeitpunkt, befinden sich die Elektronenköpfe auf den Ausgangsdrähten genau auf Höhe der Lesehilfe. Von rechts nach links gelesen ergeben sich dann die Ergebnisse der Wahrheitstabelle auf S. 14, jeweils von oben nach unten gelesen.

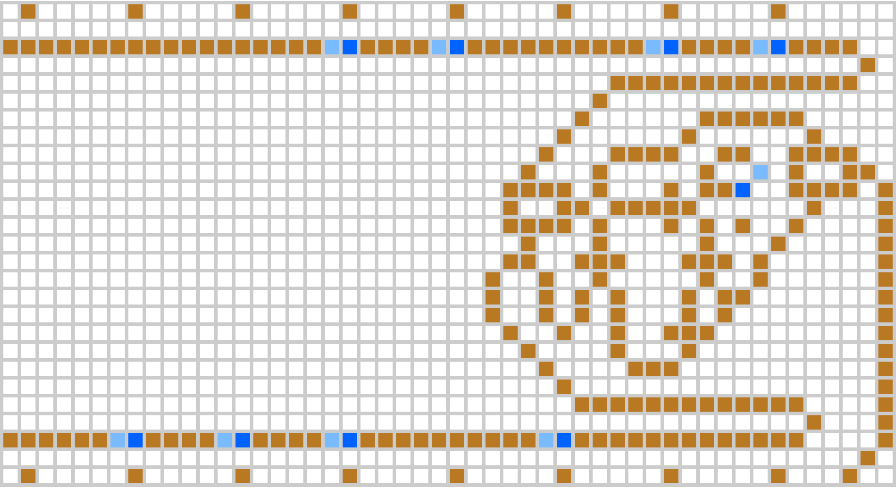

Abbildung 10: Der serielle binäre Addiere nach Moore und Owen, hier konfiguriert für die Addition 116 + 27 (dual: 110100 + 11011). Die Schaltung findet sich in `07-serial-adder-6-tick.txt`.

2.9 Der serielle Addierer

Die zuvor eingeführten Logikgatter lassen sich nun zu noch komplexeren Strukturen zusammenfügen, mit denen eintreffende Elektronen auf verschiedenste Art verarbeitet werden können. Ein Beispiel, dass hier abschließend vorgestellt wird, ist der serielle Addierer von Moore und Owen, den sie auch als Bestandteil ihres *Wireworld Computers* verwenden.[73]

Der Addierer besitzt zwei Eingänge, durch die je ein Strom aus Elektronen in die Schaltung gelangt (siehe Abbildung 10). Nach einigen Generationen Wartezeit, bedingt durch die Verzögerungsschleifen zur Synchronisation der internen Gatter, erscheint am Ausgang der Schaltung ein neuer Elektronenstrom.[74] Die logischen Eigenschaften der beteiligten Gatter sorgt dafür, dass dieser Ergebnis-Elektronenstrom die arithmetische Summe zweier dualer Zahlen darstellt (siehe Abbildung 12). Jedes Elektron aus Kopfzelle und Endzelle steht für eine duale Ziffer 1, ein im Strom fehlendes Elektron für eine 0.

Der Aufbau ist am einfachsten anhand der verwendeten Logikgatter zu erläutern. An den beiden Eingängen der Schaltung lässt sich ein EOR-Gatter erkennen. Wird die Wahrheitstabelle der Exklusiv-Oder-Funktion genauer betrachtet, so kann festgestellt werden, dass deren Ergebnisse genau mit der arithmetischen Summe zweier dualer Ziffern übereinstimmen. Diese Summe, die im schriftlichen Addieren auf dem Papier, wie es in der Grundschule erlernt wird, ‚unterm Strich‘ notiert wird, lässt sich also in einer Rechenmaschine durch ein logisches

---

[73] Vgl. Moore und Owen, *The Wireworld Computer*, Abschnitt „The binary adder“.

[74] Es sei nochmals an die Verzögerung um zwei Generationen erinnert, die am Eingang *B* eines AND-NOT-Gatters nötig ist, um seine logische Funktion zu ermöglichen, s. o., S. 17, Abschnitt 2.6.

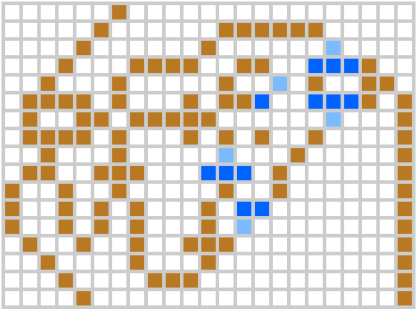

Abbildung 11: Detailansicht gegen Ende des Addierprozesses (78. Generation): Im EOR-Gatter am Ausgang des Addierers wird gerade aus der höchstwetigsten Ziffer der 116 (am oberen Eingang) und dem Übertrag der vorausgehenden Ergebniszifferberechnung (am unteren Eingang) eine duale 0 erzeugt, die vorletzte Ziffer der Berechnung. Die letzte Ziffer, die finale Übertrags-1 verlässt soeben die Flipflop-Schleife. Da keine weitere Ziffer folgt, löscht das aus dem oberhalb liegenden NOT-Gatter kommende Elektron, welches soeben die Drahtsperre am RESET-Eingang des Flipflops aktiviert hat, das bis eben noch gespeicherte Elektron und beendet damit den Elektronenstrom am Ausgang. Die Addition ist beendet.

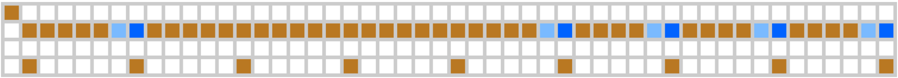

Abbildung 12: Nach genau 157 Generationen lässt sich das Ergebnis der dualen Addition ablesen: 1000111, bzw. dezimal 143.

EOR-Gatter ersetzen. Es kann beim Addieren zweier Ziffern jedoch passieren, dass man eine Ziffer in die nächsthöhere Stelle übernehmen muss – man bezeichnet dies als Übertrag, bzw. im Englischen als *carry.* Ein solcher entsteht nur dann, wenn die beiden zu addierenden dualen Ziffern 1 lauten. Man kann die Übertragsfunktion also durch ein logisches AND ersetzen. Im vorliegenden Wireworld-Addierer lässt sich das AND-Gatter etwas rechts vom Zentrum der Zellstruktur ausmachen.[75] Aus Platzgründen gibt es minimale optische Unterschiede zum oben in Abschnitt 2.8 vorgestellten AND-Gatter, die jedoch an seiner Funktion nichts ändern.

Mit diesen beiden Gattern sind alle für einen Addierer nötigen Bestandteile gegeben. Was fehlt sind lediglich ein paar Drahtschleifen und Querverbindungen, um zwei beliebig lange Zahlen im dualen Stellenwertsystem addieren zu können. Dafür sorgen die übrigen Bestandteile der Zellanordnung, wie ein Flipflop, das einen möglichen Übertrag für mindestens sechs Automatengenerationen speichert, bis das nächste Elektron – die nächsthöhere Ziffer – eintrifft.[76] Dieses Flipflop befindet sich direkt unter dem AND-Gatter. Strukturell handelt es sich dabei um eine weitere Schleife aus sechs Drahtzellen, einem SET-Eingang, der durch das OR-Gatter am unteren Ende der Schaltung gebildet wird, und einem RESET-Eingang, der durch den AND-NOT-Teil des darüberliegenden AND-Gatters gebildet wird. Sein Ausgang läuft zum unteren Eingang des abschließenden EOR-Gatters der Schaltung. Hier wird aus einem eventuellen Übertrag in der vorherigen Stelle und der Summe der zwei aktuellen Ziffern eine Ergebnisziffer erzeugt (siehe Abbildung 11).

Moore und Owen weisen auf das zusätzliche ingenieurswissenschaftliche Wissen hin, das in ihren vollständigen Addierer, über die minimal nötigen logischen Funktionen von EOR und AND hinaus, eingeflossen ist. Sie klassifizieren ihre Wireworld-Implementierung demnach wie folgt:

> The adder structure described here is called a "propagate-generate adder": A AND B generates a carry from the current bit, and A EOR B propagates a carry through it. Many fast adders implemented on real integrated circuits use exactly this technique. The simpler "ripple carry" structure, where each bit explicitely computes sum and carry outputs as direct functions of its inputs A and B and the carry output from the previous bit, are generally slower. This is especially true in Wireworld, where such a design would imply a long feedback loop in the circuit, greatly limiting the maximum speed of operation.[77]

---

75 Gut zu erkennen ist die Taktschleife des internen NOT-Gatters mit dem zirkulierenden Elektron.

76 Zur Flipflop-Schaltung in Wireworld, siehe Moore und Owen, *The Wireworld Computer*, Abschnitt „The flip-flop".

77 Ebd., Abschnitt „The binary adder". Eine Übersicht über weitere elektronische Addierschaltungen gibt z. B. Peter Pirsch. *Architekturen der digitalen Signalverarbeitung*. Stuttgart: Teubner, 1996, Abschnitt 3.2 „Addierer und Subtrahierer", S. 53 ff.

Sobald komplexere Strukturen gebaut werden, die eine bestimmte logische oder, wie in diesem Fall, arithmetische Funktion erfüllen sollen, werden alle dazu nötigen Operationen zeitkritisch oder wie Wolfgang Ernst es einleitend zu seiner aktuellen Monografie *Chronopoetik* formuliert: „Auch zelluläre Automaten sind nicht nur im Raum, sondern in der Zeit. Jedes Schalten einer diskreten Information konsumiert ein minimales Zeitintervall, mit dem buchstäblich zu rechnen ist und das zum infinitesimal kurzen Moment tendiert[.]“[78]

Das folgende Kapitel versucht, im Anschluss an dieses Zitat, das die Schalt*zeit* zellulärer Automaten betont, den medienwissenschaftlichen Blick auf den Signalbegriff näher zu untersuchen, um das bisher an praktischen Wireworld-Schaltungen demonstrierte auch medientheoretisch fruchtbar zu machen.

3 Medientheoretische Signalanalyse

Die Synchronisationsproblematik, die sich bei der sequentiellen Verschaltung grundlegender Logikbausteine zu einer komplexeren Schaltung wie dem zuvor beschriebenen seriellen Addierer einstellt, weist auf das eigentliche Problem beim Entwurf technischer Logikschaltungen hin, das weniger ein Problem der Zeichen-, bzw. Symbolinterpretation als ein Problem der genauen zeitlichen Taktung von Signalen darstellt. Was demnach der technische Begriff der Logikschaltung meint und was Medientheorie hervorheben kann, ist nicht die Funktion, Aussagen auf ihren semantischen Wahrheitsgehalt zu prüfen, sondern technische Signale auf Basis einfacher Regelsätze zeitkritisch miteinander zu kombinieren. Hierzu dient der zelluläre Automaten Wireworld in gewisser Weise als virtuelles *Messgerät* für technische Signale. Über das analogelektronische Pendant hierzu, den Logikanalysator, schreibt Wolfgang Ernst:

> In komplexen elektronischen Digitalschaltungen, die in hohem Maße auf Synchronisierung des Datenstroms angewiesen sind, dient das Meßgerät des Logik-Analysators nicht der quasi-philosophischen Überprüfung der Triftigkeit der Aussagenlogik, sondern der zeitkritischen Orchestrierung nahezu paralleler Impulsfolgen. Im Computer verhilft das *clock signal* den Schaltungen zum Abgleich im Zeitbereich; darin liegt das Dasein des Computers als Zeitvollzug.[79]

Im beschriebenen Wireworld-Addierer muss die Synchronisation der Elektronen am Eingang durch regelmäßige Lücken im Elektronenstrom vom Nutzer sichergestellt werden. Den zugrundliegenden eigentlichen Takt liefert das zelluläre Universum durch seine Diskretisierung der Zeit in einzelne Generationen. Nun soll der Begriff des *Signals,* wie ihn Wolfgang Ernst für die Medientheorie akzentuiert, näher betrachtet werden, um einige der Aspekte der

---

[78] Wolfgang Ernst. *Chronopoetik. Zeitweisen und Zeitgaben technischer Medien.* Berin: Kadmos, 2012, S. 18.
[79] Ebd., S. 314.

vorangehenden Wireworld-Beispiele aus einer dezidiert medientheoretischen Perspektive zu beleuchten. Ernst führt den Signalbegriff wie folgt ein:

> Signale, elektrophysikalisch definiert als genuine Zeitereignisse, sind ein Hauptgegenstand des medienwissenschaftlichen Blicks. Medientheorie betreibt keine Kultursemiotik strukturaler Relationen; der Signalbegriff tritt hier vielmehr an die Stelle der Zeichen.[80]

Ernst macht den Ursprung seines Signalbegriffs in der Informationstheorie und der Kybernetik aus, wo das Signal als das physikalische Trägermedium von Information definiert ist: „Rein physikalisch gesehen, ist eine Information eine in bestimmter Weise geordnete Folge von Signalen."[81] Dass hier anstatt des kanonischen Werks zur Kybernetik – Norbert Wieners *Cybernetics* von 1948 – Georg Klaus, Philosoph und maßgeblicher Wegbereiter der Kybernetik in der DDR[82] zitiert wird, liegt daran, dass Ernst sich für seinen Signalbegriff hauptsächlich auf Klaus beruft.[83] Denn im Gegensatz zu Wiener betont Klaus den Unterschied zwischen Signal und Semantik: „Information ist […] ein Ganzes aus einem physikalischen Träger und einer Semantik."[84] Klaus tut dies, um die Begriffe der Nachrichtentheorie und der Kybernetik philosophisch zu schärfen. Gleichzeitig kritisiert er Wiener, bei der Verwendung der Begriffe Information und Nachricht nur ungenau zwischen der eigentlich übertragenen Information und dem physikalischem Trägersignal zu unterscheiden. Wiener hatte in seinem zweiten Buch zur Kybernetik, *Mensch und Menschmaschine,* geschrieben: „Wenn ich […] auf einen Knopf drücke, der […] eine elektrische Kochplatte unter dem Kochtopf einschaltet, so sende ich [eine] Nachricht[…]."[85] Klaus kommentiert dies so:

> Das Knipsen eines Schalters, durch das eine elektrische Kochplatte eingeschaltet wird, hat gar nichts mit der Übertragung einer Information zu tun. Hier liegt nichts anderes vor, als daß man mit sehr geringem energetischen Aufwand größere elektrische Energiemengen steuern kann. Eine rein physikalische Ursache erzeugt hier eine rein physikalische Wirkung. Aber Wiener betrachtet die zweite Seite des Problems: „Wenn dann der elektrische Eierkocher nach einer bestimmten Anzahl von Minuten zu pfeifen beginnt, sendet er mir eine Nachricht."[86] Aber dieses Pfeifen enthält nur deswegen eine Nachricht, weil mir die Konstruktion des Eierkochers bekannt ist und die

---

80 Ernst, *Chronopoetik*, S. 17.

81 Georg Klaus. *Kybernetik in philosophischer Sicht.* Berlin: Dietz, 1961, S. 80.

82 Eine historische Einordnung von Klaus' Bemühungen um die Kybernetik in der DDR liefert Jérôme Segal. „Kybernetik in der DDR: Dialektische Beziehungen". In: *Cybernetics – Kybernetik. The Macy Conferences 1946–1953.* Bd. 2: *Essays & Documents / Essays & Dokumente.* Hrsg. von Claus Pias. Zürich und Berlin: Diaphanes, 2004, S. 227–251.

83 An einer anderen Stelle zum Signalbegriff verweist Ernst direkt auf Klaus, vgl. Ernst, *Chronopoetik*, S. 377, Anm. 776.

84 Klaus, *Kybernetik in philosophischer Sicht*, S. 81.

85 Norbert Wiener, zit. n. ebd.

86 Klaus gibt die Wiener-Zitate mit einem Verweis auf „Norbert Wiener: Mensch und Menschmaschine, Berlin 1958, S. 13" an.

> Semantik des physikalischen Trägers dieser Nachricht in meinem Bewußtsein existiert.[87]

Für Ernsts Medientheorie bedeutet die Betonung des Signalbegriffs also zunächst, die physikalischen, materiellen Grundlagen einer Übertragung, Verarbeitung und Speicherung von Information zu fokussieren – gerade auch für diskretisierte Signalspannungen im Digitalcomputer: „Der binäre Zustand ‚0' oder ‚1' ist nicht ein Zeichen, das auf ein Bezeichnetes verweist, sondern rein signalhafte Unterscheidung."[88] Was hier ebenfalls aufscheint, ist der Unterschied zwischen Syntax und Semantik in den symbolischen Codes, die ein Digitalcomputer verarbeitet. Um Ernsts medientheoretische Fokussierung auf das Signal im Unterschied zum Zeichen zu verstehen, ist ein Blick auf die Definition von Syntax und Semantik in der Semiotik Umberto Ecos erhellend:

> Der Code […] ist ein System von *syntaktischen* Regeln (er legt Vereinbarkeiten und Unvereinbarkeiten fest, wählt bestimmte Symbole aus und schließt andere als nicht zugehörig aus). […]
>
> Aber der Code unseres Ausgangsmodells macht außerdem auch noch etwas anderes: Er setzt fest, daß jedem ausgewählten Symbol eine Wasserhöhe entspricht: er sagt z. B., daß */ABC/* „Niveau 0" bedeutet. Er stellt also *semantische* Regeln auf.[89]

Wenn die Semiotik bei einer einfachen listenhaften Zuordnung von Ausdrücken eines symbolischen Vokabulars auf ein zweites (*/ABC/* ⟶ „Niveau 0") bereits von „*semantischen* Regeln" spricht, so ist dies, streng medientheoretisch betrachtet, schlicht unpräzise. Die beschriebenen Ersetzungen sind stattdessen einfache syntaktische Operationen, die in einem Digitalcomputer ständig durchgeführt werden, wie Mainzer beispielhaft darlegt:

> Eine formale Sprache, wie z. B. ein Computerprogramm besteht aus Sequenzen von Symbolen, die nach syntaktischen Regeln in Termen, Worten und Sätzen zusammengesetzt werden. Eine Bedeutung erhält der Term einer formalen Sprache dadurch, dass er einem Term in einer anderen Sprache zugeordnet wird. So steht z. B. eine bestimmte Folge von 0 und 1 im ASCII-Code eines Computers für ein Symbol, das auf dem Keyboard des Computers abgebildet ist. Die Worte unserer E-Mails stehen dann für Zusammensetzungen solcher ASCII-Codes. Die formalen ASCII-Codes haben wiederum eine operationale Bedeutung, da sie die technische Umsetzung unserer E-Mail-Nachricht in Registerzustände des Computers festlegen. Wir haben dabei also bereits drei Schichten zu unterscheiden – die Symbole und Worte auf dem Keyboard, die formalen ASCII-Codes und die technischen Registerzustände der physikalischen Maschine.[90]

Die einzige Bedeutung, die in einem Computer als symbolverarbeitende Maschine folglich zählt, ist die von Mainzer angesprochene „operationale Bedeutung". Es handelt sich dabei –

---

87 Klaus, *Kybernetik in philosophischer Sicht*, S. 81.

88 Ernst, „Signal *versus* Zeichen?", S. 329.

89 Umberto Eco. *Einführung in die Semiotik.* Aus dem Italienischen übers. von Jürgen Trabant. 5. Aufl. [Deutsche Erstausgabe: 1972; Originalausgabe: *La struttura Assente,* Mailand: Bompiani 1968]. München: Fink, 1985, S. 58.

90 Mainzer, *Die Berechnung der Welt*, S. 154.

in semiotischen Begriffen gesprochen – um rein syntaktische Operationen, die semantisch bedeutungslos sind. Aus der Perspektive der Kybernetik hält Klaus den gleichen Sachverhalt fest, erneut mit einem Seitenhieb auf Wiener: „Rein physikalische Wirkungen, die in einem geschlossenen Kontrollsystem auftreten, haben keine Semantik."[91] Medientheoretisch präzise muss hier ergänzt werden: Foglen die physikalischen Signale einem symbolischen Code, so haben sie aufgrund ihrer Syntax eine *operationale* Bedeutung, die von einer symbolverarbeitenden Maschine wie dem Digitalcomputer umgesetzt werden kann.

Unter der gezielten Umgehung der semiotischen Begriffe von Syntax und Semantik und sich stattdessen auf den physikalischen Signalbegriff berufend, will Ernst die Medientheorie begrifflich präzisieren, ähnlich wie Klaus die Kybernetik. Die Semiotik nach Umberto Eco geht zwar ebenfalls vom Signalbegriff aus, widmet sich dann aber vornehmlich der Semantik, dem „Sinn" der übertragenen Zeichen. Eco definiert diese „untere Grenze der Semiotik" als „den Punkt, an dem die Semiotik aus etwas entsteht, was nicht Semiotik ist, als das Verbindungsglied zwischer der Welt des Signals und der Welt des Sinnes".[92] Sinn, Bedeutung, Semantik entsteht demnach erst durch den Menschen, der einen „*Signifikationsprozeß* eröffnet, weil das Signal nicht mehr eine Reihe von diskreten Einheiten ist, die in *bit* Informationen berechenbar sind, sondern eine signifikante Form, die der menschliche Empfänger mit Bedeutung füllen muß."[93] Ähnlich argumentiert auch Klaus, der darauf hinweist, „daß es keine semantische Information ohne Bewußtsein gibt".[94]

3.1 Menschliche und maschinelle Perspektiven auf Signale

Wenn Wolfgang Ernst schreibt, dass „der Computer [...] eine Form der alphanumerischen *Symbol*verarbeitung [praktiziert], die quer zur analogtechnischen Welt kontinuierlicher oder diskreter *Signale* steht",[95] so meint er damit die getaktete Abarbeitung einzelner Instruktionen in ihrer operationalen Bedeutung – nicht umsonst werden diese Codes auch *Opcodes,* kurz für *operation code,* genannt. Diese numerischen Codewörter können, ausschließlich zum Zwecke ihrer Menschenlesbarkeit und -merkbarkeit, zusätzlich semantisch aufgeladen werden, indem sie in sogenannte *mnemonics* übersetzt werden; ihrerseits zunächst rein symbolische Buchstabenfolgen wie z. B. `JMP`, die aber beim menschlichen Leser bildliche Assoziationen sowie tatsächliche *Sprünge* hervorrufen können. Der Prozessor *hüpft* jedoch nach einer solchen Instruktion keineswegs wörtlich durch den Speicher. Es werden lediglich die Schalt-

---

[91] Klaus, *Kybernetik in philosophischer Sicht*, S. 82.
[92] Eco, *Einführung in die Semiotik*, S. 31.
[93] Ebd., S. 65.
[94] Klaus, *Kybernetik in philosophischer Sicht*, S. 90.
[95] Ernst, *Chronopoetik*, S. 318.

zustände, also zumeist elektronische Spannungspegel, in einem bestimmten Register namens *program counter* auf eine andere Art verändert als im normalen Programmablauf üblich. Anstatt den vorherigen Registerinhalt – eine binär codierte Zahl – durch eine arithmetische Addition um 1 zu erhöhen, wird er durch eine beliebige andere Zahl ersetzt. Der Prozessor nutzt diese Zahl im folgenden Takt, um im Hauptspeicher von einer anderen Speicherstelle als der arithmetisch folgenden den nächsten Opcode zu erhalten. Der einzige bildliche Sprung findet hierbei also, wenn überhaupt, im Menschen statt, der den Assemblercode liest und nun möglicherweise eine auf einer anderen physikalischen Seite abgedruckte Codezeile suchen muss.[96] Für den Prozessor sind jedoch alle Speicheradressen idealerweise gleich weit voneinander entfernt. Das heißt, dass jeder Speicherzugriff beim Abruf eines Opcodes, unabhängig von der zuvor abgerufenen Speicherzelle, gleichviel Zeit benötigt. Genau dafür steht auch RAM als *random access memory.* Die technische Informatik setzt hierzu beim Entwurf von Hardwareschaltungen zeitkritische Verfahren wie die Parallelisierung und das *pipelining* ein, um minimale Verzögerungszeiten im Datendurchsatz zu erreichen: „Diese Verzögerung wird gering, wenn die Schnittebenen gleichzeitig äquitemporale Ebenen des Arrays sind."[97] Gemeint ist damit, dass, wie oben im Wireworld-Addierer, die einzelnen Logikfunktionsblöcke so angeordnet werden, dass sich deren interne Verzögerungszeiten nicht nacheinadner aufaddieren, sondern möglichst parallel zueinander gesetzt sind.

Für den Menschen ergibt sich also beim *Lesen* von Computercode eine andere Perspektive als für den Computer, der die Opcodes operativ aus dem Speicher abruft, um sie zu *verarbeiten.* Ernst schließt daraus im Anschluss an Klaus: „Erst aus der Beobachterperspektive des Menschen macht es Sinn, den Signal- durch einen signifikanten Zeichenbegriff zu ersetzen".[98] An der Mensch-Maschine-Schnittstelle, die sich durch die zweidimensionale Visualisierung eines zellulären Automaten auf dem Monitor eribt, zeigt sich folglich ein Problem der Perspektive auf Signale, das Ernst mit Verweis auf einen Artikel des Informatikers und frühen Computerkünstlers Frieder Nake beschreibt:

> Je nach Perspektive findet also bei Mensch-Computer-Interaktion eine unsymmetrische Semiose (der menschliche Blick) oder eine Realisierung, die Verwandlung von Zeichen in physikalische Impulse (der Blick des Computers) statt. Beide Prozesse sind zwar unabhängig, da sie von autonomen Systemen durchgeführt werden, doch sind sie zugleich – und hier berührt Nake die Sprache der Systemtheorie – *strukturell gekoppelt.* Je nachdem, ob der Akzent hier auf Kontinuität oder auf Diskontinuität gesetzt wird (je nach hermeneutischem Willen oder nach medienarchäologischer

---

[96] In einer höheren Programmiersprache muss stattdessen eine Funktionsdefinition im Listing gesucht werden, nachdem man an anderer Stelle im Code auf ihren Aufruf gestoßen ist. Dies kann der Leser während der Analyse des ab Abschnitt 4.1 abgedruckten JavaScript-Codes selbst nachvollziehen.

[97] Pirsch, *Architekturen der digitalen Signalverarbeitung*, S. 156.

[98] Ernst, „Signal *versus* Zeichen?", S. 330.

> Insistenz also), erweist sich die Informatik als „technische Semiotik“ oder als mathematische Maschine.[99]

Die fundamentale Divergenz zwischen dem Programmcode, den ein Mensch *schreibt* und möglicherweise später noch einmal *liest,* und dem Instruktionscode, den ein Computer letztendlich *ausführt,* bzw. nach Ernst *realisiert,* wird auch anhand der Existenz zweier gänzlich verschiedener Codeklassen in einer Quellcodedatei deutlich: dem eigentlichen *Programmcode,* formuliert in der jeweiligen Programmiersprache, und den *Kommentaren,* die durch bestimmte Symbole vom übrigen Programmcode abgetrennt werden und in einer natürlichen Sprache formuliert sind – zumeist in Englisch oder der Muttersprache des Programmierers bzw. desjenigen, der den Code voraussichtlich lesen wird. Der Detailgrad der Kommentare kann sich dementsprechend fundamental unterscheiden. Wenn ein Programmierer den Code allein schreibt, sind die Kommentare häufig knapp gehalten oder fehlen ganz. Wird der Code jedoch, wie im Falle der im nun folgenden Kapitel abgedruckten Wireworld-Implementierung, gezielt für andere Programmierer oder Leser geschrieben, werden nicht selten ganze Sätze ausformuliert.[100] Die Maschine *überspringt* die Kommentare jedoch; nicht, weil sie Semantik grundsätzlich nicht versteht, sondern weil die Syntax der Programmiersprache ihr sagt, dass sie sie nicht zu interessieren hat. Für einen Compiler oder Interpreter lautet die operative Bedeutung der Symbolfolge „`//`“, die in JavaScript ein Kommentar einleitet, schlicht: *Springe zur nächsten Programmzeile* – im Sinne der oben erläuterten *operativen* Bedeutung eines Sprungs als Opcode.

4 Die Wireworld-Implementierung

Die vorliegende Implementierung von Wireworld erfolgte in der Programmiersprache JavaScript, wobei ein einfaches HTML5-Dokument mit einigen wenigen grafischen Elementen als *user interface* dient. Zur visuellen Ein- und Ausgabe wurde auf das `canvas`-Objekt des HTML5-Standards zurückgegriffen, dem Namen nach als Zeichenfläche „for rendering graphs, game graphics, art, or other visual images on the fly“ gedacht.[101] JavaScript ist eine objektorientierte Programmiersprache (OOP), allerdings wird die Objektorientierung über andere syntaktische Mittel eingeführt als dies in anderen Sprachen der Fall ist, wie z. B. in

---

[99] Ernst, „Signal *versus* Zeichen?“, S. 329.

[100] Siehe die detaillierten Kommentare im unten ab Abschnitt 4.1 abgedruckten Code, sowie die im Vergleich dazu eher spärlichen Kommentare in Johannes Maibaum. *basic-life.* Github-Repository. URL: `https://github.com/jmaibaum/basic-life` (besucht am 06. 04. 2015). *Das Repository enthält die Entwicklungsschritte der vorliegenden Wireworld-Implementierung im Unterordner* `JavaScript`.

[101] Ian Hickson u. a. *HTML5. A vocabulary and associated APIs for HTML and XHTML. W3C Recommendation.* 28. Okt. 2014. URL: `http://www.w3.org/TR/html5/` (besucht am 06. 04. 2015), Abschnitt 4.11.4.

Java oder in C++. Der wohl auffälligste Unterschied zu diesen beiden Sprachen ist das Fehlen des Schlüsselworts `class`. In JavaScript erfolgt die Erzeugung von Objekten aus der Gruppierung von Variablen und Methoden stattdessen über die Definition und Erweiterung sogenannter *Prototypen:*

> Prototype-based programming is an OOP model that doesn't use classes, but rather accomplishes behavior reuse (equivalent to inheritance in class-based languages) by decorating (or expanding upon) existing *prototype* objects. (Also called classless, prototype-oriented, or instance-based programming.)[102]

Auch wenn in der Beschreibung von Programmiersprachen semantisch-formal zwischen *klassen-* und *prototypenbasierter* Objektorientierung unterschieden wird, verbleibt dieser Unterschied *operativ* betrachtet auf der syntaktischen Ebene. Die „Vererbung" von „Verhalten" – operativ genauer formuliert: das Kopieren von Variablen und Funktionen beim Bereitstellen von Speicherplatz für neu zu erzeugende „Objekte" – wird in JavaScript schlicht über andere Schlüsselwörter erreicht als z. B. in C++. Deutlich wird dies in der folgenden Formulierung der bereits zitierten *Introduction to Object-Oriented JavaScript* im Mozilla Developer Network: „Instead [of the `class` statement, J. M.], JavaScript uses functions as classes."[103] Dass mit der Definition einer Funktion also operativ eine „Klasse" – bzw. genauer: ein *Objektprototyp* – gemeint ist, ergibt sich dann aus den verwendeten Schlüsselwörtern in der Definition und ihrer Verwendung im übrigen Programm.

```
var Cell = function ()
{
    this.state = 0;
    this.neighbours = 0;
}
```

Listing 1: Die Definition des `Cell`-Prototypen.[104]

Am Beispiel der Wireworld-Implementierung lässt sich dies z. B. am `Cell`-Prototypen zeigen, von dem alle einzelnen Zellen im Automatenuniversum abgeleitet werden (siehe Listing 1). Es handelt sich syntaktisch um die Definition einer Funktion, die an den Variablenbezeichner `Cell` gebunden wird. Dass diese Funktion in ihrer Operation später „Objekte" vom Typ `Cell` erzeugen kann, darauf deutet das Vorkommen des Schlüsselworts `this` in den Variablendefinitionen innerhalb der Funktionsdefinition hin. Es sorgt dafür, dass für jedes `Cell`-Objekt je einen eigener Bereich im Hardwarespeicher reserviert wird, auf den nur

---

[102] Mozilla Developer Network, Hrsg. *Introduction to Object-Oriented JavaScript.* 4. Apr. 2015. URL: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Introduction_to_Object-Oriented_JavaScript` (besucht am 06. 04. 2015).

[103] Ebd.

[104] Im Kontext des kompletten Programms in der Datei `common.js`, s. u., Abschnitt 4.1 ab S. 32, Zeilen 46–50.

das jeweilige Objekt – die singuläre *Instanz* – zugreifen kann, sodass dort die Zustände des Objekts gespeichert werden können. Im konkreten Beispiel handelt es sich um den aktuellen Status der Zelle und die Anzahl ihrer „lebenden“ Nachbarzellen. Eine Funktion wie diese, die neue Objekte erzeugen kann, wird auch *Konstruktor*-Funktion genannt.

Nun gehört zur objektorientierten Programmierung nicht nur die eben beschriebene Kapselung von instanzeigenen Variablen in distinkten Objekten, sondern auch von entsprechenden Funktionen, häufig *Methoden* genannt, die das *Verhalten* der erzeugten Objekte definieren. In Sprachen wie C++ würde man die entsprechenden Funktionsdefinitionen innerhalb eines `class`-Blocks angeben, um ihre Zugehörigkeit zur entsprechenden Klasse von Objekten zu erklären. In JavaScript als *prototypenbasierter* Sprache sieht das operationale Äquivalent syntaktisch leicht anders aus. Eine Methode ist wiederum eine Funktionsdefinition, die syntaktisch *an den jeweiligen Objektprototypen angehängt wird* (siehe Listing 2). Jede Instanz eines `Cell`-Objekts verfügt anschließend über eine entsprechende Methode zu Änderung ihres Zustands.

```
Cell.prototype.changeCellState = function (cellState)
{
    // ...
}
```

Listing 2: Die Syntax einer Methodendefinition in JavaScript.[105]

Der Vorteil, der objektorientierten Programmiersprachen aufgrund dieser syntaktischen Mittel allgemein nachgesagt wird, besteht in der Moularität und Austauschbarkeit der deklarierten Objekte und ihrer Methoden sowie der angeblich leichteren Verständlichkeit des ausformulierten Codes:

> OOP promotes greater flexibility and maintainability in programming, and is widely popular in large-scale software engineering. Because OOP strongly emphasizes modularity, object-oriented code is simpler to develop and easier to understand later on. Object-oriented code promotes more direct analysis, coding, and understanding of complex situations and procedures than less modular programming methods.[106]

Dem erwähnten Modularitätsaspekt kann problemlos zugestimmt werden, denn die Entwicklung des Programms begann zunächst mit einer Implementierung von *Game of Life.* Nachdem dieser Automat funktionierte, wurden lediglich entsprechende Module für Wireworld entwickelt, namentlich die Logik für die zusätzlichen Zellzustände und den entsprechen-

---

105 Die vollständige Definition der `Cell`-Methode `changeCellState` findet sich in der Datei `wireworld.js`, unten in Abschnitt 4.2 ab S. 42 in den Zeilen 18–29.

106 Mozilla Developer Network, *Introduction to Object-Oriented JavaScript*.

den Regelsatz anstatt der *Game-of-Life*-Module eingebaut.[107] Ob dieser Code anschließend wirklich *leichter zu verstehen* ist, muss jeder Leser selbst entscheiden. Gegen eine leichtere Verständlichkeit spräche beispielsweise die schiere Länge der JavaScript-Version in Relation zur im Seminar gemeinsam entwickelten Version in weitestgehend imperativem TrueBasic.[108] Allerdings verfügt die vorliegende Implementierung auch über einen deutlich höheren Funktionsumfang, wie die Möglichkeit zum interaktiven Zeichnen von Zellen mit der Maus auf dem `canvas`, sowie zum Speichern und Laden von Zellkonfigurationen.

Der Entscheidung, eine objektorientierte Programmiersprache zu verwenden, lag noch ein weiterer, letzter, Aspekt zugrunde, nämlich die konzeptuelle Ähnlichkeit zwischen den in der Programmiersprache beschriebenen Objekten und den Zellen im Automatenuniversum, auf die auch vom Mozilla Developer Network hingewiesen wird: „Each object can be viewed as an independent little machine with a distinct role or responsibility."[109] Die objektorientierte Programmierung eines zellulären Automaten schließt demnach den Kreis zur in der Einleitung vorgestellten Automatentheorie, die über den gleichen Maschinenbegriff ihre eigenen Begriffe des Automatenuniversums, der Ausgangskonfiguration einer Automatensimulation, sowie der einzelnen Zelle einführt und diskutiert:

> Before the general definitions and descriptions of tesselation structures are given in further detail, the reader should be cautioned that there are three different kinds of things which can be called machines. The tesselation itself can be considered to be a machine, although in a tesselation model of self-reproduction the tesselation is more naturally considered to be the environment or the universe (including the supplies of parts or raw materials) in which the self-reproduction takes place. A configuration (which is restricted to an array of finite size) can be considered to be a machine, and in fact it is a machine of this kind which can be shown to reproduce itself. A cell can be considered to be a machine, since it has a list of states and transitions, but it corresponds in the tesselation models of self-reproduction to one of the elementary parts out of which the machine is built.[110]

Im Programmcode lassens sich somit zwei ineinander verschachtelte ‚Maschinen' als JavaScript-Objekte lokalisieren: Das `WireField`-Objekt, das in der Implementierung über das

---

107 Hieraus resultiert auch die Aufteilung der Programmlogik in die zwei Dateien `common.js` und `wireworld.js`. Der *Game-of-Life*-Regelsatz ist ebenfalls auf der beiliegenden CD enthalten. Die entsprechenden Codemodule befinden sich in der Datei `life.js`. Der dort enthaltene Code wurde nicht detailliert dokumentiert und ist auch unten nicht abgedruckt, sollte jedoch auch anhand der Kommentare in `wireworld.js` verständlich werden. Um *Game of Life* zu starten, muss lediglich die Datei `life.html` im Webbrowser geöffnet werden.
108 Die letzte im Kurs entwickelte TrueBasic-Version von *Game of Life* umfasst lediglich 72 Programmzeilen. Vgl. Maibaum, *basic-life*, Datei `LIFE.TRU`.
109 Mozilla Developer Network, *Introduction to Object-Oriented JavaScript*.
110 Moore, „Machine Models of Self-Reproduction", S. 20. Das Moore hier, im Jahr 1962, anstatt von *zellulären* Strukturen, von *tesselation structures* schreibt, liegt daran, dass die heute übliche Terminologie erst vier Jahre später, mit der Veröffentlichung von Neumanns' *Theory of Self-Reproducing Automata* durch Burks etabliert wurde.

Schlüsselwort `new` genau einmal erzeugt wird[111] und in seinem Speicherbereich entsprechend der gewünschten Größe des zellulären Universums die benötigte Anzahl an Instanzen des `Cell`-Objekts anlegt.[112]

4.1 Kommentierter Quellcode der Datei common.js

```
 1  /*
 2   * In dieser Datei sind allgmeine Variablen, Datentypen und Funktionen
 3   * definiert, die auch zur Implementierung von anderen Zellularautomaten
 4   * nützlich sind.
 5   */
 6
 7  // Hier wird der "Einstiegspunkt" für das Programm festgelegt: Sobald das
 8  // HTML-Dokument vollständig geladen ist, löst der Browser den
 9  // "DOMContentLoaded"-Event aus, der hier mit dem Aufruf der Funktion
10  // initialize() verknüpft wird:
11  document.addEventListener("DOMContentLoaded", initialize, false);
12
13  // Die Variable Settings gruppiert alle globalen Einstellungen. Der Großteil der
14  // Werte wird zunächst auf '0' (bzw. je nach Funktion auf entsprechende
15  // Standardwerte) initialisiert. Die eigentlichen Einstellungen werden erst
16  // durch die Funktion initialize() gesetzt:
17  var Settings = {
18      boardDivision: 0,        // Enthält später die Zellunterteilung in Pixeln.
19      boardSideLenght: 0,      // Enthält später die Seitenlänge des Spielfelds in
20                               // Pixeln.
21      canvas: undefined,       // Enthält später die Referenz auf den Canvas im
22                               // HTML-Dokument.
23      context: undefined,      // Enthält später die Referenz auf den
24                               // Zeichenkontext im Canvas, damit JavaScript weiß,
25                               // wohin es zeichnen soll.
26      cellsPerLine: 50,        // Teilt den canvas in 50 Zellen pro Zeile.
27      cellSize: 0,             // Enthält später die tatsächliche Größe einer
28                               // Zelle (in Pixeln), benötigt zum Zeichnen.
29      cellColor: ["#ffffff"],  // Diese Liste wird in wireworld.js mit den übrigen
30                               // Zellfarben ergänzt.
31      cellToDraw: 1,           // Hier wird der aktuell zu zeichnende Zelltyp
32                               // festgelegt.
33      intervalID: 0,           // Enthält später eine Referenz auf das Intervall-
34                               // Timerobjekt, das den Ablauf des Automaten
35                               // steuert.
36      speed: 100               // Speichert die aktuelle Geschwindigkeit der
37                               // Simulation (in ms).
38  };
39
40
41  // Hier wird Cell als ein eigener Datentyp definiert, der eine einzelne Zelle
```

---

111 In der Datei `wireworld.js`, s. u., Abschnitt 4.2 ab S. 42, Zeile 190.

112 In der Datei `common.js`, s. u., Abschnitt 4.1 ab S. 32, Zeile 86.

```
// repräsentiert. So kann später für jede Zelle sowohl ihr Status als auch die
// Anzahl ihrer lebenden Nachbarn gespeichert werden. Beide Attribute werden mit
// '0' initialisiert. Der Status '0' enspricht in der Game-of-Life-Terminologie
// einer 'toten' Zelle, '0' Nachbarn sind selbsterklärend.
var Cell = function ()
{
    this.state = 0;
    this.neighbours = 0;
}


// Field ist der Datentyp für das Spielfeld. Mit der folgenden Konstruktor-
// Funktion wird er initialisiert:
var Field = function (rows, columns)
{
    this.rows  = rows;                // Die Attribute rows und colums werden
    this.columns = columns;           // für diverse Schleifen als Maximalwerte
                                      // benötigt und deswegen hier zentral
                                      // gespeichert.
    this.generation = 0;              // Enthält die aktuelle Generation.
    this.generationHTML = undefined;  // Wird durch initialize() gesetzt,
                                      // verweist dann auf die
                                      // Generationenanzeige im HTML-Dokument.
    this.population = 0;              // Enthält die aktuelle Population (in
                                      // Wireworld: Die Anzahl der
                                      // Elektronenköpfe.)
    this.oldPopulaton = 0;
    this.populationHTML = undefined;  // Wird durch initialize() gesetzt,
                                      // verweist dann auf die
                                      // Populationsanzeige im HTML-Dokument.
    this.data = [];                   // Der eigentliche Spielfeld-Array.

    // Fülle den Spielfeld-Array mit Cell-Objekten. Zur Vereinfachung der
    // Regelanwendung wird eine zusätzliche Zeile außen um das sichtbare
    // Spielfeld erzeugt, daher 'i < this.rows + 2', bzw. 'i < this.coloumns +
    // 2':
    for (var i = 0; i < this.rows + 2; i++) {
        // Jede Zeile ist selbst ein Array, sodass wir insgesamt ein
        // zweidimensionales Array erhalten:
        this.data[i] = [];

        for (var j = 0; j < this.columns + 2; j++)
            // Hier werden die eigentlichen Cell-Objekte mittels der
            // Arary-Methode push() erzeugt und angehängt:
            this.data[i].push(new Cell());
    }
}

// updateNeighbours() verändert den Nachbarzähler einer Zelle, indem der
// zweidimensionale Zellenarray rund um die eigenen Koordinaten entsprechend der
// Moore-Nachbarschaft durchlaufen wird. Sie muss jedesmal aufgerufen werden,
// wenn sich der Status einer Zelle entsprechend ändert. Durch den dritten
```

```
// Parameter 'increment' ist die Funktion allgemein einsetzbar, sowohl nach dem
// Zeichnen einer neuen, wie auch dem Entfernen einer bereits vorhandenen
// Zelle. Hier muss für increment lediglich '-1' übergeben werden.
Field.prototype.updateNeighbours = function (row, column, increment)
{
    this.data[row - 1][column - 1].neighbours += increment;
    this.data[row - 1][column].neighbours     += increment;
    this.data[row - 1][column + 1].neighbours += increment;
    this.data[row][column - 1].neighbours     += increment;
    this.data[row][column + 1].neighbours     += increment;
    this.data[row + 1][column - 1].neighbours += increment;
    this.data[row + 1][column].neighbours     += increment;
    this.data[row + 1][column + 1].neighbours += increment;
}

// resetAllNeighbours() wird von der nextGeneration()-Funktion aus aufgerufen,
// um anschließend, d.h. nach der Regelanwendung, alle Nachbarn neu zu zählen:
Field.prototype.resetAllNeighbours = function()
{
    for (var row in this.data)
        for (var column in this.data[row])
            this.data[row][column].neighbours = 0;
}

// increaseGeneration() erhöht den Generationszähler und aktualisiert die
// Anzeige im HTML-Dokument:
Field.prototype.increaseGeneration = function()
{
    this.generation++;
    this.generationHTML.innerHTML = this.generation;
}

// updatePopulationHTML() aktualisiert die Anzeige der Population. In Wireworld
// gibt die Population die Anzahl der Elektronenköpfe wieder:
Field.prototype.updatePopulationHTML = function ()
{
    // Aktualsiere nur, wenn sich an der absoluten Anzahl etwas geändert hat!
    if (this.population != this.oldPopulation) {
        this.populationHTML.innerHTML = this.population;
        this.oldPopulation = this.population;
    }
}

// clear() setzt alle Zellen auf Status '0', also 'leer' zurück. Wird nach
// Nutzerklick auf den 'Clear Field'-Button aufgerufen:
Field.prototype.clear = function ()
{
    for (var row = 1; row <= this.rows; row++) {
        for (var column = 1; column <= this.columns; column++) {
            this.createCell(row, column, 0, -1);
            drawCell(row, column, 0);
            this.updatePopulationHTML();
```

```
        }
    }

    // Generationszähler erst intern zurücksetzen und anschließend im HTML-
    // Dokument übernehmen:
    this.generation = 0;
    this.generationHTML.innerHTML = this.generation;
}

// initialize() ist der Einstiegspunkt des Programms: HTML-Elemente ermitteln,
// die Canvas-Zeichenoberfäche vorbereiten und EventListener für Nutzerintaktion
// setzen (Mausklicks zum Zeichnen von Zellen sowie die 'DropZone' für das Laden
// von gespeicherten Zellkonfigurationen):
function initialize()
{
    // HTML-Elemente ermitteln:
    Settings.canvas = document.getElementById("board");
    Settings.context = Settings.canvas.getContext("2d");
    field.generationHTML = document.getElementById("Generation");
    field.populationHTML = document.getElementById("Population");

    // Zeichenoberfläche vorbereiten. Es wird eine quadratische Grundstruktur
    // angenommen:
    Settings.boardSideLength = Settings.canvas.width;
    // Durch abrundendes Teilen der Seitenlänge des Canvas' durch die Anzahl der
    // zu zeichnenden Zellen wird die Unterteileilung des Canvas' (in Pixel)
    // ermittelt:
    Settings.boardDivision = Math.floor(Settings.boardSideLength /
                                        Settings.cellsPerLine);
    // Von dieser boardDivision wird die Breite des noch zu zeichnenden
    // Gitternetzes subtrahiert, um die tatsächliche Zellenbreite zu erhalten:
    Settings.cellSize = Settings.boardDivision - 2;

    // Nun kann das Gitternetz gezeichnet werden:
    Settings.context.strokeStyle = "#cccccc";  // Farbe des Gitternetzes
    Settings.context.lineWidth = 2;            // Linienbreite in Pixeln

    // Pro Schleifendurchlauf wird je eine horizontale und eine vertikale Linie
    // gezeichnet:
    for (var i = 0; i <= Settings.boardSideLength;
         i = i + Settings.boardDivision) {
        Settings.context.moveTo(0, i);
        Settings.context.lineTo(Settings.boardSideLength, i);
        Settings.context.moveTo(i, 0);
        Settings.context.lineTo(i, Settings.boardSideLength);
        Settings.context.stroke();
    }

    // EventListener für Mausklicks auf dem Canvas:
    Settings.canvas.addEventListener("mousedown", drawCellFromMouseClick,
                                     false);

```

```
    // EventListener, um den Canvas zu einer 'DropZone' für das Laden von
    // Zellkonfigurationen zu machen:
    var dropZone = Settings.canvas;
    dropZone.addEventListener("dragenter", dragenter, false);
    dropZone.addEventListener("dragover", dragover, false);
    dropZone.addEventListener("drop", drop, false);
}



/*
 * Ende des Codes für die grundsätzliche Zellularautomatenfunktionalität.
 *
 * Ab hier folgt nur noch Code, der ausschließlich für die Nutzerinteraktion
 * benötigt wird.
 */

// drawCellFromMouseClick() ist die Callback-Funktion, wenn der Nutzer innerhalb
// des Canvas' einen Mausklick ausführt. Die Koordinaten des Mausklicks werden
// ermittelt, auf die entsprechenden Spielfeldkoordinaten umgerechnet und dort
// eine Zelle gezeichnet.
//
// Der Code wurde adaptiert von:
// http://miloq.blogspot.de/2011/05/coordinates-mouse-click-canvas.html
function drawCellFromMouseClick(event)
{
    // Ermittle die absoluten Koordinaten des Mausklicks (bezogen auf das
    // gesamte Browserfenster):
    var x = event.x;
    var y = event.y;

    // Firefox unterstützt event.x und event.y nicht, folgender Workaround
    // funktioniert jedoch analog:
    if (x == undefined && y == undefined) {
        x = event.clientX;
        y = event.clientY;
    }

    // Wenn die Ansicht im Browserfenster durch Scrollen bewegt wurde, muss
    // dieser Versatz mit einberechnet werden:
    x += window.pageXOffset;
    y += window.pageYOffset;

    // Abzüglich des Pixel-Koordinaten der oberen linken Ecke des Canvas'
    // erhalten wir die korrekten Klick-Koordinaten innerhalb des Canvas':
    x -= Settings.canvas.offsetLeft;
    y -= Settings.canvas.offsetTop;

    // Übersetzung der Pixel-Koordinaten in die entsprechenden Spielfeld-
    // Koordinaten (row, column):
    var coords = convertClickToFieldCoordinates(x, y);
    var row    = coords[0];
```

```
    var column = coords[1];

    // Versuche, an den ermittelten Koordinaten eine Zelle zu erstellen. Wenn
    // dies gelingt, zeichne die entsprechende Zelle auf den Canvas. Wenn es
    // jedoch misslingt, weil sich die dort befindliche Zelle bereits im
    // gewünschten Zustand befindet, gibt field.createCell() false zurück und
    // der else-Fall tritt ein. Der Mausklick 'löscht' dann die an den
    // Koordinaten befindliche Zelle, indem die dortige Zelle auf dem Status
    // '0', bzw. 'leer' gesetzt wird.
    if (field.createCell(row, column, Settings.cellToDraw, 1))
        drawCell(row, column, Settings.cellToDraw);
    else {
        field.createCell(row, column, 0, -1);
        drawCell(row, column, 0);
    }
}

// convertClickToFieldCoordinates() rechnet pixelbasierte Klick-Koordinaten in
// die entsprechenden Spielfeld-Koordinaten um:
function convertClickToFieldCoordinates(x, y) {
    // Math.floor() rundet auf ganze Zahlen ab, das '+ 1' am Ende berücksichtigt
    // die 'unsichtbaren' Randzellen, die rund um das sichtbare Spielfeld
    // liegen.
    var row = Math.floor(y / Settings.boardDivision) + 1;
    var column = Math.floor(x / Settings.boardDivision) + 1;

    // Wird ganz am rechten, bzw. am unteren Rand des Canvas' geklickt, können
    // Koordinaten erzeugt werden, die außerhalb des gültigen Bereiches
    // liegen. Wenn dies der Fall ist, reduziere die Koordinaten entsprechend.
    if (row > field.rows)
        row--;

    if (column > field.columns)
        column--;

    return [row, column];
}

// drawCell() zeichnet eine Zelle auf den Canvas:
function drawCell(row, column, state)
{
    // context.fillStyle bestimmt die Zeichenfarbe. Setze sie entsprechend des
    // zu malenden Zellstatus (die entsprechenden Farbdefinitionen finden sich
    // zu Beginn der wireworld.js):
    Settings.context.fillStyle = Settings.cellColor[state];

    // Die Parameter der Funktion sind Feldkoordinaten, wir benötigen jetzt die
    // entsprechenden Pixelkoordinaten auf dem Canvas. Im Prinzip die inverse
    // Funktion zu convertClickToFieldCoordinates() oben. Das '+ 1' sorgt hier
    // dafür, dass das Zellgitter nicht übermalt wird:
    var drawx = Math.floor(column * Settings.boardDivision)
        - Settings.boardDivision + 1;
```

```
    var drawy = Math.floor(row * Settings.boardDivision)
        - Settings.boardDivision + 1;

    // Zeichne das eigentliche Quadrat:
    Settings.context.fillRect(drawx, drawy, Settings.cellSize,
                              Settings.cellSize);
}

// clearField() wird aufgerufen, wenn der Nutzer auf den 'Clear Field'-Button
// klickt.
function clearField()
{
    // Es wird lediglich auf die Methode field.clear() umgeleitet:
    field.clear();
}

// runAutomata() startet die automatische Simulation des zellulären Automaten.
function runAutomata()
{
    // Wenn der Automat noch nicht läuft starte einen regelmäßigen Timer, der
    // alle <Settings.speed> Millisekunden die Funktion
    // nextGenerationFromInterval() aufruft.
    if (!Settings.intervalID)
        // Die Referenz auf diesen Timer (die intervalID) wird in den Settings
        // gespeichert, um ihn später wieder stoppen zu können:
        Settings.intervalID = window.setInterval(nextGenerationFromInterval,
                                                 Settings.speed);
}

// stopAutomata() stoppt die Simulation wieder.
function stopAutomata()
{
    // Stoppe den Timer und lösche die Referenz aus den Settings:
    window.clearInterval(Settings.intervalID);
    Settings.intervalID = 0;
}

// changeSpeed() wird aufgerufen, wenn der Nutzer ein anderes Element aus der
// Geschwindigkeits-Auswahlliste auswählt.
function changeSpeed(selectItem)
{
    // Speichere den neu gewählten Wert:
    Settings.speed = selectItem.value;

    // Falls die automatische Simulation bereits läuft, stoppe kurz und starte
    // erneut mit der neuen Geschwindigkeit:
    if (Settings.intervalID) {
        stopAutomata();
        runAutomata();
    }
}

```

```
// nextGeneration() wird aufgerufen, wenn der Nutzer auf den 'Next Generation'-
// Button klickt.
function nextGeneration()
{
    // Berechne nur dann die nächste Generation, wenn die automatische
    // Simulation gestoppt ist (Settings.intervalID ist dann = 0), andernfalls
    // stoppe den Automaten, weil der Nutzer nun offenbar selbst über die
    // Generationenschritte entscheiden möchte:
    if (!Settings.intervalID)
        field.nextGeneration();
    else
        stopAutomata();
}

// Callback-Funktion für den Timer.
function nextGenerationFromInterval()
{
    // Es wird auf field.nextGeneration umgeleitet:
    field.nextGeneration();
}


/*
 * saveFile() und loadFile() ermöglichen das Speichern und Laden von
 * Zellkonfigurationen in und aus einfachen Textdateien.
 *
 * Der Code für die beiden Funktionen wurde adaptiert von:
 * https://thiscouldbebetter.wordpress.com/2012/12/18/loading-editing-and-
 * saving-a-text-file-in-html5-using-javascrip/
 */

// saveFile() speichert die aktuelle Zellkonfiguration auf dem Canvas in einer
// Textdatei. Da JavaScript im Webbrowser nur eingeschränkte Rechte besitzt,
// z.B. nicht schreibend auf das Dateisystem zugreifen darf, was nötig wäre, um
// wie in einer normalen Desktop-Applikation einen 'Datei speichern'-Dialog
// anzeigen und die Datei dann auch tatsächlich abspeichern zu können, muss hier
// getrickst werden. Es wird ein zusätzlicher (unsichtbarer) Downloadlink
// erzeugt, der von JavaScript selbst "angeklickt" wird und anschließend wieder
// entfernt wird. Ähnlich funktioniert auch das von diversen Downloadseiten im
// Internet bekannte "Ihr Download startet automatisch in 5 Sekunden..."
function saveFile()
{
    // Erzeuge die zu speichernden Textdaten aus dem aktuellen Spielfeld:
    var textToWrite = '';

    // Beginnend bei (1,1), also nur den im Canvas sichtbaren Teil:
    for (var row = 1; row <= field.rows; row++) {
        for (var column = 1; column <= field.columns; column++) {
            // field.generateChar() gibt das zum Zellstatus gehörende ASCII-
            // Zeichen zurück. Die Definition befindet sich in wireworld.js.
            textToWrite += field.generateChar(row, column);
        }
```

```
        // Beende jede Zeile mit einem 'New line'-Zeichen. Dies wäre zwar nicht
        // unbedingt nötig, sorgt aber dafür, dass die erzeugte Textdatei
        // menschenlesbar ist und z.B. von Hand in einem Texteditor bearbeitet
        // werden kann:
        textToWrite += '\n';
    }

    // Der Inhalt des Textfeldes 'FileName' dient als Dateiname:
    var fileName = document.getElementById("FileName").value;

    // Ab hier beginnt die Trickserei: Aus dem oben erzeugten textToWrite wird
    // ein temporäres Blob-Objekt erzeugt, das wie eine normale Textdatei auf
    // der Festplatte aussieht. (Blob ist Teil der File API, die aktuell von der
    // W3C für die Interaktion mit Dateien in JavaScript entwickelt wird, siehe:
    // http://www.w3.org/TR/FileAPI/
    var textFileAsBlob = new Blob([textToWrite], {type:'text/plain'});

    // Nun wird der "unsichtbare" Download-Link erzeugt:
    var downloadLink = document.createElement("a");
    downloadLink.download = fileName;

    // Die folgende if-Abfrage ist nötig, da die aufzurufende Funktion in
    // verschiedenen Browsern unterschiedliche Namen hat:
    if (window.webkitURL != null) {
        // Webkit-basierte Broser, wie Google Chrome oder Apple Safari:
        downloadLink.href = window.webkitURL.createObjectURL(textFileAsBlob);
    } else {
        // Mozilla Firefox:
        downloadLink.href = window.URL.createObjectURL(textFileAsBlob);

        // Firefox benötigt einen zusätzlichen Workaround gegenüber Webkit-
        // Browsern. Hier muss der Link tatsächlich zum DOM (Document Object
        // Model, der browserinternen Repräsentation des HTML-Dokuments)
        // hinzugefügt werden, bevor er von JavaScript "angeklickt" werden kann.
        // Die Funktion onclick() sorgt dafür, dass dies nach dem simulierten
        // Klick sofort wieder rückgängig gemacht wird:
        downloadLink.onclick = function (e) {
            document.body.removeChild(e.target);
        };
        downloadLink.style.display = "none";     // Mach den Link unsichtbar und
        document.body.appendChild(downloadLink); // hänge ihn ans Dokument an.
    }

    // Simuliere einen Klick auf den erzeugten Link, um den Download auszulösen.
    downloadLink.click();
}

// loadFile() lädt eine entsprechend formatierte Textdatei und parst ihren
// Inhalt in den Spielfeld-Canvas.
function loadFile(files)
{
    // Wähle nur die erste Datei aus der files-Liste aus (files können mehrere
```

```
    // Dateien seien, wenn der Nutzer mehrere markiert und gleichzeitig in den
    // Canvas gezogen hat).:
    var file = files[0];

    // Regulärer Ausdruck, der auf die MIME-Typen von Textdateien passt:
    var textType = /text.*/;

    // Fahre nur fort, wenn es sich wirklich um eine Textdatei handelt:
    if (file.type.match(textType)) {
        // Erzeuge ein FileReader-Objekt (Teil der File API, siehe den Link oben
        // zu Blob):
        var reader = new FileReader();

        // Die onload()-Funktion wird aufgerufen, wenn der Browser die Datei
        // fertig von der Festplatte gelesen hat.
        reader.onload = function (e) {
            field.clear();

            // Wir laden nur in den sichtbaren Teil des Spielfelds, deswegen
            // beginnen wir bei (1,1):
            var row = 1, column = 1;

            // reader.result enthält den Inhalt der Textdatei in einem Array,
            // Zeichen für Zeichen. Mit einer for-Schleife können wir also --
            // Zeichen für Zeichen -- mit dem Parsen fortfahren:
            for (var charNr in reader.result) {
                // field.parseChar() erzeugt eine dem ASCII-Zeichen
                // entsprechende Zelle auf dem Canvas. Die Definition findet
                // sich in wireworld.js:
                field.parseChar(row, column, reader.result[charNr]);

                // Erhöhe den Spaltenzähler:
                column++;

                // Wenn dies die letzte Spalte war, setze den Spaltenzähler auf
                // '0' zurück und erhöhe den Zeilenzähler:
                if (column > field.columns) {
                    column = 0;
                    row++;
                }
            }

            // Zähle alle Nachbarn und aktualisiere den Populationszähler.
            field.countAllNeighbours();
            field.updatePopulationHTML();
        };

        // Nachdem das FileReader-Objekt um die nötige Lesefunktion onload
        // ergänzt wurde, starte nun den eigentlichen Lesevorgang von der
        // Festplatte.
        reader.readAsText(file);
    } else {
```

```
        console.log("File␣not␣supported.")
    }
}

// Callback-Funktionen für die 'DropZone' auf dem Canvas. dragenter() und
// dragover() tun nichts. In diesen Funktionen wird nur verhindert, dass
// evtl. im Browser definierte Standardaktionen für die genannten Ereignisse
// ausgeführt werden.
function dragenter(e)
{
    e.stopPropagation();
    e.preventDefault();
}

function dragover(e)
{
    e.stopPropagation();
    e.preventDefault();
}

// drop() gibt die gewünschte(n) Datei(en) an die loadFile()-Funktion weiter:
function drop(e)
{
    e.stopPropagation();
    e.preventDefault();

    loadFile(e.dataTransfer.files);
}

// handleKeyPress() vereinfacht die Nutzerinteraktion: Befindet sich der Nutzer
// mit seinem Text-Cursor innerhalb eines bestimmten Textfeldes, kann mit der
// Returntaste eine Funktion ausgelöst werden.
function handleKeyPress(event, textField)
{
    if (event.keyCode === 13) {  // '13' ist der keyCode für die 'Return'-Taste.
        // Wenn im textField namens 'FileName', rufe die Speicherfunktion auf:
        if (textField == "FileName")
            saveFile();
        // Wenn im textField 'RandomCells', fülle den Canvas zufällig. (In
        // Wireworld ohne Bedeutung, Überbleibsel des Life-Automaten)
        else if (textField == "RandomCells")
            fillWithRandomCells();
    }
}
```

4.2 Kommentierter Quellcode der Datei wireworld.js

```
/*
 * Diese Datei enthält die Wireworld-spezifischen Ergänzungen zum allgemeinen
 * Zellularautomatencode in common.js, hauptsächlich die Regelanwendungen, sowie
 * den Parsercode für die Datei-laden/speichern-Funktionen.
 */
```

```
// Erweitere die Zellfarbenliste mit den Wireword-Zelltypen:
Settings.cellColor.push("#bb7733");  // "Kupferfarben" für Draht-Zellen.
Settings.cellColor.push("#77bbff");  // Hellblau für Elektronenenden.
Settings.cellColor.push("#0066ff");  // Dunkelblau für Elektronenköpfe.

// changeCellState() erweitert den Cell-Datentypen um eine Methode zum Ändern
// des Zellstatus. Die Funktion gibt in 'success' einen booleschen Wahrheitswert
// zurück: 'true', wenn der Status tatsächlich geändert wurde, 'false', wenn die
// Zelle sich bereits im gewünschten Zustand befindet. Außerdem wird in
// 'oldState' der alte Status der Zelle zurückgegeben, damit im aufrufenden Code
// ggf. die Populationsanzeige aktualisiert werden kann.
Cell.prototype.changeCellState = function (cellState)
{
    var oldState = this.state;
    var success  = false;

    if (oldState != cellState) {
        this.state = cellState;
        success = true;
    }

    return [success, oldState];
}


// WireField ist ein um Wireworld-spezifische Funktionen ergänzter Field-
// Datentyp, er "erbt" also von Field (in OOP-Terminologie):
var WireField = function (rows, columns)
{
    // Damit ein neues WireField-Objekt korrekt initialisiert wird, d.h. nach
    // dem Erzeugen wie ein Field-Objekt aussieht, muss im WireField-Konstruktor
    // (d.h. in dieser Funktion) die Konstruktor-Funktion des Field-Typen
    // aufgerufen werden. Damit dies mit der Referenz zum neuen WireField-Objekt
    // geschieht, muss dies über die Field.call()-Methode geschehen (die alle
    // neuen Objekte ihrerseits vom JavaScript-Standard-Objekttypen erben und
    // die genau für diesen Zweck vorhanden ist). Dort kann als erster Parameter
    // von Hand die neue Objektreferenz übergeben werden ('this' verweist dann
    // auf das hier erzeugte WireField-Objekt):
    Field.call(this, rows, columns);
}
// Mit der folgenden Zeile "erbt"/übernimmt der WireField-Prototyp den Inhalt
// des Field-Prototypen, d.h. alle WireField-Objekte basieren auf dem Prototypen
// des Field-Objekts (genau dies besagt "Vererbung" in der OOP-Terminologie):
WireField.prototype = Object.create(Field.prototype);
// Durch die vorige Zeile würde auch der oben definierte WireField-Konstruktor
// überschrieben (mit dem Konstruktor der Basis-Klasse). Mit der folgenden Zeile
// setzen wir ihn wieder auf den korrekten, oben definierten
// WireField-Konstruktor zurück.
WireField.prototype.constructor = WireField;

// WireField.createCell() verändert den Status einer Zelle im internen Cell-
```

```
// Array und ruft zu diesem Zweck ihrerseits die oben definierte
// Cell.changeCellState()-Funktion auf. Anhand des von dort erhaltenen
// Rückgabewerts wird ggf. die Populationsanzeige aktualisiert. Wenn der
// Funktion in 'update' ein Wert ungleich 0 übergeben wird, wird dieser (bei
// Erfolg der Zellstatusänderung) zur Aktualisierung der umgebenden Nachbarwerte
// verwendet. Durch diesen zusätzlichen Parameter ist die Funktion allgemein
// gehalten, dass sie sowohl während des interaktiven Zeichnens auf dem Canvas
// verwendet werden kann (hier wird der Funktion in 'update' ein entsprechender
// Wert übergeben), als auch während der Regelanwendung zur Berechnung der
// nächsten Generation. Hier wird 'update' bei der Statusänderung weggelassen
// und erst nach Änderung aller Zellzustände über countAllNeighbours() global
// neu gezählt. Auf diese Art wird nur ein einzelnes Spielfeld gebraucht und die
// Unterscheidung zwischen Spielfeld und Arbeitsfeld, zwischen denen hin- und
// herkopiert wird, entfällt.
WireField.prototype.createCell = function (row, column, cellState, update)
{
    // Rufe die zelleigene .changeCellState()-Methode auf.
    var retval   = this.data[row][column].changeCellState(cellState);
    // Teile den erhaltenen Rückgabewert-Array auf (zur besseren
    // Verständlichkeit des folgenden Codes):
    var success  = retval[0];
    var oldState = retval[1];

    // Wenn die Statusänderung erfolgreich war, d.h. die Zelle sich noch nicht
    // im gewünschten Status befand:
    if (success) {
        if (cellState == 3)
            // Wenn cellState == 3, dann haben wir einen neuen Elektronenkopf im
            // Universum und müssen die Population erhöhen:
            this.population++;
        else if (oldState == 3)
            // Wenn hingegen oldState == 3, dann hatte die Statusänderung zur
            // Folge, dass nun ein Elektronenkopf weniger existiert:
            this.population--;

        // Wir aktualisieren die umgebenden Nachbarn nur, wenn uns der
        // aufrufende Code darum bittet (dann haben wir im Parameter 'update'
        // einen Wert ungleich 0 erhalten).
        if (update) {
            if (cellState == 3) {
                this.updateNeighbours(row, column, update);
                this.updatePopulationHTML();
            }
            else if (oldState == 3) {
                this.updateNeighbours(row, column, -1);
                this.updatePopulationHTML();
            }
        }

        // Die Statusänderung war erfolgreich, deswegen geben wir 'true' zurück:
        return true;
    } else
```

```
        // Die Statusänderung schlug fehl, deswegen geben wir 'false' zurück:
        return false;
}

// WireField.countAllNeighbours() wird von .nextGeneration() benötigt, um nach
// der Regelanwendung die Nachbarwerte der neu berechneten Generation zu
// aktualisieren:
WireField.prototype.countAllNeighbours = function ()
{
    // Damit wir korrekt zählen, müssen wir die Nachbarwerte erst zurücksetzen:
    this.resetAllNeighbours();

    // Wir müssen den äußeren, unsichtbaren Zellrand weglassen, deswegen
    // beginnen wir bei 1:
    for (var row = 1; row <= this.rows; row++) {
        for (var column = 1; column <= this.columns; column++) {
            // Nur Cell.state == 3 hat Auswirkungen auf die Nachbarwerte:
            if (this.data[row][column].state == 3)
                this.updateNeighbours(row, column, 1);
        }
    }
}

// WireField.nextGeneration() ist die wohl wichtigste Funktion des ganzen
// Programms, denn hier findet die Wireworld-Regelanwendung statt.
WireField.prototype.nextGeneration = function ()
{
    // Wir müssen den äußeren, unsichtbaren Zellrand weglassen, deswegen
    // beginnen wir bei 1:
    for (var row = 1; row <= this.rows; row++) {
        for (var column = 1; column <= this.columns; column++) {
            // Zur besseren Lesbarkeit:
            var currentState = this.data[row][column].state;

            switch (currentState) {
            case 1:  // '1' ist der Drahtzelltyp
                // Wenn genau 1 oder 2 Nachbarzellen Elektronenköpfe sind, dann
                // wird diese Drahtzelle ebenfalls zu einem Elektronenkopf:
                if (this.data[row][column].neighbours == 1 ||
                    this.data[row][column].neighbours == 2) {

                    // this.createCell sorgt für die interne Zustandsverwaltung.
                    // Wir übergeben keinen Wert für den Parameter 'update', da
                    // sonst die Nachbarwerte für die nächsten Zellen nicht mehr
                    // stimmen würden:
                    this.createCell(row, column, 3);

                    // drawCell besorgt die visuelle Darstellung auf dem Canvas:
                    drawCell(row, column, 3);
                } // Andernfalls passiert nichts.
                break;
            case 2:  // '2' ist ein Elektronenende,
```

```
            case 3:  // '3' ist ein Elektronenkopf.
                // Der Regelcode für diese beiden Zelltypen ist identisch, da
                // sie jeweils nur für eine Generation in ihrem Status
                // bleiben. Durch die Wahl der Zustandswerte 2 und 3 für
                // Elektronenzellen (und 1 für die Drahtzellen) wird mit einem
                // einfachen 'currentState - 1' der korrekte Zustandswert für
                // die kommende Genreation ermittelt:
                this.createCell(row, column, currentState - 1);
                drawCell(row, column, currentState - 1);
            case 0:  // 'leere' Zellen bleiben in Wireworld leer.
            default:
                break;
            }
        }
    }

    // Wir haben alle neuen Zustände berechnet, jetzt müssen die Nachbarwerte
    // aktualisiert werden...
    this.countAllNeighbours();

    // ...ebenso die Generations sowie auch die Populationsanzeige:
    this.increaseGeneration();
    this.updatePopulationHTML();
}


// Das eigentliche Spielfeld wird hier als globale Variable erzeugt, damit es
// von allen Stellen im Code erreicht werden kann:
var field = new WireField(Settings.cellsPerLine, Settings.cellsPerLine);


/*
 * Ende des Wireworld-spezifischen Zellularautomatencodes.
 *
 * Alle folgenden Funktionen dienen der interaktiven Benutzeroberfläche.
 */

// changeDrawingColor() wird aufgerufen, wenn der Nutzer im HTML-Dokument einen
// anderen Zelltypen zum Zeichnen auswählt.
function changeDrawingColor(cellType)
{
    // Zur besseren Verständlichkeit:
    var oldCellType = Settings.cellToDraw;

    // Speichere die Änderung nur, wenn es wirklich eine Änderung war:
    if (oldCellType != cellType)
        Settings.cellToDraw = cellType;
}

// WireField.parseChar() und .generateChar() werden von den loadFile()- und
// saveFile()-Routinen in der common.js aufgerufen und verarbeiten beide je ein
// Zeichen der Eingabedatei, bzw. eine Zelle der aktuellen Zellkonfiguration:
```

```
WireField.prototype.parseChar = function (row, column, char)
{
    switch (char) {
    case '#':  // '#' sind Elektronenköpfe
        // Wir brauchen die Nachbarwerte nicht zu aktualisieren, dies tut
        // loadFile() global über countAllNeighbours(), deswegen übergeben wir
        // keinen Wert für den Parameter 'update'.
        this.createCell(row, column, 3);
        drawCell(row, column, 3);
        break;
    case '+':  // '+' sind Elektronenenden
        this.createCell(row, column, 2);
        drawCell(row, column, 2);
        break;
    case '*':  // '*' sind Drahtzellen
        this.createCell(row, column, 1);
        drawCell(row, column, 1);
        break;
    case '␣':  // Leerzeichen und alle übrigen Zeichen werden als leere Zellen
    default:   // interpretiert.
        break;
    }
}

WireField.prototype.generateChar = function (row, column)
{
    switch (this.data[row][column].state) {
    case 3:
        return '#';  // '#' sind Elektronenköpfe
        break;
    case 2:
        return '+';  // '+' sind Elektronenenden
        break;
    case 1:
        return '*';  // '*' sind Drahtzellen
        break;
    case 0:
    default:
        return '␣';  // Gib für leere Zellen ein Leerzeichen zurück, dann kann
        break;       // die erzeugte Datei in einem Texteditor betrachtet und
                     // bearbeitet werden.
    }
}

// Die WireField.print()-Methode wurde lediglich während der Entwicklung des
// Programms verwendet, um die interne Zellularautomatensimulation unabhängig
// von der grafischen Ausgabe zu testen. Dazu werden die aktuellen Zellzustände
// als ASCII-Grafik (ähnlich der verwendeten Zeichen beim Speichern in
// Textdateien), sowie die aktuellen Nachbarwerte in zwei Rastern nebeneinander
// ausgegeben. WireField.print() wird im fertigen Programm nicht mehr verwendet
// und ist deswegen auch nicht detailliert kommentiert. Ihre Funktion sollte
// nach dem bisherigen Code dennoch verständlich sein.
```

```
WireField.prototype.print = function()
{
    var printstring = '';

    for (var row in this.data) {
        var fieldstring = '';
        var neighbourstring = '';

        for (var column in this.data[row]) {
            switch (this.data[row][column].state) {
            case 3:  // 'head'
                fieldstring += '#␣';
                break;
            case 2:  // 'tail'
                fieldstring += '+␣';
                break;
            case 1:  // 'wire'
                fieldstring += '*␣';
                break;
            case 0:
            default:
                fieldstring += '␣␣';
                break;
            }

            neighbourstring += this.data[row][column].neighbours + '␣';
        }
        printstring += fieldstring + '|␣' + neighbourstring + '\n␣';
    }

    console.log(printstring);
    console.log(this.population + '␣Electron␣heads.')
}
```

4.3 Kommentierter Quellcode der Datei wireworld.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Wireworld in JavaScript</title>

    <!-- Zunächst wird das Stylesheet für das HTML-Dokument geladen. Die Datei
    enthält lediglich eine Änderung der Standard-Schriftart, die Farbdefinition
    des Randes um den Canvas und ein paar Größen- und Abstandsänderungen,
    deswegen wird sie hier nicht separat abgedruckt und kommentiert. -->
    <link rel="stylesheet" type="text/css" href="style.css">

    <!-- Nun wird der JavaScript-Programmcode geladen. Zunächst die allgemeinen
    Funktionen für zelluläre Automaten: -->
    <script src="common.js"></script>
    <!-- Anschließend die Wireworld-spezifischen Ergänzungen: -->
```

```
    <script src="wireworld.js"></script>
  </head>

  <body>
    <!-- Hier folgt die eigentliche "Benutzeroberfläche". Grundsätzlich gilt:
         alle Buttons rufen über ihr 'onclick'-Attribut eine JavaScript-Funktion
         auf. Wenn nicht anders angegeben, befinden sich alle diese Funktionen
         in der common.js. -->
    <h1>Wireworld</h1>
    <p><em>Choose</em>
      <!-- Zur Auswahl der "Zeichenfarbe" werden drei Radioboxen verwendet, die
           beim Anklicken (onclick) die JavaScript-Funktion changeDrawingColor()
           (in wireworld.js) mit dem entsprechenden Zellstatuswert aufrufen: -->
      <input type="radio" id="Wire" name="CellType"
             onclick="changeDrawingColor(1);"
             checked><label for="Wire">Wire,</label>
      <input type="radio" id="Tail" name="CellType"
             onclick="changeDrawingColor(2);"><label for="Tail">Tail,</label>
      or
      <input type="radio" id="Head" name="CellType"
             onclick="changeDrawingColor(3);"><label for="Head">Head,</label>
      <em>then draw on the field.</em>
      <button id="ClearButton" type="button" onclick="clearField()">Clear
        Field</button>
    </p>
    <div>
      <!-- Das Canvas-Objekt ist die eigentliche Zeichenfläche für den
           Zellularautomaten. Er muss quadratische Maße haben! -->
      <canvas id="board" width="500" height="500">
        <!-- Warnhinweis für ältere Browser. -->
        This Wireworld implementation does only run in HTML5 capable browsers.
      </canvas>
    </div>
    <p class="statusBar">
      <button id="RunButton" type="button" onclick="runAutomata()">Run</button>
      <button id="StopButton" type="button" onclick="stopAutomata()">
        Stop</button>
      <!-- Die Geschwindigkeitsauswahl funktioniert über ein <select>-Element,
           das, wenn der Nutzer hier einen neuen Wert auswählt, einen 'onchange'
           Event erzeugt. Diesen nutzen wir, um der JavaScript-Funktion
           changeSpeed() den aktuellen Zustand des <select>-Elements zu
           übermitteln (in 'this' enthalten): -->
      <select id="SpeedSelect" onchange="changeSpeed(this);">
        <option value="0">As fast as possible</option>
        <option value="10">10ms / Generation</option>
        <option value="100" selected>100ms / Generation</option>
        <option value="500">0.5s / Generation</option>
        <option value="1000">1s / Generation</option>
      </select>
      <button id="NextGenerationButton" type="button"
              onclick="nextGeneration();">Next Generation</button>
      <!-- Die folgenden beiden <strong>-Elemente bekommen je ein eindeutiges
```

```
           'id'-Attribut, damit ihr Inhalt von JavaScript aus aktualisiert
           werden kann (siehe updatePopulationHTML() und increaseGeneration(),
           beide in common.js). -->
           Generation: <strong id="Generation">0</strong>
           Population: <strong id="Population">0</strong>
    </p>
    <p class="statusBar">
      <!-- Das "FileName"-Texteingabefeld ruft, wenn es den Fokus hat, bei jedem
           Tastendruck (onkeypress) die JavaScript-Funktion handleKeyPress()
           auf. Dort wird überprüft, ob die Return-Taste gedrückt wurde. Wenn
           ja, wird von dort die saveFile()-Funktion aufgerufen. -->
      <button id="SaveButton" type="button" onclick="saveFile()">Save</button>
      the current field as <input id="FileName" type="text"
                                  value="wireworld.txt" size="12"
                                  onkeypress="handleKeyPress(event,␣this.id)">.
      Load a file via drag and drop onto the field.
    </p>
  </body>
</html>
```

5 Fazit

Zum Abschluss lässt sich, medientheoretisch präzise, festhalten: *Der zelluläre Automat Wireworld visualisiert zunächst einzig Signale.* Auf der Ebene einer einzelnen Wireworld-Zelle und ihres Regelsatzes existiert weder eine semantische, noch eine operationale Bedeutung, und höchstens letztere ergibt sich erst aus dem emergenten Zusammenspiel größerer Zellstrukturen, z. B. dem beschriebenen Addierer. Zum informationstragenden Symbol kann ein Wireworld-Elektron als pures Signal demnach nur durch seine Position auf einer aus Drahtzellen bestehenden Leiterbahn relativ zur Position anderer Elektronen werden.[113] Diese Position ist im visuell dargestellten zweidimensionalen Gitter des Zellularuniversums zwar eine räumliche, durch den Generationenzähler bekommt jede Zellkonfiguration aber auch einen diskreten Zeitindex, der die ganze Information eines einzelnen Wireworld-Elektrons ausmacht, es vom reinen Signal zum binären Symbol der 1 machen kann: „Null und Eins, *low* und *high* sind im Computer Zeit-Zustände, Zeit-Werte".[114] Christian Kassung definiert solchermaßen aus *zeitkritischen Signalen* gebildete Symbole dann auch mit direktem Verweis auf Ernst als „Signale, deren Bedeutung sich einzig und allein aus ihrem Zeitindex heraus ergibt."[115] Der Übergang von einem einfachen physikalischen Signal zu einem – aufgrund des

---

[113] Dass die Position, bzw. „die relative Stellung eines Zeichens gegenüber anderen Zeichen" sein wesentlich „bedeutungserzeugendes Moment" ist, darauf weist Kassung bereits für die ersten Stellenwertsysteme der Babylonier vor rund dreieinhalbtausend Jahren hin. Christian Kassung. *Das Pendel. Eine Wissensgeschichte.* München: Fink, 2007, S. 49.

[114] Ernst, *Chronopoetik*, S. 340.

[115] Kassung, *Das Pendel*, S. 323.

dualen Stellenwertsystems – operational bedeutungstragenden Symbol ist also ein *zeitkritischer.* Zeitkritisch heißt in diesem Fall, dass bedeutungslose Signale durch eine syntaktische Codierung strukturiert werden können, die nur funktioniert, also operational wirksam werden kann, wenn sie zeitlich synchronisiert ist. Zur Verdeutlichung eignet sich wiederum der zelluläre Automat Wireworld, das Messgerät (Ernst) für Signale in der vorliegenden Arbeit:

Zum infomationstragenden Symbol, das eine Zahl im dualen Stellenwertsystem codiert, emergieren im Wireworld-Addierer Gruppen von je neun Elektronen – acht Binärziffern plus ein eventuelles Übertragsbit – auf dem Ausgangsdraht; nach genau 157 Generationen. Tatsächlich hat das letzte Elektronenbit den Ausgang des finalen EOR-Gatters bereits nach 93 Generationen verlassen, die verbleibenden 64 Generationen dienen lediglich dazu, den Elektronenstrom der Ergebnisbits an den oberhalb der Ausgabeleitung als Lesehilfe angebrachten einzelnen Drahtzellen auszurichten. „Signale als Informationsträger sind physikalische Ereignisse, also immer schon in der Zeit“.[116] Bloß bleibt der menschlichen Wahrnehmung die Information der codierten Dualzahl verschlossen, solange sich das Signal im Fluss befindet und seine Bewegung die menschliche Wahrnehmung unterläuft. Um das Ergebnis der Zellularautomatenberechnung mit dem visuellen Wahrnehmungsapparat zu synchroniseren, muss folglich der Generationszähler im *user interface* im Blick bleiben. Nur wenn der Automat genau nach 157 Generationen gestoppt wird, kann der Signalstrom als Zahl gelesen werden.[117] Ernst hält für analoge Fernsehbilder fest: „Elektronische Bilder sind zeiträumliche Segmente eines kontinuierlichen, aber im Zeilenumbruch diskretisierten Signalstroms.“[118] Zeitkritisch serielle Signale, damit sie zu operativen Symbolen werden können – im Analogfernsehen das vollständige Einzelbild, im Wireworld-Addierer die resultierende Dualzahl –, bedürfen also grundsätzlich einer diskreten Taktung: Das Fernsehsignal weist diskrete Zeilensprünge sowie den finalen Kathodenstrahlrücklauf auf; im Wireworld-Addierer müssen die einzelnen Bits ebenso durch regelmäßige Lücken getrennt werden. Und es muss bekannt sein, nach wievielen Generationen der Ausgangssignalstrom vollständig ist.[119]

Außerdem zeigte sich in der Diskussion des Quellcodes, dass die Automatentheorie und die objektorientierte Programmierung viele Übereinstimmungen haben. Wie schon einleitend erwähnt, liegt dies vor allem an der Entwicklung und Ausarbeitung der Automatentheorie durch Forscher wie John von Neumann, Arthur Burks und Edward Moore, die die frühen Computer mitentwickelt hatten und mit ihnen arbeiteten. Was jedoch auch klar werden soll-

---

[116] Wolfgang Ernst. *Gleichursprünglichkeit. Zeitwesen und Zeitgegebenheit technischer Medien.* Berlin: Kadmos, 2012, S. 215.

[117] Geübte Nutzer können die duale Zahl womöglich auch ‚in Bewegung‘ entziffern, wenn die Simulationsgeschwindigkeit einige Generationen vor der 157. auf ein bis zwei Generationen pro Sekunde reduziert wird.

[118] Ernst, *Gleichursprünglichkeit*, S. 215.

[119] Karl Scherer bezeichnet dies als die „transit time“ seiner Wireworld-Schaltungen. Scherer, *Wireworld*.

te, ist, dass die Begriffe der Objektorientierung ebenfalls Gefahr laufen, hinter semantischen Bedeutungen wie der ‚Vererbung' von ‚Verhalten' die eigentlichen operativen Instruktionen, die der Computer ausführt, zum Verschwinden zu bringen:

> Dies geschieht nicht einfach nur dadurch, dass Datenkapselung zu Blackboxen führt, [...] sondern auch weil die epistemischen Valenzen der Objektorientierung dazu neigen, einen Aspekt der EDV zu verschleiern, der sehr oft mit dem Bedeutungsumlauf *(circulation of meaning)* und der Bedeutungsverfeinerung assoziiert wird, – der Manipulation und Auswertung von Symbolen.[120]

Das grundlegende und von Ernst angemahnte Problem bleibt demnach bestehen. Die Medientheorie darf nicht müde werden, die auf der Hardwareebene zirkulierenden Signale gegenüber den daraus auf welcher Ebene auch immer emergierenden Zeichen zu betonen. Menschen – und dies schließt auch den Autor dieser Arbeit ein, der zum Zwecke der Verdeutlichung des medientheoretischen Signalbegriffs einen zellulären Automaten programmiert hat – können „bei der Ausgabe des Signals nicht umhin, es zeichenhaft zu interpretieren", bzw. es visuell in abzubilden.[121] Das heißt, dass die Visualisierung eines 50 mal 50 Felder großen Wireworld-Universums die Medientheorie zwar verständlicher machen, aber nicht vollständig ersetzen kann. Es braucht stets begleitend in natürlicher Sprache ausformulierte Sätze, um auf die Signale hinter den Zeichen hinzuweisen. Der ausformulierte Programmcode kann helfen, die eigentlichen vom Computer bei der Simulation eines zellulären Automaten operativ ausgeführten Befehle nachzuvollziehen. Die visuelle Fassade der Automatensimulation im Webbrowser birgt jedoch immer die Gefahr, die eigentlichen Signale, hier die zirkulierenden Wireworld-Elektronen, in letzter Konsequenz zu übersehen und stattdessen ausschließlich die dual codierten Ergebniszahlen des Wireworld-Addierers zu betrachten.

Medientheorie sollte darin verbauten Logikgatter nicht in Analogien zu ihren elektrophysikalischen Pendants beschreiben. Denn die Analyse der – dank des ausführbaren Wireworld-Programms – auch praktisch nachvollziehbaren Beispielschaltungen, oder präziser: Zellkonfigurationen, hat gezeigt, dass unterhalb der semantisch aufgeladenen Einheiten, die als „Dioden" oder „Flipflops" bezeichnet werden, basalere Strukturen *operativ* am Werk sind: Abzweigungen, Sperren und Schleifen aus Drahtzellen, die nach den Regeln des Automatenuniversums namens Wireworld eine Welt voller Signale steuern.

---

[120] Matthew Fuller und Andrew Goffey. „Die obskuren Objekte der Objektorientierung". Aus dem Englischen übers. von Heinz-Günter Kuper, Agata Królikowska und Jens-Martin Loebel. In: *zfm. Zeitschrift für Medienwissenschaft* 6 (Apr. 2012), S. 206–221, hier S. 221.

[121] Ernst, „Signal *versus* Zeichen?", S. 329.

Literatur

Burks, Arthur W. „Editor's Introduction". In: Von Neumann, John. *Theory of Self-Reproducing Automata.* Hrsg. von Arthur W. Burks. Urbana/IL und London: University of Illinois Press, 1966, S. 1–28.

– „Von Neumann's Self-Reproducing Automata". In: *Essays on Cellular Automata.* Hrsg. von dems. Urbana/IL und London: University of Illinois Press, 1970, S. 3–64.

Dewdney, Alexander K. „Computer Recreations. The cellular automata programs that create wireworld, rugworld and other diversions". In: *Scientific American* 262.1 (Jan. 1990), S. 146–149.

– „Mathematical Recreations. An odd journey along even roads leads to home in Golygon City". In: *Scientific American* 263.1 (Juli 1990), S. 118–121.

Eckardt, Michael. *Medientheorie vor der Medientheorie. Überlegungen im Anschluß an Georg Klaus.* Berlin: Trafo, 2005.

Eco, Umberto. *Einführung in die Semiotik.* Aus dem Italienischen übers. von Jürgen Trabant. 5. Aufl. [Deutsche Erstausgabe: 1972; Originalausgabe: *La struttura Assente,* Mailand: Bompiani 1968]. München: Fink, 1985.

Erickson, Tim. „Game Review. The Phantom Fish Tank by Brian Silverman." In: *Simulation & Games* 19.2 (Juni 1988), S. 225–228.

Ernst, Wolfgang. „Signal *versus* Zeichen? Zeit, Medium, Maschine". In: *Kybernetik und Interdisziplinarität in den Wissenschaften. Georg Klaus zum 90. Geburtstag.* Hrsg. von Klaus Fuchs-Kittowski und Siegfried Piotrowski. Berlin: Trafo, 2002, S. 323–332.

– *Chronopoetik. Zeitweisen und Zeitgaben technischer Medien.* Berin: Kadmos, 2012.

– *Gleichursprünglichkeit. Zeitwesen und Zeitgegebenheit technischer Medien.* Berlin: Kadmos, 2012.

Fuller, Matthew und Andrew Goffey. „Die obskuren Objekte der Objektorientierung". Aus dem Englischen übers. von Heinz-Günter Kuper, Agata Królikowska und Jens-Martin Loebel. In: *zfm. Zeitschrift für Medienwissenschaft* 6 (Apr. 2012), S. 206–221.

Gardner, Martin. „The fantastic combinations of John Conway's new solitaire game 'life'". In: *Scientific American* 223.4 (1970), S. 120–123.

Heise, Nyles. *World Wide WireWorld.* URL: `http://www.heise.ws/wireworld.html` (besucht am 06. 04. 2015).

– *Letter to A. K. Dewdney.* 4. Feb. 1990. URL: `http://www.heise.ws/lettertodewdney.html` (besucht am 06. 04. 2015).

Hickson, Ian u. a. *HTML5. A vocabulary and associated APIs for HTML and XHTML. W3C Recommendation.* 28. Okt. 2014. URL: `http://www.w3.org/TR/html5/` (besucht am 06. 04. 2015).

Kassung, Christian. *Das Pendel. Eine Wissensgeschichte.* München: Fink, 2007.

Klaus, Georg. *Kybernetik in philosophischer Sicht.* Berlin: Dietz, 1961.

Logo Foundation. *What is Logo?* 2011. URL: `http://el.media.mit.edu/logo-foundation/logo/index.html` (besucht am 06. 04. 2015).

López Quintero, Manuel Ignacio. *Get the coordinates of a mouse click.* 8. Mai 2011. URL: `http://miloq.blogspot.de/2011/05/coordinates-mouse-click-canvas.html` (besucht am 06. 04. 2015).

Maibaum, Johannes. *basic-life*. Github-Repository. URL: `https://github.com/jmaibaum/basic-life` (besucht am 06. 04. 2015). *Das Repository enthält die Entwicklungsschritte der vorliegenden Wireworld-Implementierung im Unterordner* `JavaScript`*.*

Mainzer, Klaus. *Die Berechnung der Welt. Von der Weltformel zu Big Data.* München: C. H. Beck, 2014.

Moore, David und Mark Owen. *The Wireworld Computer.* 7. Nov. 2014. URL: `http://www.quinapalus.com/wi-index.html` (besucht am 06. 04. 2015).

Moore, Edward F. „Machine Models of Self-Reproduction“. In: *Mathematical Problems in the Biological Sciences.* Hrsg. von Richard Bellman. Proceedings of Symposia in Applied Mathematics 14. Providence/RI: American Mathematical Society, 1962, S. 17–33.

Mozilla Developer Network, Hrsg. *Introduction to Object-Oriented JavaScript.* 4. Apr. 2015. URL: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Introduction_to_Object-Oriented_JavaScript` (besucht am 06. 04. 2015).

Pirsch, Peter. *Architekturen der digitalen Signalverarbeitung.* Stuttgart: Teubner, 1996.

Ranganathan, Arun und Jonas Sicking. *File API. W3C Last Call Working Draft.* 12. Sep. 2013. URL: `http://www.w3.org/TR/FileAPI/` (besucht am 06. 04. 2015).

Scherer, Karl. *Wireworld. The biggest and best collection of examples of logical elements built with this amazing cellular automaton.* Sep. 2003. URL: `http://karlscherer.com/Wireworld.html` (besucht am 06. 04. 2015).

Segal, Jérôme. „Kybernetik in der DDR: Dialektische Beziehungen“. In: *Cybernetics – Kybernetik. The Macy Conferences 1946–1953.* Bd. 2: *Essays & Documents / Essays & Dokumente.* Hrsg. von Claus Pias. Zürich und Berlin: Diaphanes, 2004, S. 227–251.

Silverman, Brian. *The Phantom Fish Tank. An Ecology of Mind.* 2. Aufl. [Erstausgabe 1987]. Montreal: Logo Computer Systems Inc., 1988.

Swaminathan, Nikhil. „Digging into Technology’s Past. ‘Digital Archaeologists’ excavate the microprocessor that ushered in the home computing revolution“. In: *Archaeology* 64.4 (Juli–Aug. 2011). URL: `http://archive.archaeology.org/1107/features/mos_technology_6502_computer_chip_cpu.html` (besucht am 06. 04. 2015).

This Could Be Better (Wordpress.com-Pseudonym). *Loading, Editing, and Saving a Text File in HTML5 Using Javascript.* 18. Dez. 2012. URL: `https://thiscouldbebetter.wordpress.com/2012/12/18/loading-editing-and-saving-a-text-file-in-html5-using-javascrip/` (besucht am 06. 04. 2015).

Toffoli, Tommaso und Norman Margolus. *Cellular Automata Machines. A New Environment for Modeling.* Cambridge/MA und London: MIT Press, 1987.

Trevorrow, Andrew u. a. *Golly. An open source, cross-platform application for exploring the Game of Life and other cellular automata.* Dez. 2013. URL: `http://golly.sourceforge.net` (besucht am 06. 04. 2015).

Von Neumann, John. *Theory of Self-Reproducing Automata.* Hrsg. von Arthur W. Burks. Urbana/IL und London: University of Illinois Press, 1966.

Walraet, Matthieu. *Wireworld – Un monde cablé.* URL: `http://matthieu.walraet.net/automate/automate.html` (besucht am 21. 03. 2015).