OPEN HISTORY

Archäologie der frühen Mikrocomputer und ihrer Programmierung

Dissertation

zur Erlangung des akademischen Grades

doctorum rerum naturalium (Dr. rer. nat.)

im Fach Informatik

eingereicht an der

Mathematisch-Naturwissenschaftlichen Fakultät der Humboldt-Universität zu Berlin

von

Dr. phil. Stefan Höltgen

Präsidentin der Humboldt-Universität zu Berlin

Prof. Dr.-Ing. Dr. Sabine Kunst

Dekan der Mathematisch-Naturwissenschaftlichen Fakultät

Prof. Dr. Elmar Kulke

Gutachter:

Prof. Dr. Niels Pinkwart

Prof. Dr. Wolfgang Ernst

Prof. Dr. Dr. h.c. Raúl Rojas

Datum der Verteidigung:

11.02.2020

Inhaltsverzeichnis

https://doi.org/10.18452/21165

1. Einleitung

1.1 Fragestellung

Die Geschichte der Computer besitzt eine Eigentümlichkeit, die ihren Gegenstand von dem anderer historischer Gegenstände deutlich abhebt. Computer ist sind Medien, die aufgrund ihrer Operationen *speichern, übertragen und prozessieren* in der Lage sind alle anderen Medien zu simulieren. Um diese Eigenschaft zu erlangen, müssen Computer sich aber *im Vollzug* befinden, das heißt *operativ sein*. Erst im Verbund von prozessierender Hardware, prozessbeschreibenden Algorithmen und prozessierten Daten gelangen sie in den Medienstatus. Das bedeutet im Umkehrschluss, dass ein nicht-operativer Computer, sei er nur ausgeschaltet oder defekt, lediglich den Status von Hardware besitzt. Die Geschichte der Computer ist bislang notwendigerweise entweder als Hardware-Geschichte geschrieben worden (und zeigt sich in Technik- und Computermuseen als Ausstellung nicht-operativer Hardware) oder als Geschichte der um die Hardware, die Software und deren Nutzung etablierten Diskurse, Ingenieure, Ökonomien und Nutzer. Eine Geschichte der Computer (als operative Medien) steht bislang aus. Wie kann aber etwas, das sich im Prozess befinden muss, historisiert werden?

Die vorliegende Studie versucht diese Frage zu beantworten und das *Archiv der Computer* in Vollzug zu versetzen, denn es scheint, als seien die Methoden akademischer Geschichtsschreibung kaum anwendbar auf einen Gegenstand, der doch lediglich *in der Gegenwart* existieren kann. Dies zeichnet Computer nämlich wie alle anderen technischen Medien aus: Dadurch, dass sie erst im Vollzug zu dem werden, was sie sind (Medien), befinden sie sich als solche stets in der Gegenwart. Das hat auch Konsequenzen für ihren Status als historisches Artefakte: Es ist beispielsweise unerheblich für das Medium Radio, wann es gebaut wurde; schaltet man es heute ein, dann hört man heutige Radioprogramme (nicht nicht etwa die, die zurzeit der Entstehung des Apparates ausgestrahlt wurden). Ein operatives technisches Medium ist immer im Hier und Jetzt. Es scheint eng an das Reale gekoppelt zu sein: Es basiert auf Hardware, die Verschaltungen für Signalwege darstellt, um aus latenten Eingangs- evidente (sinnfällige) Ausgangssignale zu generieren. Die dabei ablaufenden Prozesse finden jenseits aller Symbole *im Realen* statt – das ist der Hauptgrund dafür, dass so verstandene Medien nicht (durch Symbole) historisierbar und nicht allein aus ihrer „diachronen Perspektive“ [Heilige 2004:4] verstehbar sind. Begriffe wie ‚Geschichte‘, ‚Historie‘, ‚historisch‘, ‚früh‘ und ähnliche sollen damit in der vorliegenden Arbeit kritisch reflektiert werden.[1]

1 Dort, wo solche Begriffe zur Zuschreibung einer ‚zeitlichen Situierung‘ verwendet werden, sollte dies in einem unemphatischen Sinne – etwa als *discontinued/nicht mehr lieferbar* – verstanden werden. In Retrocomputing-Szenen wird hierfür zuweilen das ™-Symbol zur ironischen Markierung des Vergangenheitsbezugs verwendet („damals™“, vgl. https://damals-tm-podcast.de/ [letzter Abruf: 15.07.2019].)

Computer unterscheiden sich dadurch von anderen technischen Medien, dass sie das Symbolische der Schrift wieder integrieren – durch die Symbole ihrer Programmierung von der Ebene des Microcodes bis zu den höheren Programmiersprachen. Damit rufen sie neben den Hardware-Geschichten auch Schrift-Geschichten und -Theorien auf, die Diskursfelder von der diachronen und synchronen Sprachwissenschaft bis hin zur Theorie der formalen Sprachen und ihren Automaten öffnen. Diese Hybridität zwischen ‚reiner' Hardware, formalisierter Logik, algorithmisch beschreibbarer Software und Daten erfordert besondere Methoden, die über den Methoden-Kanon von Elektrotechnik, Linguistik, formaler Philosophie, Informatik und Medienwissenschaft hinausgehen. Eine Geschichte des operativen Computers müsste sich daher *zwischen diesen Disziplinen* entwickeln. Damit würde sich ein solches Projekt erstmals auch die Möglichkeit einer *informatischen Computergeschichte* ermöglichen, die etwas anderes als die *Geschichte der Informatik* (als Disziplin) oder die *Geschichte des Computers* (als Technologie) sein könnte. Der methodische Ansatz, der dafür vorgestellt wird, erfordert jedoch eine scharfe Eingrenzung des untersuchten Gegenstands in Hinblick auf deren Entstehungszeitraum als auch eine beispielhafte Auswahl der Einzelanalysen.

1.2 Gegenstand

Unter der Zuschreibung „früher Mikrocomputer" werden Mikrocomputer (also Digitalcomputer auf Basis einer Ein-Chip-CPU) mit Datenbus-Breiten von 4, 8 und 16 Bit gefasst, die frei verkäuflich für Amateur-Anwender ab Mitte der 1970er-Jahre publiziert wurden. Dabei handelt es sich um Plattformen (zum Begriff der *Plattform* vgl. [Bogost/Montfort 2009a; Höltgen 2013a] sowie Kapitel 2.1 und 3.3.3), die notwendige Peripheriebestandteile (Eingabetechnologien, den Mikrocomputer selbst sowie Schnittstellen für Ausgabeperipherie) in einem Gehäuse vereinen. Ab 1975 verfügen diese Computer über Schnittstellen und Protokolle, die es ermöglichen, bereits in Privathaushalten vorhandene Medientechnologien wie Fernseher, Kassettenrecorder, Stereoanlagen, elektrische Schreibmaschinen und anderes direkt an sie anzuschließen. In diesem Fall soll von *Homecomputern* (oder Heimcomputern) in Abgrenzung zu *Personalcomputern* gesprochen werden, für deren Betrieb dedizierte Ein- und Ausgabe-Peripheriebausteine notwendig sind.

Ab 1977 entwickelt sich eine Industrie, die Homecomputer herstellt und sich dabei bis circa Mitte der 1980er-Jahre immer stärker diversifiziert [vgl. Ceruzzi 2003a:311ff.] Insbesondere der Wechsel von 8- auf 16-Bit-Architekturen, das Wiedererstarken der Spielkonsolen-Industrie und die stärkere Verbreitung günstiger *IBM*-kompatibler Personal Computer auch für Amateur-Anwendungen (insbesondere für Computerspiele) hat ab Mitte der 1980er-Jahre die Vielfalt an Herstellern und Gerätetypen stark dezimiert, so dass zum Ende des von mir betrachteten Zeitraums nur noch wenige Hersteller mit Homecomputer-Modellen am Markt verblieben sind: *Apple, Commodore*, *Atari* und *Acorn*. Letztere drei veröffentlichten mit den Modellen Acorn Archimedes A3010, Commodore Amiga 1200

und Atari Falcon 1992 die letzten Homecomputer, die Tastatur, Computer und Diskettenlaufwerk in einem Gehäuse vereinten und Anschlüsse für Atari-kompatible Joysticks verfügt. Zu dieser Zeit erscheinen neue Modelle der Konkurrenz bereits als Desktop-Computer mit abgesetzter Tastatur und Monitoranschluss. 1993 stellte *Atari* und 1994 *Commodore* die Herstellung dieser Rechnermodelle ein.[2] Acorn entwickelt die Prozessortechnik (ARM) sowie – nach verschiedenen Ausgründungen und Umfirmierungen [vgl. Mohr 2014] – das Betriebssystem RISC OS weiter. Schließlich konsolidiert sich der Markt zwischen *Apple* (mit ihren Macintosh-Computern) und *IBM*-kompatiblen Personalcomputern (zumeist mit MS-DOS- und Windows-Betriebssystemen). Die Homecomputer-Ära ist damit zu Ende.

Die hier vorzunehmende Eingrenzung der zu betrachtenden Plattformen auf die Jahre 1974-1984 hat noch einen zweiten Grund. Es sind insbesondere die in diesem Zeitraum erschienene Computer, mit denen sich das *Retrocomputing* heute beschäftigt. Für historische Homecomputer existiert heute ein lebendiger Markt mit gebrauchten Modellen (die teilweise hohe Sammler-Werte produzierten, vgl. [Brückner 2011]), sowie für alte und neue Hardware und Software, neue Zeitschriften und Bücher sowie zahlreiche Online-Plattformen (Blogs, Diskussionsforen, Facebook-Seiten und -Gruppen, Archive für Software und Dokumente). Seit einigen Jahren finden sich Nutzergruppen bestimmter Homecomputer-Plattformen oder zu Themen wie Computerspielen in Vereinen und locker organisierten Stammtischen zusammen, veranstalten Ausstellungen, Messen und Festivals, auf denen Privatsammlungen, neue Produkte (insbesondere neue Software und Hardware-Erweiterungen) vorgestellt werden, Workshops, Seminare und Vorträge stattfinden. Allein in Deutschland ereignen sich monatlich bis zu zehn Retrocomputing-Veranstaltungen [vgl. N.N. 2015a].

Bemerkenswert ist hierbei die starke Konzentration auf die Softwaregattung *Computerspiel*. Nicht nur, weil diese für die Retrocomputing-Gruppen und -Enthusiasten und innerhalb der Software Preservation die zentralen Gegenstände sind, nehmen Computerspiele in der vorliegenden Arbeit eine wichtige Rolle ein; an diesen Programmen lassen sich überdies besonders viele unterschiedliche technische Phänomene kondensieren. Computerspiele stellen immer schon einen ‚Motor' der Entwicklung sowohl privater Computerhard- und -software [Levy 1984:37-57,277ff.] als auch computertheoretischer Diskurse (wie sich am Schachspiel zeigt [Turing 1987a; Shannon 1950]).

Die Eingrenzung verfolgt jedoch auch forschungspragmatische Gründe: Eine Theorie mittlerer Reichweite, wie sie im Folgenden formuliert wird, verlangt einen überschaubaren Gegenstandsbereich, der in ausreichender Detailtiefe wissenschaftlich analysiert werden kann. Dieser soll weder das Bild einer historischen ‚Epoche' formen, noch sollen damit (ohnehin nicht mögliche) Eingrenzungen und Ausgrenzungen erreicht werden.

2 Die genauen historischen Entwicklungen stellen sich komplexer als hier skizziert dar [vgl. Bagnall 2010; Hertzfeld 2004].

Dasjenige, was Homecomputer als solche auszeichnet, hat es in Teilaspekten sowohl vor als auch nach dem hier eingegrenzten Zeitraum in der Computertechnik und -kultur gegeben. Die Homecomputer zeigen sich so, trotz meiner Eingrenzung auf die vergleichsweise kurze Existenzphase innerhalb der so genannten „Geschichte des Computers" als Gegenstand mit vielfältiger Ausprägung und langer Nachwirkung. Im Folgenden sollen überblicksartig die Methoden und ihre Herkünfte skizziert werden, mit denen dieser Gegenstand untersucht werden soll. (Im weiteren Verlauf der Studie wird detaillierter auf die einzelnen Methoden eingegangen.)

1.3 Methoden-Überblick

Eine Untersuchung der frühen Mikrocomputer und ihrer Programmierung unter den eingangs genannten Einschränkungen und Erweiterungen wird einen sorgsam abgestimmten Methoden-Kanon erfordern. Dieser muss sich sowohl an geschichtswissenschaftlichen, didaktischen, informatischen und medienwissenschaftlichen Methoden orientieren und auf einzelne Diskurse und Arbeitsfelder der jeweiligen Disziplinen zurückgreifen.

Zunächst wären – nicht nur in Abgrenzung zu deren tradierten und für den Gegenstand als defizitär ausgewiesenen Methoden – die *Technikgeschichtsschreibung* selbst zu fokussieren: Nach welcher Methodologie verfährt sie, insbesondere, wenn ‚der Computer' ihr Gegenstand ist? Welche *Historeme*[3] fasst sie dabei vor allem ins Auge (Technik, Ökonomie, Personen, Praktiken, ...)? Die Quellen der Computergeschichtsschreibung sind vielfältig. Neben akademischen Studien in Monografien und Zeitschriften (etwa den Annals of the History of the Computer) existieren zahlreiche populärwissenschaftliche und semi-professionelle Publikationen. Die Erlebnisberichte von Zeitzeugen und Mitwirkenden der Computerindustrie des frühen Mikrocomputers finden sich in zahlreichen Quellen, welche der *Oral History* zugerechnet werden müssen und damit eigene Interpretationen [vgl. Stöckle 1990; Vorländer 1990] verlangen. Zudem zeigen sich sowohl zeitgenössische technische Manuals und Handbücher (so genannte *Grey Literature)* als auch Lehrbücher, Zeitschriftenpublikationen, Dokumentar- und Spielfilme und andere Medien als informative Quellen für die Geschichte früher Mikrocomputer.

Markant an sowohl der zeitgenössischen Nutzung der frühen Mikrocomputer (Homecomputing) als auch an ihren heutigen Verwendungsweisen im Retrocomputing ist eine *spezifische Didaktik*, die sich als *Autodidaktik* darstellen lässt. Neben den vielfältigen und abwechslungsreichen Ansätzen zur curricularen Computerdidaktik entsteht erstmals mit der Privatisierung von Computertechnik in den 1970er-Jahren ein Zugang zu dieser Technologie, der freiwillige Lektüre (Lehrbücher, Handbücher, Zeitschriften) voraussetzt und

3 Zum Begriff des *Historems* als diskursivierter historischer Fakt vgl. Neumann 1999. Im Folgenden wird der Begriff im Sinne Whites verwendet: als ein Element einer „gegebenen Menge von zufällig überlieferten Ereignissen [, ... die von Historikern] zu einer Geschichte *gemacht*" werden. [White 1991a: 104 – Hervorhebung im Original]

allenfalls von Szene-internen Diskursen begleitet und geleitet wird. Die Emergenz der Homecomputer scheint eine Form des *Willens zum Wissen* in den – zumeist kindlichen und jugendlichen! – Nutzern provoziert zu haben, die eng mit ihren damaligen wie heutigen Nutzungsweisen verwoben ist. Neben der Tatsache, dass sich die Lehrliteratur für Homecomputernutzer eine historisch reichhaltige Quelle darstellt, lässt sich aus ihr – im Kontrast zu schul- und universitätsdidaktischen Publikationen – der modus operandi der *Hackerkultur* [vgl. Levy 1984:26-30] extrahieren. Ihr zugleich „kalter Blick" [vgl. Ernst 2005/6] und ‚respektloser' Zugriff auf Technik bildet auch die Basis für heutige Praktiken im Umgang mit ‚historischer Hardware'.

Zur adäquaten technischen Beschreibung eines Computers (seiner Operation, Programmierung und nicht zuletzt seiner Didaktik und seinen gesellschaftlichen Wechselwirkungen) stellt die *Informatik* etablierte Methoden bereit. Hiervon können für die vorliegende Untersuchung vor allem methodologische Aspekte der *Theoretischen Informatik* (etwa über den Aufbau formaler Sprachen und zur Automatentheorie), der *Technischen Informatik* (zur Beschreibung und Analyse logischer Gatter, von Mikroprozessoren, Computerarchitekturen und Protokollen), der *Praktischen Informatik* (in Hinblick auf Betriebssysteme, Compiler und Interpreter, Programmiersprachen und Algorithmen-Entwicklung sowie des Software-Engineering) und nicht zuletzt des Gebiets *Informatik und Gesellschaft* (in dem Diskurse zur Software-Archivierung, Informatikgeschichte, Kultur und Didaktik der Informatik, zur Emulation sowie der Geschichte und Entwicklung von Mensch-Computer-Interaktionen stattfinden) genutzt werden. Für eine Studie im Fachgebiet der Informatik müssen diese Perspektiven privilegiert eingenommen werden. Aus diesem Grund finden sich im Verlauf der Arbeit zur Argumentation meiner Thesen sowohl technische Schaltpläne als auch Programmcode. Dort, wo auf informatische Spezialdiskurse zurückgegriffen wird, findet eine Definition von Fachtermini nur kursorisch und mit Verweis auf die einschlägige Fachliteratur statt.

Schließlich liefert die *Medienwissenschaft* in der Ausprägung der so genannten ‚Berliner Schule' die dann noch fehlenden methodischen Bausteine für die vorliegende Arbeit. Die eingangs bereits skizzierte Sicht der Medienwissenschaft (im Singular) auf *Medien als Medienapparate im Vollzug* und nicht etwa als Produzenten bestimmter Ästhetiken, Verständnis- und Nutzungsweisen, Wirkungen, Märkte und anderer Felder (mit denen sich die Medienwissenschaften – im Plural – beschäftigen) hat unter Friedrich Kittler und Wolfgang Ernst eine fruchtbare Methodik hervorgebracht: die *Medienarchäologie.* Sie beschäftigt sich, knapp gesagt, mit dem *medientechnischen Apriori des Wissens* und geht auf eine Lektüre des französischen Philosophen Michel Foucault zurück. Wie dieser untersucht die Medienwissenschaft die Bedingungen der Möglichkeit des Wissens. War es bei Foucault aber vor allem die (institutionelle, diskursive, biologische, ...) *Macht*, welche die Bedingungen des Wissens darstellte, sind es nach Kittler und Ernst vorrangig *Medientech-*

nologien, mit denen Information (und damit Wissen) gespeichert, übertragen und verarbeitet [Ernst 2012b; Kittler 1993c:64] werden.

Der in dieser Arbeit verwendete Begriff *Archäologie* unterscheidet sich von dem der akademischen Facharchäologie [vgl. Schneider 2004:93-96], indem er etymologisch (wieder) näher an die ursprüngliche Bedeutung des Wortes heranrückt: „Für den neuzeitlichen Begriff von Archäologie ist es kennzeichnend, grabend zu suchen. Genau dies aber kennzeichnet nicht den klassisch-griechischen Gebrauch des Wortes; in der gleichnamigen Schrift des Dionysius von Halikarnaß meint *archaiologia* schlicht die Redaktion, das Bearbeiten alter Nachrichten." [Ernst 2004:253f.] Archäologie in diesem Sinne ist zugleich Theorie und Methode: Das *theoretische* Wesen dieser Redaktionstätigkeit in Bezug auf die Historiografie wird im Kapitel 3 erläutert; das manipulative *Hands-on-Imperativ*, mit dem sich der Computerarchäologe *methodisch* den Artefakten annähert, wird an den Beispielen des Kapitels 4 exerziert. Wo Facharchäologie das (stets historische) Artefakt von den Sedimenten der Vergangenheit freilegt, um es derartig herauspräpariert der Beschreibung der Historiker und der Bewahrung der Museen zu überantworten, legt die Computerarchäologie das medientechnische Artefakt (das nicht notwendig ein vergangenes sein muss [vgl. Ebeling 2006:12]) von den es überlagernden Diskursschichten der Historiografie frei, indem es sich durch seine Oberflächen und Schichten ‚gräbt' bis hinab auf die Ebene von Hardware und Software. Dort manipuliert der Medienarchäologe die Funde, um ihnen verborgenes Wissen zu entlocken; die so verstandene Archäologie ist also keine Konservierungspraxis. (Zeitweise kann diese Form der redaktionellen Archäologie mit facharchäologischen „Grabungspraktiken" zusammenfallen, wie etwa bei der Freilegung der LSI-Strukturen historischer integrierter Schaltkreise [vgl. Swaminanthan 2011], um Informationen verloren gegangener Entwicklerdokumente durch Prozesse von Reverse Engineering zu ersetzen.[4])

Die Analysen dieser Archäologie verfahren im Wesentlichen *techno-mathematisch* (und nicht *diskursanalytisch* wie bei Foucault) und sind bereits von daher zum Methodenkanon der Informatik ‚kompatibel'. Zudem hat den „Computer als Medium" [vgl. Bolz/Kittler/Tholen 1994; Warnke/Coy/Tholen 1997 u. a.] die Berliner Medienwissenschaft bereits sehr früh als Forschungsgegenstand entdeckt und betreibt nach wie vor Grundlagenforschung zu dem, was sich auf Arbeitsfeldern wie den „Digital Humanities" [Jones 2014] oder der „Kulturtechnikforschung" [Holl 2015] ereignet.

So eng die Zusammenhänge zwischen dieser Medienwissenschaft und den Gegenständen des Gebiets *Informatik und Gesellschaft* in der Vergangenheit gewesen sind, hat sich in den vergangenen Jahren jedoch eine merkliche Trennung beider Forschungsdiskurse ereignet (die zuletzt in der Entscheidung der GI-Arbeitsgruppe „Computer und Gesellschaft"

4 Vgl. www.vsual6502.org [letzter Abruf: 13.12.2018]. Dort werden unterschiedliche historische Schaltkreise mechanisch und chemisch geöffnet, um ihre Strukturen zu analysieren und darauf basierende Softwaresimulationen zu entwickeln.

die langjährige, stark medienwissenschaftlich geprägte Konferenz-Reihe „Hyperkult“ zu beenden und sich aufzulösen, kulminierte). Wo die akademische Informatik beständig (auch als Folge des Bologna-Prozesses) anwendungsbezogene(re) Blicke auf den Computer eingenommen hat, hat sich die Medienwissenschaft des Computers zuletzt zu einer Art „Kulturwissenschaft des Computers“ [vgl. Holl 2015] gewandelt, die nun nicht mehr die technischen Medien als solche, sondern deren Kultur, Geschichte und Ökonomie untersucht und lehrt. Dort werden Computer beispielsweise vollständig auf „Dienste“ [Pias 2015:33], also (je nach Verständnis des Begriffs) auf Angebote für Anwender oder einzelne technische Funktionen reduziert. Es ist nicht zuletzt ein Anliegen dieser Arbeit, neue Möglichkeiten der Interaktion zwischen Informatik und Medienwissenschaft aufzuzeigen und so Fäden eines ehedem fruchtbaren interdisziplinären Diskurses wieder aufzunehmen.

1.4 Aufbau

Der Aufbau der Arbeit verfährt im wesentlichen deduktiv. Im zweiten Kapitel findet die Diskussion des Forschungsstandes statt. Dabei werden Beiträge ab Mitte der 1960er-Jahre bis in die Gegenwart daraufhin vorgestellt, in welche Beziehung sie Technik zur ihrer Geschichte setzen und dabei spezifische Begriffe in den Forschungsdiskurs einschreiben, welche Auseinandersetzungen es zur Bewahrung historischer Hardware und Software gegeben hat und gibt und welche Szenen und Protagonisten sich mit der Computergeschichte und dem Retrocomputing befassen und wie sie dies tun. Es wurde hierfür versucht einen Großteil der Forschungsbeiträge zu sichten und die relevan(est)ten Beiträge daraus überblicksartig vorzustellen. Einige an dieser Stelle nicht berücksichtigte Texte werden im weiteren Verlauf der Studie diskutiert.

Im dritten *Kapitel* wird der theoretisch-methodische Ansatz der *Computerarchäologie* vorgestellt. Hierzu müssen (auf Grundlage eines Beispiels diskursarchäologischer Praxis) die Möglichkeiten und Probleme aktueller Computergeschichtsschreibung benannt werden: Mit welchen Methoden und durch welche Quellen werden Computer als historische Artefakte dargestellt? Zur Beantwortung dieser Frage wird die methodologische Auseinandersetzung zwischen Ereignis- und Strukturgeschichte, wie sie der Technikhistoriker Wolfgang König [König 2009a:40f.] dargestellt hat, erörtert. Ausgewählte, besonders diskursmächtige[5] computerhistorische Monografien sollen hernach vor dem Hintergrund ihres methodischen Ansatzes, der von ihnen verwendeten Quellen, ihrer Argumentationsstrategien und ihres thematischen Fokus analysiert werden. Die Ergebnisse dieser Lektüre werden mit dem geschichtskritischen Modell der „Archäologie des Wissens“ Michel Foucaults [vgl. Foucault 1981] konfrontiert, die im Anschluss eine Erweiterung durch die Theorien der Medienarchäologie und schließlich der Formulierung der Methoden einer Computerarchäologie erfährt. Es soll sich zeigen, dass die Praxis des Retrocom-

5 Grundlage für diese Zuschreibung stellen die Auflagenzahl dar.

putings unter diesem Aspekt auch als *operative Geschichtskritik* verstanden werden kann – insbesondere dann, wenn sie sich den durch die publizierte Geschichtsschreibung marginalisierten Objekten (Plattformen, Programmiersprachen, Betriebssystemen, Programmen, Literaturen u. a.) zuwendet.

Das *vierte Kapitel diskutiert vier Retrocomputing*-Projekte detailliert, die zwischen 2012 und 2017 im Rahmen der universitären Lehre und Forschung durchgeführt wurden. Diese sind auf den Hauptbetätigungsfeldern der Szenen – Demo-Coding, Computerspielentwicklung, Software Preservation und Emulation sowie Hardware Preservation – angesiedelt. Einige dieser Projekte fanden im Rahmen universitärer Lehrveranstaltungen statt, die sich damit neben der der Anwendung einer „Retro-Didaktik" [Höltgen 2020] auch zugleich einer computerhistorischen Wissensvermittlung als „knowledge preservation" [Agrifoglio 2015:15-20] widmeten. In Kapitel 4.1 wird die Demoprogrammierung anhand eines physikalischen Beispiels (Ballsprung-Simulation) auf unterschiedlichen Plattformen (Analogrechner, Spielkonsole, Homecomputer) und in unterschiedlichen Programmiersprachen/-systemen vorgestellt. Mit Hilfe eines hierfür entwickelten computerphilologischen Methodensets werden diese Implementierungen miteinander verglichen und zueinander in eine ‚rezeptionsgeschichtliche' Beziehung gesetzt. Kapitel 4.2 diskutiert die Entwicklung eines Simulationsspiels, dessen epistemologische Deutungsmöglichkeiten sowie die grundsätzliche Beziehung zwischen Spiel(en) und Computern. In Kapitel 4.3 wird ein Projekt zur Software Preservation durchgeführt: die Entwicklung eines modernen Speichersystems für eine historische Computerspielplattform. Dieses wird kontrastiert mit der zentralen Bewahrungsstrategie für Software, der Emulation. Die Frage, welche Möglichkeiten, Grenzen aber auch welche Mehrwerte ein Emulator mit sich bringt, bildet hierbei den argumentativen Fluchtpunkt. Das letzte Projekt in Kapitel 4.4 befasst sich mit der Reparatur eines historischen Computers. Die Tools, die hierfür zum Einsatz kommen, entstammen einer autodidaktischen Beschäftigung mit historischer Hardware und rekrutieren Methoden des Retro-Hackings, die sich deutlich von konservatorischen Praktiken der Hardware Preservation unterscheiden. Der Wissenserwerb, der – neben einem wieder funktionsfähigen Computer – aus diesem Projekt resultiert, wird als dialektischer Prozess dargestellt, der sich im wechselnden Blick auf Werkzeuge als Hilfsmittel und Objekte des Wissens, ergibt.

Das fünfte *Kapitel* wertet die vier zuvor diskutierten Projekte aus, indem sie die darin zur Anwendung gekommenen Wissens-Erwerbsprozesse in eine Didaktik überführt. Diese befasst sich mit den Methoden, Praktiken und Quellen der Aneignung autodidaktischen Wissens, welches eine Bedingung der Möglichkeit von Retrocomputing darstellt. Den schulischen und universitären Konzepten der Informatik-Ausbildung werden seit dem ersten Erscheinen der Homecomputer in der zweiten Hälfte der 1970er-Jahre Materialien und Methoden autodidaktischer Informatik-Selbstausbildung zur Seite gestellt. Letztere bilden die Wissensbasis für die private ‚Aneignung' der Computertechnologie, womit alle

Formen der Verstehbarmachung, Modifikation und des „Missbrauchs“ [vgl. Kittler 1986:149; Pias 2015:32-34] von Computerhardware durch Hacker gemeint sind. Den Kulminationspunkt bildet hier ein Überblick über Formen des informatischen Selbstausbildung. Auf diese Weise soll ein Verständnis der *Computernutzung als Spiel* evoziert werden, womit nicht das Spielen von Computerspielen, sondern das Programmieren, die Hardware-Entwicklung und die produktive Auseinandersetzung mit der publizierten Computergeschichtsschreibung (etwa in Form von Rezensionen, Internet-Kommentaren, Zeitschriftenartikel und anderem) *mit spielerischen Vorgehensweisen* gemeint ist. *Gamification,* so verstanden, meint also den Wissenserwerb als Spiel – bei dem der zu beherrschenden Computer zugleich als Spielgegenstand, Gegenspieler und Gegenstand des Spiels in Erscheinung tritt.

Im *Schlusskapitel 6* werden die Untersuchung noch einmal zusammengefasst und ihre Ergebnisse kritisch reflektiert. Zwei Anliegen sollen hier als Ausblick vorgetragen werden: Zum einen soll das zugrunde liegende Forschungsprojekt in seinen interdisziplinären Konsequenzen dargestellt werden. Zum anderen sollen – in Form eines Ausblicks – Anschlussmöglichkeiten für Forschung und Praxis (hier vor allem zu Programmiersprachen und Computerspielen) vorgeschlagen werden.

1.5 Veröffentlichungen

Teile dieser Arbeit wurden bereits vorab veröffentlicht und/oder öffentlich vorgestellt. Die Methode der Computerarchäologie in Kontrast zu technikhistorischen Historiografien wurde auf der *Jahrestagung des VDI Technikgeschichte* im Februar 2016 diskutiert. Vorüberlegungen zur Anwendung der Medienarchäologie (in Abgrenzung zu den *Platform Studies*) auf Computerspiele wurden in [Höltgen 2013a] publiziert. Einen Überblick zur Didaktik maschinennaher Programmierung für Nicht-Informatiker wurde in [Höltgen 2014a sowie Höltgen 2018b] veröffentlicht. Die Perspektive auf autodidaktischen Programmierausbildung als Gamification ist in [Höltgen 2015a] vorgestellt worden. Das Reparaturprojekt aus Kap. 4.4 wurde 2017 auf dem Workshop „Kulturen des Reparierens“ vorgestellt und in [Höltgen/Groth 2018c] in abgewandelter Form publiziert. Eine erste methodische Darlegung einer Retro-Didaktik, die sich historischer Hardware, Software und Programmiersprachen bedient, ist 2018[6] auf dem „11. Marburger Theorieforum“ der *Deutschen Gesellschaft für Erziehungswissenschaften* vorgestellt und diskutiert worden. Eine Publikation erfolgt 2020. Zur Demonstration als Forschungsmethode ist 2019 ein Sammelbandkapitel publiziert worden [Höltgen 2019a]. Darüber hinaus finden sich zahlreiche theoretische und methodologische Überlegungen dieser Arbeit in Zeitschriftenaufsätzen, Sammelbandkapiteln, Vorträgen und Podiumsdiskussionsbeiträgen wieder, die seit 2011

6 https://theorieforum.de/chronik/medienbildung-zwischen-subjektivitat-und-kollektivitat-im-kontext-des-digitalen/ [letzter Abruf: 05.06.2019].

publiziert/aufgezeichnet wurden. Dort, wo diese Aspekte in der vorliegenden Arbeit aufgegriffen werden, sind sie bibliografisch erfasst und im Literaturverzeichnis angegeben.

2. Forschungsstand

Zu einigen in dieser Arbeit fokussierten Gegenstandsbereichen und Objekten existiert bereits seit Mitte der 1960er-Jahre ein Forschungsdiskurs – sowohl in der Informatik als auch in den Kultur- und Geisteswissenschaften. Zentrale Positionen und Beiträge daraus werden im Folgenden vorgestellt. Die Publikationen werden hierfür in drei Bereiche untergliedert: Beiträge, die sich mit spezifischen Formen des Geschichtsbezugs auseinander setzen (etwa, um Begriffe wie „Nostalgie“ oder „Retro“ zu definieren und gegeneinander abzugrenzen), Beiträge, die sich mit der Bewahrung von Hardware, Software und anderen historischen Artefakten beschäftigen und Analysen, die Retrocomputing-Communities und Szenen untersuchen.

2.1 Formen des Geschichtsbezugs

Bereits 1964 konstatiert der kanadische Medienwissenschaftler Marshall McLuhan, dass (fast) jeder Medientechnologie Geschichtsbezug inhärent ist, weil „der ‚Inhalt‘ jedes Mediums immer ein anderes Medium ist. Der Inhalt der Schrift ist Sprache, genauso wie das geschriebene Wort Inhalt des Buchdrucks ist und der Druck wieder Inhalt des Telegrafen ist.“ [McLuhan 1992:18] McLuhan spart die Computertechnik aus diesen Überlegungen in Understanding Media aus. Jay David Bolters und Richard Grusins Buch Remediation. Understanding New Media ließe sich dahingehend als ‚Fortsetzung‘ [vgl. Bolter/Grusin 1999:45] rezipieren. Medientechnik-Geschichte könne als Fortschritts- und Verbesserungsgeschichte verstanden werden [59-62]. Mit dem Begriff der „Remediation“ bezeichnen sie die Wiederaufnahme einer ‚alten‘ Medienfunktion in einer neuen. Dies geschähe dadurch, dass neue Medien (womit sie vornehmlich digitale Medien adressieren) die Ästhetiken, Formate, Nutzungsweisen und andere nicht-technische Aspekte ihrer Vorgänger aufgreifen und um solche Elemente erweitern, die aus gegenwärtiger Sicht als defizitär erscheinen [60]. Als ein Beispiel diskutieren sie Ego-Shooter-Spiele wie Myst und Doom als Remedialisierungen von Film und Fernsehen [47], die sie interaktivieren, deren Rezipienten sie vernetzen [102f.] und die sie durch „immersive virtual reality“ [48] um neue Medienaspekte ergänzen. Wie bei McLuhan bleibt die Argumentation in Hinblick auf die hierfür verwendeten Technologien aber vage. Und auch wenn Bolter/Grusin mit dem Remediation-Konzept im Prinzip bereits einen Ansatz für eine Emulationstheorie vorstellen, verwenden sie diesen Begriff selbst nicht – bzw. nicht im informatischen Sinn (über ein neues Zeitungsdesign schreiben sie: „the paper emulates in print [...] the graphical user interface of a web site.“ [40]).

Jean Baudrillard [1978] definiert den Begriff „Retro-Szenario“ als eine kulturelle Praxis, in der Vergangenheit in gegenwärtigen Medieninhalten ästhetisch evoziert wird. Er bezieht sich dabei vor allem auf Spielfilme, die die Vergangenheit „ein bißchen zu schön, zu gut eingestellt, besser als andere, ohne die psychologischen, moralischen und sentimen-

talen Überzeichnungen der damaligen Zeit“ [ebd.:52f.] inszenieren. Baudrillard diskutiert solche Filme als Beispiel für seine Simulationstheorie, nach der mediale Zeichen ein referenzloses „Hyperreale[s]“ [ebd.:52] herstellen, das keine Bezüge zum Realen mehr besitzt und dieses „zum Verschwinden“ [ebd.:55] bringt. Auch ohne der kulturphilosophischen Diagnose Baudrillards zu folgen, lässt sich sein „retro“-Begriff für die hier stattfindende Diskussion fruchtbar machen, beschreibt er doch vor allem ästhetische Praktiken, bei denen für neuen Medien (hier: aktuelle Computer) neue Inhalte (zum Beispiel Computerspiele) erstellt werden, die sich an Ästhetiken vergangener Inhalte anlehnen, um einem bestimmten Stil zu folgen (Retro-Stil) oder die ästhetischen Beschränkungen von Grafik, Sound, Geschwindigkeit, I/O u. a. produktiv zu nutzen (etwa für game engines, Schwierigkeitsgrade in Spielen usw.)

Elizabeth E. Guffey [2006] übernimmt Baudrillards Retro-Begriff in Bezug auf die referenzierte Zeit (die Nahe Vergangenheit bis zurück zu den Nachkriegsjahren [vgl. 18]) und grenzt ihn aus der Warte der Kunst- und Desgin-Geschichte von den Begriffen „Revival“ und „Nostalgie“ ab: Retro sei im wesentlichen so unemotional wie Revival-Kulturen [13], verhalte sich zur Vergangenheit jedoch ironisch zitierend [12] bis kritisch [13], womit sie mit Referenz auf Fredric Jameson [13,21] den Begriff in das Repertoire der postmodernen Kulturtheorie rückt. In einer Phänomen- und Begriffsgeschichte von „Retro“ koppelt sie den Begriff schließlich auch an retrofuturistische Ästhetiken, die einen weiteren Zeitbezug herstellen: Die Zukunftsprognosen der Vergangenheit [22ff.], womit vor allem design-ästhetische Facetten begriffen werden. Sie resümiert: „Retro is not just a recapitluation of the past; it focuses in their consciousness.“ [17] Sowie: „Retro implicitly invokes what is yet to come, as well as what has passed.“ [23] Retro sei zudem eine Objekt-orientierte [vgl. 26] Form der Auseinandersetzung mit Geschichte, durchgeführt von „‘freelance‘ historians“ [ebd.] und „a form of subversion while sidestepping historical accuracy.“ [11] Auf Computer und Computerspiele nimmt sie unter dem Begriff der „technological obsolescence“ [10] Bezug und verweist auf Subkulturen, die alte Spiele auf neuen Computern spielen und solche, die dazu alte Computer nutzen „to enjoy the ‚original‘ game experience.“ [Ebd.]

Im Zentrum der *Platform Studies* steht der Begriff der *Plattform*, den die Initiatoren der gleichnamigen Buchreihe, Ian Bogost und Nick Montfort [2006 und 2009a] insbesondere auf historische „computational platforms“ [2009:3] beziehen, die auch im hier diskutierten Zeitraum der Homecomputer-Ära entstanden: das *Atari VCS* (1977) [Bogost/Montfort 2009b], der Commodore *Amiga* (1985) [Maher 2012], das *Nintendo Entertainment System* (NES, 1983) [Altice 2015], das *Nintendo Super Entertainment System* (SNES, 1990) [Arsenault 2016] und der *BBC Micro* (1981) [Gazzard 2016]. Zwei weitere Publikation beschäftigt sich mit einer jüngeren Spielkonsole und dem Animationssystem *Flash,* eine weitere mit einem Grafik-System von Ende der 1950er-Jahre. Als Plattform verstehen die Herausgeber:

> [...] a computing system of any sort upon which further computing development can be done. It can be implemented entirely in hardware, entirely in software (which runs on any of several hardware platforms), or in some combination of the two. [...] platforms are pervasive in all sorts of computing. Personal computers like the Apple][are platforms. Programming languages such as BASIC can be thought of as platforms. Culturally important systems from decades past, such as the PLATO systems of the 1960s and '70s, are platforms. Platforms support digital art, hypertext, interactive fiction, chatterbots, recreational programs that aren't standard games, and other sorts of new media production. [Bogost/Montfort 2009a:2,3]

Ihre Annäherung an diese Systeme ist vorrangig von Fragestellungen der *Cultural Studies* geprägt: „But the real question should be whether a particular system is influential and important as a platform. Something is a platform when a developers consider [sic] it as such and use it" [4, vgl. auch: Höltgen 2013a:90f.]. Sie bedient sich informatischer Theorien und Methoden nur zur Analyse [vgl. Bogost/Montfort 2009a:5] der Artefakte. Dies erzeugt vor allem Defizite und Leerstellen bei Begriffen wie Programmierbarkeit [3] oder „computer" versus „video game" [vgl. 3] in Hinblick auf die Eingrenzung, was Plattformen sind und was nicht. Insofern ist der Ansatz nur bedingt für informatische und computerarchäologische Fragestellungen nutzbar. Obwohl die Autoren das Erkenntnisinteresse deutlich auf die „Unterflächen" [Nake 2005:47] von Plattformen richten, indem sie Codes und Hardwarefunktionen analysieren, münden ihre Forschungsresultate in keine Epistemologie des technischen Dispositivs Computer, sondern situieren es in eine kulturelle Level-Hierarchie (vgl. Abb. 3.3).

Dies zeigt eine an die Platform Studies (vgl. Kap. 3.3.3.1) angelehnten Fallstudie zum Retro-Computerspiel La-Mulana, die Camper [2009] publiziert hat: Er versucht die Frage zu beantworten, welches Verhältnis ein für moderne Windows-Plattformen aber weitgehend mit den technischen Restriktionen von historischen MSX-Computern programmiertes Spiel zwischen Technik- und Computerspiel-Vergangenheit und -Gegenwart aufwirft. Die Entwickler haben historische Gameplay-Elemente (etwa den Schwierigkeitsgrad) als subtile Kritik an gegenwärtigen ‚zu leichten' Computerspielen gewählt und ihr Spiel zur Plausibilisierung in einem audiovisuellen „MSX Style" [ebd.] designt. Dabei haben sie jedoch spiel-narrative Komplizierungen gegenwärtiger Spiele in dieses pseudohistorische Setting implementiert, so dass La-Mulana komplexe Beziehungen zwischen Vergangenheit und Gegenwart entwickelt. Camper analysiert diese Beziehungen vor dem Hintergrund einer technisch informierten Platform-Studies-Methode und als Erweiterung des medienhistorischen Ansatzes der „Remediation" [ebd.; Bolter/Grusin 1999].

Eine jüngere Monografie von 2016 mit dem Titel Retrogame Archeology erscheint vor dem Hintergrund der im Folgenden vorgeschlagenen Theorie und Methode der Computerarchäologie sowie des Fokusses auf Computerspiele besonders bedeutsam. Der Informatiker

John Aycock, der Autor des Buches ist, definiert seinen Gegenstand „Retrogame“ jedoch unabhängig von der akademischen Diskussion in Bezug auf ihr absolutes „age“ [Aycock 2016:2] und den Zeitraum 1973 bis 1993 [ebd.], ohne dies näher zu begründen. Ex post zeigt sich an den im folgenden unter (fach-)archäologischer Perspektive untersuchten Computerspielen allerdings der Sinn dieser Eingrenzung: Es sind diejenigen „old games“, aus denen Aycock bestimmtes Wissen über „tools, techniques and technology“ [206] zurück gewinnen möchte, um dies für die Entwicklung neuer casual games [vgl. 206] fruchtbar zu machen. Obgleich sein Archäologie-Begriff daher der Fach-Archäologie entlehnt und seine Methoden (Hardware- und Codeanalyse sowie Analyse von Paratexten wie Kopierschutz-Beigaben zu spielen, vgl. [145-172]) archäometrischen Methoden ähneln, verfolgt er – wie zahlreiche Retrocomputing-Projekte, wie sich hier später zeigen wird – implizit auch eine computerarchäologische Sichtweise auf seinen Gegenstand. Akademisch situiert er sein Projekt explizit nicht als „game history“ [3], was sich bereits am systematischen Aufbau zeigt und stellt die Retrogame Archeology Projekten wie den Game Studies, den Platform Studies, den Critical Code Studies und den Software Studies gegenüber. Während diese Computerspiele vorrangig mit kulturwissenschaftlichen Methoden und hermeneutisch untersuchen, ist sein Vorgehen vollständig auf die Technik konzentriert. [vgl. 205]

Der US-amerikanische Medienwissenschaftler Erkki Huhtamo [2007] formuliert mit seiner „medienarchäologischen“ [21] Perspektive auf das Computerspielen eine Kritik an der auf die „Ära der Chronik“ [18] konzentrierte Computerspielgeschichtsforschung. Indem er die technologischen Vorformen von Computerspielen (und insbesondere vom Computerspielen als Praxis) in unterschiedlichen professionellen und freizeitlichen Techniknutzungen bis ins 18. Jahrhundert zurückverfolgt, leistet er eine nach eigener Angabe erste historische Darstellung, die nicht auf der hobbyistischen oder nostalgischen Involvierung der Historiker basiert. Sein Ansatz bleibt jedoch anthropologisch/anthropozentrisch – er klammert eine „Archäologie der Spiele“ [43] bewusst aus, um sich arbeitswissenschaftlichen, gendertheoretischen, kulturhistorischen und anderen Diskursen zu widmen. Damit lässt er die im engeren Sinne medienarchäologische Frage nach den Technologien hinter den Computerspielen offen, um die es in dieser Arbeit gehen soll.

Der Begriff der *Nostalgie* (insbesondere der Technik-Nostalgie) steht im Zentrum von Sebastian Felzmanns Forschungsarbeit. [Vgl. Felzmann 2014] Am Beispiel retroider Computerspiele versucht er Konzepte der Game Studies um psychosoziale Fragestellungen (Einfluss der Rezeptionsumgebung auf das Spielen [26f.]) zu erweitern und weist damit auf die Unmöglichkeit hin, mit historischen Spielen die Vergangenheit wieder aufleben zu lassen. Vielmehr konzentrieren sich sowohl die Nutzer als auch die Publisher auf bestimmte Praktiken, die historische Hardware und Software aktualisieren und dabei den zeitlichen Doppelbezug von vergangenem und gegenwärtigen in der Praxis des Retroga*ming* [29] deutlich zu machen. Felzmann diskutiert dabei ästhetische Aspekte und soziale

Nutzungskonzepte von historischer Software sowie gegenwärtige Bemühungen (sozialer Austausch über Erinnerungen, Bewahrungsstrategien und Versuche der Neuinszenierung [29f.]). Er stellt heraus, dass die Nutzer(kultur) durch ihre Bemühungen implizit an einer Kanonisierung und Computerspiel-Historiografie arbeitet. [37] Der „Verlust des ‚digitalen Heimes'" [26], der aus der sukzessiven Technik-Weiterentwicklung resultiert, mündet ihm zufolge unter anderem in die Entwicklung von Emulationen, die jedoch aufgrund der nicht mit emulierbaren sozialen Kontexte defizitär bleiben müssen. Die technischen Beschränkungen von Emulatoren (vgl. Kap. 4.3) werden von ihm nicht berücksichtigt.

Nostalgie setzen auch Mortensen/Kapper [2015] in ihrer empirischen Untersuchung zu Computerspielen im Museum als primäre Motivation für den Zugang zu historischer Computertechnologie an. Dabei differenzieren Sie zwischen „simple nostalgia" [66], die sich im bloßen Gefühl erschöpft und nicht selten als „retrotypie" [64] (eine verklärende stereotypisierte Vergangenheit) betrieben wird, „reflexive nostalgia" [66], bei der der Erinnernde Fragen nach der Validität seiner Erinnerung stellt, und „interpretative nostalgia" [66], bei der zusätzlich auch der Grund für das nostalgische Fühlen eruiert wird. Letztere beide Formen stellen das Ziel von Ausstellungen dar, die hierzu einen höheren Grad an Partizipation der Ausstellungsbesucher erreichen müssen. Die Autoren sehen eine Machtbeziehung [vgl. 65] zwischen der Institution (Museum) und den Besuchern, die vom Grad der Partizipation abhängt. Während die Frage nach der Art der Partizipation bei der Nutzung operativer Ausstellungsobjekte (hier Autorenn-Computerspiele [vgl. 67]) unberücksichtigt bleibt, stellen die Autoren nostalgische Bezüge bei Besuchern fest, die gar keine biografischen Berührungen mit den ausgestellten Objekten besitzen. Die Frage, wie es zu dieser Nostalgie der „2. generation" [70f.] in Beziehung zu Computertechnologie kommt, delegieren Sie an nachfolgende Untersuchungen. (Ein Erklärungsversuch als „Wille zum Wissen" wird im Kapitel 5.1 unternommen)

David S. Heineman [2014] konzentriert sich in seinem Beitrag auf die Frage, wie Public History (vor allem in Form von Publikationen aber auch Wiederveröffentlichungen historischer Software) im Widerstreit mit privaten Initiativen der Computerspielgeschichtsarbeit steht. Dabei werden sowohl die indentitätskonstruktiven Aspekte von Spielvergangenheit (im Sinne des Spielens damals und heute) als auch die aktuellen Auseinandersetzungen mit historischen Spielen durch die Game Studies und die Popkultur thematisiert. Im Sinne von Foucaults Diskursarchäologie [vgl. 4] wertet Heinemann spezifische Machtstrukturen, die zu einem Kanon an ‚einschlägigen' und deshalb stetig wiederveröffentlichten Spieltiteln führen [vgl. 6]. Die Technologien, auf denen diese Wiederveröffentlichungen basieren – insbesondere Neuprogrammierung von alten Spielen für aktuelle Systeme [9f.] und Verwendung von Emulatoren [8] – spielen in diesem Zusammenhang nur insofern eine Rolle, als gefragt wird, welche Art von Nostalgie durch welche Wiederveröffentlichungsform ermöglicht oder verunmöglicht wird. Für die folgende Diskussion ist zu notieren, dass Heinemann die Spiele von dem Vorgang/der Handlung des Spielens

unterschiedet [15]. Die Erinnerungsarbeit (also ein aktiver Prozess) ist dabei an Handlungen gekoppelt.

Den Begriff und die Praktiken des *Retrocomputings* stellen Takhteyev und DuPont [2013] in ihrem Beitrag zunächst musealen Bewahrungsstrategien (Preservation) gegenüber. Ausgehend von einem konkreten Fall (der Wiedergewinnung des Sourcecodes des Apple-II-Spiels „Prince of Persia“ durch den damaligen Autor sowie privaten Retrocomputing-Enthusiasten [vgl. 422f.]) stellen sie die Werkzeuge und Wissensressourcen vor, die für solche Projekte genutzt und entwickelt werden. Die Protagonisten handeln dabei vor allem privat als „hobbyists“ [423] und sind „hardly unskilled amateurs“ [423], also technische/informatische Autodidakten, die selten kommerziellen Interessen besitzen [vgl. 423]. Ihre Perspektive auf historische Computer ist vom Gedanken der Operativität [vgl. 424] bestimmt und durch „fun“ [424] motiviert. Die Autoren werten Retrocomputing-Praktiken grundsätzlich als Bewahrungsstrategien, die einerseits „itself as a historically situated practice“ [425] darstellen, andererseits einen differenzierten Blick auf Computergeschichte offenbaren: Im Sinne von Donna Haraways Theorie der „situated histories“ [Haraway zit. n. 429] entstehen dabei eine Vielzahl gleichwertiger Mikro-Historiografien, die sich aus den jeweiligen Quellen und Praktiken der Retrocomputer-Hobbyisten speisen. Der Authentizität [vgl. 427-430] des historischen Originals werden diese *situated computer histories* entgegengestellt, indem historische Hard- und Software einen „Remix“ [427] mit anderer Hardware, Software und Wissen aus unterschiedlichen Zeiten/Epochen der Computergeschichte (insbesondere der Gegenwart) erfährt: „It can also involve a mixture of historic components that were never combined in the past“ [424]. Die Autoren sehen sowohl in den Praktiken als auch in der von den Hobbyisten betriebenen Geschichtsschreibung eine Bereicherung für ‚professionelle‘ Archive, Museen und Geschichtsschreibungen und unterstreichen damit die bereits von Agar [1998] geforderte Ergänzung von Computergeschichte um ein historische Untersuchungen zu Gebrauchsformen („Use and Informatic History“ [Agar 1998:3]). Institutionen sollen im Gegenzug solche privaten Praktiken infrastrukturell fördern. Die theoretischen Positionen Takhteyevs und DuPonts sollen in dieser Arbeit an zwei Stellen erweitert werden: Die „playful[ness]“ [Takhteyev/DuPont 2013:427], die die Autoren in den Retrocomputing-Projekten sehen, werden hier (vgl. Kap. 5.2.3) detailliert als Gamification-Prozesse analysiert; die „ongoing, living practices“ [428] und die Betonung der Operativität historischer Computer in den Szenen wird dabei als eine Epistemologie des Comput*ing* gedeutet. Die in Kapitel 4 vorgestellten Projekte erfüllen die Forderungen der Autoren nach infrastruktureller Beteiligung öffentlicher Institutionen insofern, als sie Retrocomputing-Projekt darstellen, die innerhalb der universitären Forschung (sowohl mit Studenten als auch mit universitätsexternen Hobbyisten) realisiert wurden.

Den konstitutiv gegenwärtigen Charakter, der sich im *present progressive* von *Retrocomputing* ausdrückt, betont Benjamin Beil [2013] für das Retro*gaming,* indem er es als eine

„performative Aneignungsstrategie“ [320] von (Computerspiel)Geschichte definiert. Diese müsse gleichermaßen über die Handlungen im System („innerbildliche Performativität“ [321]) als auch die Handlungen innerhalb des Systems in der Retrogaming-Szene („außerbildliche Performativität“ [ebd.]) vollzogen werden. Beil ordnet drei Formen von „Re-Aktualisierungen“ [320] diesen Performanzen zu: Remakes („Retrogames im weiteren Sinne“ [327]), Retro-Remakes („neuentwickelte Spiele, die audiovisuelle [Elemente] … von älteren Spielen [...] imitieren“ [329]) und Retro-NextGen-Hybride („kombinier[en] Retro-Stile mit zeitgenössischen Formen und transformier[en] die jeweiligen Ästhetiken“ [330]). Obzwar Beil in seiner Analyse neue Spiele für historische Plattformen ignoriert, lässt sich sowohl seine Darstellung des performativen Charakters auf das Retrocomputing übertragen als mit seiner Hilfe auch die kulturtheoretischen Probleme von Emulation [vgl. 324f.] bündeln, wie in Kapitel 4.3 ausgeführt werden wird.

2.2 Preservation und Emulation

Die Nachbildung von spezifischen Computertypen durch Software, die auf nachfolgenden/jüngeren Geräten ausgeführt wird, ist seit den 1960er Jahren ein Thema in der angewandten Informatik [vgl. IBM 1962:27]. Für den hier aspektierten Zeitabschnitt der frühen Mikrocomputer beginnen die Arbeiten dazu recht früh: Für Homecomputer, die auf spezifischen Mikroprozessoren basieren, werden Emulatoren anderer Homecomputer, die über dieselbe Prozessorarchitektur verfügen, angeboten – noch während die zu emulierenden Systeme am Markt sind (vgl. Kap. 4.3.1.3). Spätestens mit dem Erscheinen des Commodore 128, der als Zweitsystem des Vorgängermodells Commodore 64 enthält, offenbart sich die vordringliche Intention der Emulatoren-Entwicklung: die zeitliche Ausdehnung des Softwareangebots eines veralteten Computersystems (beim C128/C64 vor allem die Spielesoftware betreffend). [vgl. wt 1985:113]

Omar Muñoz-Cremers [1999] widmet sich einem spezifischen Emulatoren-System, dem MAME (Multi Arcade Machine Emulator), der ab 1996 für das MS-DOS/Windows-Systeme erschien und zahlreiche Arcade-Automaten emuliert. Muñoz-Cremers unterscheidet drei Typen von Emulation: „those that make your home computer imitate another computer [...] emulators that imitate game computers [gemeint sind Spielkonsolen, S.H.]“ [107] sowie „MAME [that] is actually in a category all its one“ [ebd.], weil MAME lediglich ein Betriebssystem zum Aufsetzen spezifischer Hardware-Emulationen/Konfigurationen darstellt. Für Muñoz-Cremers stellt Emulation bereits einen Aspekt von Geschichtsarbeit („archaeology“ [ebd.]) dar: „the conservation and study of cultural artifacts that have more value for us than any random work of art from the same period.“ [ebd.] Diese Bedeutung liege in der Möglichkeit die Wandlung der Spielkultur zu untersuchen sowie Methoden der Spielproduktion, die sich seither geändert habe. Das MAME-System, das zu Beginn als ‚Bundle‘ von Emulator-Software und ROM-Dateien der zu emulierenden Spielautomaten angeboten wurde, entfachte eine juristische Auseinandersetzung, die die Ur-

heberrechte der ROM-Inhalte zum Gegenstand hatte und damit zusammen mit der zeitgleichen Debatte von MP3-Musikdownloads einen Wandel in der Frage zu den Rechten virtueller Objekte auslöste [vgl. 111f.].

Ein Jahr später reflektiert Doron Swade [2000] den Status solcher virtueller Objekte in einer an der Materialität orientierten (Museums)Kultur. Die Frage, warum nicht anstelle von historischen Originalobjekten auch materielle oder immaterielle Nachbildungen in Museen gesammelt und ausgestellt werden könnten, beantwortet er damit, dass nur Originalobjekte erweiterte Untersuchungen (etwa mittels „forensic utility" [140]) zulassen. Aus seiner Argumentation geht hervor, dass insbesondere die Virtualisierung von Objekten zukünftige Forschungsfragen im Vornherein unbeantwortbar mache – dass das Verwerfen von materiellen Objekten der „open-endness of human inquiry" [ebd.] zuwiderlaufe. Swade diskutiert dies schließlich am Beispiel zweier computerhistorischer Artefakte: der Difference Engine No. 2, die aus den Originaldokumenten Charles Babbages 1989 erstmal real konstruiert wurde [141-144], sowie anhand der „bit-level simulation" [144] des Elektronenröhren-Computers Ferranti Pegasus aus dem Jahre 1959. Wo die Konstruktion der Difference Engine No. 2 vor allem museumsdidaktische und technikhistorische Fragen beantworten soll(te), welche durch eine Detail-Simulation mittels Computeranimation noch unterstützt wird, da ist Swade zufolge die Emulation des *Ferranti*-Computers ein adäquates Surrogat zur dysfunktionalen Hardware des Originalgerätes. Hierbei referenziert Swade die Überlegungen Turings dazu, dass jede Turingmaschine eine andere Turingmaschine simulieren kann [145f.], eben weil ein emulierter historischer Computer auch noch ‚Retrocomputing' ermöglicht:

> If we wish at some future time to know whether in the 1950s there was enough computing power to solve the trajectory equations of a missile fast enough to act as a guidance system, the Pegasus simulation can be used to find out. [...] So, in the special case of electronic computers, logical simulation as a virtual object in some respects survives the forensic test of historical utility and suggests in prospect a salvation for the conservator for whom deterioration spells failure. [146]

Der (*Ferranti*-)Computer wird damit als eine bloße Emanation einer Turingmaschine angesehen; seine technikhistorischen Besonderheiten werden auf Rechengeschwindigkeit, I/O und Speicherkapazität reduziert. Jegliche materielle und physikalische Idiosynkrasie bleibt in solch einer Emulation unberücksichtigt.

Thibadeau [1994] diskutiert mögliche Erhaltungsstrategien und -methoden für digitale Informationen. Die größten Probleme stellen ihm zufolge „media fragility and technology obsolescence" [101] dar. Beide werden forciert durch die Entwicklungsgeschwindigkeit von Computertechnik und die Entstehung neuer Datenformate und Standards. Thibadeau betont, dass die Erhaltung der (semantischen) Information nur einen Aspekt darstellt; die protosemantischen Elemente (Schriftart, Auflösung, Farbräume etc.) müssten ebenfalls adäquat gesichert werden – zumal diese stark mit der später zur Wiederga-

be verwendeten Hardware und den verwendeten Datenformaten variieren können [102]. Bei bereits digital vorliegenden Inhalten (HTML, WordPerfect-Dateien, Bild-Dateien [vgl. 102] eskaliert diese Problematik: „In the simplest sense, a document can be said to be preserved if and only if it is possible to retrieve the document at some point in time and know that it is the same as it was at some prior time. But this simple concept of preservation is practically without meaning in a digital environment. As the above examples show, one cannot know how a digital document appeared at any time in the past unless one saw it at that time." [103] Die zentrale Frage von Thibadeaus Beitrag ist die nach der Erhaltung all jener Facetten, die über die semantische Information [105] hinausgehen. Hier seien zwei Strategien denkbar: „maintaining the original technology [and ...] imitating the original technology" [106]. Datenformat-Konversion (von einem in ein anderes proprietäres Format) stellt eine Möglichkeit dar, Inhalte und Metainformationen zu bewahren [107], auch wenn dies nur eine mittelfristige Lösung darstellt. Die „Transformation to Persistent Object Form" |108f.], die unabhängig von spezifischer Hard- und Software Inhalte und Strukturen (etwa mit XML) sichert, wäre eine Lösung, „for the performance of archival functions as well as the persistence of the records" [109]

Mit „imitating original technology" [106] ist die Emulation von Programmen auf jüngeren Systemen gemeint. Obgleich Thibadeau darauf hinweist, dass dies bereits zu Verlusten führt, konzentrieren sich zahlreiche Forschungsbeiträge auf diese Möglichkeit. Angefangen bei Rothenberg [1999], Emulation als die am wenigsten inadäquate Methode kennzeichnet [30f.], bis hin Borghoff et al. [2003], die der Emulation unter anderen Erhaltungsstrategien (Migration, Verwendung von Markup-Languages u.a.) das größte Potenzial für eine Langzeit-Erhaltung einräumen. Die vorliegende Arbeit greift eine Position aus diesem Kanon heraus und diskutiert sie eingehender (vgl. Kap. 4.3). Diejenige von Jens-Martin Loebel [2014]: Er evaluiert in seiner Studie „Lost in Translation" 22 aktuelle Software-Emulatoren für historische Mikrocomputer vor dem Hintergrund der Software Preservation. Insbesondere um Computerspiele als kulturelle Artefakte [16f.,26] geht es ihm dabei, weil diese die anspruchsvollsten „digitalen Objekte" für Bewahrungsstrategien darstellen. Aufgrund von aus der Sekundärliteratur zitierten Aussagen konstatiert er eine beschränkte Haltbarkeit von sowohl historischen Datenträgern als auch historischen Computern und plädiert daher für die Migration [37f.] und Emulation von Hardware und Software auf neuere Systeme als einzige adäquate Möglichkeit „dynamische digitale Objekte" [27ff.] kontextsensitiv zu bewahren. Hierbei sieht er bislang vor allem hobbyistische Szenen als Protagonisten, worin das Problem besteht, dass keine Standards für Implementierungen, ungenügende Dokumentation und nicht zuletzt keine garantierte Verlässlichkeit für die Kontinuität von Emulatorprojekten besteht. Die Frage nach der möglichen Bewahrung von Hardware (zur Wiedergabe historischer Software), die über die „medienarchäologische Geräteerhaltung" [35] von Museen hinausgeht, verneint Loebel aufgrund ökonomischer und technischer Hürden [vgl. 36], zeigt aber zugleich auch juristische und sozio-ökonomische Probleme der (professionellen) Software Preser-

vation [vgl. 22ff.] auf. Loebel sieht das Ziel einer erfolgreichen Software Preservation allein in der Emulation und ihrer authentischen Reproduktion von Ausgaben des Quellsystems auf dem Zielmediums. Sein hierfür verwendetes Schichten-Modell „Komplexer Objekte“, bestehend aus physischen, logischen und konzeptuellen Objekten [vgl. 28] läuft auf ein Blackbox-Modell hinaus. Damit vernachlässigt er alle Aspekte, die epistemologisch, historisch und technisch ‚hinter‘ Computerspielen stehen (Originalcode, zeitkritische Funktionen, Spezifikationen der Originalhardware etc.) Seine Darstellung von Emulation als „Nachahmung von bestimmten Teilaspekten eines Computersystems mittels Software oder Hardware“ [41] bleibt dabei aus Sicht der Informatik pragmatisch und wirft epistemologische Fragen – etwa zum Unterschied von Emulation und Simulation [ebd.] oder analog und digital [vgl. 33f.] – nur am Rande auf. Eine eingehendere Diskussion seiner Arbeit findet in Kapitel 4.3.1 statt.

2.3 Szenen und Protagonisten

Claus Pias hat sich insbesondere im zeitlichen Rahmen seiner Dissertation *Computer Spiel Welten* [Pias 2002a] mit einigen Diskursen und Themen auseinandergesetzt, die im Rahmen dieser Studie diskutiert werden. [vgl. Pias 2005a, 2005b] Neben den diskursarchäologischen Untersuchungen der Computerspiele ist es vor allem die Figur des Hackers, die er dabei ins Zentrum rückt. In [Pias 2002b] konstatiert er, dass der Hacker erst ex post durch den historischen Diskurs über ihn (vor allem geführt durch [Levy 1984]) als Figur mit Ethik, modus operandi und Gegenständen ausgestattet wurde. [251f.] Konstitutiv sei für ihn, dass seinem Tun eine grundsätzliche Paradoxie zugrunde liegt: „Mit jedem Hack verschwindet [...] eine Möglichkeit zu hacken. [...] Der Hacker schleppt also die Grenze, die er zu überwinden scheint, immer mit und zieht sie ununterbrochen neu.“ [263]. Den Grund dafür sieht Pias in der von ihm gehackten Technologie: „Das medientechnische Apriori des Hackers liegt in der Universalität von Turingmaschinen selbst. Jede symbolische Operation eines Computers ist eine ‚richtige‘ Benutzung, und in diesem Sinne gibt es gar keine ‚anderen‘ oder ‚falschen‘ Verwendungen, sondern nur unaktualisierte Virtualitäten.“ [260] Ohne es expressis verbis zu formulieren, kondensiert Pias die Tätigkeit des Hackings (grammatisch klar als *present progressive* geschrieben) damit als eine Praxis, die nur im Moment ihres Stattfindens Sinn erzeugt. Jeder vergangene Hack ist bereits kein Hack mehr, sondern eine Anwendung. Dies rückt den Begriff des Hacking in die Nähe des Spielens, das ebenfalls erst im Vollzug seinen Sinn erfährt [vgl. 84]. In der Konsequenz kennzeichnet Pias den Hacker damit nicht nur als „in seinem innersten Impuls ein Spieler, und seine historische Möglichkeitsbedingung ist der Digitalrechner als universale Spielmaschine“ [254], sondern verdeutlich auch die (auto)didaktischen [vgl. 254,265] Implikationen, die das „spielende Programmieren“ [265] in die Sphäre der Pädagogik und informatischen Didaktik [265] überführen. Dieser Ansatz einer „Gamification“ des Programmierens wird weiter unten methodisch näher ausformuliert und die Fragen der not-

wendigen Präsenz von Hack*ing* wie von Comput*ing* als (Technik)Geschichtskritik weiter verfolgt.

Die Demoszenen, deren Protagonisten aus der Hacker- und Crackerkultur hervor gegangen sind, gehören zu den produktivsten Retrocomputing-Gruppen, deren Protagonisten auf historischen Computern aktuelle Software programmieren und dabei nicht selten die scheinbaren Leistungsgrenzen ausweiten. Demoszenen sind verschiedentlich Gegenstand wissenschaftlicher Untersuchungen geworden – häufig im Rahmen kunsthistorischer [vgl. Hartmann 2017] mediensoziologischer Forschung, die [vgl. Reunanen/Silvast 2009 oder dies. 2014] Demoszenen als „collection of social relationships, practices and technologies, whose compositions must be discussed by the members in order to keep it afloat“ [Reunanen/Silvast 2009:291], beschreiben. Die daraus resultierende quantitative Untersuchung des szeneinternen Diskurses und seiner Topoi klammert explizit ästhetische Aspekte aus. [Vgl. ebd.] In einer umfangreichen Arbeit stellt Botz [2011] verschiedene Demoszenen in ihrer historischen Genese, ihrem modus operandi und ihren Beziehungen zu den Plattformen, für die sie programmieren, vor. Entscheidende Thesen bei Botz sind, dass Demos stets im Verhältnis zur Plattform, für die sie entstanden sind, gewertet werden müssen [vgl. 16], dass der Computer dabei nicht bloß das Werkzeug, sondern auch das Material der Praktiken darstellt [vgl. 22], dass die Arbeit der Demo Coder explizite wie implizite Momente von Geschichtskritik darstellt [vgl. 22, 102] und dass Demos von ihren Entwicklern – im Sinne des Hackings – als eine „ästhetischen Form des Widerstands“ [22] gesehen werden. Botz betont, dass es ein definierendes Kriterium einer Computerdemo ist, dass sie live gerendert [16, 289] werden müsse. Das heißt: Weder vermögen Videoaufnahmen von Demos noch Demos, die lediglich zuvor gerenderte/produzierte Bilder aus dem Speicher abrufen und zur Darstellung bringen das *Echtzeit-Phänomen Computerdemo* adäquat zu repräsentieren. Wie Retrogaming und Retrocomputing zeigen sich Computerdemos damit als radikal Gegenwärtige Praktiken und Prozesse. Die Konsequenzen dieser Zuschreibung werden in Kapitel 4.1 erörtert.

3. Computerarchäologie

Der (sogenannte) Computer besitzt zahlreiche Geschichten und Vorgeschichten, die sich – je nach Perspektive – bis zur sumerischen Mathematik [vgl. Malek 2006:198] zurück verfolgen lassen: Seine *Ideengeschichte* beginnt wahlweise mit der Herausbildung der formalen Aussagenlogik durch Aristoteles [vgl. Bolter 1990:151-182], der Entwicklung des dualen Zahlensystems durch Leibniz [vgl. Bolz 1994:11] oder der Schaltalgebra durch Shannon [vgl. Dotzler 2006:171]. Der Beginn seiner *Technikgeschichte* wird in den Rechenmaschinen von Wilhelm Schickard [vgl. Dotzler 1996:11ff.], der Entwicklung programmierbarer Automaten durch Charles Babbage und Ada Lovelace [vgl. Siegert 1993:138] oder der ersten Implementierung der Turingmaschine durch Konrad Zuse [vgl. Rojas 1997] angesetzt; *Computerkulturen* (*Kulturgeschichte*) entwickeln sich beim Einzug des Digitalcomputers in Universitäten [vgl. Höltgen 2012:271ff.], durch die Kommerzialisierung (*Wirtschaftsgeschichte*) des Computers [vgl. Ceruzzi 2003:35ff.], beim Übergang von Mainframes zu Minicomputern [vgl. Levy 1984:14ff.], durch die Privatisierung (*Sozialgeschichte*) von Digitalcomputern mit der Entwicklung des Mikroprozessors [vgl. Malone 1995:25ff.] und schließlich sogar in Hinblick auf spezifische Veränderungen des Gehäuse-Designs (etwa zur Anpassung von professionellen an private Kontexte) [vgl. Atkinson 1998].

Allein diese *Pluriperspektivität* auf die Geschichten der Computer, die sich für die Geschichte der Mikrocomputer ebenso einnehmen ließe, wenngleich auch in einem engeren Zeitraum, offenbart die Vielfalt differenter und nicht selten disparater historiografischer Ansätze. Im folgenden Kapitel sollen die Methoden und Paradigmen der Computergeschichtsschreibung diskutiert und kritisiert werden. Im Zentrum steht dabei die These, dass zwar sehr wohl eine Computer-Geschichtsschreibung existiert, diese jedoch nicht nach Fragestellungen der Informatik, sondern vor allem aus wirtschafts-, mentalitäts-, sozialgeschichtlicher und anderen Perspektiven vollzogen wird. Der Grund dafür liegt zum einem in der schwer fassbaren (und damit unmöglich historisierbaren) Operativität von Computern; zum anderen in den methodischen Paradigmen der Geschichtsschreibung selbst.

Zunächst werden hierzu beispielhaft Darstellungen der Computergeschichte (unter besonderer Berücksichtigung der Homecomputer-Ära) in ihrer Methodik, ihrer Argumentation und ihrer Besonderheiten (Auslassungen, Fokussierungen auf bestimmte Plattformen, Themen und ähnliches) vorgestellt. Hierfür wurden drei kanonische historische Monografien und zwei zeitgenössische Abhandlungen ausgewählt und zudem wird im weiteren Verlauf auf ähnliche Titel verweisen. Darstellungen, welche die Hardware, Software, Personen (Ingenieure, Nutzer) und ökonomische Aspekte fokussieren, sowie spezifische Historien (etwa die reichhaltige Literatur zur Computerspiel-Geschichte [vgl. Kent 2001]) bleiben unberücksichtigt. An den Texten soll sich zeigen, wie sich bestimmte Me-

thoden, die Auswahl historischer Quellen und „narrative Erklärungen“ [White 1991:78] etabliert haben.

Danach erfolgt eine Einordnung aus historisch-methodologischer Perspektive, die in eine Form jüngerer Geschichtskritik mündet, welche den theoretischen Ansatz zur Formulierung einer Computerarchäologie bildet. Die Methodik dieser Einordnung rekrutiert sich aus Theorien der Geschichts-, Literatur- und Medienwissenschaft, weil die Geschichte der Computer (noch) kein Gegenstand der Informatik ist, sondern der Technikgeschichte und damit dem Methodenkanon historisch-hermeneutischer Disziplinen unterliegt. Die Defizite dieser Perspektive sollen aufgearbeitet werden mit dem Ziel, eine Theorie zu entwicklen, die Computer, ihre Peripherien, Software und kulturellen Diskurse als historische Gegenstände neu bewertet und zugleich in der Lage ist, den operativen historischen Computer für eine Analyse nach informatischen Methoden zugänglich zu machen.

3.1 Methoden der Computer-Geschichtsschreibung

3.1.1 Zeit-, Sozial- und Mentalitätsgeschichte: Hackers

Die private Nutzung von Computern weist eine facettenreiche Geschichte auf, die zu dem Zeitpunkt einsetzt, als Computer rein militärische Umgebungen verlassen und in industriellen und akademischen Kontexten eingesetzt werden. Zwar gab es zuvor bereits einzelne „Missbräuche von Heeresgerät“ [vgl. Kittler 1986:149], doch eine andauernde und vor allem nach autodidaktischen Methoden vorgenommene Auseinandersetzung mit Computern durch private Nutzer beginnt erst, als Universitäten kleinere Systeme wie den TX-0 oder den SDS-940 [vgl. Höltgen 2014b, Levy 1998:139f.,157,210] mit individuell nutzbaren, alphanumerischen Terminals einsetzen, welche die vormaligen Mainframes mit Batch Processing auf Lochkartenbasis ablösen. Ab diesem Zeitpunkt beginnt die Geschichte des *Hacking*, die in den 1970er-Jahren in die Geschichte des Homecomputing übergeht.

Eine der frühesten Buchpublikationen, die sich mit der Geschichte des frühen Mikrocomputers vor dem Hintergrund seiner privaten Nutzung auseinandersetzt, ist Steven Levys Monografie Hackers. Heros of the Computer Revolution aus dem Jahre 1984. Der Autor gliedert seinen Text in drei Großkapitel, die sich mit den Hackern der 1950er- bis 1980er-Jahre befassen und die dabei spezifische ‚Epochen‘ beschreiben.

Levy umreißt in seiner Geschichte einen mit der privaten Nutzung von Computern an den Universitäten entstehenden *Mentalitätstypus* – den des Hackers –, dessen Motive und Ethiken er aus seinen Handlungen und gesammelten Aussagen ableitet. Hackers ist damit gleichermaßen eine *Zeitgeschichte* [vgl. Jordan 2009:143ff.]) wie eine *Mentalitätsgeschichte* (mit all ihren Unschärfen [vgl. Jordan 2009:165ff.]) und entwickelt überdies auch eine Art der ‚Hagiografie‘: Levy schildert von vielen der in seinem Buch vorgestellten Hackern de-

ren Lebensweg von frühester Kindheit bis hin zu ihrem Ausscheiden aus der Hacker-Community, dokumentiert ihre Taten, Erfolge und Misserfolge sowie ihre Beziehungen zueinander (sowohl in persönlicher als auch in fachlicher Hinsicht). Dabei suggeriert er nicht selten eine gewisse biografische Zwangsläufigkeit in ihrer persönlichen Entwicklung vom bastelnden Kind zum Hardware- und Software-Hacker.

Seine historische Methodik folgt vor allem der *Oral History* [vgl. Jordan 2009:162ff.]:

> As I talked to these digital explorers, ranging from those who tamed multimillion-dollar machines in the 1950s to contemporary young wizards who mastered computers in their suburban bedrooms, I found a common element, a common philosophy which seemed tied to the elegantly flowing logic of the computer itself. It was a philosophy of sharing, openness, decentralization, and getting hands on machines at any cost – to improve the machines, and to improve the world. This Hacker Ethic is their gift to us [...]. [Levy 1984:ix]

Zur Unterstreichung dieses sehr personen- und maschinengebundenen Ansatzes, stellt Levy seiner Einleitung ein „Who's who“ [ebd.:xi-xv] voran, in welchem die Hacker und die Computer namentlich genannt und in ihren Taten bzw. Features kurz spezifiziert sind. Die Eckpunkte seiner Epocheneinteilung sind: Im Kapitel „True Hacker“ [ebd.:1ff.] beschreibt er den Zeitraum von 1958 bis 1972, in welchem sich im universitären Kontext (Boston, Stanford, Berkeley) unterschiedlichste Nutzungsformen von der Computerspielprogrammierung über die Klangerzeugung, Zugriffe aufs Telefonnetz, Entwicklung von Schach-Programme und unabhängigen Betriebssystemen ereignet haben. Die Ära der „Hardware Hackers“, die Levy im zweiten Kapitel [ebd.:145ff.] aspektiert, reicht der Gründung des „Community Memory Projects“ 1973 durch Lee Felsenstein [vgl. Höltgen 2014b] bis zur Insolvenz der *Osborne Computers Corporation* 1983 und gibt die Ereignisse nach der Veröffentlichung der ersten frühen 8-Bit-Mikroprozessoren wieder, die schließlich in die Entwicklung der Homecomputer-Industrie münden. Im dritten Kapitel „Game Hackers“ [ebd.:277ff.] skizziert Levy den Zeitraum von der Gründung des Computerspiele-Herstellers *Sierra On-Line* (1980) bis zu den Aktivitäten Richard Stallmans (GNU) am MIT 1983. Hier sind es Programmierer kleiner Systeme und (Spiel-)Software- sowie -Hardware-Firmen, die im Zentrum der Darstellung stehen. Das zweite und dritte Kapitel beschäftigen sich mit Plattformen, die unter die in dieser Arbeit verwendete Definition früher Mikrocomputer fallen. Das letzte Kapitel kann in Hinblick auf die Tatsache, dass das Buch publiziert wurde, als dieser Teil der (heutigen) Vergangenheit damalige Gegenwart war, als nur im eingeschränkten Sinne historiografisch verstanden werden [vgl. Lee 1985:270].

Obgleich Levys Text großteils chronologisch strukturiert ist, sich an der Entwicklung und Vermarktung stetig neuer Plattformen und Software orientiert und schließlich sogar ein ökonomisches Ereignis (die Konkurrenz zwischen *Symbolics* und *LMI* [vgl. ebd.:421-437])

als Endpunkt der Hackerkultur setzt, versteht er ihn „in no way [as] a formal history of the computer era, or of the particular arenas I focus upon“, sondern „[as] the real story of the computer revolution“ [ebd.:x]. Der methodologische Konflikt zwischen einerseits dem Anspruch an Objektivität einer historischen Darstellung und andererseits den extrem subjektiven Quellen der Oral History [vgl. Göpfert 1996] ist in Rezensionen und nachfolgenden Lektüren dennoch seltener thematisiert worden, als die fehlende ökonomische Perspektive auf die Frage nach der Privatisierung des Computers [vgl. Ceruzzi 1998:236,264].

In zeitgenössischen Rezensionen ruft das Buch aufgrund seines sehr persönlichen Zugangs zum Thema teilweise euphorische Reaktionen hervor [z. B. Lee 1985]; zuweilen wird jedoch die allzu große Distanz des Autors zu den technischen Objekten, über die er schreibt, moniert. So stehen im Zentrum der Rezension von Williams [1985] „errors of fact concerning some of the machines Levy is describing (which suggests that the people he is describing are equally misrepresented)“ [ebd.] Insbesondere das Insistieren auf technische Korrektheit als Qualitätskriterium einer Computerhistoriografie ist ein Aspekt, der sich später auch in Rezensionen zu Retrocomputing-Büchern findet und in Einzelfällen sogar dazu geführt hat, die Community an der Entstehung der Texte als kritische Lektoren zu beteiligen. [Vgl. Gillies 2014:176]

Hackers bildet die Vorlage für eine Reihe weiterer historischer Darstellungen zur Entstehung und Entwicklung der Hackerszene. Diese differenzieren sich in Arbeiten mit generalistischem Blick [vgl. Freiberger/Swaine 2000], mit Fokus auf einzelne Szenen, Länder, Technologien [Albertz/Oldenziel 2014] oder Individuen [Mitnick 2011]. Zumeist sind diese Darstellungen populärwissenschaftlicher Natur. Insbesondere im kulturwissenschaftlichen Diskurs ist die Figur des Hackers darüber hinaus wirksam geworden als Protagonist diverser Subkulturen [vgl. Düllo/Liebl 2005] sowie als Prototyp des Autodidakten im hochtechnischen Zeitalter [vgl. Pias 2002b]. Die Diffamierung der Zuschreibung „Hacker“ ab Mitte der 1980er-Jahre [vgl. Stoll 1989] und die von Levy beschriebene „Domestikation“ des Hacking durch Professionalisierung in der Hardware- und Software-Industrie [vgl. auch Wolmeringer 2008:15] haben dazu geführt, dass insbesondere heute Hacker-Geschichte als Ingenieurs-Geschichte und Computergeschichte als „eine Unterdisziplin der Wirtschaftsgeschichte“ [Jordan 2013:136] gewertet wird.

3.1.2 Unternehmer-Geschichte: The Homecomputer Wars

Bereits Levys Text stellt ein Hybrid aus verschiedenen Genres der Geschichtsschreibung dar: Neben der chronologisch strukturierten Geschichte der Computer liefert er Biografien, Firmengeschichten, Kulturgeschichten, Marktgeschichten, und Anekdoten (im Sinne unverbundener Historeme, die ins historische Gefüge integriert sind). Tatsächlich lässt sich auf dem populärwissenschaftlichen Feld keine historische Darstellung finden, die sich ausschließlich in ein Genre einfügen lässt, was zunächst damit zu tun hat, dass Tech-

nologie-Entwicklungen eng an ökonomische, kulturelle und nicht zuletzt individuelle Historeme gekoppelt sind.

Insbesondere Unternehmensgeschichten als Subkategorie der Wirtschaftsgeschichte verknüpfen die Biografien von Firmengründern, Ingenieuren und sogar Kunden [vgl. Wieland 2011:161] mit der historischen Entwicklung eines Einzelunternehmens. Solche Historiografien [vgl. Jordan 2009:134ff.] gehen aus den „Unternehmer-Geschichten, die als Verbindung von Biografie und Firmengeschichte um 1900 zu schreiben begonnen werden“ [ebd.:134], hervor. Schon frühe Darstellungen der Homecomputer-Geschichte greifen auf dieses Narrativ zurück; so etwa Michael S. Tomczyks The Home Computer Wars aus dem Jahr 1984.

Tomczyk, ehemaliger Signal-Corps-Berichterstatter im Vietnam-Krieg, stellt die Geschichte der Homecomputer als Kriegsparabel aus der Innenperspektive der Firma *Commodore* vor, für die er ab 1976 im Marketing-Bereich tätig war. Der zeitliche Rahmen seiner (wie Levys letztes Kaptitel nicht intendierten) Historie beginnt 1976 (kurz nach der Veröffentlichung des PET-2001-Modells) und endet 1984, in welchem *Commodore* das Amiga-Modell akquiriert und der Firmengründer Jack Tramiel (über den Tomczyk fokussiert berichtet) *Commodore* verlässt und die Firma *Atari* übernimmt. Der aspektierte Zeitraum deckt also die komplette frühe Phase der Homecomputer-Industrie ab. Tomczyk berichtet über die Zeit vor seiner Einstellung bei *Commodore* 1980 unter Verweis auf ihm überlieferte Erinnerungen und Anekdoten. Während der Planung des Modells VIC-20 (dt. VC-20), an der er maßgeblich beteiligt war, rekurriert sein Bericht „on my own recollections, impressions, and viewpoint“ [Tomczyk 1984:ix].

Obgleich er sich aufgrund seiner Tätigkeit als Journalist von Beginn an auch als Chronist versteht [vgl. Tomczyk 1984:x], sind seine Darstellungen deutlich individuell gefärbt, enthalten persönliche Wertungen, basieren auf eigenen Interpretationen und – insbesondere bei der Wiedergabe wörtlich zitierter Dialoge und Aussagen – auf Erinnerungen und Rekonstruktionen, die psychischen Prozessen des Erinnerns und Vergessens unterliegen [vgl. Hobi 1988]. Es ist dieser individualistische Zugang zur Geschichte, der The Home Computer Wars zum Prototypen für nachfolgende Darstellungen zu anderen Firmen macht. [Vgl. Goldberg/Vendel 2012, Wozniak 2008, Rodney 1985 u. a.] Unternehmens- und Unternehmer-Geschichten bilden sogar den größten Teil computerhistorischer Publikationen zum Homecomputer-Zeitalter, von denen die Abhandlungen über *Commodore* (aus je unterschiedlicher Perspektive) überwiegen. [Vgl. Bagnall 2011, Tyschtschenko 2014, Coners et al. 2012, Mohr 2007, Kretzinger 2005, Maher 2012 u. a.] Die Komplexität, die sich bereits in diesen Darstellungen zu einzelnen Unternehmen zeigt, ist als der Grund dafür anzunehmen, dass nur wenige globalere Unternehmens- und Wirtschaftsgeschichten des Homecomputer-Zeitalters existieren oder dass jene, die vorliegen, enge Eingrenzungen vornehmen, wie sich am Beispiel der Monografie von Paul Ceruzzi zeigt.

3.1.3 Computergeschichte als Wirtschaftsgeschichte: A History of Modern Computing

„Auch die Geschichtsschreibung zur historischen Entwicklung des Computers und der Informatik hat eine Geschichte“, notiert Hartmut Petzold [Ceruzzi 2003a:14] in seinem Vorwort zur deutschen Übersetzung von Paul Ceruzzis Werk A History of Modern Computing.[7] Er leitet mit dieser Bemerkung zu den Versuchen Ceruzzis über, sein eigenes Werk methodisch in diese Geschichts-Geschichte einzuordnen und führt, indem er Ceruzzis US-Zentrismus und damit einhergehend dessen weitgehende Aussparung der deutschen Entwicklungen auf dem Gebiet kritisiert, vor allem ökonomische Argumente an, bestimmte „Menschen [...,] Firmen [... und] Märkte[]“ [ebd.:14] in seine Chronologie aufzunehmen. Wie sich weiter unten zeigen wird, bedarf die von Ceruzzi absolvierte Wirtschaftsgeschichte jedoch dieser (und anderer) Auslassungen, um historisch-narrative Kohärenz [vgl. Jordan 2013:19] zu erzeugen.

Petzolds Vorwort erscheint nicht allein aufgrund dieser Kritik beinahe schon wie eine Rezension zu Ceruzzis „Sicht auf die Ereignisse“ [Ceruzzi 2003:16], wenn er konstatiert: „Es kann nicht darum gehen, eine einheitliche, unverbindliche Darstellung der Geschichte des Computers zu erstellen, folgten doch die Computerhersteller und die Anwender durchaus unterschiedlichen, sich sogar bekämpfenden Sichtweisen.“ [Ebd.:15f.] Diese Einschränkung trägt bereits Züge einer wohlwollend-relativierenden Lektüre und in der Tat ähneln sich die Sichtweisen Petzolds und Ceruzzis. Ceruzzi stellt sich gleich zu Beginn von A History of Modern Computing dem Problem, seinen Gegenstand überhaupt fassbar zu machen:

> [...] that technology advances along a broad front, not along a linear path, in spite of terms like “milestone” that are often used to describe it. The history of computing presents problems under this systems approach, however. One definition of a modern computer is that it is a system: an arrangement of hardware and software in hierarchical layers. Those who work with the system at one level do not see or care about what is happening at other levels. [Ceruzzi 1998:4]

Daraus resultieren Ceruzzi zufolge die sehr disparaten Ansätze, der Geschichte(n) des Computers: Manche Autoren widmen sich der „social construction“ [ebd.:4], die sich durch das Diffundieren der Computertechnik aus Militär und Forschung in die Gesellschaft vollzieht. Dies sei diejenige Perspektive, die vor allem von technikzentrierten Historikern zumeist *nicht* eingenommen wird. Das andere Extrem stellen Ceruzzi zufolge Historiografien dar, die sich zu stark auf soziologische und ideologische Aspekte der Computergeschichte konzentrieren und dabei „advances in fields such as solid state elec-

7 Referenziert wird die erste Auflage, um die hier untersuchten historischen Abhandlungen chronologisch miteinander vergleichbar zu machen (etwa Ceruzzis Rezeption von Levy 1984 [vgl. Ceruzzi 2003a: 312]). Ceruzzis Buch ist 2003 in einer zweiten überarbeiteten und erweiterten Auflage erschienen. [Ceruzzi 2003b]. Die deutsche Übersetzung enthält zahlreiche orthografische und Übersetzungsfehler, weshalb die englische Originalfassung [Ceruzzi 1998] verwendet wird.

tronics“ [ebd.:5] außer acht lassen. Eine Geschichte des Computers müsse zwischen diesen beiden Polen angesiedelt sein. Ceruzzi benennt „Themes“ [ebd.:5], die er in seinem Ansatz verfolgt.

Gleich zu Beginn schreibt er: „The narrative that follows is chronological“ [Ebd.:5]. Die Qualifizierung als *chronologische Erzählung* ist bedeutsam für interpretatorische Zugänge, weil aus der Form der *Chronologie* [vgl. Jordan 2013:19f.] und dem Genre der *Erzählung* [White 1991a:101ff.] bestimmte Paradigmen historischer Argumentationsweisen hervorgehen. Ceruzzis Darstellung wird dramaturgisch durch „major turning points“ [ebd.] strukturiert, die sich in der Geschichte der Computer von „the late 40s [... to] the spread of networking after 1985“ [ebd.:5f.] *als Fortschritt* vollzogen haben:

> [.. A] half-century of innovation has revealed several patterns on which a structure can be built. These patterns seem to have held fast through successive waves of technology [...] [ebd.:ix.]

Gleichwohl handelt es sich bei A History of Modern Computing nicht um eine reine Chronologie. Der Autor deutet die Abfolge der von ihm ausgewählten historischen Artefakte und insinuiert als Methode dafür die strukturgeschichtliche [vgl. Jordan 2013:103] Rekonstruktion:

> Mark Twain said that historians have to rearrange past events so they make more sense. If so, the invention of the personal computer at a small model-rocket hobby shop in Albuquerque cries out for some creative rearrangement. [Ceruzzi 1998:226]

Die Quellenlage, die seiner Strukturierung unterliegt, ist aufgrund der Tatsache, dass es sich bei A History of Modern Computing um keine akademische sondern populärwissenschaftliche Arbeit handelt, nicht immer transparent. Viele Thesen werden nicht begründet, für andere führt er in Endnoten Quellen, etwa zeitgenössische Newsletter und Zeitschriften [vgl. ebd.:224], an.

Für Ceruzzi spielen weniger die technischen Basismaterialen der Schaltgatter (Relais, Röhren, Transistoren, vgl. ebd.:6] eine Rolle als die Formfaktoren Größe, Preis und damit Marktverfügbarkeit und Verbreitung von Computern. Argumentiert werden die Wendepunkte mit verschiedenen institutionellen und diskursiven Einflüssen auf die Computerentwicklung: das Militär [vgl. ebd.:7], die Forschung [vgl. ebd.:8], das Entstehen von Software als Produkt eines neuen Industriezweigs [vgl. ebd.:9] (hierzu rechnet Ceruzzi auch die Programmiersprachen-Entwicklung, die ihm zufolge von verschiedenen Autoren aber zu stark betont wird [vgl. ebd.:9]) und schließlich die Sozialgeschichte des Computers [vgl. ebd.:9f.].

Wie Petzold in seiner Einführung bemerkt, konzentriert sich Ceruzzi vor allem auf die US-amerikanischen Beiträge zur Computergeschichte und grenzt am Ende seiner Einfüh-

rung (neben dem Thema der Entwicklung künstlicher Intelligenz [vgl. ebd.:10]) andere Herstellungsländer und -regionen (Europa, Japan, die Sowjetunion [vgl. ebd.:10f.]) mit Verweis auf deren wirtschaftlich marginale Bedeutung aus seiner Darstellung aus. Markanterweise gehören auch die Homecomputer zu den aus seiner Betrachtung ausgeklammerten Geräten.

Im letzten Drittel seines Buches beschäftigt sich Ceruzzi mit den Entwicklungen von 1972 bis 1990 und fasst darunter die Jahre 1974 bis 1984 als einen durch zwei „wichtige Wendepunkte“ (siehe oben) definierten Zeitraum:

> 1974 was the annus mirabilis of personal computing. In January, Hewlett-Packard introduced its HP-65 programmable calculator. That summer Intel announced the 8080 microprocessor. In July, Radio-Electronics described the Mark-8. In late December, subscribers to Popular Electronics received their January 1975 issue in the mail, with a prototype of the “Altair” minicomputer [sic!] on the cover [...] and an article describing how readers could obtain one for less than $400. [Ceruzzi 1998:226] The introduction of IBM Compatibles and the Macintosh [1984, S. H.] signaled the end of the pioneering phase of personal computing. [Ebd.:278]

In diesen Zeitraum fällt die Homecomputer-Ära. Mit Ausnahme des Altair 8800 [ebd.:226ff.], welcher „had opened the floodgates“ [ebd. 230], sowie des TRS-80 [263f.], Commodore PET [ebd.:264] und Apple II [ebd.:264ff.] als Vertreter einer „Second Wave“ [ebd.:263] der Personal Computer, führt Ceruzzi aber keine der Geräte oder Firmen der Homecomputer-Zeit an. Anstelle dessen stellt er Implementierungen früher 4- und 8-Bit-Mikroprozessoren in Lehr- und Entwicklungssysteme wie Micral [ebd.:235], Intellec-4 [ebd.:223], Scelbi-8H [ebd.:225] oder den Mark-8 [ebd.] vor und skizziert ihre wirtschaftlichen Bedeutungen (etwa als Konkurrenten zum Altair 8800 [vgl. ebd.:229]).

Als Auslöser der privaten Beschäftigung mit Computern sieht Ceruzzi anders als Levy die zu Beginn der 1970er-Jahre erstarkende Taschenrechner-Industrie. Durch die stetige Verkleinerung der Halbleitertechnik habe sich ein Preisverfall [Ebd.:213] ereignet, der für programmierbare Taschenrechner, wie die Geräte von *Hewlett-Packard*, einen Markt etablierte und sie so immer größeren Nutzerkreisen verfügbar machte [vgl. ebd. 2003:211, 215]. Nicht etwa in den zeitgleich entstehenden Hacker-Szenen, sondern bei professionellen Taschenrechnernutzern habe sich hier zuerst eine Programmierkultur entwickelt: „Groups like the Homebrew Computer Club emphasized the ‚personal‘ in personal computer; calculator users emphasized the word computer.“ [Ebd.:216]

Diesen Nutzern spielte neben der vergünstigten Halbleitertechnik die Adaption benutzerfreundlicher(er) Betriebssysteme nach dem Time-Sharing-Modell zu: Aus dem TOPS-10-System des PDP-10, das Ceruzzi später als Vorbild für Mikrocomputer-Betriebssysteme wie CP/M [238ff.] und MS-DOS [ebd.:269ff.] sieht, entstehe schließlich der „user-friendly“ [ebd.:256, 258] Computer. Die Programmiersprache BASIC (der sich Ceruzzi vorab

ausführlicher zugewandt hatte [vgl. ebd.:232ff.]) bildet in dieser Argumentation lediglich die geschäftliche Grundlage der Firma *Microsoft* auf ihrem Weg zum *IBM*-Vertragspartner. Hier zeigt sich, was für Ceruzzi der maßgebliche Motor der Computerentwicklung war und bleibt: „But the driving force [für die Privatisierung des Computers, S. H.] was not the counterculture vision of a Utopia of shared and free information; it was the force of the marketplace." [Ebd.:236] Die hier alludierte Bedeutung der Hacker-Kultur wird von ihm auch an anderer Stelle kritisch bewertet [vgl. ebd.:264].

Ceruzzis Computergeschichte zeigt sich damit vornehmlich als eine *Geschichte wirtschaftlichen Erfolgs*. Zeitgenössische Rezensenten monieren diese Perspektive. So kritisiert Michael S. Mahoney in seiner Rezension die teleologische Sichtweise Ceruzzis, in der er die Geschichte des Computers auf einen Punkt (das Internet) zulaufen lässt und bestimmte Historeme dabei ex post daraufhin bewertet, ob sie dieser Entwicklungslinie entsprechen oder nicht:

> This way of looking at things tends to lose sight of the variety of futures that are possible at any given time, and indeed of the diversity of the present that actually emerges. The „series of 'computer ages'" is a branching, not a succession. [...] [B]y treating the Internet as the culmination of modern computing, Ceruzzi leaves the reader unaware that only a fraction of the computing performed today is visible. Left unseen are the myriads of computers at work within the appliances, vehicles, structures, and systems that are the tangible expression of an information society. [Mahoney 2000:94]

Die *teleologische Agenda* – insbesondere der Mikrocomputer-Geschichte [vgl. Mahoney 2000:94] – macht eine Filterung von Quellen, Topoi und berücksichtigten Historemen notwendig, wie Ceruzzi in einem weiteren Verweis auf seinen methodologischen Vordenker Mark Twain erklärt: „When trying to describe those years, from 1972 through 1977, one is reminded of Mark Twain's words: ‚Very few things happen at the right time, and the rest do not happen at all. The conscientious historian will correct these defects.'" [Ceruzzi 1998:207]

3.1.4 Interaktive Software-Geschichte(n)

Zwei in kurzem Abstand zueinander erschienene alternative Geschichtsschreibungen, die bereits operativen Charakter besitzen, sollen im folgenden ergänzt werden: Coding for Fun [Wolmeringer 2008] und Computergeschichte(n) – Nicht nur für Geeks [Wieland 2011]. Bei beiden handelt es sich um historische Darstellungen, die versuchen durch (Nach-)Programmierung einschlägiger Algorithmen in unterschiedlichen Programmiersprachen Geschichtswissen interaktiv zu vermitteln. Während Wolmeringer dabei einen stärkeren Akzent auf die Software-Geschichte (Betriebssysteme, Computerspiele, KI, Fraktale, Pro-

grammiersprachen) legt, finden sich bei Wieland Hardware und Software quantitativ etwa gleichrangig behandelt.

Wie bei den vorherigen historischen Darstellungen, wird auch bei Wieland der methodische Ansatz in einem Vorwort nur in Ansätzen ausformuliert: „Computer haben wie alle Dinge ihre ganz persönliche Geschichte [...]. Wir wollen versuchen, die Spuren, die der Computer in der Zeit hinterlassen hat, praktisch zu lesen." [Wieland 2011:15] Seine Vorgehensweise ist dabei „chronologisch" [ebd.:16]. Die Tatsache, dass die einzelnen Kapitel und Unterkapitel gar nicht oder nur selten kohärent zueinander sind, rückt Wielands Darstellung in den Bereich der *Ereignisgeschichte*, die ihre Narration nicht nach übergeordneten Diskursen strukturiert (Wirtschaft, gesellschaftliche Entwicklung, kultureller Einfluss etc.), sondern nach Einzelentwicklungen des aspektierten Bereichs [vgl. König 2009:40]. Hier zeigen sich aber auch bereits stilistische Ähnlichkeiten zu den *technical reports* (siehe Kapitel 3.2.3).

Homecomputer nehmen in Wielands Darstellung als Plattformen eine nebengeordnete Stellung innerhalb der Computergeschichte ein, da sie „nur ein kurzes Intermezzo in der Geschichte der EDV dar[stellen]" [Wieland 2011:147]. Dennoch zeigt sich in seinem etwa 40-seitigem Kapitel zu diesen Geräten die autobiografische Beziehung des Autors zum Gesamtthema (etwa im Unterkapitel „Mein Spectravideo" [ebd.:161]). Das Aufkommen der Homecomputer sieht Wieland als Besetzen einer „Marktlücke": „[D]er PC aus dem Büro wie der Apple II für den betuchten Enthusiasten waren für den Hausgebrauch viel zu teuer. In diese Marktlücke drang nach und nach eine ganze Reihe von Anbietern mit klingenden Namen." [Ebd.:161] Damit grenzt Wieland den Zeitraum der Homecomputer implizit auf die Jahre 1977 (der Commodore PET war ihm zufolge ein Vorläufer [vgl. ebd.:171]) bis 1992 ein (als letztes Gerät stellt er den Acorn Archimedes 3020 vor [vgl. ebd.:185]).

Bemerkenswert ist, dass Wieland (wie auch Wolmeringer) *Computerspielen* eine exponierte Bedeutung für die gesamte Geschichte der Computer zuschreibt. Sie sind die „wichtigsten Programme überhaupt" [Lektorin zit. n. Wieland 2011:Schmutztitel]. In einem dedizierten Kapitel zu dieser Software-Gattung werden insbesondere historische Spiele zur Nachprogrammierung vorgestellt. [Vgl. Wieland 2011:355-362] Wolmeringer, der Homecomputern einen noch kleineren Bereich in seiner Monografie einräumt [vgl. Wolmeringer 2008:114-119), sieht sie „als Schrittmacher"[8] [ebd.:197] des Computerspielsektors. Die Bedeutung von Computerspielen für die Geschichte der Computer sprechen auch die zuvor genannten Werke an [vgl. Levy 1984:277ff.; Ceruzzi 1998:207, 210, 230, 278; Tomczyk 1984:179].

8 Einige Textpassagen aus Wolmeringers Buch finden sich in Wielands Monografie wortwörtlich wieder, ohne dass sie mit einer Quellenangabe referenziert sind, so dass unklar bleibt, wem die Aussagen ursprünglich zuzuordnen sind. [vgl. Wolmeringer 2008: 114f. und Wieland 161f.]

Den wesentlichen Unterschied zu anderen Computer-Geschichten stellen allerdings die interaktiven Teile der Bücher von Wieland und Wolmeringer dar, die sich auch in deren Homecomputer-Kapiteln finden. Bei den Programmierbeispielen (in Assembler und BASIC [vgl. Wieland 2011:157,166f.,175f.; Wolmeringer:116]) wird der Leser allerdings nicht etwa aufgerufen, sich mit der Hardware der behandelten Plattformen auseinanderzusetzen, sondern sich diesen über Emulatoren zu nähern. Dies stellt zwar eine kostengünstige Alternative zur Anschaffung eines historischen Computers dar, führt jedoch dazu, dass der operative Zugang zum Homecomputer symbolisch bleibt, weil Emulatoren bloße Abbildungen von für eine Reihe von Anwendungen spezifisch ausgewählten Hardwareverhalten in Software darstellen. Die Idiosynkrasien eines konkreten Computers, seine vielfältigen Zeitverhalten [vgl. Höltgen 2016a] und insbesondere die *hidden features* von Hardwareplattformen, die im Retrocomputing exploriert werden, sind über Emulatoren kaum zugänglich. Im Sinne einer Geschichte, die Computer als historische Artefakte ansieht, erweist sich ein solches Surrogat zudem als inkonsequent.

Dies wirft letztlich die Frage auf, mit welcher Intention die Autoren ihre „interaktiven Computergeschichte[n]“ verfasst haben. Sieht Wieland seinen Text als „Geschichtsbuch“ [16] mit interaktiven Elementen, so ist Wolmeringers Motiv, eine IT-Geschichte zum Nachprogrammieren zu schreiben, *programmierdidaktischer* Natur, wenn er moniert:

> [...] dass aus der kleinen Wissenschaft der Hacker im Hinterzimmer mehr und mehr ein große Wissenschaft des Softwareingenieurs wurde. Zweifellos eine notwendige Entwicklung, aber der Spaß an der Sache blieb dabei auf der Stecke. Aus diesem Grund fanden sich immer weniger junge Leute, die dieser ernsten Wissenschaft nachgehen wollten. Für mich stand fest: Es wurde allerhöchste Zeit, die alten Disketten hervorzukramen, auf denen es Programme gab, die Apfelmännchen zeichneten oder Conways Spiel des Lebens spielten. Zeit, dass der Spaß am Programmieren in den Computer zurückfand. So entstand die Idee zu diesem Buch – und in diesem Geist wurde es auch geschrieben. [Wolmeringer 2018:15]

Dieser spielerische Zugang zum Computer, der typisch für das Praktiken des Hacking ist, soll im Kapitel 5 durch die Gegenüberstellung curricularer und privater didaktischer Modelle und Methoden eingehender diskutiert werden.

3.2 Geschichtskritik

Eine der Fragen, die sich aus den unterschiedlichen Historiografien der frühen Mikrocomputer ableitet, ist, wie es zu derartig unterschiedlichen, manchmal (wie im Falle von Ceruzzis und Levys Interpretation über die Genese des Personal Computers) sogar diametral entgegengesetzten Sichtweisen auf ein und denselben Gegenstand kommen kann. Die Antwort ist eng gekoppelt an die Darstellungsmodi von Historiografie: ihre diskursive

Verfasstheit, schrift-sprachliche Fixierung und die jeweilige Interpretation des Historikers.

Nachfolgend sollen diese Darstellungsmodi untersucht werden, um so die offenkundigen Widersprüche unterschiedlicher historischer Darstellungen zu erklären und um die besondere Problematik von Geschichtsdarstellungen der Technik und insbesondere der Computergeschichtsschreibung zu markieren. Über die poetologische Geschichtskritik, die Diskursarchäologie und die Medienarchäologie soll die *Kritik der existierenden Geschichtsschreibung* in die Methode einer adäquaten Darstellung der Computergeschichte münden, die hier unter dem Begriff *Computerarchäologie* vorgestellt wird.

3.2.1 Die Poetologie der Computergeschichte(n)

Außer Ceruzzi ist keiner der Autoren der oben behandelten Texte akademischer Historiker. Wieland und Wolmeringer sind Diplom-Informatiker, Tomczyk und Levy Technikjournalisten. Damit ist das Verhältnis akademischer Historiker zu nicht wissenschaftlichen Hobbyhistorikern (so genannten „Barfußhistorikern“ [vgl. Tanner 2004:79]) tatsächlich sogar noch nicht einmal repräsentativ, bilden erstere doch in der Geschichtsschreibung der frühen Digitalcomputer die absolute Ausnahme. Die fehlende oder stark marginalisierte Explikation methodischer Paradigmen begründet sich wohl vor allem darin, dass diese den Autoren eben gar nicht bekannt zu sein scheint. Anstelle ihrer argumentieren sie mit Hilfe literarischer und künstlerischer Methoden, wenn beispielsweise ein Literat wie Marc Twain als Geschichtsmethodologe [vgl. Ceruzzi 1998:207, 226] rezipiert oder die Entwicklung des VC-20 mit den Handlungen auf einem Schlachtfeld parallelisiert [vgl. Tomczyk 1984:168] werden.

Hierin unterschiedet sich die Computergeschichtsschreibung allerdings kaum von der anderer Gegenstände. Historiografie präsentiert sich oftmals nicht als akademische Praxis, sondern nimmt laut dem Geschichtstheoretiker Hayden White einen Zwischenplatz zwischen „eine[r] Art von Kunst [... und einer ...] *Quasi*-Wissenschaft“ [White 1991a:36 – Hervorh. i. O.] ein. White analysiert unter dem Programm der *Metahistory* Geschichtsschreibung daraufhin, wie diese historische Fakten zu Narrationen konstruiert. Die Vorgehensweise ist White zufolge vor allem poetologischer Natur:

> [...] Geschichtswerke [beziehen] einen Teil ihrer Erklärungswirkung (explanatory effect) daraus, daß es ihnen gelingt, aus bloßen Chroniken Geschichten (stories) zu machen; und Geschichten (stories) werden ihrerseits aus Chroniken mithilfe eines Verfahrens gemacht, das ich an anderer Stelle als „emplotment“ (Verleihung einer Plotstruktur) bezeichnet habe. Unter „emplotment“ verstehe ich einfach die Kodierung der in der Chronik enthaltenen Fakten als Bestandteile bestimmter Arten von Plotstrukturen, in eben der Weise, wie es Frye für die „Fiktionen“ allgemein behauptet hat. [White 1991a:103]

Trotz dieser zunächst als methodische Kritik anmutenden Darstellung sieht White den Status von Geschichtsschreibung keineswegs als „Ideologie oder Propaganda“ [ebd.:121]; vielmehr besitzt er ihm zufolge die Aufgabe, jene „ursprüngliche Fremdheit, das Geheimnisvolle oder Exotische der [bloßen, S.H.] Ereignisse“ [ebd.:107] aufzuheben und sie durch das emplotment für den Rezipienten „verstehbar“ [ebd.:107] zu machen. Hierzu bedarf es dedizierter Praktiken im Umgang mit den historischen Fakten. Bei diesem Prozess wird die

> Menge von zufällig überlieferten Ereignissen [...] zu einer Geschichte gemacht durch das Weglassen oder die Unterordnung bestimmter Ereignisse und die Hervorhebung anderer, durch Beschreibung, motivische Wiederholung, Wechsel in Ton und Perspektive, durch alternative Beschreibungsverfahren und ähnlichem – kurz mit Hilfe all der Verfahren, die wir normalerweise beim Aufbau einer Plotstruktur eines Romans oder eines Dramas erwarten. [ebd.:104]

Die Konstruktion des Plots orientiert sich dabei sowohl an den idealistischen Einstellungen des Autors als auch an den antizipierten Erwartungen des Lesers [vgl. ebd.:105]. Das Ziel dieses Konstruktionsverfahrens ist die Stiftung von Kohärenz zwischen den noch unverbundenen Fakten: „Der ‚Gesamtzusammenhang‘ irgendeiner gegebenen ‚Serie‘ von historischen Fakten ist die Kohärenz einer Geschichte (story), doch die Kohärenz wird nur dadurch erreicht, daß die ‚Fakten‘ auf die Erfordernisse der Geschichtenform (story form) zugeschnitten werden.“ [Ebd.:112] Der Historiker fungiert in diesem Prozess aber nicht allein als Ordner, sondern auch als ‚Filter‘:

> Unsere Erklärungen historischer Strukturen und Prozesse sind daher mehr von dem bestimmt, was wir in unseren Darstellungen weglassen, als von dem, was wir hineinnehmen. Denn gerade in dieser skrupellosen Fähigkeit, bestimmte Fakten auszuschließen, um andere zu Bestandteilen verstehbarer Geschichten zu machen, gerade darin zeigt der Historiker sein Gespür wie auch sein Verständnis. Der „Gesamtzusammenhang“ irgendeiner gegebenen „Serie“ von historischen Fakten ist die Kohärenz einer Geschichte (story), doch die Kohärenz wird nur dadurch erreicht, daß die „Fakten“ auf die Erfordernisse der Geschichtenform (story form) zugeschnitten werden. [White 1991b:112]

Das ‚Genre‘ der auf diese Weise konstruierten Historienerzählung hängt White zufolge vor allem von der Struktur der in ihr narrativ verknüpften Ereignisse ab [vgl. ebd.:114ff.] Je nachdem, *welchem Ereignis (oder welcher Ereignisfolge)* ein „privilegierte[r] Status“ [ebd.:114] vom Autor zugeschrieben wird und *an welcher Stelle* der Erzählung dieses Ereignis aufgeführt wird, erhält sie den Status eines *historischen Determinismus* (privilegiertes Ereignis zu Beginn) oder einer *Eschatologie* (privilegiertes Ereignis am Ende), woraus sich das zentrale Erklärungsmodell der jeweiligen Darstellung ergibt.

Die Konsequenz bei der Bewertung von Geschichtsschreibung ist, dass zwar einzelne historische Fakten den Status falsifizierbarer Aussagen besitzen, die aus ihnen verknüpfte Narrationen jedoch „nicht definitiv widerlegt werden können“ [ebd.:120]. Je nach individueller Sichtweise (politische Einstellung, Zugehörigkeit zu einer Historikerschule [Jordan 1991:38], hauptberufliche Herkunft u. a.) des Autors, seinen angenommenen Lesererwartungen und nicht zuletzt der Qualität und Quantität der ihm vorliegenden historischen Fakten kann die Darstellung ein und desselben historischen Prozesses also vollkommen unterschiedlich zu den Beschreibungen desselben Prozesses bei anderen Historikern ausfallen.

Da die Konstruktionsprinzipien und -methoden den Historikern (insbesondere den nichtakademischen) nicht immer vollständig explizit sind, schlägt White mit seinem Programm der *Metahistory* sowohl eine Analyse des historischen Textes als literarisches Kunstwerk als auch eine Möglichkeit für den Historiker zur Selbstvergewisserung der eigenen Arbeit vor:

> Um die Geschichte irgendeiner Forschungsdisziplin oder gar einer Naturwissenschaft zu schreiben, muß man bereit sein, Fragen über sie zu stellen, wie sie sich nicht in ihrer Praxis stellen. Man muß hinter oder unter die Voraussetzungen zu kommen versuchen, die Grundlage für eine bestimmte Forschungsrichtung sind, und muß diejenigen Fragen stellen, denen man in der Praxis ausweichen kann, um herauszufinden, warum diese Forschungsrichtung für die Lösung eben der Probleme entwickelt wurde, die sie charakteristischerweise zu lösen versucht. Das ist das Ziel der Metahistorie. Sie stellt sich solche Fragen wie: Welcher Art ist die Struktur eines spezifisch historischen Bewußtseins? Welcher Art ist der erkenntnistheoretische Status historischer Erklärungen im Vergleich zu anderen Arten von Erklärungen, die für das historische Material, mit dem sich Historiker für gewöhnlich befassen, gegeben werden könnten? Welches sind die möglichen Formen historischer Darstellung und welches ihre Grundlagen? Welche Geltung können historische Darstellungen als Beitrag zu einem gesicherten Wissen von Realität überhaupt und zu den Humanwissenschaften im Besonderen für sich behaupten? [White 1991b:101]

Dieses Programm ist in der Lage dem Historiker wie auch seinem Leser zu erklären, welche poetologischen Verfahren an der Historiografie ‚mitschreiben‘. Sie klärt jedoch noch nicht, welche Machtgefüge (Dispositive) auf die Geschichtsschreibung – vom Sammeln und Auswählen der historischen Fakten und Artefakte bis hin zum Emplotment – Einfluss auf die Historiografie nehmen. Diese Machtgefüge wirken aber stets (explizit wie implizit) bei der Entstehung von Historiografien mit und sind Ende der 1960er-Jahre Gegenstand der so genannten *Diskursarchäologie* des französischen Philosophen Michel Foucaults geworden.

3.2.2 Foucaults Archäologie der Diskurse

In einem Interview von 1968 skizziert Michel Foucault die Form und Methoden, die bis dahin die Geschichtsschreibung dominiert haben. In ihr hätten sich subtile Ideen von *Sukzession* und *Kontinuität* eingeschrieben [vgl. Foucault 2001a:892], die zugleich das Selbstverständnis des Historikers als eines Diskurskonstrukteurs der „großen ununterbrochenen Einheiten [aus dem] Gewimmel der Diskontinuitäten“ [ebd.:889f.] untermauern.

Dieses Selbstverständnis gelte es Foucault zufolge zunächst zu durchbrechen, was jedoch zu Gegenwehr führe:

> Um jedoch diese Litanei von Einwänden anzustimmen, muss man den Blick der Arbeit der Historiker umkehren, muss sich weigern, zu sehen, was gegenwärtig in ihrer Praxis und in ihrem Diskurs stattfindet [...]. Man schreit folglich jedes Mal Mord an der Geschichte, wenn in einer historischen Analyse [...] der Gebrauch der Diskontinuität allzu manifest wird. Man darf sich darin jedoch nicht täuschen: Was man so stark beweint, ist nicht die Auslöschung der Geschichte, es ist das Verschwinden derjenigen Form von Geschichte, die insgeheim, aber vollständig, auf die synthetische Aktivität des Subjekts bezogen war. [Ebd.:892.]

Was White als poetische Tätigkeit identifiziert hatte, wird hier von Foucault deutlich technischer als „synthetische Aktivität“ bezeichnet. Und anstelle es – wie White – bei der bloßen Aufklärung von Historikern und ihren Lesern über den poetologischen Charakter der Geschichtsschreibung zu belassen, fordert Foucault die „negative Aufgabe“ [ebd.:893] einer Begriffskritik an den Leitfiguren der Geschichtsschreibung: Kontinuität, Tradition, Einfluss, Entwicklung, Telos, Mentalität sowie der Sinnhaftigkeit und Übertragbarkeit historischer Phänomene auf andere:

> Man muss sich von einem ganzen Komplex von Begriffen lösen, die mit dem Postulat der *Kontinuität* verknüpft sind. Sie haben zweifellos keine sehr strenge begriffliche Struktur, ihre Funktion jedoch ist sehr präzise. So der Begriff der *Tradition*, der es möglich macht, gleichzeitig alles Neue innerhalb eines konstanten Koordinatensystems zu erfassen und einer Gesamtheit stetiger Phänomene einen Platz zuzuweisen. So der Begriff des *Einflusses*, der den Phänomenen der Übertragung und der Kommunikation – eher magisch als substantiell – als Träger dient. So der Begriff der *Entwicklung*, der es gestattet, eine Folge von Ereignissen als Manifestation ein und desselben Organisationsprinzips zu beschreiben. So der symmetrische und invasive Begriff der *Teleologie* oder der Evolution hin zu einem normativen Stadium. So auch die Begriffe der *Mentalität* oder des *Geistes einer Epoche*, die die Feststellung einer *Sinngemeinsamkeit*, *symbolischer Verbindungen, eines Spiels der Ähnlichkeiten und der Spiegelungen zwischen gleichzeitigen oder sukzessiven Phänomenen einer Epoche* gestatten. Man muss diese stereotypen Synthesen aufgeben,

> diese Gruppierungen, die man vor jeder Überprüfung zulässt, diese Verbindungen, deren Gültigkeit von vornherein zugestanden wird; man muss die dunklen Formen und Kräfte verscheuchen, durch die man gewöhnlich das Denken und die Diskurse der Menschen miteinander verbindet; man muss akzeptieren, dass man es in erster Instanz nur mit einer Menge verstreuter Ereignisse zu tun hat. [ebd.:893f. - Hervorh. S. H.]

Die von Foucault hier noch diffus als „dunkle Formen und Kräfte" der Diskurskonstruktion bezeichneten Methoden erhalten in seinem ein Jahr später publizierten systematischen Programm einer *Archäologie des Wissens* [Foucault 1981] die Form eines positiv formulierten Forschungsprogramms: „[Archäologie ..] ist nicht mehr und nicht weniger als erneute Schreibung: das heißt in der aufrecht erhaltenen Form der Äußerlichkeiten eine regulierte Transformation dessen, was bereits geschrieben worden ist. Das ist [...] die systematische Beschreibung eines Diskurses als Objekt." [ebd.:200] Foucault fordert einen „Bruch mit der traditionellen Geschichtsschreibung [..] hin zu einer Form von Analyse, welche die Regeln, Strukturen und Prozesse erfasst, die das jeweils historische Wissen konturieren." [Zimmermann 2010:26] Damit liefert seine Geschichtskritik gleichzeitig die Grundlage zu einer *Epistemologie* des (historischen) Wissens jenseits eines bloß ideengeschichtlichen [vgl. Jordan 2013:56] Verständnisses dieses Begriffs, das die historisch kontingenten Regelsysteme des Sagbaren beschreibt [vgl. Zimmermann 2010:26]:

> Nun ist aber die archäologische Beschreibung gerade die Preisgabe der Ideengeschichte die systematische Zurückweisung ihrer Postulate und Prozeduren, der Versuch, eine ganz andere Geschichte dessen zu schreiben, was die Menschen gesagt haben. [Foucault 1981:197]

Foucault liefert mit seiner Archäologie noch keine definierte Methode, sondern eher eine „Werkzeugkiste" [Foucault 1976:56] zur Erforschung historischer Diskurse. Der Archäologe sammelt gleich dem klassischen Archäologen „Monumente" [vgl. Foucault 1981:198] anstelle von Dokumenten, womit Foucault gleich darauf hinweist, dass es der Archäologie nicht um die Deutung solcher Diskurse geht: In einer archäologischen Aufstellung möglichst aller Aussagen wird weder Kausalität noch Kontinuität produziert [vgl. ebd.] und ihre Intention ist nicht auf eine Erklärung der conditio humana (durch anthropologische, psychologische oder soziologische Argumente) gerichtet [vgl. ebd.:199]. Archäologie verfolgt vielmehr gar keine Intention in ihrer Suche [vgl. ebd.:199f.], sondern versucht den „Typ von Positivität eines Diskurses zu definieren" [ebd. 182], um danach die Regeln, nach denen seine Aussagen strukturiert sind, zu finden. Im Ergebnis entsteht so eine Neustrukturierung (diskursiven) historischen Wissens.

Praktisch würde sich diese Form der Geschichtskritik als eine Suche nach möglichst vielen/allen historischen Abhandlungen, Dokumenten, Aussagen usw. (zum Beispiel zu frühen Mikrocomputern) darstellen, in der durch ihre Gegenüberstellung gezeigt werden könnte, welche impliziten Ordnungsstrukturen zuerst hinter der Archivierung dieser

„Diskurs-Monumente“ standen, um dann in deren praktische Auswertung innerhalb einer Diskursanalyse zu münden. Dieses diskursanalytische Programm ist in der Folge Foucaults methodisch ausdifferenziert, vielfältig dargestellt und angewendet worden.[9]

Man könnte die oben vorgestellten Historiografien zum frühen Mikrocomputer solch einer Diskursanalyse unterziehen, sie mit weiterem Archivmaterial anreichern und kontrastieren und erhielte dabei eine ziemlich genaue Analyse jener Dispositive, die die Computergeschichtsschreibung bestimmen. Dabei wäre man jedoch ausschließlich auf *Diskurse über Computer* beschränkt, denn Foucaults Archäologie operiert allein im Bereich des *Sagbaren* [vgl. Ernst 2004] – sei dieses nun mündlich tradiert oder schriftlich fixiert. Der Grund dafür ist, dass die Archive für den Historiker augenscheinlich vor allem sprachliche Monumente sammeln und verwalten.[10] Die Materialität des Archivs, angefangen bei den Gebäuden bis hin zu den Mikromaterialitäten von Papier und Tinte, ist allerdings kein Gegenstand dieser Diskursanalysen. Diese Reduktion der Diskursarchäologie auf die *Medieninhalte* beschränkt nun zwar das Quellenmaterial einer Diskursanalyse; warum solche non-diskursiven Faktoren der Wissensproduktion, -distribution und -speicherung von Foucault ignoriert werden, lässt sich aus ihr jedoch nicht ableiten.

3.2.3 Medienarchäologie des Non-Diskursiven

Eine Analyse der Materialität ‚unterhalb‘ der Diskurse, also der *Medientechnologien*, durch die diese gespeichert, übertragen und verarbeitet werden, scheint notwendig, wenn man berücksichtigt, wie stark sich diese Materialitäten in die Inhalte einschreiben: Die Schrift auf dem Papier und das Gemälde im Rahmen müssen statisch bleiben, die Übertragung im Radio frequenzbeschränkt, rein akustisch und ephemer, das Fernsehbild farbraumreduziert und (in Zeilen oder Pixel) aufgelöst. Die Inhalte, die diese Medientechnologien transportieren, müssen sich also an die technischen Beschränkungen ihrer materiellen Träger und Kanäle anpassen und dabei notwendigerweise für diese verändert (formatiert) werden. Eine alleinige Konzentration auf die von Medien übertragenen und gespeicherten Diskurse erscheint daher defizitär – insbesondere im Übergang vom Buchzeitalter in das Zeitalter elektronischer Medien. Diese Beobachtung erscheint zunächst trivial, sie wird jedoch brisant, weil die medientechnischen Rahmenbedingungen bereits bei der Produktion von Medieninhalten mitgedacht werden (müssen) und damit nur dasjenige in den Diskurs gelangt, was den technischen Kanal passieren kann.

9 Siehe hierzu z.B. die Diskursanalysen des *Duisburger Instituts für Sprach- und Sozialforschung* (DISS) [z. B. Jäger 2012].

10 An dieser Stelle sei auf einen aktuellen Wandel im Selbstverständnis von Archiven verwiesen, wie er gerade am Literaturarchiv in Marbach zu beobachten ist. Dort wird das Werk des Kulturwissenschaftlers Friedrich Kittler archiviert, das neben seinen Texten auch Schaltpläne, Computerprogramme und andere nicht-diskursive Artefakte enthält. Diese Veränderung archivarischer Praxis wird parallel zur Erfassung des Kittler'schen Nachlasses durch die Kultur- und Medienwissenschaft analysiert und reflektiert. [Vgl. Holl 2017; Enge 2017.]

In seinem Beitrag über Foucault und die Medien widmet sich Wolfgang Ernst [2004] der Frage, wie der Übergang von einer Archäologie der Diskurse zu einer Archäologie ihrer Medien vollzogen werden kann. Foucault selbst sei im Dispositiv des Archivs, dem Zeitalter der Schrift und dem System der Literatur und der Diskurse verhaftet geblieben und für Fragen zur materiellen Grundlage der Diskurse (den Medien) „bemerkenswert blind gewesen [...] zumindest auf einem Auge", konstatiert Ernst [2004:239,258]. In seinem Programm einer Berliner Medienwissenschaft erweitert er deshalb das Programm der Diskursarchäologie um eben jene Analyse der medialen Träger des Diskurses hin zu einer *Medienarchäologie*:

> Medien archäologisch zu wissen bleibt sein [Foucaults, S.H.] Denkauftrag an uns. So meint Medienarchäologie die Beschreibung von Diskursen auf dem Niveau ihrer apparativen oder logischen Existenz, insofern sie Funktionen medienarchivischer Elemente sind. Foucaults historische Analysen brechen zumeist mit dem 19. Jahrhundert ab. Dieser Abbruch ist implizit durch eine medienarchäologische Grenze definiert: die Emergenz technischer Medien wie Photographie, Phonograph, respektive Grammophon, Kinematograph, schließlich Radio und Fernsehen. Wenn Probleme auch auf physikalischer Ebene liegen, versagt der an literarischen oder philosophischen Texten erprobte Diskurs buchstäblich. Hier wird das Gesetz des Sagbaren auf besondere Weise berührt. [...] Durch Medienarchäologie wird [..] auf einen Raum verwiesen, der non-diskursiv verfaßt ist. Mediendispositive lassen sich als Ausprägungen des historischen Apriori auffassen. [...] Hier Aufklärung zu schaffen, ist die Aufgabe der Medienarchäologie, die Schaltpläne aufdeckt, d. h. zur Entzifferung gibt. Hinter der medialen Oberfläche stehen keine Geheimnisse, sondern schlichte Algorithmen und Maschinenbauteile – man muß sie nur zu lesen wissen. [Ernst 2004:240f.]

Die von Ernst geforderte „Transgression einer Wissens- zur Medienarchäologie, die [...] die konstitutive Leistung technischer Apparaturen der Speicherung, Übertragung und Berechnung von Daten beschreibt" [Ernst 2004:243], orientiert sich eng am kritischen Gestus der Foucault'schen Diskursarchäologie: Wiederum geht es darum die Dispositive aufzuzeigen, die zu einer bestimmten Historiografie geführt haben, und durch Flankierung existierenden Wissens mit verschwiegenen oder schlicht unsichtbarem Wissen auf Diskontinuitäten und Brüche hinzuweisen. Der entscheidende Unterschied ist hier jedoch, dass dieses neue Wissen ein *techno-mathematisch fundiertes* ist.

3.2.3.1 Beispiel: Rechenmaschinen

Der geschichtskritische Ansatz der Medienarchäologie lässt sich am Beispiel der Geschichte der Rechenmaschinen nachvollziehen. Deren Historiografie setzt zum einen an unterschiedlichen Technologien/Zeiten an, in denen technische Verfahren zur *Automatisierung diskreter Rechenvorgänge* entstehen und sich entwickeln. Die Probleme des Addie-

rens und des Dezimalübertrags bilden dabei häufig die zentralen Motive solcher Darstellungen; teilweise werden deren Lösungen sogar als Bedingung für „praktisch brauchbare[] Rechenapparat[e]“ [Lenz 1924:5] und den Beginn automatischen Rechnens definiert [vgl. Lippe 2013:113]. Insofern sieht die Rechenmaschinen-Geschichtsschreibung einen Umbruch von rein manuellen Hilfen für digitales Rechnen (wie dem Abakus [vgl. Lippe 2013:97ff.]) zu den ersten automatischen Addierern seit Wilhelm Schickard (1623). Die weitere Ausdifferenzierung der automatischen Zehnerübertragstechniken stellt ab dann das zentrale Motiv der Historiografie dar.

Ein zweites, paralleles Motiv verfolgt die *symbolische Arithmetik, die Genese des Ziffernsystems und der Zahlensysteme.* Hier steht die Formulierung der Aussagenlogik durch Aristoteles (im vierten Jahrhundert vor unserer Zeitrechnung) am Beginn [vgl. Höltgen 2017b:19f.; Bauer 2009:4]. Diese erfährt ebenfalls im Barock, mit der Einführung des „dyadisches System“ [vgl. Zacher 1973] durch Leibniz (1697, [vgl. Leibniz 1697]), einen Umbruch. Auf Basis dieses Zahlensystems entsteht durch George Boole [2001] eine aussagenlogische Algebra, die später (theoretisch) durch Claude Shannon [1938] und zeitgleich (praktisch) durch Konrad Zuse [vgl. Rojas 1997] die Grundlage logischer Rechenwerke wird. Mit Zuses Implementierung erfährt zugleich eine Idee Leibniz‘ ihre Konkretisierung: Bereits 1679 hatte er den Entwurf für die *Machina arithmeticae dyadicae*[11] vorgenommen, welche auf Basis des dualen Zahlensystems arbeitet. Die Formulierung mathematischer Algorithmen durch symbolische Systeme erfährt mit den Vorschlägen Ada Lovelaces [Hammerman/Russel 2015:30] eine weitere Eskalation, die hin zu turingvollständigen Hochsprachen für Computer führt.

Aber bereits mit der Symbolifizierung von Logik, die diesen Sprachen zugrunde liegt, wird der Universalrechner möglich. Mit Zuses Implementierung des Logik-basierten Mikroprogramms laufen daher beide Geschichten, die der mechanischen Rechenmaschine und die der symbolischen „Rechenvor*schriften*“ [Zuse 1975:11-15, Hervorh.: S. H.] zu einer Geschichte der Computer zusammen. Die meisten historischen Darstellungen zu Computern setzten an diesem Punkt ein und betrachten die mechanischen Rechenmaschinen als „bedeutenden Schritt in Richtung“ [Hoffmann 2007:13] und die Aussagenlogik als „Grundlage“ [Lippe 2013:123] der Digitalcomputer(geschichte). Dieses nach H. Whites o. g. Kategorisierung *eschatologische Geschichtsnarrativ* erfordert allerdings die Filterung alternativer Rechenmaschinen-Technologien (die zeitgleiche Entwicklungen *kontinuierlich operierender* Analog-Rechenmaschinen) aus ihrem Diskurs und ignoriert ebenso *a-chronologische Rückgriffe* und technologische *Brüche*, die den Eindruck von technologischer Kontinuität und technischer Progression abschwächen könnten.

Einer der bedeutsamen kleineren Schritte der Symbolifizierung des automatischen Rechnens ist die Einführung der *Mikroprogrammierung* durch M. Wilkes [1951], bei der die seit Zuse praktizierte Festverdrahtung logischer Gatter des Mikroprogramms als symbolische

11 http://dokumente.leibnizcentral.de/index.php?id=95 [letzter Abruf: 08.12.2016].

Mikrooperationen codiert werden, um so zur Grundlage vielfältiger arithmetischer, logischer und funktionaler Opcodes des Rechenwerks zusammengestellt werden können. Dieses Verfahren erfährt bis in die Gegenwart Anwendung und Weiterentwicklung (bei der Entwicklung von Funktionsspeichern [vgl. Schildt et al. 2005:64-67; Hoffmann 2007:303-308]). Mit der Einführung des Mikroprozessors im Jahre 1971 findet diesbezüglich jedoch ein ‚Rückschritt' zu festverdrahteten Rechenwerken statt, der erst mit der Veröffentlichung des mikroprogrammierbaren Mikroprozessors Zilog Z180 (1985) überwunden wird. Die Frühgeschichte des Mikrocomputers und des Mikroprozessors ist diesbezüglich epistemologisch betrachtet also ein Rücksprung auf protosymbolische Logikschaltgatter. (Vgl. Abb. 3.1 und 3.2. Eine ähnliche Zeitfigur findet sich bei der Hardware-Implementierung der Gleitkomma-Arithmetik.)

Abb. 3.1 & 3.2: Illustrationen aus Homecomputer-Zeitschriften der 1980er-Jahre.

In der schaltlogischen Realisierung von Additionsrechenwerken zeigt sich ein weiteres Moment, das in der Geschichte des automatischen Rechnens als überwundene historische Vorstufe gesehen wird: Alle Formen binärer Volladdierer arbeiten mit kombinierten Bit-Schiebe- und -Addier-Schaltkreisen. In den Schieberegistern werden die Summanden bitweise in das Addierwerk geschoben und dort in einer Addierschaltung summiert. Der entstehende binäre Übertrag (0 oder 1) eines Halbaddierers wird als neuer Summand zum nächsten Halbaddierer transportiert. Dieses Verfahren ruft nicht zuletzt bei seiner Visualisierung Erinnerungen an die *Kugelrechenmaschine* (eine europäische Version des *Abakus*) auf, bei dem die diskreten Werte des Summanden in einer Zeile und die Dezimalüberträge in der folgenden Zeile angezeigt werden [vgl. Shores 2009].

Wie bei der Kugelrechenmaschine muss bei den meisten frühen Mikrocomputern jegliche Arithmetik grundsätzlich auf Additions- und Subtraktionsrechnungen reduziert werden, was ebenfalls einen ‚Rückschritt' gegenüber den Möglichkeiten vorheriger CPUs und ihrer komplexen mathematischen Opcodes zur Multiplikation und Division darstellt. Zudem ruft die ‚Kugel-Arithmetik' einmal mehr Leibniz' oben erwähnte *Machina arithmeticae dyadicae* in Erinnerung, welche seiner Beschreibung nach „kleine Würfel oder Kugeln"[12] zum Rechnen verwenden sollte. In beiden Beispielen zeigen sich die frühen Mikrocomputer als interessante *epistemische Objekte*: „Technische Gegenstände haben min-

12 http://dokumente.leibnizcentral.de/index.php?id=95 [letzter Abruf: 08.12.2016].

destens die Zwecke zu erfüllen, für die sie gebaut worden sind; sie sind in erster Linie Maschinen, die Antworten geben sollen. Ein epistemisches Objekt hingegen ist in erster Linie eine Maschine, die Fragen aufwirft.“ [Rheinberger 2001:24]

3.2.3.2 Medienepistemologie

Diese Umbrüche und Rückgriffe finden nicht auf der *diskursiven Medienoberfläche* statt und lassen sich nicht auf dieser analysieren, sondern beschreiben Vorgänge auf den *technischen Unterflächen* (Zur Metapher der „Unterfläche“ vgl. [Nake 2005:47ff.]) Unter Berücksichtigung des oben geschriebenen wirken diese Unterflächen (also die konkreten Techniken) jedoch auf die medialen Oberflächen ein und nehmen damit Einfluss auf das durch Medien proliferierte Wissen:

> Wohl heißt Wissen in diskursanalytischer Perspektive nur, was positiv ausgesagt werden kann. Medienwissen hingegen ergründet die Bedingungen solcher Aussageregeln, insofern sie selbst nicht wahrgenommen werden oder konstitutiv ausgeblendet sind. [...] Für technische Medien sind das, im Analogen wie im Digitalen, subliminale Optionen der Prozessierung. [Holl 2015, 85]

Das Programm der Medienarchäologie versteht sich daher zunächst als *Geschichtskritik*, jedoch in einem doppelten Sinne: Zum einen wird Mediengeschichtsschreibung (analog zur Foucault'schen geschichtskritischen Methode) mit techno-mathematischen Methoden auf ihre Rekurse und Friktionen hin untersucht, ergänzt und kritisiert. Zum anderen wird dabei zugleich die mediale Verfasstheit des Wissens und seines Transfers im Sinne einer *Medienepistemologie* verdeutlicht. Es wird also gefragt, welchen Anteil die Medientechnologien an der Wissensproduktion haben und (wiederum im Sinne einer Geschichtskritik) immer schon hatten. Technische Medien werden damit von bloßen Dispositiven zu *Agenten* der Wissensproduktion und -distribution, deren Agenda sich aus den Schaltplänen und Codes, die ihren Prozessen der Signal- und Symbolverarbeitung zugrunde liegen, herauslesen lässt.

Medienarchäologie verweilt aber nicht bei der Kritik, sondern versucht der Mediengeschichtsschreibung ein technisch adäquates Notationssystem zur Seite zu stellen, das jenseits historiografisch-diskursiver Sprachspiele [vgl. Wittgenstein 1985:277f.] auch technische Daten zu speichern und übertragen in der Lage ist. Hierzu muss jedoch die *Operativität als konstitutives Kriterium von Medien* berücksichtigt werden, was nur mit Beschreibungen möglich ist, die zeitlich nicht grundsätzlich erst *nach* den Medienprozessen erfolgen.[13] Wolfgang Ernst umschreibt dies mit dem Begriff der *Rekursion*, die nicht mehr

13 Anzumerken ist, dass auch Medienarchäologie in Form schriftlich fixierter Diskurse auftritt und damit scheinbar in den Modus des kritisierten Gegenstandes zurückfällt. Ein Ausweg aus diesem Dilemma stellt die offen diskutierte *Selbstreflexivität* dar, die bereits Hayden White den Historikern zur Offenlegung ihrer Arbeit als literarische Kunstwerke vorschlägt [White 1994:155]. Eine zweite Möglichkeit wird im weiteren Verlauf mit Hilfe der *Demonstration* vorgestellt [vgl. auch Höltgen 2019a].

als historische Metapher, sondern im mathematischen Sinne und aus ihrer algorithmischen Bedeutung für konstruktive Medienarchäologie genutzt werden soll:

> [D]er Begriff der Rekursionen ist nur in Grenzen auf Mediengenealogie übertragbar. Die von der Algorithmik induzierte, letztlich zumeist immer noch historiographisch gedeutete Figur der Rekursion wird von der Eigenlogik medienarchäologischer Zeit durchkreuzt. Die Entwicklung technischer Objekte zeitigt eigene Folgen und Rhythmen, die durchaus nicht immer mit ihrer zeitgenössischen Wissenschaftsgeschichte korrespondieren. Es gilt also, sich für einen Moment unabhängig von Erfinderbiographien zu machen und eine der technologischen Konfigurationen ihrer konkreten medientechnischen Existenzweisen zu schreiben. [...] Der medienarchäologische Ansatz verankert Rekursionen nicht innerhalb der menschlichen Kultur als Geschichtsfigur, sondern vielmehr exklusiv in der Eigengesetzlichkeit der in ihr als technische Medien verhandelten Elektrophysik, Logik und Mathematik. Es ist eine genuine Operativität hochtechnischer Medien, die eine alternative Form der Zeitschreibung nahelegt: die algorithmische Zeit, die zeitliche Logik der Programmierung in Schleifen und Bedingungen. [Ernst 2012a:444-448]

Solche Zeitfiguren lassen sich nicht mehr in Diskursen fassen, sondern verlangen nach einer Schreibung, die ihre Operativität (das heißt: ihre vollständige Gegenwärtigkeit im operativen Zustand) aufzuschreiben in der Lage ist. Ernst schlägt hierzu eine *Archäographie* vor, die sich aus den Methoden der Informatik speist:

> Bleibt für den Chronisten also nur Nachträglichkeit oder eine andere Schriftkompetenz namens Informatik; diese findet längst nicht mehr im Medium der Historie, der Erzählung, statt. Wenn Archäographie im Unterschied zur Geschichtsschreibung Programmieren heißt (*arché* verweist auf den Befehlsmodus) und die Archive der Zukunft nicht mehr Akten, sondern Algorithmen speichern, heißt das für die Retrospektive, „an den Blaupausen und Schaltplänen selber, ob sie nun Buchdruckerpressen oder Elektronenrechner befehligen, historische Figuren [...] abzulesen." {zit. n. Kittler 1986:3f.} Von Geschichtskörpern gibt es immer nur das, was Medien speichern und weitergeben können. [Ernst 2001:262]

Schaltpläne, Diagramme (Flussdiagramme, Automaten, ...), Messdaten, Formeln und andere Notationsverfahren der Natur-, Formal- und Ingenieurswissenschaften könnten historische Diskurse ergänzen, kommentieren und suspendieren. Einen konkreten Vorschlag hierfür hat N. Montfort gemacht: „technical reports" [Montfort 2013] sollten für geisteswissenschaftliche Diskurse genutzt werden, um auf diese Weise von narrativen zu technologischen Argumentationsmustern zu wechseln. Der *technical report* steht ihm zufolge zwischen den publizistischen Sachzwängen von peer-reviewten Journal-Beiträgen und der Unverbindlichkeit von wissenschaftlichen Weblog- und Twitter-Beiträgen. Wichtiger erscheint ihm jedoch der technische Stil solcher *reports*, die dem Genre der *grey lite-*

rature (Betriebsanleitungen, technische Dokumentationen, Sourcecodes, ...) zuzurechnen sind. Montfort versammelt seit 2012 auf seiner Webseite Trope Tank eine „Series of Technical Reports“[14], die sich mit computerhistorischen und -archäologischen Detailthemen und -fragen beschäftigen.

Montforts *technical reports* sind gattungsbedingt eher papers über einzelne Gegenstände als umfangreicheren Untersuchungen zu einem Arbeitsfeld (etwa in Monografien) zuzurechnen. Eine thematisch umfangreichere *Archäologie der Computer* – verstanden als maßgebliche Konvergenzmedien der Gegenwart [vgl. Heilige 2004:3] – im hier skizzierten, techno-mathematischen und informatischen Sinne und mit dem Ziel ihre Geschichte(n) zu analysieren und dabei die eingangs dokumentierten disparaten Historiografien zu kritisieren, steht noch aus. Zwar existieren ähnlich zu Montforts *technical reports* bereits Ansätze, die Computertechnologie, ihre Diskurse und Kulturen archäologisch zu betrachten [vgl. größer angelegte Projekte wie Bolz/Kittler/Tholen 2004; Dotzler 2006; Becker 2012], jedoch ist diesen gemein, dass sie markanterweise das aufgeführte, bereits seit Jahrzehnten bestehende Instrumentarium, welches die Informatik als dedizierte Computerwissenschaft in formaler, technischer, praktischer und gesellschaftswissenschaftlicher Hinsicht entwickelt hat, nicht nutzen.

Eine dezidierte *Computerarchäologie* überführt die Theorie und Methode der Medienarchäologie auf den Computer, indem sie das Instrumentarium der Informatik integriert und damit das Forschungs- und Lehrfeld der vormals wirtschafts- und technikhistorisch situierten *Geschichte des sogenannten Computers* für diese Disziplin öffnet. Dabei muss sie die schwierige Gratwanderung zwischen (allzu) holistischen Betrachtungen und (allzu) akribischen Detailanalysen leisten. Eine solche Theorie wird notwendigerweise einen engen Forschungsfokus besitzen (siehe Kapitel 3.3.3.2), auch um dem Paradigma wissenschaftlicher Falsifizierbarkeit genügen zu können. Ihre Methodik soll im Folgenden theoretisch umrissen und schließlich an einem Beispiel veranschaulicht werden.

3.3 Computerarchäologie als Methode

Im Vorwort zum 1994 publizierten Sammelband *Computer als Medium* [Bolz/Kittler/Tholen 1994], das den folgenden Überlegungen zugleich als Ausgangspunkt und Projektplan eines Forschungsdiskurses dient, konstatieren die Herausgeber:

> Die Informatik kann aber schon aus Gründen ihrer mathematischen Effizienz keine Mediengeschichte sein. Was die hier versammelten Beiträge suchen, ist folglich eine Wissenschaft von Computern, die keine Computerwissenschaft wäre. Sie untersucht weder Algorithmen noch Schaltkreise, sondern das Faktum, daß die Gegenwart von Algorithmen und Schaltkreisen gemacht wird. [7]

14 http://www.nickm.com/trope_tank/ [letzter Abruf: 23.02.2016].

Der Sammelband widmet sich folglich einer *Medienarchäologie von Computern* und ihrer Geschichte/Geschichtsschreibung auf Basis der unter 3.2.2f. beschriebenen Methodologie. Die Themen der einzelnen Beiträge reichen dabei von Vor- und Alternativgeschichten (etwa zum Analogcomputer, zu Rechenmaschinen und papiernen Archivmaterialen), Nutzungsweisen und -geschichten (Computer im Krieg, Kryptografie und andere) bis hin zu Detailfragen (Programmiersprachen, Vernetzung, Computerkunst und andere). Die meisten Beiträge folgen der Vorgabe des Vorwortes und verzichten auf technische Detailangaben (eine Ausnahme hiervon bildet [Hoelzer 1994]). Der expliziten *Pluralisierung* des Gegenstands im Vorwort („Computer*n*“) folgen die Beiträge jedoch nicht, sondern aspektieren ihren Gegenstand mit dem Kollektiv-Plural „der Computer“ [vgl. beispielsweise Bolz 1994:12-16], was, wie sich zeigen soll, bereits die technikferne Sichtweise forciert.

Für eine Überführung der Medienarchäologie des Computers in eine dedizierte *Computerarchäologie* wäre angesichts der oben zitierten Passage zu fragen: *Erstens,* ob das Argument der „mathematischen Effizienz“ ein valider Grund für die vermeintliche Ignoranz der Informatik gegenüber ihrer eigenen Geschichte ist – also, ob eine „Wissenschaft von Computern“, die sich der Historie der Computer widmet, auf „Algorithmen und Schaltkreise“ verzichten muss oder darf, ohne dadurch zugunsten ihrer Beschreibungsreichweite an analytischer Detailgenauigkeit zu verlieren. *Zweitens, wie* der hier insinuierte Übergang von „[G]eschichte“ [7] zu einer „Archäologie der Gegenwart“ [7] zu verstehen ist. Und *drittens*, ob die Verwendung des Plurals von Computer (der sich implizit bereits im Buchtitel findet) gegenüber der Kollektivum-Schreibweise in den Artikeln des Sammelbandes eine Eskalation mit epistemologischem Zusatznutzen darstellt. Hierzu wird im Projektteil der Arbeit (Kapitel 4) ein Set von Methoden zur Analyse der jeweiligen Fragestellungen erarbeitet.

3.3.1 Informatikgeschichte und Retrocomputing

Die Geschichte der Informatik ist selbst fester Bestandteil des Studien- und Forschungsgebietes Informatik. Die (deutsche) *Gesellschaft für Informatik* führt seit 1993 eine Fachgruppe *Informatik- und Computergeschichte*, die Bestandteil ihres Fachbereiches *Informatik und Gesellschaft* ist. [vgl. Heilige o. J.] „Ihre Aufmerksamkeit gilt insbesondere der Geschichte der Fachdisziplin und Institutionen, der Software und der Programmierung, der Mensch-Computer-Interaktion, der Computernetze und Computerkommunikation, der Sozialgeschichte der EDV-Berufe und der Alltagsgeschichte des Computers und der Datenverarbeitung.“ [Ebd.]

Heilige skizziert die Entwicklung der Informatik seit ihren Anfängen Mitte der 1950er-Jahre [Heilige 2004:2] als den Versuch über die Definition dessen, was „der Computer“ ist, ein stabiles Forschungsparadigma für die Disziplin zu finden. [vgl. Heilige 2004:4] Die Ansätze sind daher bis heute pluri-perspektivisch geblieben: Je nachdem, welche Computerdefinition zugrunde liegt,

> [...] konkurrierten in der Informatik von Beginn an immer verschiedene Theorien um den disziplinären Kern: so die als Gruppe auftretende Berechenbarkeitstheorie, Automatentheorie und die Formale Sprachen, dann die Theorie der Programmierung, die Informationstheorie, Kommunikationstheorie und die allgemeine Theorie der ‚Zeichenverarbeitung', schließlich die multidisziplinär angelegten Theoriekomplexe der Kybernetik und der ‚Automatique'. [Heilige 2004:9]

Daher erscheint Heilige der Versuch, *eine* Geschichte der Informatik zu zeichnen bereits als eine Reduktion auf *eine* Sichtweise, weswegen der Herausgeber den *Plural* Geschicht*en* (auch im Sinne einer Sammlung von historischen Erzählungen) bevorzugt. Die einzelnen Beiträge betrachten aus der Perspektive mittlerer theoretischer Reichweite Einzelphänomene (Informatik-Geschichte*n*) in ihrer Historizität und Technizität (Diagramme, Explosionszeichnungen, Formeln, Schaltpläne und anderes). Ein Beispiel-Kapitel aus dem Buch kann dies verdeutlichen:

Ganz im Sinne eines *technical reports* rekonstruieren Rojas et al. [2004] Konrad Zuses Entwurf der Programmiersprache Plankalkül aus den Archivmaterialen [232f.] und verwenden hierzu Datenfluss-Diagramme [215], Schaltlogik-Schemata [218] und deren aussagenlogische Form [219], Prozedur-Hierarchien als Baumdiagramm [219] und Source Codes [221-225, 227]. Die archäographische Eskalation stellt dabei nicht allein die Pointe des Beitrags dar, die die Implementierung eines Plankalkül-Compilers aus dem Jahr 2000 vorstellt [229-232], sondern bereits die Verwendung der oben aufgeführten Darstellungsformen, die eine (mental vorvollziehbare) Operativität ermöglichen: Es ist grundsätzlich möglich auf Basis des Datenfluss-Modells [215] eines Plankalkül-Programms seine Funktionsweise für beliebige Werte zu präjudizieren.

Im Unterschied zur Befürchtung von Bolz/Kittler/Tholen [1994] zehren „die Informatik" und ihre Geschichtsschreibung hier sogar wechselweise voneinander: Mit Hilfe der informatisch präzisieren Beschreibung des Plankalküls wird es beispielsweise erst möglich die Genese von Algorithmentheorie (beispielsweise des Computerschachs [vgl. Reinefeld 2006]), Compilerbau [Rojas et al. 2004:228] und nicht zuletzt Informatik-Didaktik zu verstehen:

> Für einige unserer studentischen Mitarbeiter war dies der erste von ihnen erstellte Compiler, insofern hatte das ganze eine didaktische Komponente. Das wichtigste war aber, das mit solche[n] Projekten Computergeschichte plötzlich und unerwartet lebendig wird. Die Geschichte des Computers ist zu wichtig, um sie allein den Historikern zu überlassen. Man kann heute Informatiker für die Geschichte unseres Faches durch solche Rekonstruktionsprojekte interessieren und begeistern. Für uns war die Erstellung eines Plankalkülcompilers eine beeindruckende Reise zurück in die Ursprünge des Computerzeitalters. [Rojas et al. 2004:229]

Für den operativen Nachvollzug dieser Überlegungen muss das Ergebnis (der Plankalkül-Compiler) zugänglich sein. Rojas und seine Arbeitsgruppe haben daher einen Editor[15] und Compiler[16] für den Einsatz im Webbrowser erstellt. Als Beispiel-Algorithmus wurde Zuses eigener Entwurf für ein Schachprogramm implementiert[17]. Diese Arbeiten gerieten zur Grundlage eines Computerschach-Kurses und -Projektkurses, der im Wintersemester 2015/16 im Masterstudium Medienwissenschaft durchgeführt wurde[18] und im Rahmen dessen unter anderem verschiedene historische Schach-Engines miteinander verglichen und gegeneinander zum Spiel gebracht wurden. Hierbei konnte beispielsweise die Überlegenheit des TuroChamp-Algorithmus[19] von Alan M. Turing gegen Zuses Programm gezeigt werden. (Letzteres ist lediglich ein Zuggenerator für regelgültige Züge, der das Schachbrett von links oben nach rechts unten auswertet und auf dieser Basis den nächstmöglichen Zug wählt.) Dieses kontemporäre, spielerische Experimentieren mit historischen Algorithmen ist eine typische Praxis des *Retrocomputing*.

Für den Gegenstand der frühen Mikrocomputer wird gerade die Verbindung von Didaktik und Archäographie (siehe Kapitel 5) zum zentralen Einsatzpunkt, welcher Archivarbeit, Epistemologie und Informatik miteinander verbindet. Dabei werden im Sinne Ernsts [2012:444] beide Richtungen *rekursiv* beschritten: Mit der Entwicklung von Emulatoren, die alte Hardware-Systeme als Software in neuen Systemen operativ aufrufen, findet eine zu Rojas et al. analoge Archäographie statt.

Für eine Methodologie der Computerarchäologie erweisen sich daher informatische und techno-mathematische Darstellungen als konstitutiv – insbesondere dort, wo historische Fragestellungen in systematische oder didaktische überführt werden. Die Vergegenwärtigung historischer Aspekte der Informatik verlangt also nachgerade nach Methoden, die jenseits historischer Zeitbegriffe ansetzen. Die Darstellung eines Projektes, wie der Entwurf und Bau einer Schaltung oder die Analyse obfuskierter Algorithmen wären ohne solche Methoden nicht durchzuführen und ohne die spezifischen Darstellungsweisen nicht zu erklären.

15 http://zuse-z1.zib.de/simulations/plankalkuel/editor/pk.html [letzter Abruf: 23.02.2016].

16 http://zuse-z1.zib.de/simulations/plankalkuel/compiler/plankalk.html [letzter Abruf: 23.02.2016].

17 http://zuse-z1.zib.de/simulations/plankalkuel/chess/applet/chess.html [letzter Abruf: 23.02.2016].

18 https://agnes.hu-berlin.de/lupo/rds?state=verpublish&status=init&vmfile=no&publishid=104661&moduleCall=webInfo&publishConfFile=webInfo&publishSubDir=veranstaltung [letzter Abruf: 05.06.2019].

19 https://chessprogramming.wikispaces.com/Turochamp [letzter Abruf: 23.02.2016].

3.3.2 Archäologie der (Computer-)Gegenwart

Bei der Realisierung aktueller informatischer Projekte auf historischen Computerplattformen wird in Gegenrichtung des Projektes von [Rojas et al. 2004] der Anachronismus (der in der Historiografie einen Fehler darstellt [vgl. Schmidt-Biggemann 2003]) operativ vollzogen und damit zu einer Provokation von traditioneller Computergeschichtsschreibung. Wenn beispielsweise (wie in Kapitel 5.4.3 dargestellt) auf einer Computerplattform von 1985 ein modernes Multitasking-Betriebssystem mit grafischem Userinterface aus dem Jahre 2000(ff.), in welchem wiederum ein aktueller Interpreter einer historischen Programmiersprache implementiert wird, ist damit nicht nur die Behauptung, frühe Homecomputer „waren noch nicht gut genug für ernsthafte Anwendungen." [Ceruzzi 2003:313f.] widerlegt; hier verliert jegliche historiografische Darstellung gegenüber einer informatisch argumentierenden Archäologie, die ganz andere Zeitfiguren und -arten (der Maschine, des Betriebssystems, des Programmcodes) in den Fokus rückt, ihre Beschreibungskompetenz.

Hierin unterscheidet sich der Begriff der Archäologie im Sinne Foucaults und Ernsts am deutlichsten von dem der Fach-Archäologie. Wo letztere eine Bergung von Artefakten der *Vergangenheit* (aus der Gegenwart heraus) darstellt, unternimmt erstere eine Analyse der *Gegenwart*, ihrer Diskurse, Dispositive und Technologien. ‚Freigelegt' wird dabei das Wissen über Technologien ganz unterschiedlicher technischer und historischer Zeiten, das – bildlich gesprochen – von den Diskursen, Historiografien, Anekdoten und Mythen verdeckt ist. Daher verfährt eine am technischen Artefakt ausgerichtete Computerarchäologie bei der Wahrheitssuche nicht nach diskursanalytischen oder hermeneutisch-geisteswissenschaftlichen, sondern positivistischen-falsifizierbaren naturwissenschaftlichen Methoden.

Dass diese Methoden auch auf historische technische Artefakte, wie frühe Mikrocomputer, angewendet werden können, liegt in deren *theoretischer* und *technischer* Fundierung: „Computer sind vielmehr das einzige technische Medium, das es ohne seine Theorie gar nicht gäbe." [Bolz/Kittler/Tholen 1994:7] Das bedeutet: Computer sind aus der Theorie hervorgegangen. Medienwissenschaftliche Beiträge, die zur Zeit des Sammelbandes *Computer als Medium* erschienen sind, setzen sich verstärkt mit diesen informatiktheoretischen Paradigmen von Digitalcomputern und deren Begründungsschriften (Turing, von Neumann sowie Leibniz, Boole und Shannon) auseinander. Deren Beschreibungen von Computerfunktionen liegen in Form von logischen Gattern, Automaten-Modellen, Flussdiagrammen Codebeispielen und „auf Englisch, wo nötig unter Zuhilfenahme mathematischer Symbole" [Turing 1987:121] vor. Solche formalen Modelle und Beschreibungen sind, wie oben ausgeführt, *prinzipiell* operativ, das heißt: Sie lassen sich lesend (nach)vollziehen. Aus diesem Grund bieten die Notationssysteme der theoretischen, technischen und praktischen Informatik bereits vielfältige Möglichkeiten für eine Computerarchäolo-

gie die formalisierbaren Aspekte historischer Computer *theoretisch* in die Gegenwart zu projizieren.

Für *reale* Computer ist hingegen, wie für alle technischen Medien, *Operativität konstitutiv*. Ein ausgeschalteter oder dysfunktionaler Computer kann nur prinzipiell (beziehungsweise virtuell, also in seinen *potenziellen* Funktionen) dargestellt werden, wofür die Werkzeuge der Theorie nutzbar sind. Eingeschaltet reduziert sich der virtuelle Horizont der möglichen Funktionen zu einer realen Option unter vielen anderen Alternativen. Aus der universellen Turingmaschine und der Von-Neumann-Architektur wird dann beispielsweise ein konkreter Schachcomputer, ein Finanzbuchhaltungssystem, ein Internetbrowser und so weiter. Der Unterschied dieser beiden Ein-/Ausschaltzustände zeigt sich in der Temporalität, die Ernst als *Kairos* (dem realen, gegenwärtigen *Zeitpunkt*) und *Chronos* (dem historischen *Zeitfluss*) unterscheidet:

> Auf welche Zeitmodelle also läßt sich der Begriff von Medienzeit (für den Fall elektronischer Medien) bauen? Das eine ist die emphatische Zeit von *chronos*, mithin der Diskurs der Historie; das andere ist die Zeit des *kairos*, das zeitkritische Element in Mediensystemen selbst – zwischen Gedanken und Affekt. Transponiert in einen technomathematische Zustand, wird die Differenz von Bewegtbild und Momentbild unterlaufen – in einem signalverarbeitenden Medium, dessen Wesen selbst ein prozessuales, also strikt zeitkritisches ist (der Computer in der Von-Neumann-Architektur). [Ernst 2012a:184 – Hervorh. i. O.]

Computerarchäologie muss sich daher stets auf Computer im operativen Zustand beziehen (vgl. [Ernst 2012b:365-378], der dort ebenfalls den Begriff „computerarchäologisch" einführt) und metaphorische Sprechweisen damit ebenso ausschließen, wie soziale, politische, ästhetische oder ökonomische *Diskurse über Computer*. Anstelle dessen wird sie mit realen Computern und ihren Funktionen argumentieren und damit konkrete Argumente oder Gegenargumente in diese Diskurse einbringen. Hierzu muss notwendigerweise zur diskursiven Darstellung (in Texten und Vorträgen) die *Demonstration* des bezogenen Objektes hinzutreten. Auf diese Weise generiert sie empirisch nachprüfbare Aussagen. Zum Beispiel: Zu konstatieren, „daß die Gegenwart von Algorithmen und Schaltkreisen gemacht wird." [Bolz/Kittler/Tholen 1994:7], bedürfte daher (neben der Konkretisierung, was mit „der Gegenwart" gemeint ist) zumindest die Ausführung, *welche Algorithmen und Schaltkreise* auf welche Weise etwas mit der Gegenwart „machen". Dabei könnte sich zeigen, dass eine solche Aussage nicht falsifizierbar und damit methodologisch streng genommen nicht wissenschaftlich ist.

3.3.3 Der (sogenannte) Computer

Hier deutet sich bereits an, dass vom ‚Computer' in allgemeiner (das heißt kollektivpluraler) Weise zu sprechen problematisch für die Falsifizierbarkeit wissenschaftlicher Hypo-

thesen sein kann. Friedrich Kittler hat an zahlreichen Stellen in seinem Werk auf ironische Weise auf solche Generalisierungen aufmerksam gemacht, indem er Kollektivpluralformen das Adjektiv „sogenannt“ vorangestellt hat. In seinen 1993 erschienen *Technischen Schriften* verwendet er diese Phrase 13 mal [Kittler 1993:58,60,62,126,133,144,147, 173,210,217,236,237,240]. Markanterweise findet sich jedoch weder dort nach an anderen Stellen seines Werks eine Erwähnung des „sogenannten Computers“, obgleich er sich wissenschaftlich mit realen Computern auseinandergesetzt hat. Der zuvor dokumentierte Zugang zu dieser Technologie auf Basis ihrer *theoretischen (Begründungs-)Diskurse* könnte einen Grund hierfür sein; Kittler selbst hat die Schriften Alan Turing auf deutsch übersetzt, mit herausgegeben [vgl. Turing 1987] und damit diesen historischen Diskurs der theoretischen Informatik für die Medien- und Kulturwissenschaft akzentuiert. Einen anderer Grund zeigt die Schwierigkeit, einen real existierenden Apparat in einem generalisierenden wissenschaftlichen Diskurs zu behandeln, wie ein zweiter Blick auf das oben genannte Zitat zeigt: „Computer sind vielmehr das einzige technische Medium [...].“ [Bolz/Kittler/Tholen 1994:7] Die stilistische Auffälligkeit entsteht dadurch, dass das „sind“ suggeriert, hier wären „[Die] Computer“ gemeint, dann jedoch „das Medium“ in Numerus-Widerspruch zum Satzsubjekt steht. Der Halbsatz changiert zwischen den Lesarten „Der Computer ist vielmehr das einzige technische Medium“ und „Die Computer sind vielmehr die einzigen technischen Medien“.

Naturwissenschaftliche Methodologie fordert nachprüfbare Formulierungen von Forschungshypothesen. Dies ist nur möglich, wenn der Gegenstand der Hypothese eindeutig definierbar ist. (Sogenannte) „Computer“ sind kein solcher Gegenstand, weil unklar ist, welche Computer hiermit gemeint sind. Die Bedeutung von „Computer“ ist vielfältig und reicht von „rechnenden Menschen“ [vgl. Ceruzzi 1998:1f.] über mechanische und elektronische Analogcomputer, Relais-, Röhren- oder Transistor-basierten Digitalcomputern, Hybridcomputer, Mainframes, Minicomputern, Mikrocomputern, Mikrocontrollern bis hin zu Computermodellen. Selbst dann, wenn sich ein Verständnis von „Computer“ auf Mikrocomputer einer spezifischen Generation (etwa definiert an den Datenbus-Breiten, vgl. 1.2) bezieht, ist damit keine Forschungshypothese generierbar, wenn diese nicht konkrete Informatik-Systeme aspektiert.

Daher kann eine Aussage wie die bereits zitierte, frühe Homecomputer „waren noch nicht gut genug für ernsthafte Anwendungen.“ [Ceruzzi 2003:313f.] selbst wenn der Autor dies mit „Rechner wie der TRS-80“ [313] spezifiziert, nicht wissenschaftlich überprüft werden. Dies liegt zum einen daran, dass Ceruzzi das, was er mit „ernsthafte Anwendungen“ meint, nicht spezifiziert. Zwar lässt sich durch seine vorherige Angabe „Sommer 1977“ [311] schließen, dass er sich auf das TRS-80 Model I Level I bezieht, nicht jedoch, welche „Rechner wie der TRS-80“ gemeint sind. Diese technikhistorische Vagheit stellt für computerarchäologische Fragestellungen allerdings nur ein Randproblem dar. Nicht erst mit heutigen Mitteln lässt sich der TRS-80 Model I Level I zu einem System auf- und

umrüsten, das auch „ernsthafte Anwendungen“, wie sie Ceruzzi im Sinn gehabt haben mag, genügen könnte. Bereits zur Zeit seiner Erstveröffentlichung [vgl. Comicro 1978:94] wurden die Rechner – teilweise durch eigene Modifikationen, teilweise durch Erweiterungen von Drittanbietern – so verändert, dass sie den gewünschten und geforderten Zwecken dienen konnten.[20]

3.3.3.1 Platform Studies

Eine Verengung der Perspektive und zugleich Aktualisierung historischer Hardware versuchen die *Platform Studies*. In Ihrem ersten Beitrag Racing the Beam [Bogost/Montfort 2009] untersuchen beide Autoren die Hard- und Software der Atari-VCS-Spielkonsole. Die Programmierung dieser Plattform geriet zur Zeit ihrer Veröffentlichung aufgrund der spartanischen Hardware-Ausstattung förmlich zur Kunst und noch heute werden Spiele und Demonstrationen für die VCS entwickelt (vgl. Kap. 4.1.3.1). Ein zentraler Aspekt der *Platform Studies* ist, dass Hardware im Verbund mit der Software zu einer Emergenz produzierenden Einheit – der *Plattform* – wird. Erst dies ermöglichte der VCS ihren Erfolg und ließ die Homecomputer-Entwicklung auf ganz ähnliche Architekturen setzen. Die Autoren definieren eine so verstandene Plattform demzufolge:

> A platform in its purest form is an abstraction, a particular standard or specification before any particular implementation of it. To be used by people and to take part in our culture directly, a platform must take material form, as the Atari VCS certainly did. This can be done by means of the chips, boards, peripherals, controllers, and other components that make up the hardware of a physical computer system. [...] Whatever the takes for granted developing, and whatever, from another side, the user is required to have working in order to use particular software, is the platform. In general, platforms are layered – from hardware through operating systems and into other software layers – and they relate to modular components [Bogost/Montfort, 2009:2f.]

Die Beiträge der *Platform Studies* untersuchen nun also stets *eine* solche Platform in Hinblick auf ihre Hardware-Besonderheit(en), ihre Software und der daraus ablesbaren Programmier-Historie, aber auch ihrem Einfluss auf die Wirtschaft, die Kultur und das ‚Echo‘, als das die jeweilige Platform in späteren Architekturen vernehmbar ist:

> Our approach is mainly informed by the history of material texts, programming, and computing systems. Other sorts of platform studies may emphasize different technical or cultural aspects, and may draw on different critical and theoretical approaches. To deal deeply with platforms and digital media, however, any study of this sort must be technically rigorous. The detailed analysis of hardware and

20 Es gehört in den Bereich meiner privaten Oral History, dass mein Vater 1981/82 einen TRS-80-kompatiblen Computer durchaus sinnvoll als Arbeitsrechner eingesetzt hat, liefert aber zumindest ein konkretes Gegenbeispiel zu Ceruzzis These.

code connects to the experience of developers who created software for a platform and users who interact with and will interact with programs on that platform. [Bogost/Montfort, 2009:3]

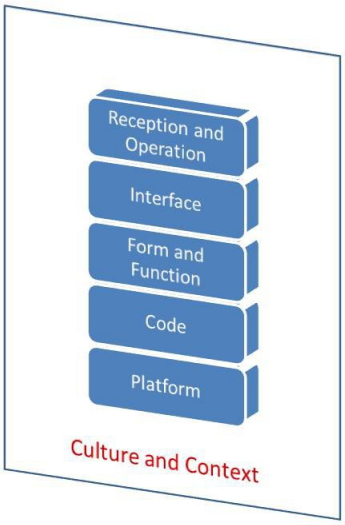


Abb. 3.3: Layer der Platform Studies [Bogost/Montfort 2009]

Auch wenn der methodische Zugang der *Platform Studies* dem der Computerarchäologie zunächst ähnelt, basiert diese nicht auf demselben theoretischen Ansatz, sondern verortet sich deutlich im Programm der US-amerikanischen *Cultural Studies*: „A computational platform is not an alien machine, but a cultural artifact that is shaped by values and forces and which expresses views about the world“ [Bogost/Montford, 2009:148]. Durch die Rückbindung an kulturwissenschaftliche Fragestellungen gewinnen die *Platform Studies* einen ‚größeren Abstand' von ihrem Gegenstand (Computer bzw. hier: Computerspielkonsole), um beispielsweise nach der kulturellen, ästhetischen und technikhistorischen Wirkung der Atari-VCS-Konsole zu fragen und diese aus den in ihren Studien analysierten Hardwares und Software-Beispielen abzuleiten – mit allen Konsequenzen für wie wissenschaftliche Theoriebildung. Theoretische Beschreibungen von Computern können die *Platform Studies* daher nicht integrieren, weil sich diese zu wenig direkte kulturelle Relevanz besitzen. Aber auch das einzelne, reale Gerät kann nicht ihr Gegenstand sein, weil sich aus ihm keine kulturwissenschaftlichen Verallgemeinerungen ableiten lassen.

3.3.3.2 Singuläre Computer

Die generalisierende Sprechweise von Computern und einzelnen Plattformen wird vor allem auf einer wesentlich niedrigeren Stufe problematisch: Dort, wo der Computer als elektronisches und physikalisches System gesehen wird, verlangt eine adäquate Beschreibung eine weitere Einengung – und zwar auf das *konkrete Gerät* und seine technischen Spezifikationen, die sich *zu einem bestimmten Zeitpunkt* und *unter bestimmten Bedingungen* darstellen lassen. Ein im Zuge dieses Forschungsprojektes reparierter TRS-80

Model I Level II hat einen spezifischen Kondensator-Schaden, der ihn von einem anderen Gerät der Sammlung unterscheidet. Sowohl der original verbaute Kondensator als auch das Austauschbauteil besitzt je spezifische Eigenschaften (die noch hinter die Unterschiede der Herstellerfirmen, Produktionsdaten und elektrischen Toleranzbereiche zurückreichen). Je nachdem, auf welcher Ebene über ‚den' TRS-80 Model I Level II gesprochen werden soll, müssen diese Merkmale Berücksichtigung finden (vgl. Kap. 4.4).

Eine derartig starke Einengung des Fokus stellt keine Spitzfindigkeit dar, sondern folgt einem theoretischen Programm: Das, was für historische unikale Computer wie den ENIAC, die Z3 oder den Harvard Mark 1 in der historischen Forschung gilt, kann auch für einen spezifischen Mikrocomputer (als Exemplar einer Baureihe) gelten: ‚Den' Computer als *besonderen* Apparat zu analysieren und dabei *verallgemeinerbare* Erkenntnisse darüber produzieren. Diese Beziehung zwischen Allgemeinem und dem Besonderen bildet die Schnittstelle zwischen Natur- und Geisteswissenschaft, zwischen Informatik und Medienwissenschaft. (Vgl. Küppers 2000). In pragmatischer Hinsicht ermöglicht diese Einengung bei einigen historischen Computern erst das Verständnis spezifischer Idiosynkrasien (die sich beispielsweise in Platinen-Revisionen derselben Modellreihe zeigen), die Herstellung besonderer Hardwareerweiterungen, die Erfordernis bestimmte maschinennahe Programmierungen und nicht zuletzt eine adäquate, funktionsgenaue Reparatur/Restauration.

Sinnfällig wird dies am Beispiel des Exidy Sorcerer, den Horst Völz ab 1984 zur Entwicklung der KC-85-Textverarbeitung TEXOR verwendet und eigens für diesen Zweck mit individuellen Hardware- und Software-Modifikationen versehen hat. Die Reparatur dieses spezifischen Computers erforderte die genaue Analyse der (teilweise sowjetischen) Bauteile sowie des von Völz selbst entwickelten BIOS-ROMs unter Zuhilfenahme von technischen Manuals (zum originalen Sorcerer-Modell von *Exidy*), Programmiertutorials (zum im Computer verbauten Z80A-Mikroprozessor), Messapparate zur Analyse der Hardware-Veränderungen und nicht zuletzt Rückfragen bei Zeitzeugen zu den verwendeten Bauteilen und Materialien (vgl. Abb. 3.4 und 3.5).



Abb. 3.4 & 3.5: Der von Horst Völz modifizierte Exidy Sorcerer

Um solche Praktiken untersuchen zu können, muss sich Computerarchäologie neben den oben skizzierten formalen und modellhaften Beschreibungen grundsätzlich auch mit konkreten, operativen Apparaten befassen können. Projekte, wie die in den folgenden Kapiteln dokumentierten, basieren auf solchen lauffähigen und laufenden historischen Computern und fordern zu ihrer Analyse die Berücksichtigung der vielfältigen und komplexen Strukturen ihrer Hard- und Softwares. Hierfür stehen neben den sinnlich erfahrbaren Input- und Output-Vorgängen dedizierte Messtechniken und -methoden, die die technische Informatik, Physik, Chemie, die Elektro- und Messtechnik anbieten, zur Verfügung. Mit diesen Methoden ist es möglich, konkrete Materialitäten und Mikrostrukturen ebenso wie aktivierte Schaltungen, prozessierende Programme und Datenflüsse sowohl im Innern des Computers als auch an seinen Peripheriegeräten *in statu nascendi* zu untersuchen. Als Werkzeuge stehen hierfür Disassembler, Debugger sowie Oszilloskope, Frequenz- und Logikanalysatoren, Multimessgeräte und anderes elektrotechnisches Testequipment zur Verfügung.

Analysen einzelner Artefakte auf dieser Ebene ermöglichen allerdings keine beliebig verallgemeinerbaren Aussagen und lassen sich (aus der theoretischen Distanz des wissenschaftlichen Diskurses) oft auch nicht falsifizieren. Umso notwendiger ist hier eine formal exakte, detaillierte Beschreibung der Artefakte und der ihnen zugehörigen Projekte. Wissenschaftstheoretisch kann Computerarchäologie daher zu den *Theorien mittlerer Reichweite* gerechnet werden. Diese Zuschreibung geht auf die Methodologie des Soziologen Robert K. Mertons zurück. Er hat beobachtet, dass weitreichende soziale Theorien angesichts mikrosozialer Beobachtungen oft ihre Beschreibungskompetenz verlieren, mithin individuelles soziales Verhalten Emergenzeffekte zeitigt, die durch universalistische Theorien nicht prognostizierbar sind und vice versa nicht zur Induktion von Theorie mit universalistischer Beschreibungspotenz nutzbar sind. Merton definiert:

> Neben solcher allgemeinen Theorie sind andere, ebenfalls analytische und systematische Theorien von weit begrenzterem Bezug entwickelt worden. Sie enthalten Kombinationen von Vorstellungen, die als Theorien mittlerer Reichweite bezeichnet werden können. [...] Diese Theorien enthalten natürlich auch Abstraktionen, die aber nicht so weit von den Daten der soziologischen Beobachtung entfernt sind. [...] Theorien mittlerer Reichweite [..] bieten hinreichende Erklärungen für ausgewählte Aspekte eines begrenzten Erscheinungsbereichs, und sie lassen sich mit anderen gleichartigen Theorien zu einem umfassenderen Bündel von Ideen zusammenfassen. [Merton 1973:319-321, vgl. auch Merton 1968:60-64]

Derartige Theorien zeigen sich flexibler gegenüber Mikro-Erscheinungen und erlauben es neue empirische Forschungsergebnisse zu integrieren. Dieses Theorie-Modell ist von soziologischen auf historische Theorien übertragbar, wie Ernst anmerkt. Historische Theorien sollen ihm zufolge beständig an „[a]rchivische Datenmengen“ [Ernst 2003:30] zurückgebunden werden, um damit neue Konfigurationen [vgl. ebd.] historischen Wis-

sens möglich zu machen. Computerarchäologische Fragestellungen orientieren sich allerdings stets am konkreten Gerät/Problem, das sie im Idealfall als Basis für eine Verallgemeinerung (mittlerer Reichweite) nutzen können. Der Objekt-Bezug bleibt jedoch auch deshalb maßgeblich, weil nur durch die *Demonstration* der Argumente am Objekt einer rein diskursiven Formulierung (die sich wieder der unter 3.2 vorgestellten Kritik unterziehen müsste) entkommen werden kann. Dieses Unterlaufen diskursiver Beschreibungen von Prozessen *durch Zeigen* zieht zugleich eine mit naturwissenschaftlichen Methoden vergleichbare empirische Überprüfbarkeit von Thesen im Nachvollzug des Experimentes nach sich. Die praktische Demonstration am Medium selbst wird von verschiedenen Medienwissenschaftlern als neue Methode vorgeschlagen. Unter dem Begriff der „carpentry" fordert Bogost [2012:85ff.] das (technische) Experiment, um Medientechnologien aus ihrer bislang am Subjekt/an der Kultur ausgerichteten Interpretation zu entziehen. In der Medienarchäologie stellt die Demonstration (als ‚Vergegenwärtigung' historischer Medientechnologie) ein zentrales Moment dar, in welchem Medien ihre Eigenzeit inszenieren. Wolfgang Ernst führt hierzu den Begriff des *Re-Enactments* ein.

Das *Re-Enactment* wird von Collingwood [1955] bereits 1946 als historische Forschungsmethode vorgestellt. Collingwood konstatiert, dass historische Forschung immer schon in der Gegenwart stattfindet – im Bewusstsein des Historikers, der den historischen Vorgang durch Erinnerung aktualisiert: „Denn historische Kenntnis und Erkenntnis ist nichts anderes als der Nachvollzug der Erfahrungen der Vergangenheit im Geist des Denkers der jeweiligen Gegenwart." [340f.] Der Historiker vollzieht diesen Akt bewusst [vgl. 300f.], wodurch dieser *Nachvollzug* zu einer geschichtsphilosophischen Forschungsmethode wird. Hatte White den konstruktivistischen Charakter von Geschichts*schreibung* betont, so richtet Collingwood das Augenmerk nun auf das Geschichtsdenken selbst, das gar nicht anders als im Modus des Re-Enactments stattfinden kann: „In so far as all history is the history of thought, this must be so: for one can only apprehend a thought by thinking it, and apprehend a past thought by re-thinking it." [Collingwood 1999a:223] Dies weitet Computerarchäologie methodologisch aus:

> Der Historiker vollzieht nicht nur das Denken der Vergangenheit in sich nach, er vollzieht es vielmehr im Zusammenhang mit seiner eigenen Erkenntnis. Er verbindet also mit dem Nachvollzug eine kritische Beurteilung des Denkens der Vergangenheit, er bildet sich sein eigenes Urteil vom Wert dieses Denkens und verbessert die Irrtümer, die er in ihm entdeckt. Die Kritik an dem Denken, mit dessen Geschichte er sich befaßt, ist nicht etwa von sekundärer Bedeutung dieser Geschichte gegenüber; sie ist vielmehr eine unerläßliche Voraussetzung der historischen Erkenntnis selbst. Man könnte sich hinsichtlich der Geschichte des Denkens nicht gründlicher irren als mit der Annahme, daß der Historiker als solcher lediglich das Denken der jeweiligen Geschichtsträger feststellt, indem er einem anderen die Entscheidung darüber überläßt, „ob dieses Denken richtig war". Alles Denken ist kriti-

> sches Denken; das Denken, das Gedanken der Vergangenheit nachvollzieht, kritisiert daher diese Gedanken im Nachvollzug. [Collingwood 1955:226]

Die Konsequenz hieraus führt zu einer *Aktualisierung des Historischen*: „if history is the re-enactment of the past in the present: for a past so re-enacted is not a past that has finished happening, it is happening over again." [Collingwood 1999b:240]

Von dieser Überlegung ausgehend schlägt Wolfgang Ernst den Begriff des „re-enactment" [Ernst 2012a:200ff.,313ff.; Ernst 212b:363ff.] als „Nachvollzug des Mediums" [Ernst 2012a:313] in einem „Medientheater" [Leeker 2001] vor, in welchem die Medientechnologien selbst die Akteure sind [vgl. Ernst 2012b:147]. Er exemplifiziert dies am Beispiel der Emulation des Commodore 64 [vgl. ebd. 363-366], um den Begriff der Emulation als medienarchäologisches Konzept dem Konzept der Simulation gegenüberzustellen. In den Projekten des Kapitels 4 wird das Re-Enactment noch einmal ‚wörtlich' im Sinne Collingwoods genommen: als reflektiertes gegenwärtiges Nachvollziehen historischer „thoughts", wenn etwa aktuelle Programmierprojekte Funktionen historischer Programme auf derselben oder einer anderen Plattform nachvollziehen sollen. Dieser Nachvollzug geschieht im Modus der Demonstration als Medientheater und bezieht sich dann auf einen konkreten Computer (und nicht dessen Emulation, weil ansonsten über den emulierenden Computer gesprochen werden müsste). Die Aufführung und Inszenierung computerhistorischer Prozesse in der Gegenwart überführt die geschichtsphilosophische Theorie des Re-Enactment in eine Form der *experimentellen Archäologie* [Keefer 2006:26]. Eine Medienwissenschaft, die sich dieser Methode bedient, inkorporiert damit naturwissenschaftliche Methodik (das Experiment als Forschungs-, Beweis- und Prüfmittel) in ihre Methodologie.

Abschließend soll an einem Beispiel gezeigt werden, wie die theoretischen und methodischen Ansätze von Computerarchäologie bereits implizit auf einen konkreten historischen Diskurs und einen aktuellen Gegenstand angewendet wurden. (Das Beispiel wurde im Rahmen verschiedener Vorträge operativ belegt. [z. B. Höltgen 2014d])

3.3.4 Fixing (the history of) E.T.

Im Frühjahr 2014 berichteten zahlreiche internationale Medien [stellvertretend vgl. Reißmann 2014] von einer archäologischen Ausgrabung historischer Computerspielehard- und -software: Die Firma *Microsoft* hatte ein Dokumentarfilm-Team finanziert, um auf der Müllhalde der Stadt Alamogordo (New Mexico, USA) nach ‚angeblich' 1983 dort deponierten Gegenständen der seinerzeit in Finanznot geratenen Firma *Atari* zu suchen. Obwohl es schon zur Zeit der damaligen Lagerräumung *Ataris* Medienberichte über die Deponierung nicht-verkäuflicher Spiele und Hardwarekomponenten gab [vgl. N. N. 1983], entwickelte sich aus der Aktion über die vergangenen drei Jahrzehnte „einer der größten

Mythen der Videospielindustrie“ [Freundorfer 2009], der in zahlreichen Publikationen kolportiert wurde.

An diesen Mythos ist insbesondere ein Spiel *Ataris* gekoppelt: E.T. – The Extraterrestrial (1982), von dem der bei der Ausgrabung entstandene Dokumentarfilm handelt. Diesem Spiel wird eine bedeutende Rolle in der Wirtschafts- und Firmen- und Personengeschichte der Computerspiele und mithin der Mikrocomputer-Industrie zugeschrieben. Allgemeinhin als „worst video game ever“ [Reinhard 2014] apostrophiert, soll E.T. – The Extraterrestrial maßgebliche Schuld am wirtschaftlichen Untergang der Firma *Atari* sowie in der Folge der gesamten Computerspiel-Branche der frühen 1980er-Jahre gehabt haben. Goldberg/Vendel [2014:592ff.] vergleichen den so genannten *Videospiel-Crash* in seiner historischen Bedeutung sogar mit dem Überfall auf Pearl Harbor im Jahr 1941.) Die Diskursproduktion zu diesem ökonomischen Umbruch und im Zuge dessen zum Mythos des auf dem Müll deponierten Spiels hat sich über die Jahrzehnte perpetuiert, wie eine Websuche nach den Begriffen „E.T.“, „Atari“ und „Video game crash“ zeigt.[21]

Im Frühjahr 2013 haben Hobbyisten um David Richardson[22] das Spiel auf der Codeebene analysiert und dabei herausgefunden, dass die in der Literatur aufgeführten Argumente für das Scheitern des Spiels nicht nachvollziehbar sind: Die als Programmierfehler angesehenen Sprite-Kollisionen zwischen der Spielfigur und der Hintergrundgrafik des Spiels erwiesen sich dabei als Konsequenz einer gemischten Darstellung von zweidimensionalem Vordergrund und perspektivisch verzerrtem, dreidimensionalem Hintergrund. Diese Mischperspektive findet sich auch in anderen zeitgenössischen Spielen, in denen sie jedoch keine vergleichbaren Kritiken hervorgerufen hat. Grund dafür, dass sie bei E.T. – The Extraterrestrial moniert wurde, ist, dass sich aus der Mischperspektive Grafik-Kollisionen zwischen der Spielfigur und dem Hintergrund ergeben, die für den Spieler schwer nachvollziehbar sind. Richardson et al. haben dies als vom Programmierer beabsichtigte Steigerung des Schwierigkeitsgrades gewertet und die sich daraus ergebenden Schwierigkeiten umprogrammiert. Überdies hat das Team weitere oft monierte Programmierfehler als „Myths“ [Ebd.] erkannt und kleinere tatsächliche Fehler in der Darstellung der Spielfigur (etwa deren falsche Farbe) korrigiert sowie zusätzliche Features in das Spiel eingebracht.

Für den hier diskutierten Zusammenhang ist an diesem Projekt die Konfrontation von ‚offizieller Geschichtsschreibung‘ und deren Kritik durch Argumente auf der technischen/informatischen Ebene bedeutsam. Richardson et al. gehen in ihrem Projektbericht explizit auf Vorwürfe der Spiel-Presse und in historischen Darstellungen zum Spiel ein. Dieser Projektbericht selbst ist stilistisch nach dem von Montfort vorgeschlagenen Textsorte *technical report* verfasst. Die Tatsache, dass es die Verfasser nicht bei der Widerle-

21 Circa 14.400 Suchergebnisse liefert google.com: http://bit.ly/1M4dxfM [letzter Abruf: 08.03.2016].

22 http://www.neocomputer.org/projects/et/ [letzter Abruf: 04.03.2016].

gung falscher Zuschreibungen und der Dokumentation der technisch korrekten Sachverhalte belassen, sondern Hand an das historische Maschinencode-Artefakt legen, fügt sich in das methodologische Verständnis von Archäologie als „Redaktion“ [Ernst 2004:253f.] historischer Sachverhalte.

Dabei haben sich wichtige Erkenntnisse über die zur Zeit der Veröffentlichung von E.T. – The Extraterrestrial vorhandenen Möglichkeiten und Grenzen der Hardware und ihrer Programmierung gezeigt. Das Debugging und die Erweiterung des Codes um zusätzliche Routinen haben Speicherplatz erfordert, der auf dem nur 4 Kilobyte großen ROM-Chip nicht ohne weiteres verfügbar war. Richardson et al. mussten daher Teile des vorhandenen Programmcodes neu organisieren, komprimieren (*program bumming* [vgl. Levy 1984:31]) und Programmiertechniken nutzen, bei denen vorhandene Programmteile in eigene Routinen integriert wurden. Das daraus entstandene ‚gefixte‘ E.T. – The Extraterrestrial wurde von ihnen danach als ROM-Datei veröffentlicht. [Vgl. Höltgen 2015a]

Anders als die zeitgenössischen Programmierer von Atari-VCS-Spielen konnten sie bei ihrer Arbeit aber auf Werkzeuge zugreifen, die diese Aufgabe wesentlich erleichterten. Wurden Computerspiele aus Gründen der Stabilität und Sicherheit bereits zur Zeit von E.T. – The Extraterrestrial im damals vergleichsweise teuren Cross-Platform-Development-Verfahren entwickelt [vgl. Levy 1984:414], so finden Entwicklungen für historische Computer heute nahezu ausschließlich mit IDEs und Emulatoren auf modernen PCs statt. (Dass diese Praxis nicht immer problemlos für die Funktionalität derartig entwickelten Codes ist, zeigt [Fritz 2014] sowie Kapitel 5.3) Richardson et al. haben für ihre Arbeit den Emulator/Debugger Stella genutzt (vgl. Abb. 3.6), der es nicht nur erlaubt, den Code virtualisierter ROMs zu lesen und auszuführen, sondern auch zu disassemblieren. Dabei interpretiert der Disassembler aufgrund bekannter Eigenschaften der emulierten Hardware (Atari VCS) spezifische Adressen und weist sie konventionalisierten System-Variablen[23] zu.

Eine Besonderheit bei der Nutzung eines solchen Tools ist, dass es neben Step-by-step- auch Live-Debugging erlaubt: Während der Programmcode ausgeführt wird, kann er verändert und das Ergebnis der Veränderung am laufenden Programm in Echtzeit geprüft werden. Die Stella-Tools zum Ändern von Inhalten der Farbregister erlauben es beispielsweise, die Farbe von Spiel-Objekten live zu editieren. Während das Spiel auf der Oberfläche (dem Bildschirmfenster) abläuft und Spieleingaben über die Standard-Interfaces (Joysticks, Paddle, Schalter an der Spielkonsole) erwartet, kann der Stella-Nutzer mit dem Spiel auf der Unterfläche des Systems experimentieren und auf diese Weise die Atari VCS ‚spielend‘ programmieren lernen. Auf welche Weise diese spielerische Variante hardwarenaher Programmierung systematisch zum Programmierenlernen genutzt werden kann, wird in verschiedenen Tutorials (vgl. Trionfo 2012) beschrieben.

23 http://www.alienbill.com/2600/101/docs/stella.html#W [letzter Abruf: 06.03.2016].

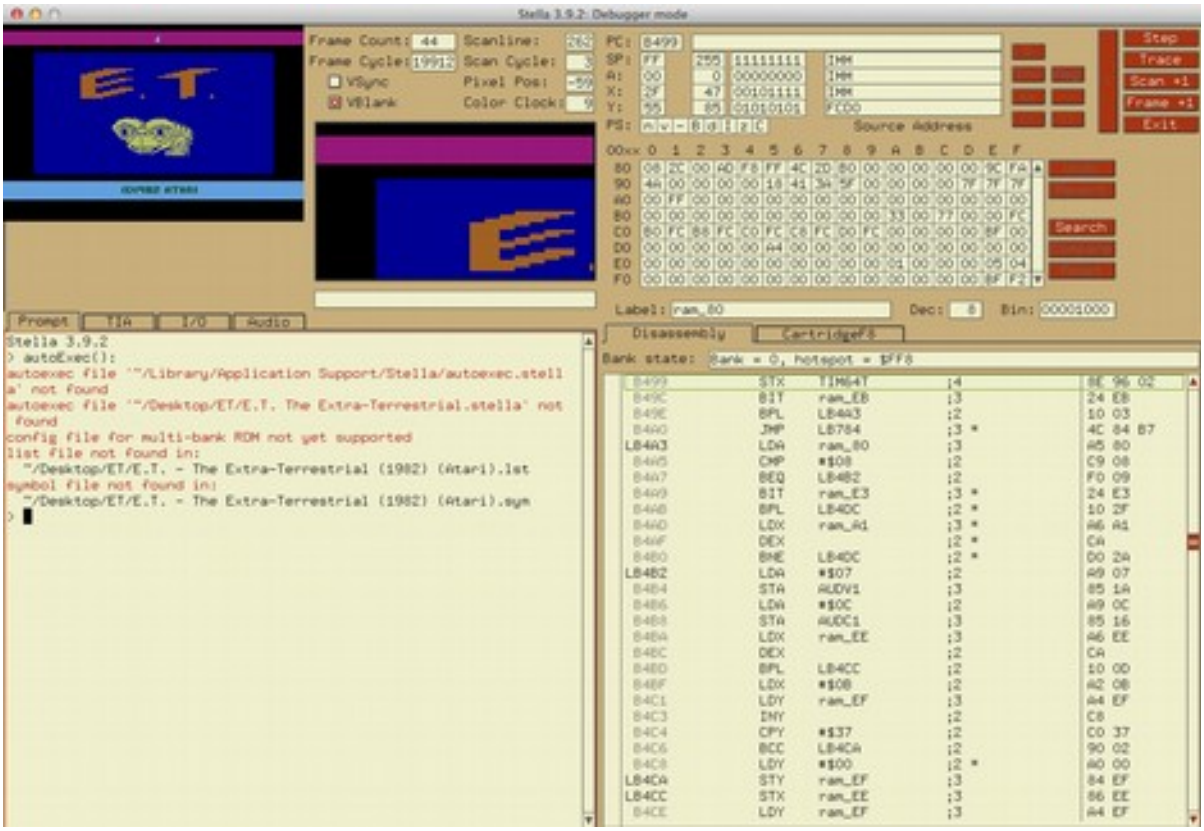


Abb. 3.6: Der Emulator/Debugger Stella mit geladenem E.T. – The Extraterrestrial

3.4 Zusammenfassung

Computerarchäologie stellt sich damit als Forschungsmethode zwischen einer kritischen Geschichtsschreibung, medienarchäologischer und informatischer Analyse digitaler Medientechnologien in Vergangenheit und Gegenwart dar. Die kritische Reflexion existierender Formen der Computerhistoriographie durch Kombination der White'schen Poetologie und der Collingwood'schen Aktualisierungstheorie historischen Denkens und Schreibens, kombiniert mit der Foucault'schen Machtanalyse und der medienarchäologischen Akzentuierung der medientechnischen Aprioris fordert gänzlich neue ‚Schreibweisen' von Computergeschichte. Detaillierte technische Sachverhalte müssen dabei ebenso berücksichtigt werden wie die Tatsache, dass sich operative Computer Historisierungsversuchen konsequent entziehen. Damit verliert ‚der Computer' (im Kollektivsingular) seinen Status als archäologisches Forschungsobjekt und auf diesem Begriff/Konzept basierende Theorien müssen als ‚zu weitreichend' von Theorien mittlerer Reichweite suspendiert werden. Das Methodenset der Informatik bietet hierfür das angemessene Instrumentarium, lassen sich mit seiner Hilfe doch natur-, ingenieur- und formalwissenschaftliche *Beschreibungen* jenseits subjektiver *Deutungen* finden. Die Dokumentation der so gewonnenen Forschungsergebnisse kann in Archäographien ‚kondensieren', wobei sich technische Prozesse selbst schreiben können (etwa mit Hilfe von Messinstrumenten) oder in diagrammatischen und formalsprachlichen Notationen erfasst werden.

Das folgende (zentrale) Kapitel dieser Arbeit stellt vier solcher Archäographien vor und entwickelt aus den daraus gewonnenen Erkenntnissen neue (computerarchäologische) Forschungsmethoden. Dies geschieht durch eine Kombination aus informatischen, medienepistemologischen, philologischen, zeichentheoretischen und didaktischen Ansätzen (siehe Tabelle 3.1).

Kapitel	**4.1**	**4.2**	**4.3**	**4.4**
Gegenstand	Demonstrationen	Computerspiel	Software Preservation	Hardware Preservation
Informatische Aspekte	Genealogie der Programmierung	Toy Computing	Retro Fitting	Autodidaktik und Gamification
Medienwissenschaftliche Aspekte	Computerphilologie	Simulation als Medienprozess	Emulation als Medienprozess	Medien und Wissen

Tabelle 3.1: Informatik und Medienwissenschaft des Retrocomputings

4. Retrocomputing als Archäographie

Im nachfolgenden Kapitel werden vier Projekte detailliert vorgestellt, die sich mit aktuellen Arbeiten zu frühen Mikrocomputern beschäftigen. Diese Projekte haben zwischen 2012 und 2018 am Lehrstuhl für Medientheorien des *Instituts für Musikwissenschaft und Medienwissenschaft* der *Humboldt-Universität zu Berlin* stattgefunden und folgen damit dem Vorschlag Takhteyevs/DuPonts [2013:431] einer Kooperation zwischen Institutionen (hier: dem *Signallabor* des o. g. Instituts) und privaten Hobbyisten: Zu einem Teil stellen die Projekte Ergebnisse und Weiterungen dortiger Lehrveranstaltungen dar und bringen eine spezifische ‚Retro-Didaktik' zur Anwendung; zum anderen Teil werden Aktivitäten dokumentiert und interpretiert, die während Ausstellungen, Workshops und der öffentlichen Laborpraxis von Hobbyisten im Signallabor stattfanden und zeigen computerarchäologische Überlegungen zum Retrocomputing.

Die vier ausgewählten Projekte lassen sich zu einigen der hauptsächlichen Betätigungsfelder der Retrocomputing-Szenen zählen. Sie befassen sich mit:

- dem *Retro Demo Coding* (4.1),
- den *Computerspielen und ihrer Geschichte* (4.2),
- Praktiken der *Software Preservation* (4.3),
- Praktiken der *Hardware Preservation* (4.4).

Die Darstellung der vier Projekte erfolgt vor dem zuvor skizzierten theoretischen Hintergrund der *Computerarchäologie* und im Modus von *technical reports*. Die vier Analysen versuchen zu zeigen, wie sich nondiskursive Quellen (Codes, Prozesse, Architekturen) für eine Interpretation und daraus aufbauende technik- wie kulturhistorische Einordnung fruchtbar machen lassen. Dazu werden Historeme mit Hilfe informatischer Methoden detailliert (beispielsweise spezifische Codefragmente) untersucht, die Ergebnisse in informatische Fachdiskurse (im Beispiel: die historische und kontemporäre Programmierlehre) eingeordnet und medienwissenschaftlich als Episteme (im Beispiel: Unterschiede menschlicher und automatischer Lektüre formaler Sprachen) gedeutet. Jedes Unterkapitel enthält daher sowohl eine informatisches als auch eine medienwissenschaftliches Fazit der jeweils vorausgegangenen Analyse, mit denen Retrocomputing in einen wissenschaftlichen Diskurs überführt werden kann.

Im Ergebnis soll sich zeigen, dass Retrocomputing wesentliche Aspekte hobbyistisch erworbenen informatischen Wissens mit Praktiken des Hackings kombiniert. Dabei wird neben der Arbeit am Gegenstand auch eine ‚Arbeit am Wissen' betrieben, die stets implizite geschichtskritische, epistemologische und anderweitig theoretische Effekte besitzt. Diese sollen als Deutungsangebote den einzelnen Projekten hinzugefügt werden.

4.1 Demonstration, Simulation und Codierung

Ein integraler Bestandteil des Retrocomputings ist die Erstellung neuer Software für historische Computerplattformen. Hierzu zählen neben Spielen, Anwenderprogrammen und sogar Betriebssystemen (vgl. Kap. 5.4.2) vor allem so genannte Demos. Dabei handelt es sich um Programme, die die technischen Möglichkeiten einer Plattform als audiovisuelle ‚Leistungsschau' vor Augen führen sollen. Mit Demos zeigen die Programmierer aber nicht allein ihre Erfahrungen im Programmieren und der Beherrschung der jeweiligen Plattform, sondern sie schreiben mit ihren Elaboraten auch Computergeschichte – und zwar nicht als Diskurs, sondern als Code. Diese Form der ‚Geschichtsarbeit' ruft (im Sinne des *Re-Enactment*) historische Elemente auf (Algorithmen, Routinen, Programmiertricks, ...) und aktualisiert sie. Wie jede Programmierung bedeutet also auch Demoprogrammierung ein Sich-in-Bezug-setzen zur Softwaregeschichte. Durch das Aufgreifen bestimmter Motive oder Genres und mehr noch die Überarbeitung historischer Codes zu neuen Demos erhält die Demo-Programmierung jedoch einen sehr bewussten historiografischen Akzent.

Ähnlich wie frühe Computerspiele sind Demoprogramme sehr eng auf eine Hardware (ihre Begrenzungen, Möglichkeiten, Schnittstellen usw.) zugeschnitten. Insofern spielt bei der Programmierung von Demos die bezogene Hardware eine zentrale Rolle. Demoprogrammierung ist oft sogar eine Methode der Hardware-Exploration. Nicht selten werden über die Programmierung von Demos neue Features in historischen Computerhardwares entdeckt, die weder offiziell dokumentiert wurden noch den Ingenieuren und Erbauern bekannt waren [vgl. Steil 2014:134-138]. Die Hardware spielt aber noch eine weitere Sonderrolle bei Computerdemos: Aufgrund der engen Beziehung des Programms zur Plattform ist das vollständige Verständnis einer Demo nur möglich, wenn diese auch operativ ist. Diese Notwendigkeit zeigt sich umso deutlicher, je begrenzter die Möglichkeiten der Hardware sind und je explorativer die Programmierer daher bei der Erstellung ihrer Programme vorgehen müssen. Die *Demonstration* zeigt sich daher bei Computerdemos (sozusagen *expressis verbis*) besonders vordergründig als Erkenntnisinstrument.

Das folgende Kapitel stellt eine Reihe solcher, einander ähnelnder Demoprogramme vor und diese in ihren spezifischen Differenzen einander gegenüber. Es handelt sich dabei um Implementierungen von Ballsprung-Demonstrationen. Untersucht werden hierbei die hardware- und softwarespezifischen Möglichkeiten und Grenzen der jeweils bezogenen Systeme und die Anforderungen der Entwickler, einen solchen physikalischen Prozess als Programm zu kodieren. Ziel des Vergleiches ist die Anreicherung historiografischer Diskurse durch eine Archäographie der Hard- und Software. Hierzu wird der Vergleich zwischen unterschiedlichen Implementierungen (also Code oder Schaltung) als *computerphilologische Analyse* durchgeführt.

Da die zumeist durch Hobbyisten vorgenommene Entwicklung von Computerdemos, wie geschrieben, stark explorativen Charakter besitzt, kann Demo Coding auch als ein Aspekt des learning by doing dargestellt werden:

> Demoprogrammierung ist als grundsätzlich zweckfreie, eigenmotivierte, hardwarespezifische, nicht-kommerzielle und nur auf sich selbst verweisende Leistung die womöglich freieste aller Nutzungsarten eines Computers und demonstriert auf diese weise die Unterwanderung sämtlicher Kontrollmöglichkeiten des Mediums. [Botz 2011:393]

Dabei

> wird eine Relation zwischen dem spielerischen Ausprobieren, dem daraus resultierenden tiefen Verständnis und der Kontroll-Idee deutlich. In den Handlungsweisen der Demoszener zeigt sich ein deutlicher und zugleich diachroner Zusammenhang dieser Trias. [Hartmann 2017:176]

Auf diese Weise wird implizites Wissen der phänomenalen Schau („Wie kann ich etwas programmieren, das so aussieht, wie das, was ich hier sehe?“) explizi(er)t. Diese Explikation findet also zwischen zwei Zeichenkategorien (Index – Symbol) als Versprachlichung bzw. Codierung statt.

Mit Verweis auf Jean-François Lyotards Essay *Das Postmoderne Wissen* konstatiert Claus Pias,

> daß es einen dichten Zusammenhang gibt zwischen dem Spielen von Computerspielen und dem Spielen von Sprachspielen, zwischen dem Spiel des Hackers und dem Spiel des Users; zwischen Herstellen und Konsumieren; zwischen dem Schreiben von Programmen und dem (lesenden) auf Information – einen Zusammenhang, dem unterschiedliche ‚Zukünfte des Computers‘ entspringen. [Pias 2005b:216]

Lyotard hatte 1979 in seinem erwähnten Bericht [Lyotard 2012] festgestellt, dass Computer Ende der 1970er-Jahre die Art, wie Wissen in der so genannten Postmoderne erworben, gespeichert und proliferiert wird, maßgeblich bestimmen und künftig bestimmen werden. Pias unternimmt den Versuch, die besondere Qualität des Wissens und seines Erwerbs in der Computer-Kultur zu untersuchen. Als Ausgangspunkt wählt er hierfür die Geschichte einer Computer-Demonstration: des springenden Balls und ihrer Interaktivierung als (Tennis-)Computerspiel. [Vgl. Maher 2012:12; Pias 2005b:218-222]

Dieses Motiv wählt Pias vor allem deshalb, weil es für ihn einen Ausgangspunkt der Computerspielgeschichte und der Hacker-Bewegung darstellt, das heißt: weil der historische Ursprung dieser Demonstration und seiner Fortsetzungen stets mit dem ‚Missbrauch‘ von professioneller Hardware zu Spielzwecken, dem Willen zum Wissen und autodidaktischer Aneignung desselben verbunden war. Von einem Demoprogramm für den Compu-

ter AN/FSQ-7 aus dem Jahre 1950/51 [Pias 2005b:218] über das frühe Computerspiel Tennis for Two von 1958 [vgl. ebd.], entstanden am *Brookhaven National Laboratory,* und dem zwei Jahre später programmierten Tennisspiel auf den Kontrolllampen eines IBM 704 am MIT [vgl. ebd. 119] bis hin zum Spiel Pong [vgl. ebd.:221f.], das Nolan Bushnell und Alan Alcorn (die später die Firma *Atari* gründeten) 1972 zeitgleich mit einem ähnlichen Table-Tennis-Spiel von Ralph Baer [vgl. ebd.:222-227] für dessen Videospielkonsole Odyssey entwickelte, verfolgt Pias das Motiv. Dabei zeichnet er die medienhistorische und -kulturelle Oberfläche dieser Entwicklungen nach, basiert seine Beschreibungen der diesen zugrunde liegenden Technologien jedoch *rein phänomenologisch auf ihren Schnittstellen (Ein- und Ausgaben)*. Außer in der Benennung einiger Computermodelle und typischer Hardware-Besonderheiten bleibt seine Analyse dabei aber technikfern.

Das folgende Kapitel will deshalb auf eine Zusatzperspektive hinweisen, die sich ergibt, wenn die Analyse der technischen Prozesse, welche den mannigfaltigen Implementierungen springender Bälle zugrunde liegen, mit in die Historiografie einfließt. Dabei soll diese im computerarchäologischen Sinne durch eine Archäographie ersetzt werden, um so zeigen zu können, welche Interdependenzen sich zwischen den einzelnen, oft Jahrzehnte auseinander liegenden Implementierungen des *augenscheinlich selben* Phänomens offenbaren. Aus der heutigen Perspektive des Retrocomputing führt die Hacker-Kultur deshalb auch nicht notwendigerweise [vgl. Pias 2005b:239 und Pias 2015:41] in die Apokalypse [vgl. White 119:114] am Ende aller „Zukünfte des Computers“ (so der Titel seines Sammelbandes, in welchem Pias den Hacker-Text [Pias 2005b] veröffentlicht). Springende Bälle sind auch noch nach seiner Prognose und nicht bloß aus historischer oder nostalgischer Rückbesinnung von Hackern programmiert worden.

Die folgende Analyse orientiert sich allerdings nur kursorisch an den Kapiteln von Pias‘ Darstellung. Neben den bei ihm erwähnten Tennis for Two werden ähnliche Programme untersucht, für:

- die Atari-VCS-Spielkonsole, für welche zwischen 1999 und 2013 zwei Ballsprung-Demonstrationen entstanden,
- die Computerplattformen Commodore Amiga (1984) und Amstrad CPC (1985),
- die im Rahmen von Lehrveranstaltungen implementierten Re-Enactments für den Lerncomputer Signetics Instructor 50 (2014)[24] und die Modifikation der Ball-im-Kasten-Simulation für den Analogcomputer Telefunken RA-742 aus dem Jahre 2012 [vgl. Höltgen et al. 2012].

Dabei sollen, transversal zu den von Pias aufgeführten historischen Motiven verlaufend, zunächst folgende technische Fragen aufgeworfen und beantwortet werden:

24 Vgl. http://www.simulationsraum.de/blog/2016/05/24/tennis-2650/ [letzter Abruf: 24.05.2016].

1. Wie unterscheidet sich die theoriegeleitete Implementierung physikalischer Vorgänge durch Spezialisten von heuristischen und experimentellen Implementierungen der Hobbyisten? Worin ähneln sie sich und wie lässt sich diese Ähnlichkeitsbeziehung klassifizieren?
2. Auf welche Weise werden die Möglichkeiten der unterschiedlichen Hardwares durch Algorithmen gezielt zur Evokation von Bewegung ausgenutzt? Wie werden deren Begrenzungen durch Hacking-Verfahren überwunden?
3. Welche Rolle spielen die Programmiersprachen und ihre Möglichkeiten bei der Implementierung?

Zuerst erfolgt eine Beschreibung des durch Computer zu simulierenden Phänomens; danach werden die einzelnen Implementierungen vorgestellt und abschließend in einen theoretischen Kontext gestellt.

4.1.1 Analogien

Die Darstellung eines springenden Balls muss, um vom Betrachter als ‚realistisch' wahrgenommen zu werden und dem in der Natur beobachtbaren Verhalten eines geworfenen Balls zu ähneln – aber erst recht, um den Anforderungen einer physikalischen Simulation zu genügen, spezifische Bedingungen erfüllen. Hierzu sind unterschiedliche Parameter zu berücksichtigen: Der Ball muss wie eine schwere Masse den Gesetzen der *Gravitation* folgen, sein *Impuls*, seine *Geschwindigkeit(sänderung)*, sein *Ort* und seine *Richtung* sind dabei von der *Kraft des Wurfs*, der *Gravitationskraft* sowie verschiedenen *Dämpfungsfaktoren* abhängig. Neben der Simulation des *Vorgangs* und den daraus resultierenden Folgebewegungen ist der Ball selbst ebenfalls als physikalisches *Objekt* zu modellieren: Er hat die Materialeigenschaften *Masse, Elastizität* und *Reibung*. Diese Eigenschaften kommen ebenso dem Material zu, von dem der Ball beim Sprung abprallt. Je nach Anspruch werden diese Faktoren bei der Implementierung genau oder weniger genau berücksichtigt (und zweitweise sogar unberücksichtigt gelassen).

Mit der Anzahl und Genauigkeit der berücksichtigten Parameter wächst die Komplexität der der Implementierung. Die physikalischen Verhältnisse des Vorgangs sind bekannt und in der Theorie der Kinematik mathematisch beschrieben [vgl. etwa Holzmann u.a. 2010]; sie können daher durch ein Programm dargestellt werden. Abhängig ist die Beschreibungsgenauigkeit dann noch von den Spezifikationen der Zielplattform und vom Anspruch, den der Programmierer mit seiner Implementierung verfolgt. Insbesondere bei der Programmierung von grafischen Demonstrationsprogrammen zeigen die nachfolgend untersuchten Beispiele nicht nur starke Abweichung in der Anzahl der berücksichtigten Parameter; zeitweise wird bei der Entwicklung der Programme ein geringeres Augenmerk auf physikalische Korrektheit (etwa die Adaption der korrekten Gesetzmäßigkeiten) als auf den zu erzielenden visuellen Effekt gelegt. Insbesondere hierin offenbaren

sich die deutlichsten Unterschiede und zugleich die markantesten epistemischen Brüche zwischen den Implementierungen. Die folgende Aufführung historischer Programme und Schaltungen auf historischen Plattformen ist geeignet, die unterschiedlichen Intentionen, Beschränkungen und Verfahren zu kennzeichnen, mit denen die Simulationen des oben genannten physikalischen Vorgangs realisiert wurde und wird.

Die Demonstrationsprogramme sind, wie jede Computersoftware, *erst und nur im Vollzug auf der Original-Hardware* vollständig analysierbar. Wie in der Forschungsliteratur zur Demoszene dargestellt [vgl. Botz 2011:289-302, Hartmann 2017:144-200], stellen Foto- oder Filmaufnahmen von Demo-Programmen einen Widerspruch zur Intention dieser Softwaregattung dar, welche Echtzeitberechnungen für eine audiovisuelle Ausgabe vollziehen und dabei die Eigenzeitlichkeit der Hardware ausnutzen und vorführen soll. Aus demselben Grund kann eine alleinige Betrachtung von Demos in einem Emulator ebensowenig Grundlage für ihre Analyse sein, weil diese in zeitkritischen Momenten allenfalls die Performanz der Emulationssoftware und des Host-Systems vorführt – so wie ein Video die Computerausgaben dem zeitlichen Regime des Video-Players unterwirft. Die nachfolgende Untersuchung basiert aus diesem Grund auf den Ausgaben der jeweils untersuchten Hardware-Software-Systeme. Abbildungen müssen unter dieser Prämisse betrachtet vor allem als Illustrationen angesehen werden.

4.1.1.1 Ball im Kasten

Den Ausgangspunkt für die Betrachtung bildet die (augenscheinlich) am genauesten an die physikalischen Gegebenheiten angepasste Demonstration Ball im Kasten. Sie gilt mutmaßlich als die Grundlage für das 1958 implementierte Spiel Tennis for Two (vgl. Abb. 4.1.1).

Die Analyse von Tennis for Two wurde im Rahmen einer Lehrveranstaltung als Re-Enactment vollzogen. Im Wintersemester 2011/2012 fand das BA-Seminar „It's more fun to compute! – Computer & Spiel / Analog & Digital / Theorie & Praxis“[25] statt, in welchem die historischen und epistemologischen Besonderheiten der Softwaregattung Computerspiel thematisiert wurden. Das Seminar wurde nach einer kurzen Einführungsphase in zwei Gruppen aufgeteilt: Die eine begann mit der Programmierung eines Digitalcomputerspiels [vgl. Höltgen 2016e]. Die andere Studenten-Gruppe programmierte das Spiel Tennis for Two als Re-Enactment auf dem Analogcomputer Telefunken RA-742 (ca. 1970). Als Grundlage hierfür dienten die Ballsprung-Simulation Ball im Kasten[26], die so verändert

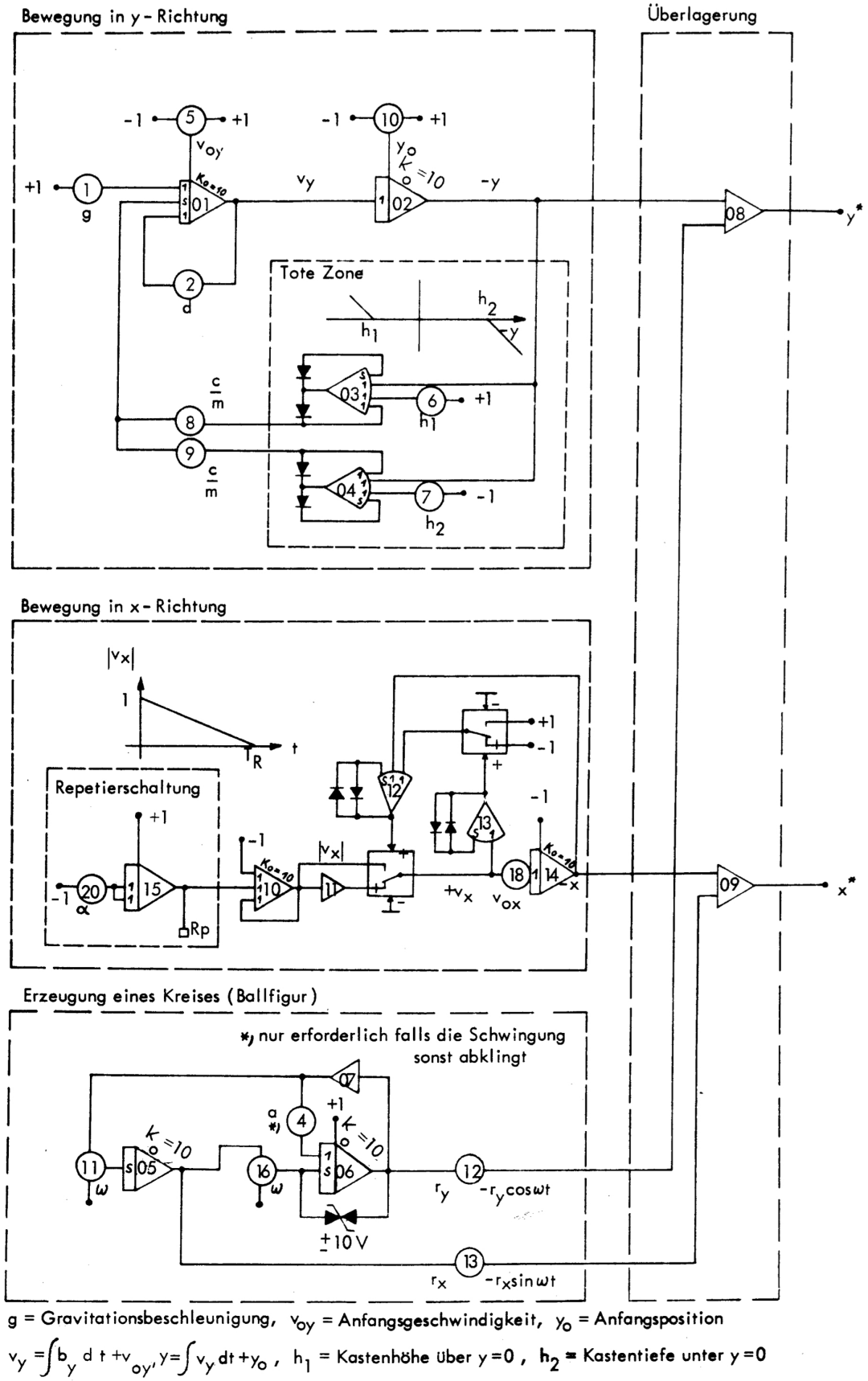
25 https://agnes.hu-berlin.de/lupo/rds?state=verpublish&status=init&vmfile=no&publishid=43852&moduleCall=webInfo&publishConfFile=webInfo&publishSubDir=veranstaltung [letzter Abruf: 06.06.2019].

26 Das Demonstrationsbeispiel Nr. 5: Ball im Kasten für Analogrechner entstammt einem Lehrgangshandbuch für Analog- und Hybridrechner, das um 1969 von der Firma AEG/Telefunken herausgegeben wurde. Auf neun Seiten werden darin die Beschaltungen der Analogrechner Telefunken RAT-700, RA-741 und

wurde, dass der Ball durch Interaktion seine Bewegungsrichtung und -geschwindigkeit ändert. Die Ausgangsimplementierung soll hier zunächst in ihren Spezifikationen beschrieben werden. Der in Abb. 4.1.1 dargestellte Prinzip-Schaltplan bildet die Programmiergrundlage.

Beim Ball im Kasten handelt es sich um ein typisches Demonstrationsprogramm für Analogcomputer, das in Manuals [vgl. Heath 1959:29f.] und Fachbeiträgen [vgl. Pfalzgraff 1969; Ulmann 2017] vorgestellt wird. Es wurde im Handbuch des TA-742 „als Werbemaßnahme veröffentlicht [..], um die Leistungsfähigkeit der Telefunken-Tischanalogrechner zu demonstrieren und gleichzeitig Programmiertricks einer breiteren Öffentlichkeit, hierunter nicht zuletzt auch Bereichen der Forschung und Lehre, d.h. potentiellen Neukunden bekannt zu machen.“ [Ulmann 2010:164f.] Das Programm simuliert das Verhalten eines Balls, der mit einer bestimmten Anfangsgeschwindigkeit in einen Kasten geworfen wird, wo er von den Wänden, dem Boden sowie dem Deckel elastisch abprallt, durch die Gravitation beschleunigt und durch den Luftwiderstand (sowie die Federkonstanten) gedämpft wird.

RA-742, RA-770 sowie RA-800-HYBRID vorgestellt, die mit leistungsstarken und präzisen Analogrechenelementen und einer variablen Anzahl von Operationsverstärkern (bis zu 35 beim RA-800-HYBRID) ausgestattet sind.



g = Gravitationsbeschleunigung, v_{oy} = Anfangsgeschwindigkeit, y_o = Anfangsposition

$v_y = \int b_y \, dt + v_{oy}$, $y = \int v_y \, dt + y_o$, h_1 = Kastenhöhe über y = 0, h_2 = Kastentiefe unter y = 0

Abb. 4.1.1: Prinzipschaltplan zum Ball im Kasten [AEG/Telefunken o.J.a:5]

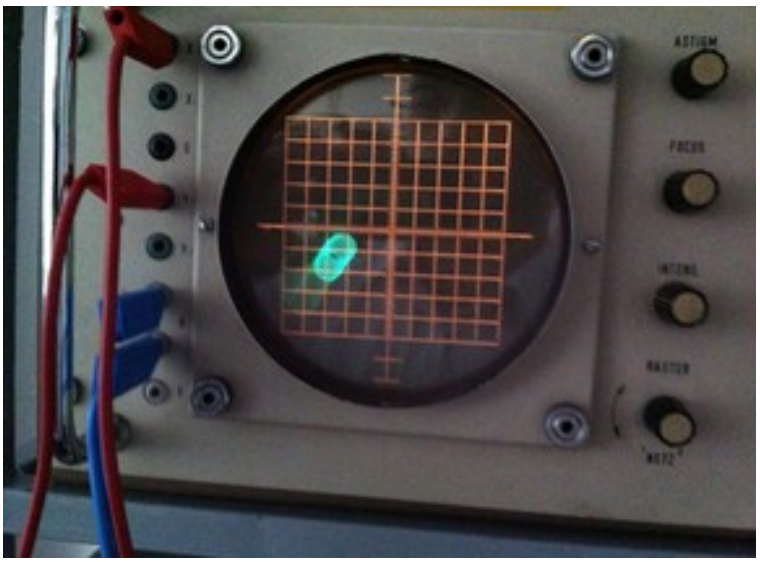


Abb. 4.1.2: Oszilloskop-Darstellung vom Ball im Kasten (In der Momentaufnahme zeigt sich der Ball als Spirale, die aufgrund ihres kurzen Nachleuchtens als bewegter Kreis erscheint.)

Dort, wo das Demo-Programm vorgestellt wurde,

> führt die vorliegende Behandlung des in einer Kiste eingeschlossenen, springenden Balles die hohe Interaktivität und Anschaulichkeit einer analogen Rechnung vor Augen. Da alle die Bewegung bestimmenden Koeffizienten wie Dämpfung, Erdbeschleunigung etc. mit Hilfe von Koeffizientenpotentiometern in die Rechenschaltung eingehen, können diese auch ohne weiteres im Verlauf einer Rechnung manuell geändert werden, so dass hieraus resultierende Effekte sofort an der Bewegung des Balles deutlich werden. [Ulmann 2010:169]

Die Demonstrations-Qualitäten für Maschine, Ausbildung und beispielhafte Physik-Implementierung werden auch andernorts in der Literatur hervorgehoben: „A problem which is interesting to watch, as well as one which illustrates the more complex type of problem that the computer is capable of handling“ [Heath 1959:28] oder: „the usefulness of the analog computer as a demonstration tool for undergraduate students“ [Pfalzgraff 1969:1011f.].

Zu eben jenen Demonstrationszwecken und aufgrund technischer Beschränkungen vereinfacht die Simulation im Analogcomputer den physikalischen Prozess auf vielfache Weise:

1. „Der Vorgang wird als ebenes Problem betrachtet.“ [AEG/Telefunken o.J.a:1] - das heißt, er findet zweidimensional (in Seitenansicht) statt. Dies ist nicht allein auf die Begrenzung des Ausgabemediums Oszilloskop zurückzuführen, sondern auch auf die notwendigerweise gering zu haltende Komplexität der Rechenschaltung. (Hierin findet sich ein erstes ‚Indiz‘ dafür, warum die Computerspiel-Geschichtsschreibung, Ballsprung-Simulationen in einen Motiv-Komplex integriert: Sie alle bieten zweidimensionale Ansichten des Spielfeldes.)

2. Während der Kasten auf dem Oszilloskopschirm nicht dargestellt und nur indirekt durch seine Einwirkungen auf die Bewegungsänderungen des Balls erfahrbar wird, ist der Ball als *Kreis* realisiert. Der durch die Schaltung berechnete Ball ist allerdings ein idealisierter „center of the ball (the assumed center of mass)“ [Pfalzgraff 1969:1008].
3. Die Kreis-Gestalt selbst wird durch die Auslenkung des Kathodenstrahls auf Basis einer Schwingungsfunktion erreicht, wobei die Frequenz der Schwingung so hoch gewählt werden muss, dass „für eine möglichst flimmerfreie Darstellung“ (Ulmann 2010:166) „die schnelle Kreisbewegung“ [AEG/Telefunken o.J.a:1] die Distinktionsgrenzen der Wahrnehmung unterläuft (beziehungsweise die Nachleuchtzeit der Kathodenstrahlröhre soweit ausnutzt), „daß als stehendes Bild ein Kreis entsteht.“ [Ebd.:1]
4. Ball und Kasten haben eine Höhe und eine Breite, aber keine Tiefe. Dies hat nicht nur Auswirkungen auf die Darstellung des Balls, der immer gleich dargestellt wird, weil er während der Simulation nicht perspektivisch verlagert oder beim Abprallen verformt wird[27], sondern auch auf die physikalische Simulation, die nicht von einer Reibungs*fläche* zwischen Ball und Kastenboden/-wänden/-deckel ausgeht, sondern lediglich von einer Tangentialberührung.
5. Ebenfalls wurde „zur Vereinfachung der Schaltung die Dämpfung als Näherung proportional der Geschwindigkeit angenommen“ [ebd.:3] sowie die „Beschleunigungsänderung durch den elastischen Stoß [...] proportional angenommen“ und beides lediglich auf die vertikale Bewegung appliziert. Ähnliches gilt für den Luftwiderstand, welcher den Ball hier nur in seiner Horizontalbewegung bremst. Auch bei der „Bewegung des Balles in x-Richtung wurde für die Schaltung als Näherung angenommen, daß der Betrag der Geschwindigkeit linear mit der Zeit abnimmt.“ [Ebd.]

Der offensichtliche Grund für die meisten der genannten Einschränkungen ist die Reduktion der Schaltungskomplexität. Auf dem Telefunken RA-742 stehen 23 Operationsverstärker (8 Integrierer/Summierer, 7 Summierer, 4 Invertierer/Summierer, 4 Invertierer) und 19 Koeffizienten-Potentiometer [Vgl. AEG/Telefunken o.J.b] für Rechenschaltungen zur Verfügung. Die vorliegende Ball-im-Kasten-Demonstration verwendet 13 Operationsverstärker und 14 Potentiometer. Eine viel komplexere Schaltung wäre daher gar nicht implementierbar gewesen. Die Rechenschaltung stellt mithin ein Kompromiss zwischen den Hardware-Restriktionen und der Heuristik der Implementierung dar: Die schätzungsweise wichtigsten Parameter der Simulation, das heißt: diejenigen, die das Erkennen des simulierten Vorgangs ermöglichen, haben Eingang in die Schaltung gefunden.

27 Die Implementierung von Pfalzgraff berücksichtigt die Verformung des Balls beim Aufprallen auf Boden und Deckel. [Vgl. ebd.:1011f.]

Dennoch simuliert das Demonstrationsprogramm in zweierlei Hinsicht den physikalischen Prozess adäquater als das etwa zeitgleich entstandene Digitalcomputer-Programm für den IBM 704 [vgl. Pias 2005b:218]: *Erstens* liefert das Programm BALL IM KASTEN, aufgrund der Tatsache, dass Analogcomputer keine diskreten Daten in Programmen verarbeiten, sondern kontinuierliche Spannungswerte in elektronischen Schaltungen als Analogien zu den zu simulierenden Prozessen verändern, wesentlich genauere Resultate als das IBM-Tennisprogramm. *Zweitens* ermöglicht es jene Analogie-Bildung mathematische Prozesse, die in Digitalcomputern vergleichsweise aufwändig symbolisch kodiert werden müssten, in Teilschaltungen zu *real*(-)isieren. Insbesondere die für die vorliegende Simulation des schiefen Wurfs nötigen Differenzialgleichungen lassen sich leicht auf Analogcomputern einrichten.

Die horizontale Auslenkung des Kathodenstrahls beruht auf physikalischen Annahmen (angenommene Dämpfung durch Luftreibung) sowie der Änderung der Bewegungsrichtung durch Vorzeichenumkehr (mit Hilfe der Komparatoren). Die Bewegung des Balls in vertikaler Richtung wird im Programm mathematisch bestimmt durch:

Weg: $y = \int v_y \, dt + y_0$

Geschwindigkeit: $v_y = \int a_y \, dt + v_{oy}$

mit Beschleunigung: $a_y = -g + dv_y + (c/m) \bullet (|y|-h_2)$ für: $y<h_2$

bzw. Dämpfung: $a_y = -g + dv_y - (c/m) \bullet (y-h_1)$ für: $y>h_2$

[Vgl. AEG/Telefunken o.J.a:2]

Diese Gleichungen beschreiben das Weg-Zeit-Gesetz sowie dessen Ableitungen zur Ermittlung von Geschwindigkeit, Weg und Ort. Die Variablen g (Gravitationskonstante), h_1 (oberes Kastenende), h_2 (unteres Kastenende), v_0 (Anfangsgeschwindigkeit, aufgelöst in v_{0x} für die horizontale und v_{0y} für die vertikale Anfangsgeschwindigkeit), c/m (die Federkonstante dividiert durch Masse) sowie d (als Dämpfungskonstante) werden über die Koeffizientenpotentiometer des Analogcomputers voreingestellt und sind zur Laufzeit änderbar. Hierbei zeigt sich bereits eine Eigenart des Analogcomputers: Alle Funktionen werden gegen die Variable t (Maschinenzeit) abgeleitet. Sie ist die einzige Veränderliche, die nicht über die Koeffizientenpotentiometer vorgegeben wird.

Die Auflösung in zwei Raumdimensionen ermöglicht es, die x- und y-Auslenkung des Kathodenstrahls durch zwei voneinander unabhängige Funktionen zu verwirklichen (in Abb. 4.1.3 links-oben und links-mitte). Beide daraus entstehenden Werte r_x und r_y werden mit Hilfe einer Integrierer-Schaltung und der daraus realisierten Differenzialgleichung[28]

28 Der Wert für den Kreisradius wird aus den Koeffizienten r_x und r_y generiert, die über die Potenziometer 12 und 13 voreingestellt werden. Die Zener-Diode kompensiert die Ungenauigkeiten des beteiligten Integrierer-Verstärkers, der Werte minimal kleiner oder größer als 1 liefert, welche sich im Feedback-Prozess „aufschaukeln". Die Diode begrenzt den Wertezuwachs/-schwund auf den von den Potentiometern vorgegebenen Wert.

$$y'' + \omega^2 y = 0$$

für eine Kreisfunktion aus Sinus und Kosinus von ω genutzt (im nachfolgenden Bild links-unten):

$$y = r_y \bullet \cos(\omega t) \text{ sowie } x = r_x \bullet \sin(\omega t),$$

deren Ausgabe schließlich in einer Summierer-Schaltung zu y_t und x_t hinzu addiert wird (in Abb. 4.1.3 rechts):

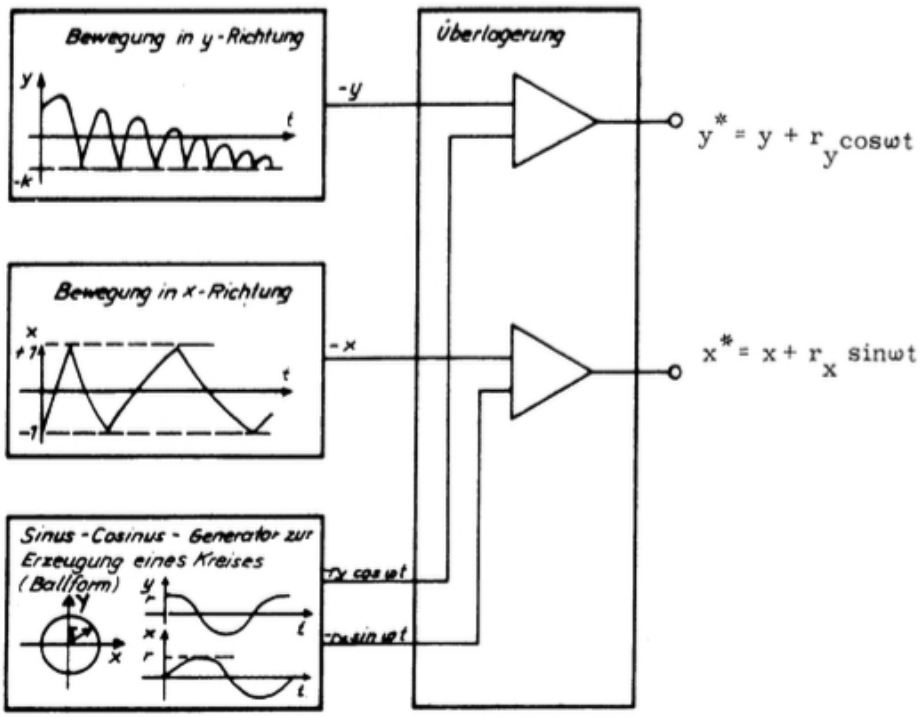


Abb. 4.1.3: Separate Funktionen der Schaltung [AEG/Telefunken o.J.a:2]

Der Analogcomputer muss mithilfe seiner eingebauten Funktionen diese mathematischen Gleichungen ‚numerisch' lösen. Zur Verfügung stehen ihm dabei in dieser Simulation die Möglichkeiten, Funktionswerte zu integrieren, zu summieren, Vorzeichen umzukehren oder zu ignorieren (Beträge zu bilden). Der Wertebereich richtet sich dabei nach der so genannten Maschineneinheit, hier also den Gleichspannungsgrenzen von +10 Volt und -10 Volt. (Daraus ergibt sich schon, dass eine Beschleunigung mit der Gravitationskonstanten 9,81 m/s^2 lediglich proportional nachgebildet werden kann).

Insbesondere seine Fähigkeit zur Lösung von Infinitesimalgleichungen hat die Geschichte und Entwicklung des Analogrechners beeinflusst. Im Ball im Kasten generiert er aus der Integration der Momentan-Beschleunigung die Momentan-Geschwindigkeit und daraus wiederum durch Integration den Momentan-Ort des Balls. Die mathematische Beschreibung der implementierten physikalischen Gesetze ermöglicht es, diese Gesetze in der Rechenhardware als Schaltung abzubilden.

Zunutze gemacht wird sich hierfür vor allem das Ladungs- und Entladungsverhalten von Kondensatoren, die in der Rückführung des Ausgangs auf den (invertierenden) Eingang eines Operationsverstärkers geschaltet werden. Ein derart beschalteter Rechenverstärker „ist sozusagen bemüht, die Summe der am Knotenpunkt des invertierenden Eingangs zusammentreffenden Ströme, die sich zum einen aus der Summe der durch die Eingangswiderstände [...] fließenden Ströme sowie zum anderen dem durch den Kondensator C fließenden Strom ergeben, zu 0 werden zu lassen.“ [Ulmann 2010:98] Die Zeit (t), die einzige mögliche Variable in einem Analogrechner, wird durch den Integrierer in die Gleichung gebracht: „Der Integrierer ist das einzige Rechenelement eines elektronischen Analogrechners, in dessen Ergebnis die Zeit als freie Variable Eingang findet - letztlich bestimmt der Integrierer den zeitlichen Verlauf einer jeden auf einem elektronischen Analogrechner durchgeführten Rechnung.“ [Ulmann 2010:99] Die Echtzeitberechnung des Balls im Kasten bezieht sich also auf das Zeitverhalten seiner Integrierer.

An der Implementierung zeigt sich bereits, dass der Analogrechner die physikalische Gegebenheit in ein zeitvariantes *Diagramm* überträgt und deshalb typischerweise auf jene Aspekte reduziert, die von Interesse sind (oder sich als ‚Demonstrationsbeispiel‘ leicht in einer Schaltung integrieren lassen). Wie Diagramme, so produziert auch der Analogrechner *Ikons* [vgl. Peirce 1983:64ff.] – insbesondere den kreisförmigen Ball und seine Bewegungsveränderung (die auf Basis der implementierten mathematischen Funktionen in Echtzeit stattfinden). Eine direkte Beziehung (Proportion) besteht zwischen der zugrunde liegenden Physik und ihrer Simulation in ihrer zeit- und parameterabhängigen Wertebeeinflussung.

Die Anzeige des Oszilloskops wiederum steht in einer proportionalen Beziehung zur Spannungsveränderung der Analogrechner-Ausgänge: Sind diese am Ausgang des Operationsverstärkers Nr. 2 niedrig, so wird der Kathodenstrahl zum unteren Ende des Oszilloskopschirms ausgelenkt, sind sie hoch, dann wandert er nach oben – lediglich ‚abgelenkt‘ durch die aufaddierte Kreis-Funktion für die Balldarstellung. Ähnliches gilt für die horizontale Auslenkung: Der Kathodenstrahl bewegt sich von Links nach Rechts, wenn die Spannung am Operationsverstärker Nr. 14 positiv ist, in die andere Richtung bei negativer Spannung. Die *Illusion der Simulation* wird hier also einerseits erst durch diagrammatische Reduktion möglich (dadurch, dass nur die ‚wesentlichen Eigenschaften‘ einer solchen Bewegung berücksichtigt werden), andererseits durch die proportionalen Beziehungen zwischen den Spannungsverhältnissen in der Rechenhardware und der elektromagnetischen Auslenkungsqualität und -quantität in der Oszilloskop-Bildröhre.

4.1.1.2 Tennis for Two und Tennis für Drei

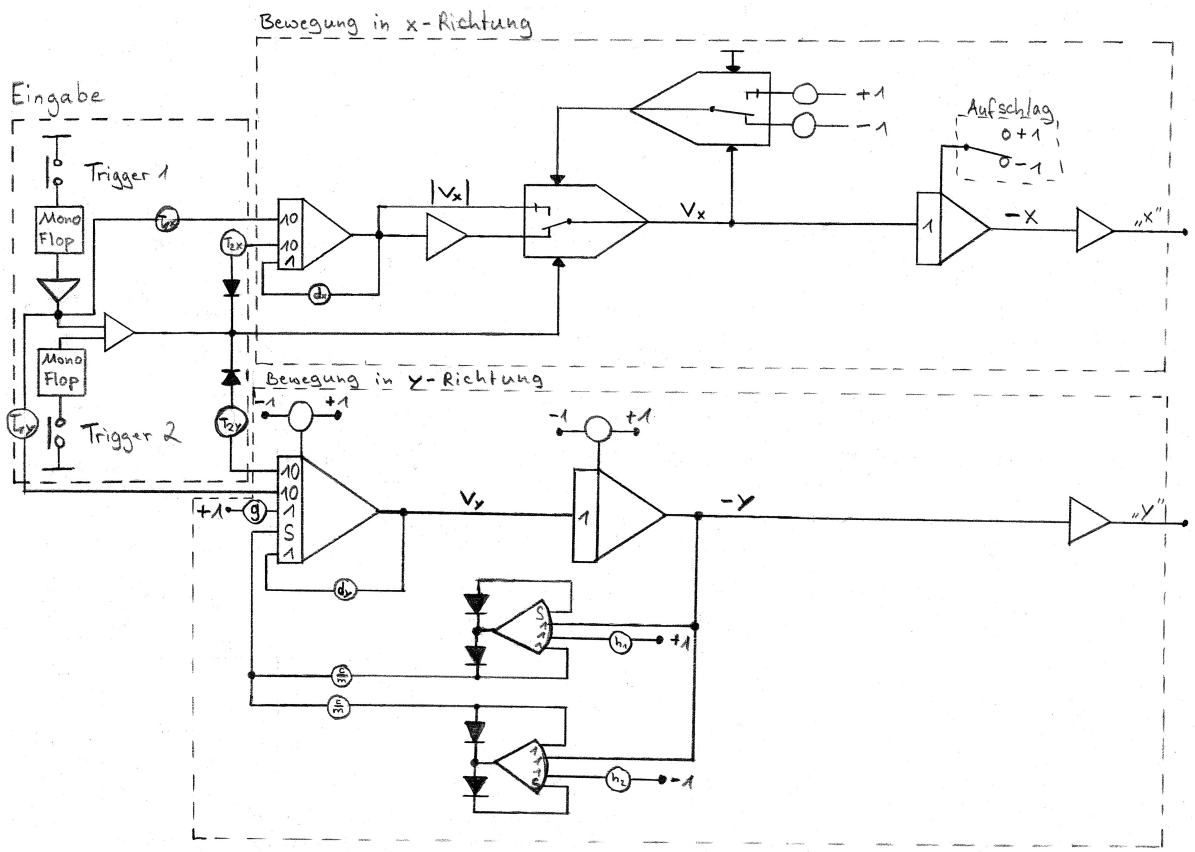


Abb. 4.1.4: Tennis für Drei

Für das Re-Enactment von Tennis for Two unter dem Titel Tennis für Drei [vgl. Höltgen/Maibaum/Rech 2012] musste die vorliegende Schaltung zunächst reduziert werden, um freie Operationsverstärker zu bekommen. Hierzu wurde entschieden die Ballfunktion zu deinstallieren. Die Operationsverstärker 5, 6 und 7 der o.g. Schaltung konnten damit für die Ballsteuerung genutzt und die ‚wild' vor dem Steckpanel angebrauchte Zenerdioden-Schaltung (zur Stabilisierung der Ballgröße) entfernt werden. Die Darstellung des Balls schrumpfte damit auf (s)einen (Mittel-)Punkt zusammen. Durch diese Reduktion konnte eine hinter der originalen Ästhetik verborgene technische Fiktion sichtbar gemacht:

Zum einen bestand William Higinbothams Spiel aus *drei Teilschaltungen*, von denen eine den Ball sowie die Spiel-Physik berechnete und animierte, die zweite das Tennis-Feld von der Seite darstellte und die dritte die Ausgaben der beiden ersten so kombinierte, dass sie auf *einem* Oszilloskop dargestellt werden konnten. Das Oszilloskop, das Higinbotham hierfür verwendete, war *einstrahlig*, was bedeutet, dass es nur einen Kathodenstrahl zur Darstellung aller Spielgrafiken besaß. Damit Higinbotham trotzdem Ball und Spielfeld simultan darstellen konnte, griff er auf das Verfahren mit den Teilschaltungen zurück. Aktuelle Re-Implementierungen wie T42 [vgl. MEGA 2012], bei denen das gesamte Spiel

auf *einer Platine* (und dazu noch mit integrierten Bauteilen) emuliert wird, machen diese Besonderheit von Tennis for Two unsichtbar.

Die Integration der Monoflop-Trigger in die Schaltung von Tennis für Drei benötigte zwei zusätzliche Operationsverstärker und vier weitere Koeffizientenpotentiometer (siehe Abb. 4.1.4). Die Generierung des Tennisfeldes konnte deshalb nur konzipiert nicht aber implementiert werden. Die Lösung für die mathematische Darstellung des Graphen für das Tennisfeld in Seitenansicht ist (vgl. Abb. 4.1.5):

$$y = 1 - | SGN(x) |$$

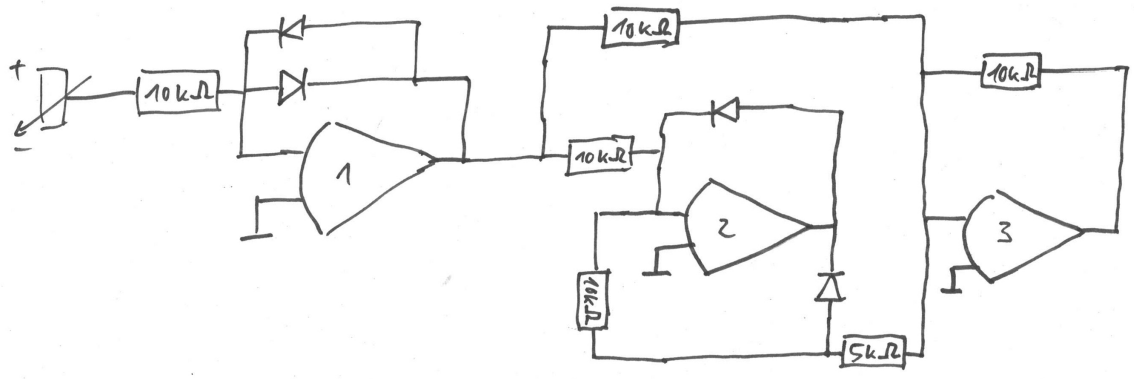


Abb. 4.1.5: Manueller Entwurf eines Prinzip-Schaltplans für ein Tennisfelde (noch ohne Invertierung) [vgl. Sydow 1974:141, 143]

Der Schaltungsteil um den Operationsverstärker Nr. 1 bildet die Vorzeichenfunktion (SGN). Die Elemente um die Operationsverstärker Nr. 2 und 3 sind für die Betragsfunktion zuständig. Um die Ausgabe der Schaltung von 1 zu subtrahieren wird ein vierter Operationsverstärker benötigt, in dessen nicht-invertierenden Eingang man eine Maschineneinheit einspeist und im invertierenden Eingang den Ausgang der o. g. Funktionsschaltung. Diese zu Testzwecken auf einem *Leybold-Haeraus*-Analogcomputer implementierte Schaltung erweis sich als defizitär: Zwar ergibt der Graph der o.g. Funktion im Funktionsplotter an der Stelle x=0 ein unstetigen Sprung nach y=1, der im Oszilloskop aufgrund des nicht abschaltbaren Kathodenstrahls eine senkrechte Linie hätte darstellen müssen; die Langsamkeit der Schaltung hat jedoch stets einen ‚unsauber' *gekrümmt* auf- und absteigenden Kathodenstrahl und damit keine Senkrechte Linie ergeben. Eine schnelle Trickschaltung, wie sie Higinbotham in seiner zweiten Teilschaltung realisiert hatte, hätte hier ein adäquateres Ergebnis erbracht.

Allerdings wären für die Implementierung der Feldgrafik die hierfür benötigten weiteren vier Operationsverstärker auf dem *Telefunken*-Rechner nicht mehr verfügbar. Aus diesem Grund wurde das Tennisfeld als Foliengrafik auf den Oszilloskop-Schirm aufgebracht. Diese Implementierung ähnelt also nicht nur in der Punktdarstellung des Balls, sondern auch in der Semantisierung des Spielfeldes [vgl. Pias 2002a:110] Ralph Baers 1972 veröffentlichtem Table Tennis für die Odyssey-Konsole, sowie der Vectrex-Spielkonsole, die

einen Vektorbildschirm implementiert hat. Beide Systeme verwenden ebenfalls Bildschirmfolien für die Suggestion eines Spielfeldes und zur Einfärbung ihrer Schwarzweiß-Grafik (vgl. Abb. 4.1.6).

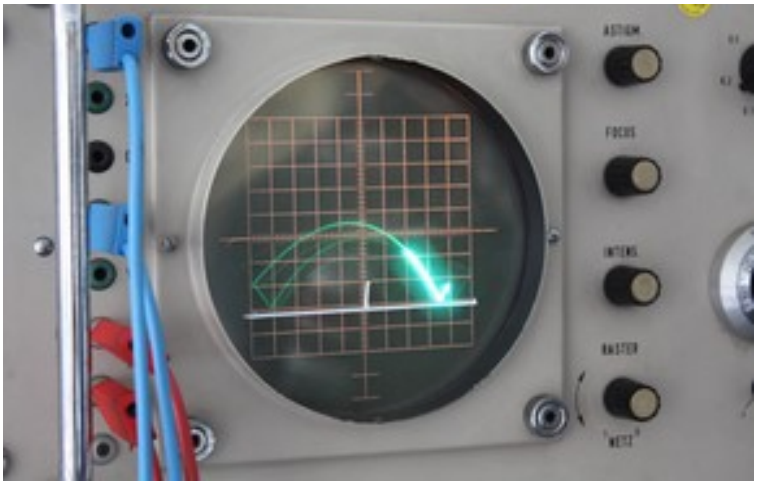


Abb. 4.1.6: Das Spielfeld von Tennis für Drei mit der Tennisfeld-Folie

Sowohl in Higinbothams Tennis for Two als auch in der Hardware-Emulation T42 interagieren die Tennisfeld-Grafik und der Tennisball augenscheinlich miteinander: Bei Berührung prallt der Ball vom Netz ab. In der von uns implementierten Version muss ein dritter Mitspieler darauf achten, dass die Spielregeln eingehalten werden und die Berührung von Ball und Netz ahnden. Dieses Kriterium der menschlichen bzw. maschinellen Spielregel-Überwachung ist in einem Gerichtsverfahren von Baer gegen *Atari* [vgl. Kent 2001:45-58] zu einem maßgeblichen Argument geworden, denn darin unterschieden sich Baers elektronisches Bildschirmspiel und *Ataris* Spielcomputer voneinander. Das analoge bzw. hybride Tennisspiel (Tennis for Two, Table Tennis) besitzt im Gegensatz zu Pong keine Punktezählung, sondern lagert diese – und andere nicht implementierte Spielbestandteile – ebenfalls aus:

> Was noch nicht auf dem Bildschirm implementierbar war, wurde analog supplementiert, und noch einige Jahre lang werden an Spielautomaten farbig bedruckte Folien die Grenzen von spielexternem Display und spielinterner Grafik zu verwischen suchen. Das spezifische Manko der Odyssey ist, daß sie nicht mit diskreten Werten rechnen kann und daher alle numerischen Operationen wie Kartenwerte, Punktestände und Zufallszahlengenerierung auslagern muß. Ein nicht minder bedeutsamer Aspekt ist Baers am Scheideweg zwischen Hockey und Tennis gefundene Erkenntnis, daß Programm und Semantisierung nichts miteinander zu tun haben. [Pias 2002a:110]

4.1.2 Metaphern

Diese Unabhängigkeit von Programm, semantischer Zuschreibung seiner Funktion und seiner historischen Bedeutung ist auch in Pias‘ kulturwissenschaftlicher Analyse von

Computerspielen selbst zu beobachten, wie ein zweites re-implementiertes Spiel und der Vergleich mit seinen Quellen zeigt. Claus Pias beschreibt eine frühe Variante des computerisierten Tennisspiels auf einem IBM-Großrechner der 1960er-Jahre:

> [...] 1960 war eine dritte Art von Ballspiel zu sehen, das Studenten des *MIT* für den *IBM704* geschrieben hatten. Dieser besaß eine Kette von Kontrolllämpchen, über die die Funktionsfähigkeit einzelner Bauteile getestet werden konnte. Das Programm bestand nun darin, die einzelnen Teile des Rechners in einer bestimmten Reihenfolge zu testen, um so die Lämpchen nacheinander aufleuchten zu lassen und einen wandernden Lichtpunkt zu erzeugen, der rechts verschwand um links sofort wieder zu erscheinen. Drückte man dann pünktlich beim Aufleuchten des letzten Lämpchens eine Taste, so kehrte der Lichtpunkt seine Laufrichtung um, schien abzuprallen und zurückzuwandern. Die Kontrolltafel des *IBM704* war zu einer Art eindimensionalem Tennisspiel geworden. [Pias 2005b:218f.]

Diese wenigen Informationen bildeten den ersten Ausgangspunkt für ein Re-Enactment des beschriebenen Programms. Eine Orientierung, wie LED-basierte ‚Ballspiele' implementiert werden können, lieferte darüber hinaus das 1975 für den Altair 8800 entstandene Spiel KILL THE BIT [vgl. McDaniel 1975], bei dem ein im Akkumulator der CPU gespeichertes Bit auf den Frontpanel-Leuchtdioden des Computers visuell durch-rotiert und vom Spieler rechtzeitig mit dem dazugehörigen Binärschalter ‚getroffen' werden muss[29]:

```
Adr.  Opcodes u. Arg.    Label    Mnemonic  Argumente  Kommentar
[…]
0000   21 00 00          lxi      h,0                 ;initialize counter
0003   16 80                      mvi      d,080h              ;set up initial display
bit
0005   01 0E 00          lxi      b,0eh               ;higher value = faster
0008   1A       beg:    ldax     d                   ;display bit pattern on
0009   1A                       ldax     d                   ;...upper 8 address lights
000A   1A                       ldax     d
000B   1A                       ldax     d
000C   09                       dad      b                   ;increment display counter
000D   D2 08 00          jnc      beg
0010   DB FF                     in              0ffh                ;input data from
sense switches
0012   AA                       xra      d                   ;exclusive or with A
0013   0F                       rrc                          ;rotate display right one
bit
0014   57                       mov      d,a                 ;move data to display reg
0015   C3 08 00          jmp      beg                 ;repeat sequence
[…]                      end
```

[McDaniel 1975]

Das obige, in Intel-8080-Assembler programmierte Spiel bildete die Vorlage für das Re-Enactment des IBM-704-Programms in Assembler auf dem Signetics Instructor 50 (vgl. Abb. 4.1.7). Dieser auf dem 8-Bit-Prozessor 2650 (der Firma *Signetics*) basierende Lern-

29 https://www.youtube.com/watch?v=ZKeiQ8e18QY [letzter Abruf: 07.02.2018]. Unter der Adresse (http://altairclone.com/downloads/pong.pdf [letzter Abruf: 01.06.2019] findet sich der Code für ein PONG-Spiel für den Altair 8800, das jedoch erst 2014 von Mike Douglas entwickelt wurde.

computer bildete die Grundlage für einen 14-tägigen Brückenkurs, in dem interessierte Programmier-Neueinsteiger den Zugang zum laufenden, auf vier Semester angelegten Programmierworkshop erhalten sollten (vgl. Kap. 4.2 und 4.3). Ausgehend von der Problemstellung ein Tennisspiel auf den acht Leuchtdioden der Parallel-I/O, die die Spannungssignale am Datenbus des Prozessors anzeigen, zu realisieren, entstand der unter [Höltgen 2016d] publizierte Code:

```
; PROGRAMM:
; ********************************************************************************
;ADR    Label          Opcode u. Arg.  Mnemonic u. Arg.      Kommentar
;
0000    START          1F 01 00       BCTA,UN INIT           ; INT überspringen
;
0007    INT            76 20                  PPSU 20        ; INT-Taste deaktivieren
0009           1F 01 50       BCTA,UN TOLFT          ; Sprung zu TOLFT
;
0100    INIT           77 02                  PPSL 02        ; COM=logisch
0102           76 20                  PPSU 20        ; INT-Taste deaktivieren
0104           04 00                  LODI,R0 00     ; Allzweckregister
0106           05 01                  LODI,R1 01     ; Ballregister laden ...
0108           F1                     WRTD,R1        ; ... und ausgeben
0109           06 00                  LODI,R2 00     ; Schleifenregister
010B           07 00                  LODI,R3 00     ; Schleifenregister
;
; Hauptschleife
;
010D    MAIN           1F 03 30       BCTA,UN AUFR           ; Rechts beginnt
;
0150    TOLFT    3F 03 00             BSTA,UN WAIT           ; Warteschleife
0153           D1                     RRL,R1         ; Ball bewegen
0154           F1                     WRTD,R1        ; Ball auf dem Feld darstellen
0155           01                     LODZ,R1        ; Pos. in R0 zwischenspeichern
0156           44 80                  ANDI,R0 80     ; Ball schon ganz links?
0158           1C 01 50       BCTA,01 TOLFT          ; Wenn nicht, zurück
015B           06 71                  LODI,R2 71     ; Dauer innere Schleife
015D           07 C2                  LODI,R3 C2     ; Dauer äußere Schleife
015F    TSENS1   B4 80                TPSU 80        ; SENS-Taste gedrückt?
0161           1D 01 90       BCTA,00 TORGHT         ; ja, dann zu TORGHT
0164           FE 01 5F       BDRA,R2 TSENS1         ; nein, dann weiter innere ...
0167           FF 01 5F       BDRA,R3 TSENS1         ; ... und äußere Schleife
016A           1F 03 50       BCTA,UN LLOSE          ; verpasst: zu LLOSE
;
0190    TORGHT   3F 03 00             BSTA,UN WAIT           ; Warteschleife
0193           51                     RRR,R1         ; Ball bewegen
0194           F1                     WRTD,R1        ; Ball auf dem Feld darstellen
0195           01                     LODZ,R1        ; Pos. in R0 zwischenspeichern
0196           44 01                  ANDI,R0 01     ; Ball schon ganz rechts?
0198           1C 01 90       BCTA,00 TORGHT         ; Wenn nicht, zurück
019B           74 20                  CPSU 20        ; INT-Taste aktivieren
019D           06 71                  LODI,R2 71     ; Dauer innere Schleife
019F           07 C2                  LODI,R3 C2     ; Dauer äußere Schleife
01A1           FA 7E                  BDRR,R2 7E     ; innere Schleife
01A3           FB 7C                  BDRR,R3 7C     ; äußere Schleife
01A5           1F 04 00       BCTA,UN RLOSE          ; verpasst: zu RLOSE
;
; SUBROUTINEN
;
; Warteschleife (0,5 Sekunden):
;
0300           06 71                  LODI,R2 71     ; Dauer äußere Schleife
0302           07 C2                  LODI,R3 C2     ; Dauer innere Schleife
0304           FA 7E                  BDRR,R2 7E     ; innere Schleife
0306           FB 7C                  BDRR,R3 7C     ; äußere Schleife
0308           17                     RETC,UN        ; RETURN
[...]
```

[Höltgen 2016d]

Der groben ‚Auflösung' und Eindimensionalität der grafischen Darstellung ist es geschuldet, dass aus den Bewegungsabläufen des Ballwurfs bzw. -schlags hier lediglich die Impuls-Umkehr (ohne Verluste durch innere und äußere Reibung) implementiert werden konnte. Es hätte nicht nur zu Problemen mit den Ressourcen des Zielsystems geführt (512 Byte RAM, 303/895 Kilohertz Prozessortakt [vgl. Signetics 1978b:6-24], fehlende Gleitkomma-Hardware usw.) hier komplexe physikalische Prozesse zu implementieren. Auch ist vom historischen IBM-704-Code ebenfalls nicht bekannt, wie genau er sich diesbezüglich an den physikalischen Vorgängen orientierte. Insofern stellt sogar die Darstellung der Beschleunigung lediglich eine heuristische Näherung dar.

Der obige Code-Ausschnitt zeigt die Initialisierung (Adressen 0000_{16}-$010B_{16}$), die Hauptschleife mit den Ball-Bewegungsroutinen (Adressen $010D_{16}$-$01A5_{16}$) sowie die Warteschleifen-Subroutine (Adressen 0300_{16}-0308_{16}). (Nicht abgebildet sind die Routinen für den Aufschlag sowie für den Ballverlust.) Gespielt wird das Spiel mit den Tasten SENS und INT (Abb. 4.1.7). Das Drücken der INT-Taste löst ein Interrupt-Signal aus, bei dem der Programmcounter des Prozessors auf die Adresse 0007_{16} gesetzt wird. Dort wird das Label TOLFT angesteuert, in welchem der Ball nach links geschlagen wird. Das Drücken der SENS-Taste wird in der Polling-Routine $015F_{16}$-$016A_{16}$ abgefragt und führt den Programmteil ab Label TORGHT aus, in dem der Ball nach rechts geschlagen wird. Ein Ball gilt nur dann als ‚getroffen', wenn die jeweils äußerste (linke oder rechte) Leuchtdiode der Anzeige leuchtet und in dieser Zeit entweder die INT- oder die SENS-Taste gedrückt würde.

Abb. 4.1.7: Der Signetics Instructor 50

Die Leuchtdauer (mithin die Zeit, in der der Ball geschlagen werden kann) wird in der Warteschleife definiert und beträgt 0,5 Sekunden (vgl. Kap. 4.1.5.2). Dies ist auch die Zeitdauer, in der die übrigen Leuchtdioden aufleuchten. Der ‚Flug' des Balls auf die gegenüberliegende Seite dauert daher 3,5 Sekunden. Die Grobheit der Bewegungsauflösung in acht Schritten sowie die Geschwindigkeit stellen die einzigen Schwierigkeiten für die Spieler dar. Nach dem idealisierten Vorbild von Kill the Bit findet weder eine Beschleuni-

gung nach dem Schlag noch seine Verlangsamung während des Flugs durch Reibung statt. Ebenso (und im Gegensatz zu Kill the Bit) ist es nicht möglich etwas anderes als das reine ‚Grundlinienspiel' zu spielen: Während des Ballflugs werden die Tasten INT und SENS nicht abgefragt (in ersterem Fall ist der Interrupt maskiert, vgl. 0102_{16}).

Dass es sich bei Tennis 2650 wie auch bei seinem Vorbild vom IBM 704 um eine Tennis-Simulation handelt (und nicht etwa um das Hin und Her spielen eines Hockey-Balles), lässt sich allein aus der Pias'schen Zuschreibung ableiten. Im Effekt scheint das Spiel ähnlicher mit Kill the Bit, das bereits mit seiner Handlungsanweisung „kill the rotating bit" [McDaniel 1975] als ein *Spiel mit der Technik* beschrieben wurde; Tennis 2650 wäre demnach adäquater durch ‚reverse the rotating bit' zu paraphrasieren.

4.1.3 Mimikrys

Der Ball im Kasten als Ausgangsproblem einer Implementierung physikalischer Vorgänge in Computersimulationen wird 1984 mit einem Demoprogramm, an welchem die Leistungsfähigkeit einer neuen Homecomputer-Generation vorgeführt werden soll, wieder aufgegriffen. Mit der so genannten Boing!-Demo (entwickelt von Dale Luck and Robert J. Mical während der *Winter Consumer Electronics Show* 1984 in Las Vegas [vgl. Maher 2012:18]) für den Amiga-Computer der Firma *Commodore* wurde das seinerzeit 25 Jahre alte Motiv des springenden Balls abermals ins Zentrum der öffentlichen Betrachtung einer Hardware-Show gerückt.[30]

Der Lorrain-Prototyp des Amiga war „a cobbled collection of circuit boards and wires that failed constantly." [Maher 2012:16] Der Computer war noch auf die ISA-Ebene reduziert [Tanenbaum 2016:22f.]; ein Betriebssystem, Software oder höhere Programmiersprachen existierten noch nicht. Nicht einmal eine Tastatur stand den Entwicklern der Demo zur Verfügung. Sie mussten computerhistorisch zwei Epochen zurück gehen und ein Terminal für die Programmierung der Hardware über eine serielle Schnittstelle verwenden. Der Code selbst entstand im Cross-Platform-Development in der Programmiersprache C auf einem anderen System mit 68000-CPU [vgl. Maher 2012:18;289FN25].[31]

Maher [2012] hat von diesem Demonstrationsprogramm 2012 für den Amiga eine „reconstruction" [ebd.:33] in der Programmiersprache C zu Forschungszwecken erstellt; aber bereits zuvor hatte es Adaptionen von Boing! sowohl für den Amiga (in Form von Ak-

30 Eine Informationsseite über die Spezifikationen und die Ausstellung dieses Prototypen mit dem Namen Lorraine findet sich unter: http://www.amigahistory.plus.com/prototypes/lorraine.html [letzter Abruf: 18.07.2016].

31 Dennoch bezeichnet Maher das auf seiner Webseite verlinkte Disassembly irrigerweise als „Assembly language source code to the original demo, with comments by Harry Sintonen" (vgl. http://amiga.filfre.net/?page_id=5 [letzter Abruf: 14.02.2018]). Kai Scherrer hat die Versionsgeschichte des Programms untersucht und dabei die Residuen des Sourcecodes noch in zahlreichen Versionen entdeckt [vgl. Scherrer 2018].

tualisierungen [vgl. Scherrer 2018]) als auch auf anderen Systemen gegeben, von denen einige wesentlich *leistungsschwächer* als der Amiga sind. Die Geschichte dieser Demonstrationsprogramme (mit denen die Autoren zu zeigen beabsichtigten, dass man keinen Amiga benötigt, um derartige Grafik-Effekte generieren zu können [vgl. hs 1986a:116]) eskaliert in dem 1999 entstandenen und 2013 erweiterten Port für die Spielkonsole VCS der Firma *Atari* – und läuft damit zugleich historisch zurück, denn Jay Miner, der Entwickler der Amiga-Hardware, hatte auch den TIA-Grafik/Sound-Chip für die VCS designt.

Im Folgenden werden zunächst Teile des Disassemblys einer historischen Implementierung und die daraus ablesbaren Anpassungen an die Hardware vorgestellt. Danach werden diese Mahers „reconstruction" in C, einer Adaption für Amstrad CPC in Locomotive-BASIC und schließlich der Portierung in MOS-6507-Assembler für die Atari-VCS-Spielkonsole von Moll [2014] gegenüber gestellt.

4.1.3.1 Boing!

Der Amiga-Computer von *Commodore* erschien 1985 als früher 16-Bit-Homecomputer (nach dem erfolglosen Versuch der Firma *Sinclair*, ihr Modell QL 1984 als 16-Bit-Computer auf dem Homecomputermarkt zu etablieren [vgl. Adamson/Kennedy 1984]) und Nachfolger von *Commodores* 8-Bit-Produktlinie. Entwickelt wurde der Rechner in den zwei Jahren davor von Jay Miner, der, wie geschrieben, bereits für die VCS-Spielkonsole von *Atari* den Spezialprozessor TIA (*Television Interface Adapter*) zur Grafik- und Sound-Generierung konstruiert hatte. *Atari* hatte diese Technologie, Sound, Grafik und I/O nicht mehr von der CPU verwalten zu lassen, danach bereits in ihren 8-Bit-Homecomputern übernommen: Die Spezialprozessoren POKEY (*Potentiometer and Keyboard Integrated Circuit),* zuständig für die Signalverarbeitung der Tastatur- und Spielcontroller-Eingaben sowie für die Sounderzeugung, und ANTIC (*Alphanumeric Television Interface Controller*), der zusammen mit den GTIA- bzw. CTIA-Bausteinen die Grafik erzeugt, entlasten die niedrig getakteten 6510-CPUs in den Rechnern (und besitzen teilweise Direct Memory Access). Die Nutzung von dedizierter und zumeist vom Nutzer austauschbarer Grafik- und Sound-Hardware in modernen PCs geht auf genau diesen Ursprung zurück [vgl. Maher 2012:13f.].

In den Amiga floss dieses Designprinzip ebenfalls ein: Hier wurde die 68000-CPU von drei Spezialprozessoren unterstützt: AGNES enthält einen Blitter (mit dem schnelle Speicher-Blockkopiervorgänge vollzogen werden können) und einen Copper (ein mathematischer Co-Prozessor für Grafikspeicher-Operationen), PAULA fungiert als Soundprozessor (und zugleich Controller-Bausteilen für Diskettenlaufwerke und die serielle Schnittstelle) und DENISE verwaltet als Grafik-Chip die Sprites und enthält die Farbregister. Die Funktions- und Leistungsfähigkeit dieser drei Prozessoren sollte mit dem Boing!-Demonstrationsprogramm ausgestellt werden. Dieses schon kurz nach seinem Erscheinen *kanonisch* [Jauß 1975a:386] gewordene Programm soll nun in verschiedenen (historischen) Varianten untersucht werden.

Hier werden drei Aspekte der Demo beleuchtet: die Erzeugung der Ballgrafik, der Ball-Rotation und der Ball-Sprungbewegung.

Abb. 4.1.8: Original Boing!-Demo auf einem Commodore Amiga 500.

Der *Grafikhintergrund* der Boing!-Demo (vgl. Abb. 4.1.8) stellt einen dreidimensionalen Kasten dar, dessen Boden und Hinterwand durch ein Gitter stilisiert sind. Die Seitenwände sind grafisch ‚offen', werden in der Demonstration aber als Hindernisse für den Ball berücksichtigt. Die Vorderseite, Decke und die Rückwand des Kastens werden nicht vom Ball berührt und verändern dessen Bewegungen daher nicht. Der Betrachter der Demo schaut, wie beim Ball im Kasten, von vorn in den Kasten hinein – wie sich zeigt, ist diese phänomenologische und technische ‚Sichtöffnung' (*viewport* [Maher 2012:29; Pöpsel u.a. 1994:10f.]) durchaus auch epistemologisch zu verstehen.

Wie vorherige Homecomputer verfügt auch der Amiga über einen dedizierten RAM-Bereich der als Grafik-Speicher dient. Das auf dem Monitor dargestellte Bild ist eine Visualisierung dieses Speicherbereichs:

> [.. T]he image obviously exists on the screen of the monitor used to view it, it also exists within the memory of the computer itself. The latter image is in fact the original of the image that is mirrored to the monitor and is changed as needed by the programs running on the computer when they wish to modify the image on the screen. [Maher 2012:23]

(Dieses Prinzip lässt sich bis zur Williams-Kilburn-Tube zurückverfolgen, vgl. Kap. 4.2). Mit zunehmender Speichergröße und Grafik-Leistung (Auflösung, Farbtiefe, Spriteanzahl) wird auch mehr Speicher für die Grafik reserviert. Ist für die monochrome Grafik in Homecomputern wie dem Amstrad CPC noch für jedes Bildschirm-Pixel ein Bit reserviert, so wächst der Grafikspeicher-Bedarf nicht nur höherer Auflösungen, sondern auch polynomisch, wenn mehr Farben angezeigt werden müssen. Beim Amiga sind hierbei zwei Verfahren zu berücksichtigen:

1. Durch die Farbtiefe von 32 (aus 4096 möglich) Farben müssen pro Pixel bereits 12 Bit für Farben reserviert werden: Je 4 Bit für den Rot-, Grün- und Blau-Anteil einer Farbe. Der Amiga verfügt über verschiedene Grafikmodi: Je mehr Farben dargestellt werden sollen, desto geringer ist die dafür zur Verfügung stehende Pixelauflösung. Die höchste Auflösung beträgt 640×512 Bildpunkte bei maximal 16 (aus 4096) Farben, die niedrigste 320×256 Bildpunkte.[32] Die Farbpixel sind dabei (je nach Farbtiefe) auf bis zu 5 separaten *bitplanes* im Speicher abgelegt, die mittels *planar method* ‚übereinander gelegt' [vgl. Maher 2012:24]) dargestellt werden.

2. Die Grafikdaten des Amiga können einen wesentlich größeren Bereich im RAM belegen als Darstellungsfläche auf dem Monitor verfügbar ist. Mittels der *viewport*-Methode werden die Pointer des Grafikprozessors dabei auf den Anfang und das Ende des darzustellenden Bereichs aus diesem RAM-Bereich gesetzt. Der Effekt ist vorstellbar als „a camera lens looking down into selected areas of the Amiga memory and transmitting what it sees to the monitor." [Maher 2012:29f.]

Die Boing!-Demo macht sich diese Verfahren auf kreative Weise zunutze, um die Evokation eines sich drehenden und springenden Balls in einem Kasten zu ermöglichen. Die Darstellung Mahers [2012:26-37] seien im Folgenden paraphrasiert:

1. Für die Demo wird die niedrigste Auflösung (320×200 mit 32 Farben – also 5 Bitplanes) verwendet. Von den 32 möglichen Farben werden die folgenden 7 auf die Farbregister verteilt: zwei Grautöne für Hintergrundfarbe und Ballschatten (Bit 0-1), 7 identische Rottöne für den Ball (Bit 7-13), 6 identische Weißtöne für den Ball (Bit 2-6 und Bit 15) und ein Rot-Weiß-Ton (Bit 14). Diese Farbaufteilung findet sich beinahe identisch in den oberen Farbregistern gespiegelt: 7 identische Rottöne für den Ball (Bit 18-19 und 27-31), 6 identische Weißtöne für den Ball (Bit 21-26) und ein Rot-Weiß-Ton (Bit 20). Nach dem Start der Demo werden die Rot-, Weiß- und Rot-Weiß-Farbtöne durch die Farbregister rotiert. Dabei wechseln die Farben des Balls sukzessive von Weiß zu Rot und umgekehrt. So entsteht der Eindruck der Ballrotation. (Der Rot-Weiß-Ton stellt hierbei den Farbübergang für die Bewegungsunschärfe dar.)

2. Die Ball-Sprungbewegung wird durch das Verschieben des *viewports* mittels Pointer-Arithmetik realisiert. Während diese Verschiebung die bitplanes 0-3, in denen sich die Grafik- und Farbdaten des Balls befinden, sichtbar verändert, scheint die 4. bitplane, die die Hintergrundgrafik enthält, unbewegt. In dieser befinden sich die Grautöne von Hintergrund und Schatten sowie die beiden Violett-Töne des Kasten-Gitters. Diese vier Farbregister werden im oben genannten Prozess nicht mit rotiert (lediglich beim Wechsel vom Hellgrau des Bildschirmhinter-

32 Diese Angaben beziehen sich auf die Auflösung des Amiga 1000 nach dem PAL-Standard. [vgl. Klöter 2008] Es existieren weitere Modi, in denen bis zu 4096 Farben gleichzeitig dargestellt werden können.

grundes zum Dunkelgrau des Schatten wird an dieser Stelle auch das hellere Violett des Gitters zum dunkleren Violett des ‚beschatteten' Gitters gewechselt). Das bitplane 4 wird von der Pointer-Arithmetik der übrigen bitplanes ausgespart; die *viewport*-‚Kamera' zeigt immer dieselbe Stelle des Grafik-RAMs, weshalb dieses Bildelement unbewegt erscheint.

Beide Programmierverfahren sorgen nicht nur für eine sehr schnelle und flüssig wirkende Bewegung der Grafik; sie sind auch noch überaus speichersparsam realisiert. Es werden keine großen Grafik-Datenmengen bewegt, und keine Einzelbilder zu einer Bewegung animiert. Dies macht die Demo umso eindrucksvoller, weil sie als ein Prozess neben mehreren anderen laufen kann, ohne dass ein Performance-Verlust bemerkbar wäre: „The fact that the demo as a whole does not unduly strain the hardware is one of its key attributes because this feature makes it ideal for running in tandem with other programs to demonstrate the Amiga's multitasking capabilities." [Maher 2012:39]

Um den Diskurs aus Kapitel 4.1.1 und 4.1.2 wieder aufzugreifen, werden im Folgenden die drei Aspekte der Boing!-Demo und deren unterschiedliche Realisation im Original-Code und in Mahers Reimplementierung dargestellt: die Ball-Grafik, die Ball-Drehung und die Animation desselben. In seiner Argumentation verzichtet Maher auf die Diskussion dieser Fragen (den Code sowohl der disassemblierten Version als auch seiner Adaption in C stellt er auf der zum Buch gehörenden Homepage vor.[33])

Der Ball der Boing!-Demo ist, wie die obigen Ausführungen gezeigt haben, kein Blitter-Objekt (also kein vom Grafik-Prozessor generiertes und verwaltetes Grafik-Objekt, das sich unabhängig vom Bildschirmhintergrund bewegt[34]) und auch keine vorgefertigte Grafik, die vom Demo-Programm nachgeladen wird oder als Datensatz im Code enthalten ist. Der Ball wird in der Routine „_init_globe" (Zeilen 581-691[35]) berechnet und dann in „_draw_globe" (Zeilen 693-1122) gezeichnet. In „_init_globe" werden zur Berechnung des Kreisrandes sowie der Kreisbögen (für die Trennlinien der Rot-Weiß-Karo-Textur) die Funktionen „_Sine16" (Zeilen 597, 653) und „_Cosine16" (Zeilen 612, 632) aufgerufen, die sich weiter hinten im Code (2302-2339) befinden. Wie im Disassembly deutlich wird, verzichteten die Programmierer auf eine Sinus-/Kosinus-Wert-Generierung mit Hilfe eines Algorithmus und verwenden anstelle dessen eine Lookup-Tabelle (Zeilen 2334-2339) mit 65 Sinus-Werten und einer 15-Bit-Auflösung (im Intervall [0-32767] mit einem Abstand von 804).

33 http://amiga.filfre.net/?page_id=5 [letzter Abruf: 20.07.2016].

34 Vgl. http://amigadev.elowar.com/read/ADCD_2.1/Hardware_Manual_guide/node00AE.html [letzter Abruf: 21.07.2016].

35 Aus Platzgründen werde hier und im Folgenden die Datei http://amiga.filfre.net/misc/Chapter2/boing.asm [letzter Abruf: 20.07.2016] referenziert, aus der die Zeilennummern einer Texteditor-Darstellung angeben werden.

Die Berechnung des Balls aus Sinuswerten verzögert den Start des Demoprogramms etwas; die Nutzung einer solchen Tabelle macht allerdings noch zeitaufwändigere Iterationen, wie sie etwa in CORDIC-Algorithmen enthalten sind [vgl. Volder 1959:331] überflüssig – auf Kosten der Wertegenauigkeit. Die so entstehenden Grad-Werte für die Kreisfunktionen dienen als Approximationen für später benötigte Werte. Im Gegensatz zur Berechnung der Sinus-/Kosinus-Werte mit Hilfe einer Differenzialgleichung, wie beim Ball im Kasten, scheint der Anspruch der Boing!-Demonstration weniger auf mathematisch-physikalische Exaktheit zu zielen, sondern richtet sich eher an den niedrigen Anforderungen der menschlichen Wahrnehmung geometrischer Figuren aus.

Die Sprung-Bewegung des Balls der Boing!-Demo wird im Original-Code in den Zeilen 2076-2128 realisiert. Der Pointer des *viewport* für die bitplanes 0-3 wird dort über eine quadratische Funktion (eine unten offene Parabel: $y=-ax^2$) verändert. Diese Funktion ist in einer zweifachen Multiplikation realisiert (Zeilen 2107, 2109). Der Hintergrund in der bitplane 4 wird dabei (wie oben dargestellt) durch eine AND-Funktion maskiert (Zeile 2089), damit er nicht mit bewegt wird. Eine Dämpfung der Sprungbewegung ist nicht implementiert; der Ball springt nach dem Abprallen am Kastenboden stets gleich hoch. „Luck and Mical's only goal was to impress the viewer, not to simulate the physical reality of a bouncing soccer ball within the Amiga's memory." [Maher 2012:40]

4.1.3.2 reversing, re-enacting, reconstructing: Boing!-reconstruction

Mahers eigene „reconstruction" der Amiga-Ball-Demonstration aus den Jahren 2009/2010 in der Programmiersprache Lattice C 5.02 [SAS 1990] lässt sich als Übersetzung des Originals beschreiben.[36] Da der Sourcecode nicht vorlag, musste Maher sich eines *technischen Lektüreprozesses*, des *Reverse Engineering,* bedienen. Diese Praxis definiert Eilam [2005] wie folgt:

> Reverse engineering is the process of extracting the knowledge or design blueprints from anything man-made. The concept [...] is very similar to scientific research, in which a researcher is attempting to work out the ‚blueprint' of the atom or the human mind. [...] Reverse engineering is usually conducted to obtain missing knowledge, ideas, and design philosophy when such information is unavailable. In some cases [...] the information has been lost or destroyed. [...] Not too long ago, reverse engineering was actually a fairly popular hobby, practiced by a large number of people (even if it wasn't referred to as reverse engineering). Remember how in the early days of modern electronics, many people were so amazed by modern appliances such as the radio and television set that it became common practice to take them apart and see what goes on inside? [3f.]

36 http://amiga.filfre.net/?page_id=5 [letzter Abruf: 11.01.2019].

Dort, wo Software reverse engineert werden soll, stellt dies hohe Anforderungen an die Fähigkeiten und die Lernbereitschaft des Lesers von fremdem Code und erfordert besondere Tools, wie Eilam [2005] fortsetzt:

> Software reverse engineering requires a combination of skills and a thorough understanding of computers and software development, but like most worthwhile subjects, the only real prerequisite is a strong curiosity and desire to learn. Software reverse engineering integrates several arts: code breaking, puzzle solving, programming, and logical analysis. [4]

Maher benötigte für sein „reversing" (womit „Software Reverse Engineering" [4] gemeint ist) der BOING!-Demo ein Disassembly einer der weithin verfügbaren, ausführbaren Binary-Dateien. Hierbei genügte es nicht, bloß die Daten in Opcodes und deren Argumente zurück zu übersetzen; die eigentliche Arbeit stellt *das bearbeitende Verstehen des Codes* dar:

- Das Programm sollte zunächst in Sinneinheiten eingeteilt (strukturiert) werden.
- Diese Sinneinheiten sollten mit semantisch passenden Labels (als ‚Sprungzielen' für den Leser) versehen werden.
- Schleifen sollten ebenfalls optisch strukturiert werden und mit Labels statt nur konkreten Adressen versehen werden.
- Adressen, die in Sprungbefehlen als Argumente angegeben sind, sollten auf die o. g. Sinneinheiten bezogen werden.
- Systemadressen sollten mit ‚tradierten' Systemvariablennamen[37] versehen und diese sollten im Code verwendet werden.
- Im Programm integrierte Daten-Tabellen (wie hier die Lookup-Tabellen für die Sinus-/Kosinus-Werte) dürfen nicht als Opcodes missinterpretiert werden.
- Die Programmteile müssen kommentiert werden, um ihren Sinn im Gesamtzusammenhang des Programms nachvollziehbar zu machen. (Hierbei half Maher eigenen Angaben zufolge Harry Sintonen[38])
- Eventuell eingebaute Obfuskierungen müssen erkannt und als solche behandelt werden. (Solche können auf verschiedenen Weise in den Code eingebaut worden sein, zum Beispiel als sinnlose Programmanweisungen, die vom Code aber nie ausgeführt werden, oder als ebenfalls sinnlose Dateneinschübe, die vom Disassembler als Code übersetzt werden usw. [vgl. Eilam 2005:344f.].)

37 http://wandel.ca/homepage/execdis/exec_disassembly.txt [letzter Abruf: 21.03.2018] stellt das Amiga-ROM als kommentiertes Disassembly vor und benennt die Systemvariablen.

38 http://amiga.filfre.net/?page_id=5 [letzter Abruf: 11.01.2019].

Je nach Umfang des zu disassemblierenden Programms kann diese Tätigkeit sehr langwierig sein, fördert aber ein – im Sinne der Hermeneutik – maximal tiefes Verständnis des Codes und seiner Funktionen zutage. Voraussetzung hierfür ist, dass die bezogene Hardware (insbesondere der genaue Prozessor-Typ, die Memory-Map und die Peripheriebausteine), die verfügbaren/im Programm aufgerufenen Betriebssystemroutinen, Systemvariablen und gegebenenfalls nachgeladene Libraries sowie weitere in andere Binary-Dateien ausgelagerte Programmteile bekannt sind und mit-verstanden werden. Bei nur eingeschränkt konfigurierbaren Systemen, wie den Homecomputern der 1970er- bis 1990er-Jahre, ist der Erwerb solch umfassender Kenntnisse noch möglich. Beliebig skalierbare und individuell konfigurierbare PC-Systeme stellen hier bereits eine große Hürde dar.

Nachdem der erste Schritt des Disassemblierens vollzogen war, stand der zweite Schritt an: Die Übersetzung des Programms aus der rekonstruierten Assembler-Sprache zurück in die Programmiersprache C. Diese „reconstruction“ fand bei Maher nicht allein aus didaktischen Erwägungen (etwa zur Visualisierung der Codestruktur) statt; mit der Adaption betrieb er auch Retrocomputing im hier verstandenen Sinn: als Aktualisierung eines historischen Konzeptes mit moderne(re)n Mitteln. Dabei kompensierte er die historischen ‚Probleme‘ (fehlende FPU, keine vorhandenen Grafik-Tools etc. [vgl. Maher 2012:18]) mit modernen Werkzeugen. Die Ball-Grafik etwa entwarf er mit einem Amiga-Grafikprogramm und hinterlegte sie als Bitmap im Programmcode. Dies kommentierte er im Code wie folgt:

```
/* [..] For the sake of clarity and simplicity, I have chosen to store the image of the ball
within my version of the program and merely paint it onto the screen.

The original demo having been created before Amiga paint and graphical manipulation programs
existed, Luck and Mical obviously did not have the luxury of approaching the problem in this
way. This is by far the largest single difference between this reconstruction and the original
demo.*/
```

Durch diese Änderung konvertiert er das Boing!-Demoprogramm allerdings mehr in Richtung einer *Animation* und entfernt sich so zwar von den Vorgaben der Demoscene [vgl. Botz 2011:17], nähert sich aber den ‚Intentionen‘ der CPU an: „The 68000 also hosted an extensive instruction set designed to support programming in *high-level languages* as C rather than the tedious *assembly code* that was the norm of the time [...]“ [Maher 2012:15]. Die hohe Performanz seiner Adaption im Vergleich zum in Assembler programmierten Original scheint seine These zu bestätigen. Das im Rahmen seiner Forschungsarbeit durchgeführte Projekt steht also in der epistemologischen Tradition des Hackings.

Allerdings vollzieht sich diese Übersetzung in mehreren Instanzen:

1. Zunächst übersetzt ein Disassembler ein Binary File in Assembler-Sprache.

2. Dann übersetzt Maher[39] den erzeugten Assemblercode durch Strukturierung (über die Suche nach Koähsionsmarkern, vgl. Kap. 4.1.4.3).
3. Danach übersetzt er die Funktionen des Assembler-Programms in C-Sourcecode.
4. An dieser Stelle setzt wieder eine mehrschichtige [Griffith 2002:7f.] maschinelle Übersetzung ein, wenn der C-Code kompiliert wird. Hierbei wird, wie bei C-Compilern üblich [ebd.:8], zunächst wieder eine Assembler-Datei erzeugt.
5. Diese Datei wird in einem weiteren Schritt in ausführbaren Code assembliert und als Binary File gespeichert.

Mahers „reconstruction" stellt sich damit als vielfacher Übertragungs- und Übersetzungsprozess heraus, bei dem sowohl menschliche als auch maschinelle Instanzen Sourcecodes in unterschiedliche Sprachen transkodieren. Der C-Compiler lässt sich hier nicht mehr allein als ‚Zusammensteller' verstehen, sondern auch als ‚Zusammen*schreiber*' (beides Übersetzungsmöglichkeiten des englischen Begriffs *to compile*). Auf welche Weise der C-Compiler den Sourcecode in mehreren Schritten in ein ausführbares Programm übersetzt, ist zudem Abhängig von den Compiler-Entwicklern, denn diese legen die Übersetzungsregeln fest.

4.1.3.3 Illusion of Rotation: Blitter

Dieser Prozess der Adaption einer Software von einem System auf ein anderes (älteres oder weniger komplexeres) wird von Hobbyisten oft mit der Aussage des „wissen wollen, ob es geht" begründet. Er fand am Beispiel der Boing!-Demo schon sehr früh statt. Die mit dem Amiga aufkommende ‚Verheißung' einer neuen audiovisuellen Qualität und Quantität im Homecomputing [vgl. hs 1986b:18] hat insbesondere Nutzer von 8-Bit-Systemen dazu motiviert, gleichartige Demo-Programme zu entwickeln, um zu beweisen, dass weniger komplexe Systeme zu ebensolchen Ausgaben fähig sind [vgl. ebd.]. An der obigen Diskussion der Original-Implementierung für den Amiga konnte bereits gezeigt werden, dass die Grafikausgaben und -animationen im Wesentlichen auf ‚Tricks' basieren, bei denen durch Farbänderungen und Bildausschnittverschiebungen ein Bewegungseffekt suggeriert wird.

Das Zusammenspiel von 16-Bit-CPU und Spezialchips hat bei der Boing!-Demo auf dem Amiga die notwendig hohe Performanz erzeugt, um das Programm selbst noch neben anderen im Multitasking-Betrieb des Amiga OS flüssig ablaufen zu lassen [Maher 2012:39]. Diesem Anspruch mussten die Adapteure der 8-Bit-Varianten nicht nachkommen, weil für diese Systeme damals (noch, vgl. Kap. 5.4.3) keine solchen Betriebssysteme existierten. Der so genannte „Amiga-Ball" wurde zu dieser Zeit für Systeme wie den

39 Ein C-Decompiler existiert für den Commodore Amiga bislang nicht.

Commodore 64[40], *Ataris* 8-Bit-Computer[41], *Sinclairs* ZX Spectrum[42], TRS-80 Color Computer[43] u.a.[44] adaptiert. Diese Programme wurden allerdings allesamt in Assembler geschrieben. Eine Umsetzung in BASIC, wie das nachfolgend diskutierte Programm BLITTER von Paul Bond [1986] stellt insofern eine besondere Herausforderung dar, da interpretierte BASIC-Programme sehr viel langsamer sind als solche in Maschinensprache (vgl. Kap. 4.1.5.5). Diese Tatsache hebt der Begleittext allerdings noch als kompetitives Argument hervor: „Blitter mimics a demo program Commodore use[d] to show off their enormously expensive Amiga personal computer [...] Arnold [ein Szene-Spitzname für die CPC-Computer, S. H.] can match computers nearly ten times more expensive – not bad, for 3K of Basic!“ [Bond 1986:78]. Es ging/geht hierbei also auch um ein spielerisches Messen von Programmier-Skills, scheinbar motiviert vor einem ökonomischen Hintergrund. Solche Gamification-Praktiken der Demoscene(n), können im Sinne eines spielerischen Austauschs/Wettbewerbs gedeutet werden [vgl. Hartmann 2017: 109-141] (vgl. Kap. 5.2.3.2).

Der Amstrad CPC ist ein 8-Bit-Computer, basierend auf der Z80A-CPU, getaktet mit 4 Megahertz. Als Bausteine für die Grafik- und Soundgenerierung verwendete Amstrad Standard-ICs: Der CRTC-Chip[45] generiert die Grafiken und findet ebenso wie der AY-3-8912[46], zuständig für den Sound, in zahlreichen anderen Computern und elektronischen Geräten dieser Zeit Einsatz. Die erste Generation der CPCs war in drei Ausführungen erhältlich: Zwei Modelle mit 64 Kilobyte RAM, eines davon (CPC 464) mit eingebautem Kassetten-, das andere (CPC 664) mit eingebautem 3-Zoll-Diskettenlaufwerk. Die dritte Variante (CPC 6128) besitzt ebenso ein eingebautes 3-Zoll-Diskettenlaufwerk sowie 128 Kilobyte RAM-Speicher (wobei die zweiten 64 Kilobyte vom Z80A nur per bank switching adressierbar sind). Alle drei Modelle wurden es zusammen mit Grün- oder Farbmonitor verkauft. Als Betriebssystem dient in diesen CPCs das Locomotive-BASIC (Version 1.0 bei CPC 464 und CPC 664, Version 1.1 bei CPC 6128). Die CPCs mit eingebautem Diskettenlaufwerk verfügen zudem über das Betriebssystem AMSDOS und ermöglichen zusätzlich die Nutzung von CP/M. Als Programmiersprache ist in allen drei Computern ein BASIC-Inter-

40 http://csdb.dk/release/?id=20935 [letzter Abruf: 07.02.2018].

41 https://www.pouet.net/prod.php?which=10096 [letzter Abruf: 07.02.2018].

42 http://zxdemo.org/productions/12626/ [letzter Abruf: 07.02.2018].

43 https://www.pouet.net/prod.php?which=51322 [letzter Abruf: 07.02.2018].

44 Es existieren ebenso zeitgenössische Adaptionen für 16-Bit-Systeme, die, wie der Amiga, auf der Motorola 68000-CPU basieren, jedoch nicht über vergleichbaren Spezialchips verfügten: Sinclair QL (https://qlwiki.qlforum.co.uk/doku.php?id=qlwiki:animationsgrafiK [letzter Abruf: 07.02.2018]) und Atari ST (https://www.pouet.net/prod.php?which=27109 [letzter Abruf: 07.02.2018]).

45 Cathode Ray Tube Controller, http://www.cpcwiki.eu/index.php/CRTC [letzter Abruf: 01.03.2018].

46 http://www.cpcwiki.eu/index.php/PSG [letzter Abruf: 01.03.2018].

preter im ROM verbaut sowie bei den Diskettenlaufwerk-Modellen zusätzlich Teile eines Logo-Interpreters (diese können nur zusammen mit der Logo-Diskette, die den Rest der Programmiersprache enthält, aktiviert werden; eine lizenzrechtliche Lösung). Vor dem Hintergrund des hier zu besprechenden Programms Blitter muss erwähnt werden, dass der CRTC-Grafikchip weder über Spritegrafiken noch über Blitter-Funktionen oder die *viewport*-Technologie verfügt.

Das Programm Blitter wurde als Type-In-Listing (gedruckter BASIC-Sourcecode zum Abtippen) in der britischen Zeitschrift *Amstrad Action* veröffentlicht. Es spielt mit seinem Titel auf die Funktion des oben genannten Amiga-Customchips AGNES an, der an der Grafik und Animation der originalen Boing!-Demo zwar keinen Anteil hatte, jedoch als Synecdoche für die Grafikhardware des Amiga im Diskurs war. Selbst die Funktionalitäten eines Blitter-Chips (das hardwarebasierte Kopieren der Inhalte von Speicherblöcken) realisiert das Programm nicht als Softwarefunktion, obgleich der Z80A-Mikroprozessor des CPC über umfangreiche Block-Operationen verfügt [vgl. Zaks 1987:154-156] und im BASIC-Programm auch hardwarenahe Programm-Elemente untergebracht sind.

Programmiert ist Blitter in Locomotive-BASIC 1.0 und damit lauffähig auf allen Amstrad-CPC- und -kompatiblen Computern. Das Programm umfasst 60 BASIC-Zeilen:

```
10 a=0.2
20 x=120
30 DEG
40 c%=3
50 c1%=9
60 MODE 0
70 FOR x%=&C000 TO &F7FF STEP 2:POKE x%,128:NEXT
80 FOR x%=&F800 TO &FFFF:POKE x%,192:NEXT
90     GOSUB 310:ORIGIN 0,0,0,640,0,400
100    col=1
110    FOR kkk=1 TO 100
120            FOR kk=1 TO col
130                    x=x-4
140                    IF x=0 THEN a=-0.2
150                    IF x=-120 THEN GOTO 290
160                    d=0
170                    PLOT 320,300,1
180                    FOR t%= 90 TO 270 STEP 4
190                           chk=INT(200+100*SIN(t%))
200                           IF chk=293 OR chk=243 OR chk=156 OR chk=106 THEN d=1
210                           IF chk=276 OR chk=203 OR chk=128 THEN d=0
220                           IF d=1 THEN DRAW 320+x*COS(t%),200+100*SIN(t%),c% ELSE DRAW
320+x*COS(t%),200+100*SIN(t%),c1%
230                    NEXT t%
240            NEXT kk
250    col=col+a:c%=c%+1:c1%=c1%+1
260    IF c%=15 THEN c%=3
270    IF c1%=15 THEN c1%=3
280    NEXT kkk
290 GOTO 420
300 END
310 c=15
320 x%=0:y%=100:ORIGIN 380,200:PLOT -2,-2,c
330 d%=3-2*r
340 WHILE x%<y%+2
350    PLOT x%,y%:DRAW -x%,y%:PLOT y%,x%:DRAW -y%,x%:PLOT -x%,-y%:DRAW x%,-y%:PLOT -y%,-x
%:DRAW y%,-x%
360    IF d%<0 THEN d%=d%+4*x%+6: GOTO 390
370    d%=d%+4*(x%-y%)+10
```

```
380     y%=y%-2
390     x%=x%+2
400 WEND
410 RETURN
420 d=2:'Change variable D for speed
430 KEY 1,"call&bc02:mode 2:list"+CHR$(13)
440 ENV 1,14,-1,2:'Set up envelopes and variables
450 ENT 1,100,5,1
460 dr=1:cl=3:cl2=9:fr=1: xd%=-1:yd=-0.5:x%=30:y=15
470 FOR x=2 TO 14 STEP 2:INK x,6:INK x+1,26:NEXT:'Set inks to correct colours
480 INK 15,1:INK 1,2:INK 0,11:BORDER 11
490 WHILE mainloop=0
500     INK cl,6:INK cl2,26
510     OUT &BC00,12:OUT &BD00,48+INT(t%/256):OUT &BC00,13:OUT &BD00,t% MOD 256:'Uses CRTC
register 12 & 13 to set OFFSET for hardware scroll
520     IF x%>35 OR x%<15 THEN IF xd%=-1 THEN SOUND 1,1500,0,1,1,1 ELSE SOUND 4,1500,0,1,1,1
ELSE a=a
530     yd=yd-0.5:x%=x%+xd%
540     IF x%>35 OR x%<15 THEN xd%=-xd%:dr=-dr
550     t%=x%+(80*y):FOR a=1 TO d:CALL &BD19:NEXT:'Use FRAME instead of CALL&BD19 on 664 &
6128
560     y=y+yd:IF y<11 THEN yd=2.5
570     IF yd=2 THEN SOUND 2,1000,0,1,1,1
580     cl=cl+dr:IF cl=15 THEN cl=3 ELSE IF cl=2 THEN cl=14:'Keeps control of INK values to
give illusion of rotation
590     cl2=cl2+dr:IF cl2=15 THEN cl2=3 ELSE IF cl2=2 THEN cl2=14
600 WEND
```

[Bond 1986:78 – Formatierung: S. H.]

Bevor die einzelnen Elemente des Programms diskutiert werden, soll zunächst dessen grobe Struktur (anhand der BASIC-Zeilennummern) erläutert werden:

10-60: Initialisierung der Variablen und Einrichten des Bildschirmmodus
70-80: Zeichnung des Hintergrundrasters
100-280: Zeichnung der Ballsegmente
310-400: Zeichnung des Ballschattens
430: Bildschirm-Reset-Funktion zum Verlassen des Programms
440-480: Ballrotation und Soundgenerierung
490-600: Animation

Das BASIC-Programm stellt ein Hybrid zwischen BASIC-Strukturen und -Elementen und hardwarenahen Programmier-Elementen dar. Dies beginnt bereits bei der Zeichnung des Hintergrundrasters innerhalb zweier aufeinander folgender FOR-NEXT-Schleifen, die die Bildschirmspeicher-Adressen ($C000_{16}$-$FFFF_{16}$) des Amstrad CPC mit spezifischen Bit-Masken füllen. Mit diesem Verfahren wird das benötigte Muster deutlich schneller generiert als durch die Verwendung der BASIC-Eigenen Befehle (DRAW, PLOT, etc.). Nur dort, wo die darzustellenden Grafiken erst algorithmisch berechnet werden müssen – bei der Zeichnung der Ballsegmente mittels Sinus-Cosnus-Funktionen (Zeile 220) und des Ballschattens mit dem Bresenham-Kreis-Algorithmus [vgl. Van Aken 1984] (Zeile 350) – werden solche BASIC-Befehle benutzt, weil diese durch die Möglichkeit der Koordinatenübergabe intuitiver einsetzbar sind und der Code dadurch kürzer wird (was zugleich seine Laufzeit verringert).

Vor allem der letzte Abschnitt, in dem die Animation stattfindet, bedient sich maschinennaher Operationen: Der OUT-Befehl (Zeile 510) implementiert die gleichnamige Funk-

tion der Z80-Maschinensprache, um auf einen Hardware-Kanal zu schreiben (hier zum Ansteuern des CRTC-Grafik-Bausteins). Mit CALL (Zeile 550) werden Maschinensprache-Routinen gestartet (hier: Routinen des Betriebssystems). Die Verwendung maschinennaher Operationen dient einerseits zur Beschleunigung der Ausführung, da diese in interpretierten BASIC-Programmen ein Problem darstellt. Andererseits kann von BASIC aus nur über diese Funktionen *direkt* auf den Grafikchip (CRTC) des Computers zugegriffen werden. Ein anderes hier eingesetztes Verfahren zur Beschleunigung ist die Verwendung von Integer-Variablen (gekennzeichnet durch ein %-Zeichen am Ende der Variablennamen), da BASIC-Dialekte aus didaktischen Gründen standardisiert mit Fließkommazahlen arbeiten [vgl. Kurtz 2009:83].

Die Ballgrafik wird in den Zeilen 100-280 aus Kreisbögen konstruiert. Dieses Verfahren ermöglicht es bereits beim Zeichnen ein perspektivisch verzerrtes Raster (für die roten und weißen Felder) auf die Balloberfläche aufzubringen (Zeile 220). Der Zeichenprozess lässt sich aufgrund seiner Langsamkeit gut verfolgen (siehe Abb. 4.1.9). Die hierbei zunächst noch zahlreichen Farben (die Demo benutzt die niedrigste Grafikauflösung MODE 0 mit 16 Farben bei 160×200 Pixeln) werden nach dem Zeichnen in 6 weiße und 6 rote Farbwerte geändert (Zeile 470). Während der Animation werden diese Farben durchrotiert (Zeilen 580, 590). Drei ‚statische' Farben bestimmen den Bildschirmhintergrund (0), die Farbe des Rasters (1), und des Schattens (15) (Zeile 480).

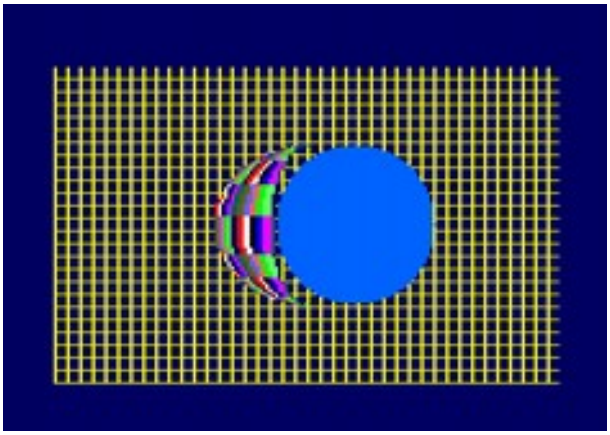


Abb. 4.1.9: Das Blitter-Programm (im Emulator JavaCPC) während der Generierung der Ballgrafik, rechts ist der kreisförmige Schatten bereits gezeichnet, links zeigt sich der halb fertige Ball bestehend aus vielfarbigen Kreisbögen

Die Verteilung von Rot und Weiß auf 13 Farbwerte ermöglicht die Rotationsillusion: In Zeile 490 werden die Farben in einer Endlosschleife von Rot zu Weiß gewechselt. Dies zitiert den Prozess der Boing!-Demo auf dem Amiga. Es ist nicht bekannt, ob der Autor von Blitter das trickreiche Verfahren der Amiga-Implementierung kannte. Es ist jedoch anzunehmen, dass er es aus denselben Gründen gewählt hat: Die Illusion einer Rotation durch Farbwechsel stellt kaum Anforderungen an die Systemperformanz, was bei einem mit 4 Megahertz getakteten 8-Bit-Computer und unter BASIC besonders wichtig ist.

Aufwändiger bewältigt das Programm die Sprungbewegung des Balls. Da die Amstrad CPCs in ihrer kleinsten Konfiguration über nur 64 Kilobyte RAM verfügen und weder das

Betriebssystem (BASIC) noch der Prozessor *viewport*-Prozeduren bereitstellt[47], musste eine andere Lösung gefunden werden. Diese basiert auf einer zweiten Täuschung: Über die OUT-Befehle (Zeile 510) werden jene Register des Grafik-Chips CRTC manipuliert, die für Hardware-Scrolling-Effekte genutzt werden. Mit dieser Funktion verschiebt der CRTC die Start- und Endpunkte des dargestellten Bildes (unabhängig von dessen Inhalt und ohne dafür die RAM-Adressen des Grafikspeichers zu manipulieren). Hier wird also die Videohardware, die für die Generierung des Ausgabebildes zuständig ist, selbst beeinflusst, um die Ballbewegung zu suggerieren. Die Werte, mit denen die CRTC-Register zyklisch geändert werden, entsprechen einer nach unten geöffneten Parabel (Zeile 510).

Der Effekt gelingt unter anderem deshalb, weil – wie bei der Amiga-Demonstration – der Hintergrund unbewegt scheint. Dies ist jedoch ebenfalls eine Illusion: Bei der Veränderung der CRTC-Register werden Bildbereiche, die ‚aus dem Bildschirm hinaus geschoben' werden, auf der gegenüberliegenden Seite ‚hinein geschoben'. Dieser Effekt lässt sich beobachten, wenn man das BASIC-Programm unterbricht, wodurch eine Textausgabe auf dem Bild erscheint, und dann mit dem Befehl CONT fortsetzt. Nun bewegt sich der Text ebenfalls auf dem Bildschirm und zeigt die Überlappungen der Ränder (vgl. Abb. 4.1.10). An diesem Text zeigt sich ebenfalls, dass die vertikalen und horizontalen Bewegungen nicht in Einzel-Pixel-Abständen stattfinden, sondern 8 Pixel sowohl vertikal als auch horizontal ‚überspringen'. Das ist genau die Distanz zwischen den Hintergrund-Rasterlinien und führt dazu, dass das Raster in den selben Abständen ‚springt', was aber aufgrund der Randüberlappung als Stillstand erscheint (Zeile 550).

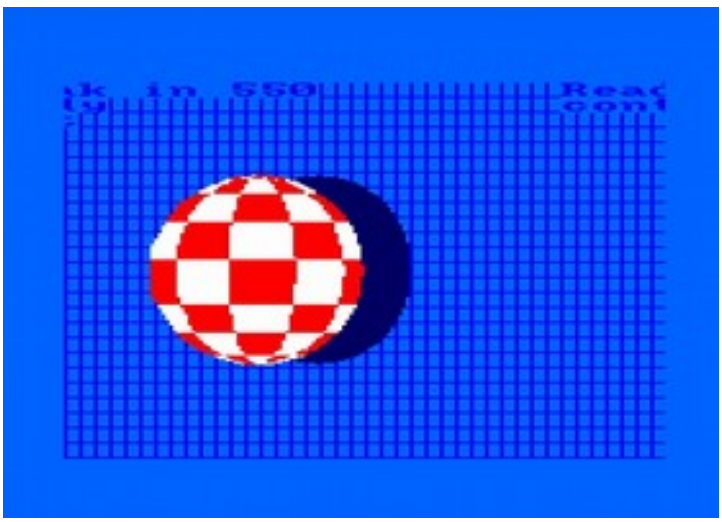


Abb. 4.1.10: Das unterbrochene und mit CONT fortgesetzte Blitter-Programm offenbart seine Scrolling-Illusion: Die Bildränder sind verschoben.

47 Auch die mit 128 Kilobyte RAM ausgestattet Variante CPC 6128 wäre hierfür nicht geeignet, weil die Bildschirmspeicher-Adressen bei allen CPC-Systemen gleich groß (16 Kilobyte) und im selben RAM-Bereich situiert sind ($C000_{16}$-$FFFF_{16}$), um die Kompatibilität der unterschiedlichen Modelle zueinander zu wahren.

4.1.3.4 Bouncing with the Beam: VCS Boing

Die letzte hier diskutierte Variante der Boing!-Demo stellt die unabhängig voneinander in zwei Etappen entwickelte Demo VCS Boing dar. 1999 entstand Amiga Boing! vom US-amerikanischen Hobbyisten David Galloway. Dieser hatte den Code der Demo Ho ho ho #1 (Amiga)[48] von Rob Kudla als BiggerBoing26[49] upgedated und später zu einer eigenständigen Amiga Boing Demo[50] ausgebaut. Der deutsche VCS-Hacker Sven Oliver Moll hat auf dieser Codebasis 2013 dann die im Folgenden untersuchte Demo VCS Boing! erstellt.

Die Systemspezifikationen [vgl. Hugg 2016:23-70; Montfort/Bogost 2009:12-15] der 1977 erschienenen Spielkonsole Atari VCS/2600 stehen im krassen Gegensatz zu denen des Commodore Amiga. Die VCS basiert auf der 8-Bit-CPU 6507 von *MOS Technology Inc.*, die dieselben Funktionen wie die 6502 desselben Herstellers besitzt – jedoch nur 13 anstatt 16 Adressleitungen, womit sie lediglich 8 Kilobyte Speicher verwalten kann. Dieser Speicher liegt komplett auf den ROM-Modulen, auf denen sich die Software (vor allem Spiele) für die Konsole befindet. Um beim Programmablauf Daten zwischenspeichern zu können, verfügt die VCS über insgesamt 128 Byte RAM, der auf dem RIOT-Chip[51] untergebracht ist. Diese engen Speicherbegrenzungen haben immer schon eine Herausforderung für VCS-Programmierer dargestellt – zeitgenössische Entwickler mussten ihre später zu vermarktenden Spiele in 8 Kilobyte unterbringen; heutige VCS-Hacker versuchen aufwändige Software auf der VCS zu implementieren oder sogar die 128 Byte Scratchpad-RAM für lauffähige Programme zu nutzen.[52]

Der TIA-Chip verwaltet die analogen Controller-Inputs, die Joystick-Feuerknöpfe und entlastet die 6507-CPU von der audiovisuellen Signalgenerierung: Er erzeugt die Grafik und stellt einen Soundgenerator zur Verfügung, der sowohl zur Klangerzeugung als auch für den Pseudozufallszahlen-Generator genutzt wird [vgl. Braguinski 2018:161ff.]. Er besitzt jedoch kein Video-RAM, sondern kann stets nur ein Pixel verwalten und zur Darstellung bringen. Dies zwingt Programmierer dazu, Grafiken in Echtzeit (im ‚Wettrennen' mit dem Kathodenstrahl des Bildschirms) generieren zu lassen. Um auf diese Weise Grafiken darzustellen, die über sich mehr als eine Rasterzeile erstrecken, muss auf das „Racing the Beam"-Verfahren zurückgegriffen werden: „Noch während der Rasterstrahl die Spritedaten darstellt, muss der Inhalt der Register geändert werden." [Moll 2014:73]

48 http://www.biglist.com/lists/stella/archives/199912/msg00006.html [letzter Abruf: 28.02.2018].

49 http://www.biglist.com/lists/stella/archives/200307/msg00008.html [letzter Abruf: 28.02.2018].

50 https://www.atariage.com/store/index.php?l=product_detail&p=321 [letzter Abruf: 28.02.2018].

51 RAM, I/O, Timer - http://www.ionpool.net/arcade/gottlieb/technical/datasheets/R6532_datasheet.pdf [letzter Abruf: 28.02.2018].

52 Das Spiel RAM Pong residiert ausschließlich in diesem Speicher und kann mit ‚gezogenem' Cartridge gespielt werden (vgl. https://www.pouet.net/prod.php?which=65469 [letzter Abruf: 28.02.2018]).

Die Grafikauflösung der VCS-Konsole beträgt 40 mal 192 Bildpunkte, wobei die horizontale Menge aus 20 Pixeln besteht, die auf unterschiedliche Weise gespiegelt dargestellt werden können. Die vertikale Auflösung beträgt in der PAL-Version 228, in der NTSC-Version 192 Zeilen. Große Bereiche des darstellbaren Bildschirmbereiches werden von der VCS nicht berücksichtigt, wie Abb. 4.1.11 zeigt:

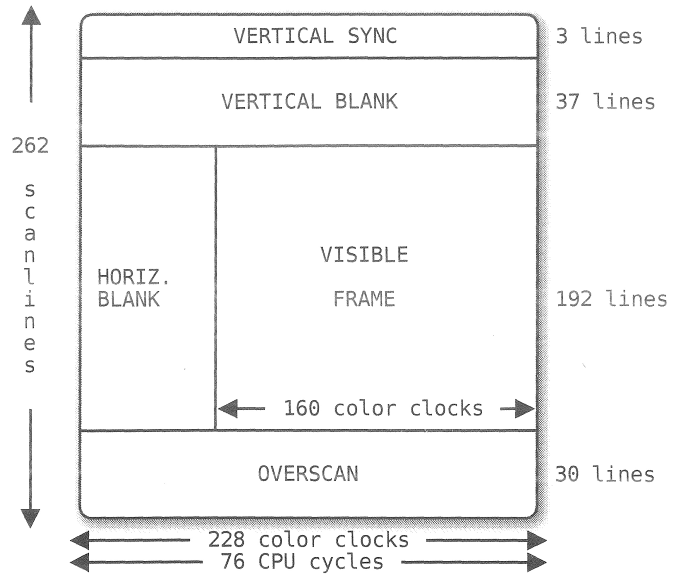


Abb. 4.1.11: Der Bildschirmaufbau des Atari VCS/2600 [Hugg 2016:34]

Diese limitierenden Faktoren hatten einen wesentlichen Einfluss auf die Adaption und die Ästhetik von VCS Boing. Zum einen kann der Ball nicht als Pixelmenge (48×42 Pixel [Moll 2014:70]) komplett auf den Bildschirm gebracht werden, sondern muss Bildpunkt für Bildpunkt im „Racing the Beam"-Verfahren zur Laufzeit generiert werden. Während der Animation muss dieser Prozess ständig wiederholt werden. Die Animation des Balls und die Realisation der Sprungbewegung resultieren aus diesem Darstellungsverfahren: Beim Zeichnen des Bildschirminhaltes wird die derzeitige Position Balls (das heißt: ob an der entsprechenden Bildschirmposition ein Ballpixel zu sehen ist oder nicht) sowie die Farbverteilung auf seiner Oberfläche (ob an der entsprechenden Bildschirmposition ein rotes oder weißes Pixel dargestellt werden soll) vom Programm in Echtzeit in die Register des TIA-Chips geschrieben.

Das Programm VCS Boing befindet sich in einem 4-Kilobyte-ROM, worin neben den Programm-Routinen auch die Grafikdaten Platz finden. Die Ballgrafik wird, anders als in der Amiga-Originalversion, hier nicht aus trigonometrischen Lookup-Tabellen zur Laufzeit berechnet, sondern bedient sich eines „vorberechne[te]n" [Moll 2014:70] Sets von Spritedaten (vgl. Datei *ballgfx.s*). Diese hat der Entwickler mit Hilfe von externen Tools generiert. Hierzu ließ sich Moll von einem Ball-Logo auf einer Internetseite inspirieren [vgl. ebd.]. Dennoch ließ sich der fertige Ball nicht als ganzes Darstellen: Die VCS kann aufgrund der o. g. Beschränkungen des TIA-Koprozessors nur ein 8 Pixel breites und 1

Pixel hohes einfarbiges Sprite generieren. Die Routine *initdone* berechnet die Ballgrafik an der aktuellen Position während des vertical blanks:

```
@initdone:
        lda #$01                        ; der 48-Pixel-Sprite benötigt die verzögerte
        sta VDELP0                      ; Ausgabe von Sprites (Vertical DELay Player 0
        sta VDELP1
        lda #$03                        ; die Anzahl der Sprites auf 3 mit einer 8 Pixel großen
Lücke
        sta NUSIZ0                      ; setzen (NUmber and SIZe of sprites 0) (player und
missile)
        sta NUSIZ1
        lda #$64                        ; die Farbe des Balls setzen: ein dunkles aber nicht
        sta COLUP0                      ; zu dunkles Rot (COLoUr Player 0)
        sta COLUP1
        ldx xpos                        ; die X-Position des Balls ins X-Register laden
        lda yindex                      ; Bit 7 enthält die Richtung, in die sich der Ball
bewegt:
        bpl @add                        ; 0 bedeutet nach rechts, 1 nach links
@sub:
        dex                             ; X-Position = X-Position - 1
        bne @xok                        ; ein Erreichen von 0 bedeutet
        and #$7f                        ; Richtungswechsel nach rechts
        bpl @setdir
@add:
        inx                             ; X-Position = X-Position [+] 1
        cpx #111                        ; ein Erreichen von 111 bedeutet
        bcc @xok                        ; (111 = 160(Bildschirmbreite)-48(Spritebreite)-1)
        ora #$80                        ; Richtungswechsel nach links
@setdir
        sta yindex                      ; geänderte Richtung speichern
        txa
        jsr spritepos           ; Position der beiden Sprites setzen
        lda xpos                        ; die Nummer des Animationsframes abhängig von der X-
Position
        lsr                             ; berechnen: jeden 2. Frame in Betracht ziehen, und dann
        sec                             ; Rest der Division durch 6 verwenden
@sbloop:
        sbc #$06                        ; solange 6 von der X-Position abziehen, bis ein
        bcs @sbloop                     ; Unterlauf stattfindet
        adc #>boing00 + 6               ; die 6 wieder aufaddieren zusammen mit [] dem Highbyte
        sta s0+1                        ; der Startposition im ROM und diese dann in allen
        sta s1+1                        ; Vektoren speichern
        sta s2+1
        sta s3+1
        sta s4+1
        sta s5+1
        jsr waitblank           ; alle Berechnungen sind erledigt, warten bis zur Darstellung
```

[Moll 2014:76 – formatiert: S. H.]

Konnte die originale Boing!-Demo auf dem Amiga noch auf den Multiplikations-Opcode `mulu` zurückgreifen, um die Zwischenwerte der Lookup-Tabellen-Einträge zu berechnen, so verfügt die Adaption auf der VCS-Konsole nicht über die Möglichkeit ihre Parabelwerte aus Multiplikationen zu ermitteln. Der Programmierer hat sich anderweitig beholfen, indem er einen Quadratur-Algorithmus als Additionsroutine geschrieben hat, der 64 x-Werte der Funktion $y=x^2$ (im Intervall $0<y<127$) im RAM der Konsole (in den Adressen 91_{16}-$D0_{16}$, vgl. Abb. 4.1.12) ablegt:

```
        ldx #$00
@paraloop:
        clc
        lda s0                  ; 16 Bit Addition: s0 = s0 + 1
        adc s1                  ; s0 wird dabei als Festkomma-Wert verwendet:
        sta s0                  ; das Highbyte sind die Vorkommastellen,
        lda s0+1                        ; das Lowbyte die Nachkommastellen
```

```
        adc s1+1
        sta s0+1                        ; 16 Bit Addition Ende
        sta bouncetab,x                 ; nur der Vorkommawert wird als Position verwendet
        lda s1                  ; 16 Bit Addition: s1 = s1 + $0010
;clc                                    ; nicht nötig, weil vorherige Addition
        adc #$10                ; nicht überlaufen konnte
        sta s1
        bcc @skiphi
        inc s1+1                        ; 16 Bit-Addition Ende
@skiphi:
        inx                             ; nächster Wert
        cpx #$40                        ; sind schon 64 Werte berechnet?
        bcc @paraloop           ; wiederholen, wenn nicht
[...]
```

[Moll 2014:75f. – strukturiert: S. H.]

Diese Implementierung steht am Ende einer Überlegung, die der Autor mit der Vorversion des Demo-Programms begonnen hatte:

> Bleibt noch die Bewegung auf der Y-Achse, die das Springen simuliert. In der ersten Version hatte ich dafür eine halbe Sinuskurve (also nur die obere Halbwelle) verwendet, die das recht ansehnlich simuliert. [...] Kurz darauf habe ich in einem ‚Wiki', das sich mit der Programmierung des C64 beschäftigt, einen Artikel entdeckt, der die Berechnung einer Sinuswelle über die Annäherung mit Hilfe einer Parabel durchführt (Holz 2010). Nun ist die Parabel eigentlich genau das, was einen fallenden Gegenstand beschreibt. Ich habe die Routine so angepasst, dass sie mir die Hälfte des RAMs des Atari VCS/2600 mit der „fallenden" Hälfte der Parabel füllt. Für die steigende Hälfte laufe ich die Tabelle einfach rückwärts ab. [...] Unterm Strich hat mir das Umbauen von einer Sinustabelle auf die Berechnung der Parabel etwas weniger als 100 Byte Speicherplatz im ROM gespart. [Moll 2014:71]

Hier zeigt sich der archäologische *modus operandi* von Retrocomputing-Projekten besonders deutlich: Hier wird eine Lookup-Tabelle als Vorbereitung der Animation per Routine generiert und nicht – wie bei Boing! - als Set von Daten im Programmcode hinterlegt. Dies spart wertvollen Speicherplatz (der oben zitierte Code hierfür ist 42 Byte lang, die 64 Einzelwerte hätten 64 Byte belegt) – erst recht, weil hier nicht mehr auf eine Sinusfunktion zurückgegriffen wurde, sondern auf eine im 6507-Assembler wesentlich einfacher und kürzer zu implementierende schnelle Quadratfunktion. Seine Intention war es ursprünglich gewesen, eine Bewegung entlang einer Sinuskurve zu generieren. Über den Umweg, einen Sinus durch einen Parabel anzunähern (ein Verfahren, das bereits in der indischen Mathematik des 6. Jhd. genutzt wurde [vgl. Joyce 2015]) hat der Programmierer dann unbewusst den physikalisch korrekten Algorithmus (Wurfparabel) in sein Programm eingeschrieben. Damit ruft er zugleich die Ball-im-Kasten-Implementierung in Erinnerung, die ebenfalls eine Wurfparabel als kontinuierlichen Verlauf von Spannungswerten auf ihre Ballfigur addiert hatte.

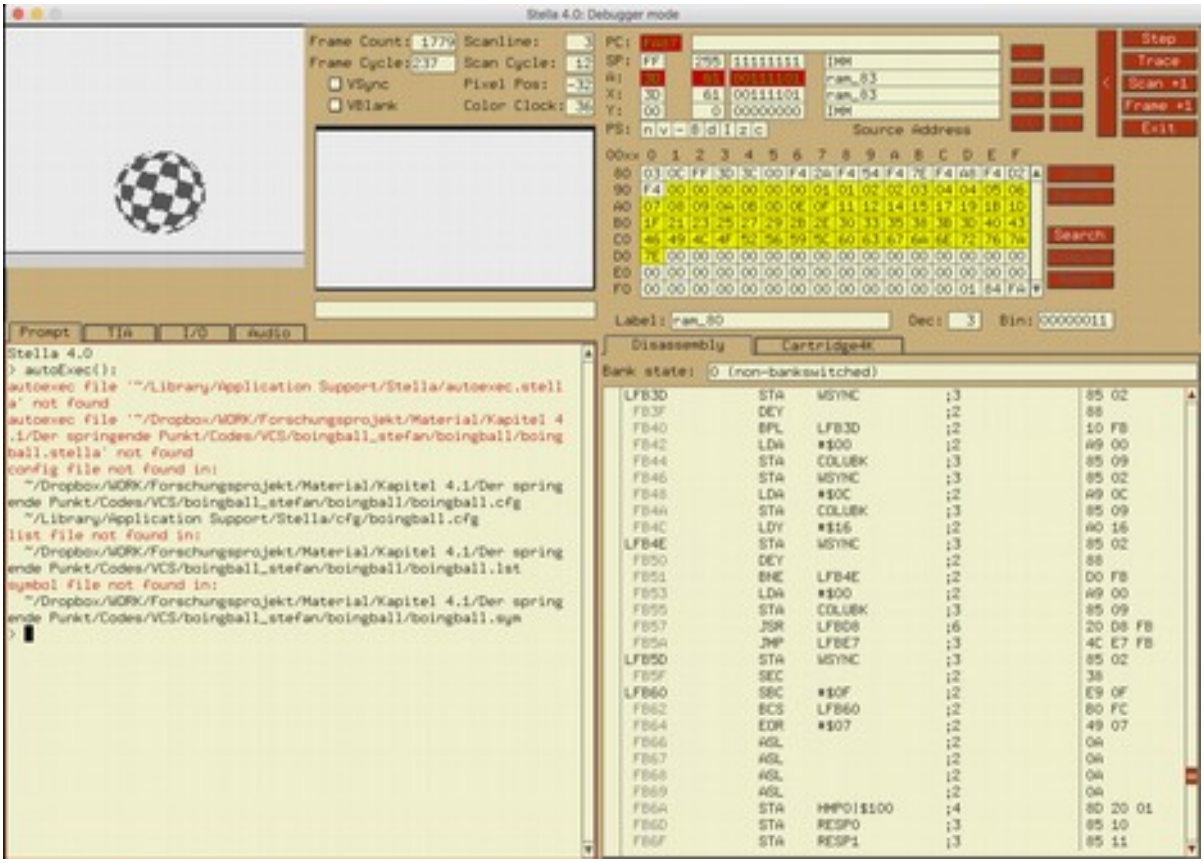


Abb. 4.1.12: Screenshot des VCS-Emulators Stella im Debugger-Modus, während VCS Boing ausgeführt wird. Die gelb eingefärbten Bereiche der 128-Byte-RAM-Tabelle enthalten die von der obigen Routine generierten Bahnkoordinaten der Wurfparabel.

4.1.4 Computerphilologie I: Menschen lesen Code

Die Ergebnisse der oben durchgeführten Programmanalysen sollen nun in einen theoretischen Zusammenhang gestellt werden. Dazu sollen deren Codes zunächst einer *linguistischen Analyse* unterzogen werden. Code wird dabei grundsätzlich als symbolisches (Programmtext), ikonisches (Schaltungen) und indexikalisches (Signale) Zeichensystem verstanden, das von Menschen und/oder von Maschinen gelesen und ‚verstanden' werden kann und muss. Während modernere Programmiersprachen (insbesondere objektorientierte Programmiersprachen) eine solche Lesart bereits aufgrund des ihnen zugrunde liegenden Paradigmas nahelegen [Baranovska 2018], lassen sich auch maschinennahe Sprachen wie Assembler oder „Sprache[n] ‚mittlerer Ebene' (middle-level languages)" [Sommergut 1994:18], mit Möglichkeiten des direkten Hardwarezugriffs, wie C und BASIC, linguistische Deutungskonzepte in Anschlag bringen – sowohl auf der synchronen wie auch auf der diachronen Ebene.

Für eine solche Analyse wird zunächst ein Begriff von *Computerphilologie* vorgestellt, der versucht die genannten Symbolsysteme für eine computerarchäologische Beschreibung

nutzbar zu machen und zugleich Möglichkeiten für das Verstehen historischer Programme aufzuzeigen. Die Schritte hierbei sind:

1. Die einzelnen Demoprogramme sollen im Sinne einer Software-Geschichte als historisch progredienter *Rezeptionsprozess* verstanden werden.
2. Die Codes der einzelnen Demoprogramme sollen in ihrer *intertextuellen Beziehung* zu einander unterschiedlichen Paratexten betrachtet werden.
3. Die Verweisstrukturen innerhalb der Codes sollen auf Merkmale der *Kohäsion* untersucht werden.

Hierdurch werden Möglichkeiten eröffnet, Code-*Lektüren* in einem philologischen Diskurs zu situieren und so für eine Software Preservation fruchtbar zu machen. Im darauffolgenden Kapitel (4.1.5) wird dann eine zusätzliche Perspektive eingenommen:

4. Die Eigenzeitlichkeit des Mediums Computer im Vollzug der Demoprogramme soll als Basis der *maschinellen Lektüre* von klassisch-philologischen Beschreibungen abgegrenzt und für eine technische Analyse fruchtbar gemacht werden.

Unter *maschineller Lektüre* soll dann jedoch weder die Auswertung von Code-Texten durch technische Verfahren noch eine computergestützte Analyse natürlicher Sprachen, wie sie die Computerlinguistik erforscht, verstanden werden, *sondern die vom Computer vollzogene Ausführung des Codes als maschineller Lektüreprozess*. Mit dieser Sichtweise soll noch einmal auf die Notwendigkeit der Operativität von Computern hingewiesen werden, die bei der Ausführung eines Programm(code)s ein gänzlich anderes ‚Lektüreergebnis' produziert als die menschliche Rezeption desselben.

Die Methodik für diese Analyse ist aus der Philologie, insbesondere der Literatur- und Sprachwissenschaft abgeleitet. Der Terminus *Computerphilologie* weist bereits eine längere Forschungstradition auf. Dabei haben sich zwei Verständnisweisen herausgebildet. Diese sollen zunächst vorgestellt, danach die spezifischen philologischen Methoden definiert und auf die oben vorgestellten Programmcodes angewendet werden.

Die Konfrontation der Philologie(n) mit Computern hat bereits in den 1970er- und 1980er-Jahren stattgefunden, wie Balke und Garderer [2017] notieren: „Computerphilologie [.. widmete] sich digital gestützten Editionsaufgaben" [ebd.; Rieger 1970], womit der Begriff hier als *Philologie mithilfe des Computers*[53] verstanden werden kann – was eine Protoform der *Digital Humanities* innerhalb heutiger philologischer Disziplinen darstellt [vgl. Lauer 2009]. Dem gegenüber steht ein Verständnis des Begriffs, wie es Friedrich Kittler [1996] vorgeschlagen hat: als eine *Philologie angewendet auf Computer* (ihre Prozesse und Texte). Diese sei angesichts einer durch Computer maßgeblich veränderten Sicht auf Texte notwendig geworden. Kittler weist der Computertechnologie einen Einfluss auf den Begriff Text und unseren Umgang damit zu: Dadurch, dass Computer Texte (Programme)

53 https://www.digitalhumanities.tu-darmstadt.de/index.php?id=37 [letzter Abruf: 16.03.2018].

verwenden, um andere Texte (Daten, z.B. Textdateien) zu kodieren [vgl. ebd.:241], ergeben sich neue Formen von Alphabetismus und Analphabetismus. Der Mangel an code literacy[54] wird vor allem deshalb als problematisch angesehen, weil Computertechnik maßgeblichen Einfluss auf das durch sie gespeicherte, übertragene und verarbeitete Wissen bekommt:

> Was schließlich mit denen geschieht, die Codes nicht einmal lesen können, ist in der Theorie klarer als in zukünftiger Empirie. Theoretisch verrät es schon die stehende Redewendung, derzufolge Benutzer, was immer sie an der Konsole tun, es ‚unter' einem bestimmten Betriebssystem wie Unix oder Windows tun: Der Computeranalphabet als solcher ist, mit anderen Worten, zum Subjekt oder Untertan einer Corporation geworden. Er unterliegt dem digitalen Code genauso massiv und undurchschaubar wie etwa seinem genetischen Code. Davor behütet auch der geläufige Alphabetismus der Buchstaben in keiner Weise. [ebd.:244]

Von Kittlers These ausgehend, entwerfen Balke und Gardener einen neuen Begriff von Medienphilologie, der diesen „geläufigen Alphabetismus" für Medien einfordert. Ohne jedoch bloß die Methoden der Philologie auf Medieninhalte zu übertragen [Balke/Garderer 2017:11f.] oder, wie Kittler, allein das medientechnische Apriori ins Bewusstsein des Philologen rücken zu lassen, fordern sie „die Wirkung des Medialen auf der Ebene der Ereignisse" [13] zu betrachten. Damit sind die zeitkritischen Prozesse des Mediums gemeint, die – im Falle des Computers – durch Code auf der Hardware initiiert aber noch nicht repräsentiert werden.

Medienphilologie erweitert also das Arbeitsfeld der klassischen Philologie, indem sie ihr einen „anderen Arbeits- und Schauplatz eröffne[t]" [13]. Neben die *Texte in den Medien (Codes)* und *den Einfluss der Medien auf die Texte (medientechnisches Apriori)* rücken hierbei auch *editionsphilologische* [vgl. Van Hulle 2008] Fragen in den Fokus. Untersuchen Philologen Texte auch diachron und problematisieren dabei „den historischen Abstand zwischen uns und Texten längst vergangener Epochen" [Balke/Garderer 2017:15], widmen Medienphilologen den Schnittstellen zwischen Mensch und Maschine und zwischen Maschinen untereinander ihr Augenmerk [vgl. ebd.]. Die hier stattfindenden mikrozeitlichen Prozesse lassen sich allerdings nicht mehr mit historischen Kategorien benennen.

Eine für solche Zeitprozesse sensible Medienphilologie stellt Wolfgang Ernst [2016] vor und versteht dabei Digital Humanities als „humanities of the digital" [Ernst 2018] – also eine Anwendung von geisteswissenschaftlichen Methoden auf Computerprozesse. Besonderes Augenmerk richtet er hierbei auf die philologische Berücksichtigung von operativen Medien:

54 http://www.zeit.de/digital/internet/2013-10/code-literacy-zuckerberg-gates [letzter Abruf: 13.03.18]. Zu dieser Forderung ist angesichts der elektronischen Verarbeitung großer Forschungsdaten-Mengen jüngst eine Forderung nach „data literacy" (https://gi-radar.de/216-data-literacy/ [letzter Abruf: 06.06.2019]) hinzugekommen.

> Medienphilologie [stellt] die logische Eskalation der klassischen Philologie dar. Das aus der Literaturwissenschaft vertraute *close reading* textueller Semantik verschiebt sich hin zur syntaktischen Grammatologie logischer Schaltpläne und Quellcodezeilen; an die Stelle von textgenetischer Historie treten Zeitdiagramme[] in ihrer realen elektronischen Verkörperung. Damit kommen textkritische Tugenden, wie sie aus der Diplomatik, also der Urkundenforschung als Hilfswissenschaft der Historie ve[r]traut sind, für alphanumerische Textartefakte zum Zug. [Ernst 2016]

Medienphilologie wendet also zunächst Methoden der Philologien (sowohl von Literatur- und Sprachwissenschaft als auch von Komparatistik – jeweils in synchroner und diachroner Perspektive) an, um damit technische Prozesse zu analysieren. Dadurch, dass die Zeichensysteme sowohl symbolischer (Code) als auch ikonischer (Diagramme, Schaltungen) als auch indexikalischer (Signale) Art sein können, erfordert dies eine Erweiterung des philologischen Instrumentariums dort, wo nicht mehr dezidierte *Texte* der Gegenstand der Analyse sind.

Während die epistemologisch orientierte Medienwissenschaft [vgl. Ernst 2016] den Diskurs der Medien- und Computerphilologie bereits theoretisch und theoriegeschichtlich untersucht, soll hier ein Methoden-Set vorgestellt werden, mit dem Computerphilologie praktisch betrieben werden kann. Theorien hierzu existieren in der (vor allem Theoretischen) Informatik bereits seit der Adaption des Chomsky-Hierarchie-Modells [Chomsky 1956] auf formale Sprachen und der (generativen) Phrasenstrukturgrammatik [Chomsky 1965] zur Backus-Naur-Form [Backus 1959] – jedoch vor allem als *synthetisches* Instrument der Sprachgenerierung und -klassifizierung. Die Applikation der Linguistik natürlicher Sprachen auf informatische Gegenstände findet sich vereinzelt – etwa zur Beschreibung von Hash-Funktionen [Wüster 1974:98-103] oder in der HCI-Forschung [Zeller 2005]. Das nachfolgende *analytische Set* setzt sich aus Elementen der drei philologischen Bereiche Rezeptionstheorie, Intertextualitätstheorie und Textkohäsionstheorie zusammen.

4.1.4.1 Rezeptionstheorie: Evolution der Problemlösungen

Rezeption meint die Aufnahme einer Information durch einen Empfänger (Rezipient) und wird von der Informationstheorie, Kommunikationswissenschaft, Physiologie und den Kunstwissenschaften untersucht. Letztere widmen sich dabei sowohl der Frage, wie Kunstwerke in der *synchronen* Rezeption aufgenommen und verstanden werden (Wirkung auf den Betrachter) als auch, wie die Rezeption von Kunstwerken andere Künstler *diachron* beeinflusst (Wirkung auf die Geschichten der jeweiligen Kunstgattung) und in eine ‚produktive Rezeption' einmündet, wie der Literaturhistoriker Hans Robert Jauß schreibt:

> Die rezeptionsästhetische Theorie [...] erfordert auch, das einzelne Werk in seine ‚literarische Reihe' einzurücken, um seine geschichtliche Stelle und Bedeutung im Erfahrungszusammenhang der Literatur zu erkennen. Im Schritt von einer Rezeptionsgeschichte der Werke zur ereignishaften Geschichte der Literatur zeigt sich diese als ein Prozeß, in dem sich die passive Rezeption des Lesers und Kritikers in die aktive Rezeption und neue Produktion des Autors umsetzt oder in dem – anders gesehen – das nächste Werk formale und moralische Probleme, die das letzte Werk hinterließ, lösen und wieder neue Probleme aufgeben kann. [Jauß 1975b:141]

Die Literaturgeschichte widmet diesem Prozess seit der zweiten Hälfte der 1960er-Jahre mit der so genannten *Rezeptionstheorie* ein eigenes Forschungsfeld innerhalb der Literaturwissenschaft. Wurde Literaturgeschichte bis dahin als eine auf den Autor und die Epoche konzentrierte Forschung betrieben [vgl. ebd.:147], so fragen die Vertreter der Rezeptionstheorie nach der kommunikativen Autor-Text-Leser-Trias und den Verständnisprozessen von Texten vor den jeweiligen historischen Rezeptionshintergrund des Lesers. Wenn es sich bei ihm um einen „informierte[n] Leser" [Fish 1979:215] handelt, der neben der Sprache des Textes und dem notwendigen Weltwissen auch über „literarische Kompetenz verfügt" [ebd.], dann entfalten sich auch literaturhistorische Bezüge eines Textes im Leseprozess.

Die Literaturgeschichte wird von der Rezeptionstheorie daher nicht als ‚vergangen' betrachtet, sondern sie vergegenwärtigt/aktualisiert sich im Prozess der Rezeption. Jauß schreibt über die Tradierung von (literarischen) Texten, dass

> [...] eine literarische Vergangenheit also auch nur wiederkehren kann, wo eine neue Rezeption sie in die Gegenwart zurückholt, sei es, daß eine veränderte ästhetische Einstellung sich Vergangenes im gewollten Rückgriff wieder aneignet, sei es, daß von dem neuen Moment der literarischen Evolution auf vergessene Dichtung ein unerwartetes Licht zurückfällt, das etwas in ihr finden läßt, was man zuvor nicht in ihr suchen konnte. Das Neue ist also nicht nur eine ästhetische Kategorie. [Jauß 1975b:144]

Autoren (also schreibende Leser) setzen sich mit ihren Werken selbst zur Literaturgeschichte ins Verhältnis und kodieren literaturhistorische Bezüge bewusst oder unbewusst in ihre Texte, die wiederum von Lesern verstanden werden (oder nicht). Im Schreiben aktualisieren sie damit Literaturgeschichte, wie sich diese beim Leser (als *re-enactment* [vgl. Collingwood 1999b:240]) mental vergegenwärtigt.

Dieser Prozess findet auch bei Computer-Code statt und trifft in der hier geführten Diskussion auf alle Implementierungen der Ballsprung-Demonstrationen zu. Die historisch distinkten Rezeptionen des Ballsprung-Motivs stellen dabei jeweils dessen Aktualisierung („Evolution" [Jauß 1975:141ff.]) dar. Das neue Element, das sie damit in die Chronik dieser

Demoprogramme einschreiben, ist, wie sich Jauß weiter interpretieren lässt, aber stets auch historisch:

> Das Neue wird auch zur historischen Kategorie, wenn die diachrone Analyse der Literatur zu der Frage weitergetrieben wird, welche historischen Momente es eigentlich sind, die das Neue einer literarischen Erscheinung erst zum neuen machen, in welchem Grade dieses Neue im historischen Augenblick seines Hervortretens schon wahrnehmbar ist, welchen Abstand, Weg oder Umweg des Verstehens seine inhaltliche Einlösung erfordert hat, und ob der Moment seiner vollen Aktualisierung so wirkungsmächtig war, daß er die Perspektive auf das Alte und damit die Kanonisierung der literarischen Vergangenheit zu ändern vermochte. [Jauß 1975b:144]

Betrachtet man bestimmte (distinkte) Elemente der Softwaregeschichte, wie zum Beispiel ein Programm-Genre, die Implementierung eines bestimmten Algorithmus, die Verwendung einer spezifischen Schnittstelle oder die Simulation eines physikalischen Prozesses, so lässt sich für diese auf der Oberfläche (der Ausgaben) wie auch auf der Unterfläche (der Codes und Schaltungen) eine rezeptionstheoretisch organisierte Chronologie erstellen. Der Wert einer solchen wäre ein zweifacher:

1. Codes werden zu einer ‚gewöhnlichen' Textsorte, indem sie philologischen Prozessen zugänglich gemacht werden. Ihrer Unsichtbarkeit als graue Literatur ließe sich damit im Sinne der Software Preservation ebenso entgegenwirken wie der Annahme, sie seinen technisch zu kompliziert, um sie als Texte verstehen zu können.

2. Bestehende und neue Historiografien könnten durch eine Software-Rezeptionstheorie mit belastbare(re)n Argumenten und konkreten Beispielen angereichert werden. Das Archiv der Softwaregeschichte könnte auf diese Weise für eine valide(re) Form der Computergeschichtsschreibung konsultiert werden.

Für die literarische Rezeptionstheorie bieten die Archive allerdings nur wenige empirische historische Belege für private Rezeptionsprozesse (also von Lesern, die weder Rezensionen noch wissenschaftliche Sekundärliteratur dazu verfasst haben). Im Unterschied zur *empirischen Rezeptionsforschung* richtet die diachron ausgerichtete Rezeptionstheorie ihr Augenmerk auf einen idealisierten Leser, den sie im Text als „impliziten Leser" vom Autor adressiert annimmt [vgl. Warning 1975:32]. Rezeptionstheorie untersucht dann, wie ein Autor die Kommunikation mit seinen Leser über diesen impliziten Leser gestaltet und ist also in den meisten Fällen gezwungen ihre Belege für Rezeptionsprozesse über die Interpretation herbeizuführen [vgl. Janich 2008:195f.]. Konkreter kann diese Betrachtung jedoch dann werden, wenn sich die Rezeption in Texten niederschlägt. Autoren referenzieren das von ihnen zuvor gelesene aber zumeist nicht explizit, sondern implizit (und ‚vermischt' mit unterschiedlichsten Einflüssen/Diskursen [vgl. Foucault 2001b]).

Bei Computerprogrammen sind die Bezüge zu vorgängigen Programmtexten ebenfalls nicht immer explizit bestimmbar. Dennoch lassen sich hier Beobachtungen bezüglich verwendeter Algorithmen unternehmen. Die oben skizzierten technischen ‚Mängel' der Computer, seien sie nun als kreative Beschränkungen verstanden oder schlicht als Defizite, ließen die Programmierer auf jeweils unterschiedliche „Urtexte" [Van Hulle 2008:149] der Programmierung zurückgreifen – die (Standard-)Algorithmen [vgl. Zaks 1986:19]:

In Ermangelung eines Speichers und überhaupt einer numerischen Erfassung von Funktionswerten zum Beispiel nutzt die Analogcomputer-Simulation Ball im Kasten die Möglichkeiten von Integriererschaltkreisen, um über Differentialgleichungen den Kathodenstrahl sinus- und kosinusförmig abzulenken – ein Algorithmus aus der Zeit der barocken Mathematik Roger Cotes' [vgl. Katz 1987:315]. Aufgrund des Fehlens einer FPU, die trigonometrische Funktionen zur Verfügung stellen könnte, hinterlegen die Boing!-Programmierer die Sinus- und Kosinuswerte als Tabelle im Programmtext – ein Verfahren, das bereits in der spätmittelalterlichen Nautik [vgl. Peschel 1869:7] bekannt ist. Moll nähert den für seine Wurfparabel gewünschten Sinusgraphen über eine Parabelfunktion an – das Verfahren geht auf die indische Mathematik des 6. Jhd. [Joyce 2015; Gupta 1967] zurück. Die Langsamkeit des BASIC-Interpreters lässt Bond für sein Blitter-Programm auf den Bresenham-Kreisalgorithmus zurückgreifen [Bresenham 1965; Horn 1976; Van Aken 1984]. Die Rotationsillusion des Balls erzeugen die meisten Programmierer mittels Farbwert-Rotation – ein ebenfalls ‚kanonisches' Verfahren der Bewegungssuggestion [kunsthistorisch: Schneider 1976], wird der Effekt doch bereits auf der Ebene der Digitallogik (die den maschinensprachlichen Ballrotations-Routinen explizit zugrunde liegt) als *Rotierfunktion* [technikhistorisch: Flowers 1983:246] beschrieben (vgl. Tab. 4.1.1).

Neben der Rezeption von Algorithmen, finden sich auch explizitere Formen der Wiederaufnahme von Code. Programmtexte haben häufig dazu eingeladen von Personen, die nicht ihre Autoren sind, modifiziert zu werden. Ein solcher Umschreibprozess stellt eine Rezeption im hier verstandenen Sinne dar. Insbesondere im Bereich der Open-Source-Bewegung werden Leser sogar direkt dazu aufgefordert selbst zu Autoren zu werden und den vorliegenden Sourcecode zu bearbeiten.

Auch Blitter lädt den Computernutzer explizit dazu ein, seinen Code zu studieren und zu modifizieren:

```
550 t%=x%+<80*y>:FOR a=1 TO 4:CALL&BD19:NEXT:'Use FRAME instead of CALL&BD19 on 664 & 6128
```

[Bond 1968:78]

Titel	System	Implementierung	Ballgrafik	Ballrotation	Sprunganimation
Ball im Kasten	Analogrechner Telefunken TA-742	Hardware (Verstärker-Schaltungen)	Sinus/Kosinus aus Differential-gleichung	Keine	Physikalisch (Wurfparabel)
LED Pong	Signetics Instructor 50	Signetics-2650-Assembler	Datenbus-Bit in LED-Anzeige	Keine	Parametrisch (kontingent)
Boing!	Commodore Amiga	Motorola-68000-Assembler	Sinus/Kosinus aus 16-Bit Lookup-Tabelle	Farbwert-Rotation	Mathematisch (Parabel)
Boing! Reconstruciton	Commodore Amiga	Lattice C 5.02 kompiliertes Programm	Grafik als Bitmap im Code hinterlegt	Farbwert-Rotation	Mathematisch (Parabel)
Blitter	Amstrad CPC	Locomotive-BASIC 1.0 mit Hardware-Zugriff	Sinus/Kosinus und Bresenham-Kreis-Algorithmus	Farbwert-Rotation	Mathematisch (Parabel)
VCS Pong	Atari VCS	MOS-6507-Assembler	Grafik als Bitmap im Code hinterlegt	Farbwert-Rotation	Mathematisch (Parabel)

Tab. 4.1.1: Verwendete Algorithmen in den Ballsprung-Demonstrationen

Der Text hinter dem Apostroph ist ein Kommentar des Programmautors; ein Aufruf an den Leser/Abtipper seines Programmtextes, ihn zu modifizieren. Blitter lädt grundsätzlich zum Experimentieren ein: Bond hat in seinen Code eine Funktion implementiert, die nach dem Abbruch des Programms durch drücken der Taste F1 den Bildschirm in den Ursprungszustand zurückversetzt und den Programmcode mittels LIST-Befehl auf dem Bildschirm ausgibt. Sogar das Drücken der RETURN-Taste zum Starten des Prozesses ist hier bereits implementiert (vgl. Programmzeile 430). Konnte sich der Nutzer beim Abtippen von Blitter bereits erste Gedanken über die Funktionalitäten des Programms machen, so bekommt er hiermit die Möglichkeit den Code und seine Ergebnisse zueinander in Beziehung zu setzen. Mit Hilfe des CPC-Handbuchs sowie der verfügbaren Sekundärliteratur ist er in der Lage die Tricks von den Algorithmen bis hinab auf die Hardwareebene zu durchschauen. Er kann auf diese Weise seine Wahrnehmung der Ballsprung-Simulation zum Programmtext in Beziehung setzen und das Programm nach eigenen Vorstellungen modifizieren.

Auch dort, wo sich Leser am Gelesenen nicht programmierend, sondern kommentierend geäußert haben – durch glossierende Randanmerkungen – existiert ein belegbarer Einblick in den Rezeptionsprozess. Diese Möglichkeit wird bei der Lektüre von Code häufig angewandt: Durch das Hinzufügen von Kommentaren strukturiert und erklärt sich ein Leser die Bedeutung von fremdem Code. Damit erhält eine auch an der Rezeptionsge-

schichte von Code interessierte Software Preservation belastbare empirische Belege über menschliche Verständnis von Programmcodes.

Am Disassembly des Boing!-Programms lässt sich dies exemplarisch zeigen:

```
                        lea   (sc_RastPort,a0),a1     ;poking screen rastport.. nasty
[...]
.bgrenderloop           move.l (rp_BitMap,a3),a0      ;poking screen bitmap pointer.. even
nastier
[...]
.L115                   move.l d0,d1                  ;poking colortable directly.. again nasty
[...]
.L5                     move.l a2,-(sp)               ;BUG: does AddPort before msglist is
initialized!
```

Die hier ausgewählten Codezeilen enthalten laut Maher[55] Kommentare des Lesers Harry Sintonen. In den ersten drei hier zitierten Zeilen bewertet Sintonen die direkte Manipulation von Hardware-Registern des Copper im Programmcode als „nasty“. Solche Direktzugriffe sind von Assembler aus zwar möglich, sollten aufgrund der Kompatibilität von Programmen mit künftiger Hardware (die vielleicht andere Adressen für diese Register einführt) vermieden werden; anstelle dessen sollten Einsprungvektoren[56] genutzt werden. Die von den Codeautoren (Mical/Luck) vorgenommene direkte Programmierung der Grafikhardware könnte auf die knappe Entwicklungszeit und den Prototypen-Status des Amiga-Computers zurückzuführen sein. In der vierten Codezeile hat Sintonen überdies einen Programmierfehler, der zu einem Timingproblem beim Aufbau der Bildschirmgrafik führen könnte, identifiziert und dies kommentiert.

Sintonen rezipiert den disassemblierten Boing!-Code vor dem Hintergrund von dessen Historizität und historischer Bedeutung und vergleicht ihn mit seinem kontemporären Wissen über Programmierung und das aktuelle Wissen über das System. Damit stellt er einen „informierten Leser“ im Sinne der Rezeptionstheorie dar. Seine Lektüre basiert auf dem Erkennen von Referenzen. Diese bedürfen bei Codes (anders als bei literarischen Werken) keiner diskursiv verhandelbaren Textauslegung, sondern können auf Basis von empirischen Beobachtungen belegt werden. Code-Rezeption kann jedoch auch in Form eines neuen Programms auftreten (wie oben dargestellt, wenn ein Autor das Werk eines anderen Autors rezipiert), wie im Fall der Boing! Reconstruction zeigt:

```
/*
File: boing5.c

What follows is the fifth of five stages of a reconstruction of the original Amiga Boing demo
that was written by Dale Luck and R.J. Mical in 1984 to 1985. This version was coded by Jimmy
Maher in 2009 to 2010, and may be freely distributed.
```

55 http://amiga.filfre.net/?page_id=5 [letzter Abruf: 16.03.2018].

56 Betriebssystemroutinen wurden bei Homecomputern von den Entwicklern oft über Einsprungadressen (so genannte Vektoren) erreichbar gemacht. So konnte einerseits Code später geupdatet werden (wobei nur die der hinter der Einsprungadresse stehende Vektor geändert werden musste) und andererseits gaben die aufgelisteten Einsprungadressen [vgl. Brückmann u.a. 1985:115-140] dem Nutzer die Möglichkeit Betriebssystemroutinen leicht von eigenen Programmen aus aufzurufen (vgl. Kap. 4.1.4.2).

```
This program was developed with Lattice C 5.02 running under KickStart and Workbench 1.3, and
therefore should be certain compile successfully in that environment. Your milage may vary
with other environments.

This final stage of the reconstruction adds the sampled "boom" sound.
*/[57]
```

Hier bezieht sich der Autor (Maher) implizit auf das Disassembly Sintonens und explizit auf das Programm Micals/Lucks. In einem solchen Fall lassen sich die konkreten Rezeptionsbezüge zwischen den unterschiedlichen Codes als *Intertextualität* kategorisieren.

4.1.4.2 Intertextualität: reCALLing code history

Der Begriff Intertextualität entstammt der Literaturtheorie der 1960er-Jahre und hat von dort ausgehend eine äußerst heterogene Bedeutungsänderung erfahren [vgl. Janich 2008:178f.]. „Intertextualität [...] bezeichnet als theoretischer Begriff zunächst nichts mehr [als] das Phänomen einer wie auch immer festzulegenden Relation zwischen Texten." [Holtius 1993:29] Diese Beziehung kann *unterschiedlich direkt* (ein Text zitiert/paraphrasiert/alludiert/... einen anderen Text) und vermittelt durch *unterschiedliche Signale* (das, „was der Text selbst zur Identifizierung einer intertextuellen Relation beitragen kann" [Holtius 1993:33]) hergestellt werden. Mit dem Grad der Vagheit der Signale, wird das Erkennen von Intertextualität zu einem Deuten innerhalb einer (literaturwissenschaftlichen) Interpretationstätigkeit.

Diese Vagheit vermeidet die sprachwissenschaftliche Textlinguistik, die den Begriff der Intertextualität restriktiver versteht als eine „nachweisbare[] Bezugnahme" [Janich 2008:182], bei welcher verschiedene Grade von intertextuellem Bezug operationalisierbar [ebd.:181f.] sind und in ihrer kommunikativen Wirkung untersucht werden. Der französische Literaturwissenschaftler Gerard Genette hat, obgleich er Literaturhistoriker und kein Textlinguist war, eine sprachwissenschaftlich nutzbare [vgl. Janich 2008:186] Klassifikation intertextueller Bezüge und ihrer Wirkungen vorgenommen. Sie sollen im Folgenden verwendet und hier vorgestellt werden:

- *Intertextualität* meint bei Genette die „Beziehung der Kopräsenz zweier oder mehrerer Texte, d. h. in den meisten Fällen, eidetisch gesprochen, als effektive Präsenz eines Textes in einem anderen Text." [Genette 1993:10] – etwa durch Zitate, Anspielungen etc.
- *Metatextualität* ist „die üblicherweise als ‚Kommentar' apostrophierte Beziehung zwischen einem Text und einem anderen, der sich mit ihm auseinandersetzt, ohne ihn unbedingt zu zitieren (anzuführen) oder auch nur zu erwähnen". [Genette 1993:13]

57 Vgl. „C source codes" 1-5 auf: http://amiga.filfre.net/?page_id=5 [letzter Abruf: 13.2.2018].

- *Hypertextualität* bezeichnet, abweichend von der heutigen, technischen Definition[58], „jede Beziehung zwischen einem Text B (den ich als Hypertext bezeichne) und einem Text A (den ich, wie zu erwarten, als Hypotext bezeichne), wobei Text B Text A auf eine Art und Weise überlagert, die nicht die des Kommentars ist." [Genette 1993:14f.]
- Bei *Architextualität* „handelt es sich um eine unausgesprochene Beziehung, die bestenfalls in einem paratextuellen Hinweis auf die taxonomische Zugehörigkeit des Textes zum Ausdruck kommt (in Form eines Titels wie *Gedichte, Essays* oder *Der Rosenroman* usw. oder, was häufiger der Fall ist, eines Untertitels, der den Titel auf dem Umschlag ergänzt, etwa Hinweise wie *Roman, Erzählung, Gedichte* usw.)." [Ebd.:13]
- Der *Paratext* wird von Genette schließlich definiert als „jenes Beiwerk, durch das ein Text zum Buch wird und als solches [...] vor die Öffentlichkeit tritt." [Genette 2001:10] Paratextualität bezeichnet grundsätzlich „die gemeinsame Anwesenheit mehrerer Texte in einem, also die klassischen Formen der Bezugnahme eines Phänotextes auf einen oder mehrere Referenztexte wie Zitat, Anspielung, Plagiat u. a." [Janich 2008:185] Paratextualität wird von Genette noch weiter differenziert:
  - *Epitexte* sind all jene Paratexte, die „im Umfeld eines literarischen Werkes, mit denen ein Autor, beispielsweise in Form von Selbstanzeigen und Interviews, ein Werk aus seiner Sicht erläutert." [Genette 2001:7]
  - *Peritexte* finden sich „im Umfeld des Textes, innerhalb ein und desselben Bandes, wie der Titel oder das Vorwort, mitunter in den Zwischenräumen des Textes, wie die Kapitelüberschriften oder manchen Anmerkungen" [Genette 2001:12].

Die vom Leser vorgenommene Text-Glossierung [Genette 2001:305-307], die oben bereits als eine Randnotiz erwähnt wurde (vgl. Kap. 4.1.4), fällt, weil sie keinen offiziellen Charakter besitzt, sondern rein privat ist, nicht in die Rubrik der Paratexte. Sie erfüllt aber eine ähnliche Funktion wie der Metatext: Sie kommentiert, erklärt, stellt Bezüge des nebenstehenden Textes zu anderen Texten oder zu Gedanken des Lesers her.

Vereinzelt sind einige dieser Kategorien bereits (implizit) in die Programmiersprachen-Entwicklung eingeflossen – und zwar vor allem dort, wo Programmiersprachen entworfen wurden, die sich (etwa zur leichteren Erlernbarkeit) an natürlichen Sprachen orientieren: Grace Hoppers Proto-COBOL-Sprache FLOW-MATIC [Knuth/Pardo 1976:89f.] als eine „application of basic English to describe both data and the procedures to be executed" [Beyer 2009:291] nutzt das INPUT-Statement zur hypertextuellen Referenzierung externer Programme. Thomas E. Kurtz' und John G. Kemenys Sprache BASIC implementiert mit dem READ-Befehl die Möglichkeit für den Programmierer peritextuelle Querver-

58 Vgl. https://de.wikipedia.org/wiki/Hypertext [letzter Abruf: 14.03.2018].

weise zu Datenlisten innerhalb des Programms zu konstruieren [vgl. Höltgen 2018a: 98-101; Dijkstra 1978]. In objektorientierten Programmiersprachen können Variablen als global oder lokal (innerhalb einer Funktion) verwendbar deklariert werden [vgl. Maibaum/Höltgen 2018:228], was spezifische Verweis-Distanzen darstellt.

Genettes Klassifizierung von Text-Text-Beziehung ermöglicht eine qualitativ genaue Kennzeichnung von Verweisen, die Computercodes auf andere Texte (andere Codes, Algorithmen, Randnotizen usw.) vornehmen. Zudem erlaubt sie die nachvollziehbare Einordnung von Programmcodes in Programm(code)genres (Architexte) und von Codefragmenten/-ausschnitten zum Gesamtcode (Paratexte). Letzteres zeigt sich an einigen der oben diskutierten Programmcodes.

Als Paratextualität lässt sich die Programmier-Erleichterungen fremden Code in eigenen Programmen zu verlinken deuten. Auf diese Weise können in höheren Sprachen zum Beispiel Standard-Bibliotheken für I/O-Funktionen, mathematische Bibliotheken und anderes genutzt werden, ohne dass sich der Programmierer mit deren Sourcecodes auseinandersetzen muss. Darüber hinaus können Programmierprojekte so auf verschiedene Dateien (die Code für unterschiedliche Funktionen enthalten) verteilt werden. Auf die erste Möglichkeit greift Mahers „reconstruction“ von Boing! zurück. Molls VCS Boing nutzt die zweite Variante.

In Blitter wird eine dritte Form der Paratextualität verwendet, bei der im ROM-Speicher befindliche Programmteile des Betriebssystems in den eigenen Code integriert werden. Programme, wie das original Boing! oder LED-Tennis machen von diesen Möglichkeiten keinen Gebrauch. Solche Entwicklungen werden unter den Begriff *bare metal programming* geführt und genießen in Hacker-Kreisen besondere Achtung, weil sich die Software direkt auf die Hardware (bare metal[59]) bezieht und keine Routinen der Firmware benutzt werden, sondern jeder Code-Teil selbst entwickelt und verstanden wird. Solche Programme verbrauchen oft den wenigst-möglichen Speicher und besitzen die höchste Performanz. Die originale Boing!-Demo zählt hierzu, denn zu ihrer Entwicklungszeit existierten noch keine Betriebssystemroutinen, auf die die Programmierer Bezug nehmen konnten. Der von Sintonen als „nasty“ (siehe oben) bezeichnete Programmierstil kennzeichnet diese Hacks als Form des bare metal programming.

Das Verweisen auf und das Einbeziehen von vorprogrammiertem, fremdem Code (in C mit der Präprozessor-Direktive #include) aus *Bibliotheken* (libraries) oder ROM-Inhalten innerhalb der eigenen Programme lässt sich mit Genette als Hypertext-Hypotext-Beziehung verstehen. Diese Beziehung erhält archäologische Relevanz, sobald der Hypotext nicht mehr allein synchron, sondern auch *diachron* betrachtet wird [Genette 2001:378f.]: Zitiert werden kann nämlich nur ein *zeitlich vor dem eigenen Text publizierter Text*; auch dies gilt für Programmtexte. Insofern stellen die Einbindung einer Library und der Aufruf einer ROM-Routine auch einen *Aufruf der Software-Geschichte* dar.

59 https://en.wikipedia.org/wiki/Bare_machine [letzter Abruf: 11.01.2019].

Von den oben diskutierten Programmen finden sich in dreien derartige intertextuelle Bezüge unterschiedlicher Graduierung: Der Sourcecode von VCS Boing ruft zwei verschiedene ausgelagerte Programmteile über Pseudo-Instruktionen[60] auf:

```
.include "vcs.inc"
.include "globals.inc"
```

Erstere enthält allgemeine Variablen-Zuweisungen.[61] Deren Variablennamen sind in der Geschichte der VCS-Programmierung konventionalisiert und tradiert und werden daher sogar von Programmier-Manuals verwendet [vgl. Hugg 2016:221-223]. Die zweite Library, gobals.inc, enthält speziell für VCS Boing bestimmte Zuweisungen und definiert den Zusammenhang der unterschiedlichen Dateien, die die Sourcecodes enthalten. Molls Demonstrationsprogramm liegt nicht in *einer* Assembler-Datei vor, sondern ist auf unterschiedliche funktional differenzierte Dateien verteilt, die in einem Hypertext-Hypotext-Verhältnis zueinander stehen. Erst nach dem Assemblieren werden sie zu einer Datei (boingball.bin) zusammengeschrieben.

Weiterhin referenziert der Programmierer die Quellen, denen er seinen Algorithmus als Zitat entlehnt hat, innerhalb der Sourcecode-Kommentare:

```
; calculate a falling parabol, as describe here:
; http://codebase64.org/doku.php?d=base:generating_approximate_sines_in_assembly
```

[Datei boing.s]

Mahers reconstruction der Boing!-Demo für den Amiga verfährt ganz ähnlich, wenngleich die Programme, die über die Präprozessordirektive #include in das C-Programm eingebunden sind, weit über die Funktionalitäten der VCS-Boing-Ergänzungen hinausgehen. Hier werden *Bibliotheken* wie *stdio.h* integriert. Dabei handelt es sich um eine Sammlung von Maschinensprache-Routinen für Input-Output-Operationen, die in eigenen Programmen genutzt werden kann (möchte man Ein- und Ausgaben nicht selbst – bare metal – programmieren). Diese Standard-Bibliotheken sind bereits bei der Entwicklung der Sprache zusammengestellt worden [vgl. Kernigan/Ritchie 1988:241ff.]. Sie werden noch vor dem Compilieren des Programms geladen und in den eigenen C-Code integriert. Maher verwendet davon die folgenden:

60 Pseudo-Instruktionen (auch: Pseudo-Opcodes) stellen keine Opcodes im Sinne von im Prozessor implementierten Funktionen dar, sondern werden vom Assemblierer in solche übersetzt. Auf diese Weise lassen sich komplexere Funktionen höherer Programmiersprachen in Assemblerprogrammen nutzen. Allerdings hängt der Vorrat solcher Pseudo-Opcodes vom jeweiligen Assemblierer ab, so dass die Kompatibilität von Sourcecodes zwischen verschiedenen Assemblierern nicht in jedem Fall gewährleistet ist, wenn solche Funktionen zum Einsatz kommen. Für VCS Boing wurde der Assemblierer CA65 (http://www.cc65.org/doc/ca65.html [letzter Abruf: 20.03.2018]) verwendet, der über eine große Anzahl von Pseudo-Variablen, -Funktionen und -Opcodes verfügt.

61 In Assembler bedeutet dies, dass eine konkrete RAM-Adresse reserviert wird und ein symbolisches Label bekommt; im Programm kann dieses Label dann wie eine Variable verwendet werden, um Werte an der dahinter stehenden Adresse zu speichern, zu verändern oder in anderen Funktionen zu nutzen.

```
#include <exec/types.h>
#include <exec/memory.h>
#include <intuition/intuition.h>
#include <graphics/gfxbase.h>
#include <dos.h>
#include <devices/audio.h>[62]
```

Ein dritter Fall liegt bei Blitter vor. Dieser soll detaillierter diskutiert werden. Das Programm stellt implizite und explizite Intertextualitätsbezüge zu Routinen des Betriebssystems her.

Ersteres geschieht mit jedem Aufruf eines BASIC-Befehls oder einer BASIC-Funktion, die als ausführbarer Maschinensprache-Code im ROM-Speicher des Computers hinterlegt ist. Ein Befehl wie PLOT (Zeile 170) ruft, ohne dass der Programmierer dies in irgend einer Weise berücksichtigen muss, die Routine „GRA PLOT“ [Janneck/Mossakowski 1985:427] auf, die die Anweisung und ihre Argumente interpretiert und ausführt. Kenntnis über diese Routinen kann ein Programmierer durch so genannte ROM-Listings erhalten, in denen die in den BASIC-ROMs gespeicherten Maschinenspracheprogramme disassembliert, kommentiert und auf unterschiedliche Weise aufbereitet werden. Für die meisten Homecomputer gibt es solche ROM-Listings – zumeist in der zeitgenössischen Sekundärliteratur; selten geben diese die Sourcecodes wieder, sondern bereiten eigene Disassemblys für die Publikation auf. Mit ihrer Hilfe kann der Programmierer nicht nur den Ablauf des BASIC-Interpreters verstehen, sondern die Routinen auch für eigene Zwecke verwenden.

Dies geschieht im zweiten Fall, dem expliziten Routinen-Aufruf des Programms mit CALL &BD19 (Zeile 550) und CALL &BC02 (Zeile 430). Mit CALL wird von BASIC aus eine Maschinensprache-Routine im Speicher an der Adresse gestartet, die hinter dem Befehl angegeben ist. Die angesprungene Routine wird ausgeführt; endet sie mit dem Opcode $C9_{16}$ (RET, unbedingter Rücksprung), so wird zurück ins BASIC gekehrt. Die beiden im Blitter-Programm angegebenen CALL-Sprünge implementieren folgende Funktionen in das BASIC-Programm:

CALL &BD19 ruft eine Routine mit folgender Funktion auf: „Synchronisiert den Schreibvorgang auf dem Bildschirm mit dem Strahlrücklauf (engl. Frame flyback). Dadurch wird eine weichere Bewegung von Zeichen oder Graphik auf dem Bildschirm bewirkt, ohne Flackern und Flimmern.“ [Spital u.a. 1985:3/31] Diese Funktion ist in der Locomotive-BASIC-Version 1.1 als der Befehl FRAME implementiert worden (vgl. Kommentar in Programmzeile 550), der ebenfalls einen Sprung an diese Adresse auslöst. An ihr liegt die Routine mit dem Titel „MC WAIT FLYBACK“. Das „Systembuch“ [Woigk 2012] beschreibt ihre Funktion wie folgt:

> Mit jedem Strahlhochlauf des Monitorbildes erzeugt der CRTC ein Signal (VSYNC), das einerseits die Interrupt-Erzeugung des Gate Arrays synchronisiert (und einen Interrupt auslöst) und andererseits über Bit 0 von Port B der PIO eingelesen werden

62 Vgl. „C source codes“ 1-5 auf: http://amiga.filfre.net/?page_id=5 [letzter Abruf: 13.2.2018].

> kann. Die Zeit, in der der Elektronenstrahl des Monitors vom Bildschirmende wieder zu dessen Anfang hochläuft, ist vergleichsweise lang und damit ideal geeignet, umfangreichere Aktionen auf dem Bildschirm durchzuführen, ohne dass es eventuell zu unangenehmen Flimmereffekten kommt. Dieser Vektor lässt die CPU so lange im Leerlauf kreisen, bis das Ende eines Bildes auf dem Monitor erkannt wird. Wenn möglich, sollte man hierfür aber immer ein FRAME FLYBACK EVENT programmieren. [Ebd.:537]

Im CPC-6128-RAM steht ab Adresse $BD19_{16}$:

```
BD19  CF       RST   &08
BD1A  B4       OR   H
BD1B  87       ADD   A
BD1C  CF       RST   &08
BD1D  76       HALT
BD1E  87       ADD   A
BD20  CF       RST   &08
BD20  C0       RET   NZ
[Disassembliert mit dem JavaCPC-Emulator (V. 2.2).]
```

Dieser Code (genau genommen nur die zweite Spalte von links) wird vom BASIC-Programm aus mit dem CALL-Befehl direkt referenziert. Er wurde vier Jahre vor der Entstehung von Blitter! von Chris Hall programmiert, dem innerhalb eines Teams die Entwicklung der Echtzeit-Prozesse des Betriebssystem oblag. [vgl. Smith 2014] Im Speicher des Computers befinden sich außer dem Code keine weiteren Informationen über das Programm – insbesondere nicht über dessen Autor oder die einzelnen Funktionsweisen von Routinen. Diese wurden simultan im Firmware-Handbuch [Godden 1984] publiziert, worin die Routinen kommentiert und strukturiert dargestellt sind. Noch zugänglicher wurden sie durch Publikationen, wie die von Woigk [2012] für Programmierer aufbereitet. Während der Computer die aufgerufene Routine problemlos ‚versteht' (also ausführen kann), stellt sich dem menschlichen Leser ihre Funktion erst nach einer Recherche der zeitgenössischen und aktuellen Metatexte zum System.

CALL &BC02 ruft eine Subroutine des „Graphic Pack (SCR)" [vgl. Janneck/Mossakowski 1985:91-95 sowie 252-271] auf. Hier befindet sich ein Zeiger auf die Routine „Screen Reset", die im ROM des CPC 6128 ab Adresse $0AD0_{16}$ lokalisiert [vgl. Janneck/Mossakowski 1985:170] ist und eine „Kleine Initialisierung des Screen-Pack[s]" [Woigk 1987:498] auslöst. Dabei werden die Standardwerte des Screen Packs, die Anzeigefarben, Cursor-Blinkperioden und der Grafikmodus restauriert [vgl. ebd.] – also die für die Demo veränderten Einstellungen des Bildschirms zurück genommen. Janneck/Mossakowski [1985:251] listen die Routine wie folgt auf:

```
************************************ SCR RESET
                                                        (Tabellen initialisieren)
0AB1    AF                      XOR             A       Null
0AB2    CD 49 0C                CALL            0C49    Force-Mode f. SCR WRITE
0AB5    21 BE 0A                LD              HL,0ABE Adresse der ROM-Tabelle
0AB8    CD 8A 0A                CALL            0A8A    Indirections kopieren
0ABB    C3 D2 0C                JP              0CD2    Farbwerttabellen initialisieren
0ABE    09                                              Anzahl der zu kopierenden Bytes
0ABF    E5 BD                                           Zieladresse im RAM
```

```
0AC1    C3 82 0C            JP          0C82        SCR READ
0AC4    C3 68 0C            JP          0C68        SCR WRITE
0AC7    C3 F7 0A            JP          0AF7        SCR MODE CLEAR
```

Hier zeigt sich, anders als im obigen Beispiel, ein stark bearbeiteter Ausschnitt aus den Betriebssystem-Routinen, das ein Verständnis der Funktion wesentlich erleichtert. Die im Code enthaltenen weiteren CALL- und JP-Aufrufe offenbaren, dass die Routine selbst weitere Betriebssystem-Programmteile aufruft. Der Leser wird zum vollen Verständnis der Routine also dem Programmcode ‚folgen' müssen, so lange, bis er auf den Opcode $C9_{16}$ (RET) stößt, der zurück in das BASIC-Programm führt. (Solche internen Verweisungsstrukturen werden weiter unten als Kohäsionsmarker diskutiert.) Das Verfolgen der Paratexte eines Programmcodes geht also einher mit dessen Verständnis.

Das Blitter-Programm musste vom zeitgenössischen Leser aus der Zeitschrift *Amstrad Action* abgetippt werden. Dieser Vorgang initiierte den (autodidaktischen) Verstehensprozess bereits, bei dem unter anderem die Paratexte eine wichtige Rolle spielten:

1. weil die REM-Kommentare als Metatexte und die Einleitung als Peritext Informationen über das Programm für Leser/Abtipper enthalten,
2. weil sich über die mühevolle und zeitaufwändige Eingabe des Programms auch Verständnisprozesse (etwa über die Grob- und die Feinstruktur des Programms, die Nomenklatur von Variablen, über INPUT und PRINT vermittelten Bildschirmausgaben des Programms etc.) ergeben,
3. weil im Zuge des späteren, nicht unwahrscheinlichen Abtippfehler-Suchens tiefergehende Verstehensprozesse (etwa über die Syntax der verwendeten Programmiersprache) ergeben.

Gerade bei langen und komplexen Abtipp-Programmen haben solche Rezipienten auf das Mittel der *Glossierung* zurückgegriffen und das gedruckte Programmlisting mit handschriftlichen Anmerkungen (vgl. Abb. 4.1.13 & 4.1.14) und Strukturierungselementen[63] versehen. Solche Leser-Kommentare stellen – wie handschriftlich abgefasste Code-Scribbles generell – ein gleichermaßen wertvolles wie schwer archivier- und erfassbares Element der Rezeptionsgeschichte von Computercodes dar. Eine Software Preservation, die zugleich eine Rezeptionsgeschichte der Codes mit bewahren will, muss für derartige Annotationen sensibel bleiben, nicht zuletzt, weil sie vielleicht Verständnisse über die Funktionsweise von Programmen geben können, deren Hardwareplattformen nicht mehr existieren.

63 Im Rahmen eines Seminars über Computerschach hat ein Student ein BASIC-Programm abgetippt und dessen Struktur auf verschiedenen Grafiken visualisiert. (https://www.musikundmedien.hu-berlin.de/de/medienwissenschaft/medientheorien/signallabor/praxisarbeiten/silvio-lippert-2016-schachcomputer-und-computerschach.zip/view [letzter Abruf: 16.03.2018]).

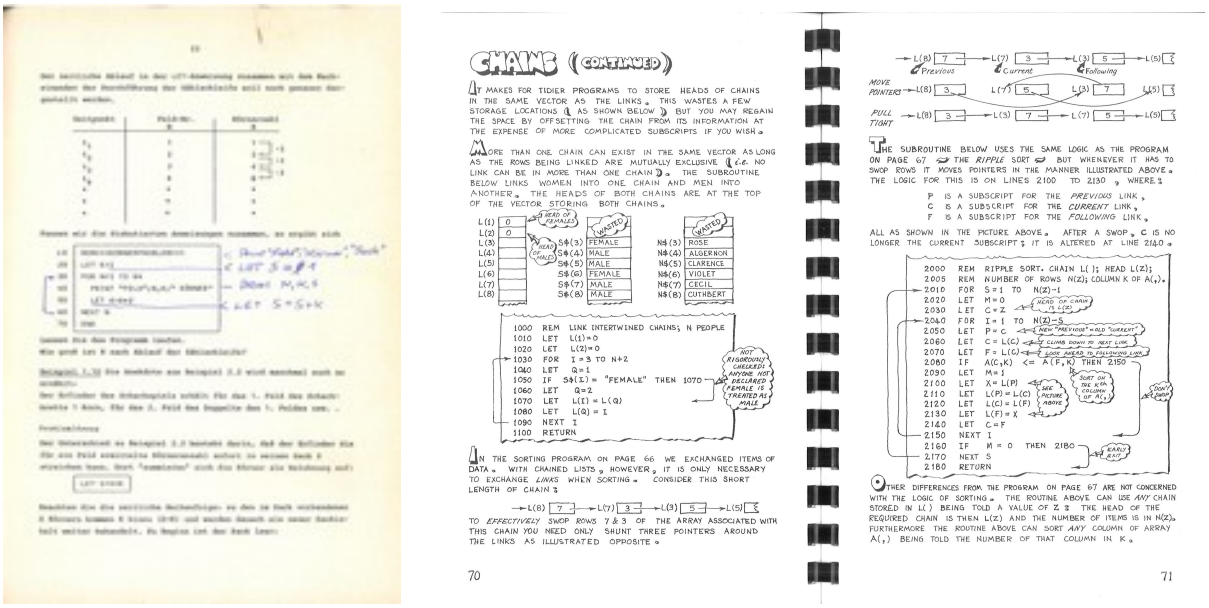


Abb. 4.1.13 & 4.1.14: Ein BASIC-Listing mit handschriftlichen Anmerkungen des Lesers/Abtippers aus [Löthe/Quehl 1982:23]. Diese Form des interlinearen Kommentierens von BASIC-Listings wurde teilweise in Programmierlehrbüchern empfohlen (rechts das Beispiel aus [Alcock 1977:70f.]

In dem Fall, wo vom eigenen Code auf einen Fremdcode referenziert wird (durch Präprozessor-Direktiven, Links, Aufruf von ROM- oder Betriebssystem-Routinen, ...), wird vom Programm zusätzlich eine softwarehistorische Referenzierung vollzogen, denn der jeweils bezogene Code muss bereits (vorher) existieren, um überhaupt referenziert werden zu können. Ob er vom selben Autor am selben Tag oder von einem anderen Programmierer Jahre zuvor entwickelt wurde, ist dabei nebensächlich. Zuletzt können selbst implizite Referenzen, wie die Verwendung eines spezifischen Algorithmus (der, wie in dem Ball-Generierungsroutinen gezeigt, mehrere Tausend Jahre alt sein kann), als diachrone intertextuelle Prozesse verstanden werden. Programmierer arbeiten also stets mit der Geschichte, weshalb eine Untersuchung der diachronen Textverhältnisse ihr Tun zugleich als Archäographie kennzeichnen kann.

4.1.4.3 Kohäsion: Jumps, Loops und Feedbacks

Häufiger, als auf Fremdcode verweist Code allerdings auf sich selbst bzw. seine eigenen Bestandteile. Diese Selbstreferenzen spielen eine entscheidende Rolle bei der menschlichen (Re-)Lektüre von Programmen. Genette untersucht sie als Thema-Rhema-Bezüge [2001:77-80, 86-88] in Texten, allerdings lediglich in Hinblick auf die Beziehung des Textes zu seinem Titel. Analysen innertextlicher Zusammenhänge werden in der Textlinguistik als Kohärenz-Kohäsionsbeziehungen betrachtet. Während *Kohärenz* hierbei die *semantischen* Verbindungen zwischen verschiedenen Textteilen bezeichnet [de Beaugrande/Dressler 1981:5], beschreibt *Kohäsion deren grammatische* Beziehungen und ist damit konstitutiv für die Wahrnehmung von Textualität. De Beaugrande und Dressler nennen Textkohäsion daher als erstes Kriterium für Textualität [3f.]. Rothstein u.a. [2014] definieren sie wie folgt:

> Unter Textkohäsion verstehen wir die explizit sprachlich textuellen Verknüpfungen zwischen den Ausdrücken eines zusammenhängenden Texts. Kohäsionsmarker sind beispielsweise Pronomen, die als Anaphern zu einem passenden Antezedenten fungieren. Je nach syntaktischer Bezugsgröße wird zwischen globaler und lokaler Kohäsion unterschieden. Globale Kohäsion liegt zwischen nicht-adjazenten Sätzen vor, lokale zwischen adjazenten. [38f.]

Kohäsion muss vom Autor aktiv gestaltet werden, will er beim Leser den Eindruck eines zusammenhängenden Textes erzeugen. Sie „liegt vor, wenn grammatisches Wissen verwendet wird, um einen Zusammenhang herzustellen.“ [Kunkel-Razum u.a. 2016:1079] Hierfür werden u. a. folgende *Kohäsionsmarker* gesetzt:

- Interpunktionen: z. B. Komma, Semikolon, Doppelpunkt, Gedankenstrich, Klammer u.a. [ebd.:1079-1083]
- Konnektoren: Junktionen, Relativwörter, Adverbien, Abtönungspartikeln, Präpositionen [ebd.:1083-1088]
- Artikelwörter und Pronomen: Anaphora, Kataphora, Deixis u.a. [ebd.:1120-1125]
- Tempus, Verbmodus und Diathese [ebd.:1126-1135]

Neben der Klassifizierung der *Kohäsionsmarker* lassen sich noch unterschiedliche *Kohäsionsarten* differenzieren, die danach unterschieden werden, wie weit voneinander entfernt (im räumlichen Sinn) die durch Kohäsion verbundenen Textteile sind. Diese Abstände reichen von aufeinander folgenden Sätzen bis hin zu weit voneinander entfernten Textteilen. Rothstein u.a. [2014] stellen bezüglich der syntaktischen Bezugsgröße eine Tabelle auf (vgl. Tab. 4.1.2).

Vor dem Hintergrund der Code-Analyse ist die Betrachtung von Kohäsion nicht nur deshalb interessant, weil mit ihr spezifische Konstruktionen von Routinen (z.B. Schleifen mit `LOOP … UNTIL`) möglich werden und sie teilweise syntaktische Notwendigkeiten darstellen, auf deren Basis Computer Programme erst übersetzen/ausführen können. Auch für das nachträgliche (menschliche) Lesen von Code sind Kohäsionsmarker von entscheidender Bedeutung, ermöglichen sie doch die (Re-)Organisation von Code in Sinneinheiten. Insbesondere reverse engineerter/disassemblierter Code wird über das Erkennen von Kohäsion erst verständlich (vgl. Kap. 4.1.3.2). Die Suche nach ‚Zusammenhängen‘ – im wörtlichen wie übertragenen Sinn – beginnt mit der Suche nach Kohäsionsmarkern. Die Strukturierung von Code durch Einrückung stellt eine auffällige räumliche Markierung von Kohäsion dar. Das gilt für das Verständnis von Code wie auch anderer Texte – Textkohäsionsforschung findet daher insbesondere im Bereich der kontrastiven Linguistik, Didaktikforschung und Lernpsychologie statt [vgl. Schmitz 2016].

Kohäsionsart	Bezugsgröße	Realisierungsmöglichkeiten	Beispiel
Lokale Kohäsion	Satz	Teilsatzverbindungen (z.B. Konjunktionen)	*weil*
	Satzpaar	Satzverbindungen (z.B. Konnektoren)	*daher*
Globale Kohäsion	Absatz	Teilabsatzverbindungen (z.B. satzwertige Querverweis-Ausdrücke)	*zusammenfassend*
	Absatzpaar	Absatzverbindungen (z.B. satzwertige Querverweis-Ausdrücke)	*wie im vorhergehenden Abschnitt bewiesen*
	Kapitel	Teilkapitelverbindungen (z.B. satzwertige Querverweis-Ausdrücke)	*(siehe Absatz 1)* [geschrieben ab Absatz 3 desgleichen Kapitels]
	Kapitelpaar	Kapitelverbindungen (z.B. satzwertige Querverweis-Ausdrücke)	*(vgl. das vorhergehende Kapitel)*
	Text	Textverbindungen (z.B. satzwertige Querverweis-Ausdrücke)	*(siehe Kapitel 1)* [geschrieben ab Kapitel 3]

Abb. 4.1.2: Texttopografische Kohäsion [Rothstein u.a. 2014:38]

Als syntaktische Elemente sind Kohäsionsmarker auch in unterschiedlichen Programmiersprachen realisiert, jedoch auf unterschiedliche Weise. Auch finden nicht alle Kohäsionsverfahren, die natürliche Sprachen erlauben/benötigen, in formalen Sprachen Einsatz. Die folgende (unvollständige) Auflistung ordnet die Kohäsionsmarker, wie sie Kunkel-Razum et al. [2016:1079-1135] vorstellen, den Ballsprung-Demos zu. Die Kohäsionsverfahren beim Ball im Kasten werden aufgrund ihrer nicht-symbolischen Notation zuletzt gesondert beschrieben.

Interpunktionszeichen [vgl. Kunkel-Razum u.a. 2016:1079-1083] trennen einzelnen Programmzeilen oder -anweisungen und ermöglichen ihre separate Wahrnehmung: Im Assemblersprachen werden Werte in Datentabellen durch *Kommata* getrennt. Hier trennt das *Semikolon* Programmtext von Kommentartext. Mit dem *Klammeraffen* (@) und dem *Doppelpunkt* werden Label (als Pseudo-Opcodes) markiert, die als Strukturierungsmittel und Sprungziele dienen. In C dient das *Semikolon* als Marker für das Ende einer Anweisung(szeile). Mit *geschweiften Klammern* werden in dieser Sprache größere Struktureinheiten (wie zum Beispiel Schleifenkörper, Funktionen u.a.) gruppiert. *Runde und eckige Klammern* besitzen in allen hier aspektierten Programmiersprachen gruppierende Bedeutung für die Angaben innerhalb der Klammern (etwa beim Einklammern von Funktionsargumenten, Datenlisten, Arrays u.a.). In Assemblersprachen drücken *Klammern*

häufig auch indirekte Adressierung aus. Locomotive-BASIC verfügt, wie viele andere BASIC-Dialekte über unterschiedliche Interpunktionszeichen: *Doppelpunkte* trennen unterschiedliche Anweisungen innerhalb einer Programmzeile. Das *Hochkomma* (‘) dient als Abkürzung für die REM-Anweisung (mit der ein Kommentar in oder ans Ende einer Programmzeile geschrieben wird). In C werden solche Kommentare über *Schrägstrich-Asterisk* (/*) gekennzeichnet. Überdies dienen *mathematische und logische Operatoren* (+, ++, &&, ...) in vielen Sprachen zur kohäsiven Verknüpfung von Ausdrücken innerhalb mathematischer oder logischer Operationen. Werden solche Operatoren für Vergleiche benutzt (<, >, !=, ==, ...), dann gehören sie zur die Klasse der *komparativen Konnektoren* [vgl. Kunkel-Razum u.a. 2016:1117-1119].

Junktionen [vgl. Kunkel-Razum u.a. 2016:1084f.], wie etwa Konjunktionen, Disjunktionen, Subjunktionen u.a. sind kohäsive Mittel, die bereits in der formalen Logik Aussagen miteinander verknüpfen. Junktionen werden auch in Programmiersprachen verwendet, etwa als *logische Operatoren* AND, ORA, EOR, CMP u.a. (in 6507-Assembler [vgl. Zaks 1986:99-101). Locomotive-BASIC nutzt solche logischen Junktoren zum selben Zweck, ermöglicht jedoch auch eine mehr semantische Verwendung, die auf der Ebene der Verarbeitung dann allerdings wieder formallogisch interpretiert wird [vgl. Spital u.a. 1985:3.5]. Als solche lassen sie sich unter die Klasse der *kopulativen Konnektoren* subsumieren [vgl. Kunkel-Razum u.a. 2016:1092f.].

Temporalkonnektoren [vgl. Kunkel-Razum u.a. 2016:1095-1099] finden Einsatz, um Zeitverhältnisse zwischen verschiedenen Textteilen herzustellen. Diese Funktion nutzen in Programmiersprachen insbesondere bei *Schleifenkonstruktionen*, z.B. in C mit FOR(){...} [vgl. Kernighan/Ritchie 1990:13f.], WHILE(){...} [ebd.:59ff.] oder DO{...}WHILE() [ebd.:62f.] oder in Locomotive-BASIC mit FOR-TO-NEXT [Spital u.a. 1985:3.30, 3.50], WHILE-WEND [ebd.:3.91] sowie den in Assemblersprachen verwendeten *konditionalen Sprungbefehlen* wie den DB*cc*-Befehlen[64] (68k-Assembler zum *Prüfen einer Bedingung, Dekrementieren eines Registers und Verzweigen* mit nur einem Opcodes [Hilf/Nausch 1984:4.134-4.138]).

Konditionale Konnektoren [vgl. Kunkel-Razum u.a. 2016:1099-1103] verknüpfen Bedingungssätze miteinander und finden sich in Programmiersprachen daher vor allem in *Kontrollstrukturen*. IF-THEN-ELSE-Konstruktionen in Locomotive-BASIC [Spital u.a. 1985:3.34] gehören ebenso dazu, wie ON-GOTO/GOSUB [ebd.:3.51-55]. Solche Anweisungen sind in dieser Sprache in einer Programmzeile realisiert. In C können sich IF-ELSE-/ELSE-IF- [vgl. Kernighan/Ritchie 1990:55-58] und SWITCH/CASE-Anweisungen [ebd.:58f.] als Blöcke über mehrere Programmzeilen erstrecken. In Assemblersprachen werden Bedingungen über Statusregister-Bits geprüft. 6507-Assembler testet die unterschiedlichen Bedingungen innerhalb der *Branch-Befehle* [Zaks 1986:102-105]. Außerdem finden sich im VCS Boing Pseudo-Opcodes für if-then-Bedingungsprüfungen. Im 68k-

64 *Decrement and Branch if cc*. *cc* steht für „condition code“, also die Abhängigkeit der Operation von Status- und Condition-Code-Register-Inhalten.

Assembler gibt es neben einem solchen Statusregister ein zusätzliches Condition Code Register [Hilf/Nausch 1984:2.22-25], das für bedingte Verzweigungen mit B*cc*[65] [ebd.:4.69-4.72] abgefragt werden kann.

Restriktive Konnektoren [vgl. Kunkel-Razum u.a. 2016:1115-1117] kennzeichnen Ausschließungen, also negative Bedingungen, mit denen Texte verbunden werden. In *Bedingungsanweisungen* höherer Programmiersprachen lassen sich diese mit dem logischen NOT [vgl. Spital u.a.:1985:30.50f.] oder „<>“ (ungleich) bzw. „!“ (NOT [vgl. Kernighan/Ritchie 1990:41f.,198]) und „!=“ (ungleich [vgl. ebd.:16,41,201]) realisieren. In 6507-Assembler ist die Ausschlussbedingungen wiederum in einen Branch-Befehl integriert: BNE steht für „branch on not equal to zero“ [Zaks 1986:119]), womit der Inhalt des Zero-Flags geprüft wird – etwa, um bei Rückwärtszählschleifen die Abbruchbedingung („bereits 0?“) zu testen. In Verbindung mit dem Compare-Befehlen CMP/CPX/CPY [vgl. ebd.:125-130] wird eine Test-Subtraktion zwischen den übergebenen Argumenten und den Inhalten der Register (A, X oder Y) vorgenommen, bei der alle Ergebnisse ungleich 0 dann „Ungleichheit“ bedeuten. Die CMP-Befehle in 68k-Assembler besitzen eine ähnliche Funktionalität [vgl. Hilf/Nausch 1984:4.109-133], können jedoch auch RAM-Speicherzelleninhalte [CMPM] miteinander vergleichen.

Eine für das Verständnis von Programmtexten besonders bedeutsame Form von Kohäsion wird über die Erzeugung von *Phorik und Deixis* erzeugt. In natürlichsprachlichen Texten werden hierzu Artikelwörter und Pronomen verwendet [vgl. Kunkel-Razum u.a. 2016:1120-1125]. Phorik betrifft Kohärenzbeziehungen zu vorangegangenen (Anaphorik) oder nachfolgenden (Kataphorik) Textteilen. Beide erzeugen *implizite* Thema-Rhema-Strukturen [vgl. Warning 1979:30; Genette 2001:80; Schwarz-Friesel u.a. 2007:81-102] im Text. Deixis erzeugt hingegen *explizite* Textbeziehungen durch das „Zeigen oder Hinweisen auf einen Textgegenstand“ [Kunkel-Razum u.a. 2016:1120]. Jegliche Sprunganweisungen in Programmiersprachen lassen sich unter diese Kohäsionsart fassen, weil sie von einem Teil eines Programmtextes explizit auf einen anderen verweisen und damit den linearen Leseprozess unterbrechen. Unterschieden werden können dabei folgende Arten deiktischer Kohäsion:

- *Direkte Verweise* auf absolute Sprungziele durch GOTO, GOSUB [Spital u.a. 1985:3.32], B*cc* [Zaks 1986:113-122; Hilf/Nausch 1984:4.134-4.138], JMP, JSR [Zaks 1986:141-143; Hilf/Nausch 1984:4.174-179] und Nennung eines Sprungziels, angegeben als Zeilennummer, Label oder Adresse.
- *Indirekte Verweise* durch den Aufruf von Funktionen und Prozeduren in Locomotive-BASIC über DEF FN [Spital u.a. 1985:3.15] und in C [vgl. Kernighan/Ritchie 1990:67-90], wobei in letzterer Funktionen die zentrale Rolle für Verweisstrukturen innerhalb von Programmen spielen.

65 *Branch if* cc. Vgl. Fußnote 64.

- *Relative Verweise* durch Sprünge zu Zielen, die sich im Programmablauf ergeben, etwa auf Basis der Adressarithmetik (Assembler vgl. [Zaks 1986:47,]) oder den zwischengespeicherten Rücksprungzielen am Ende von Unterprogrammen: in Locomotive-BASIC mittels RETURN [Spital u.a. 1985:3.70] und RESUME [ebd.:3.69f.], in 6507-Assembler durch RTI und RTS [Zaks 1986:87-95,163f.] und in 68k-Assembler durch RTE, RTR und RTS [Hilf/Nausch 1984:4.306-311].

Die vom Mikroprozessor zur Verfügung gestellten Adressierungsarten spielen bei Assemblersprachen eine besondere Rolle in Hinblick auf diese Art der Kohäsion im Programmtext, erlauben sie doch vielfältige Möglichkeiten auf Speicherinhalte zuzugreifen (bzw. auf diese im Sinne der Deixis *zu zeigen*). Die Datenmanipulation eines Programms auf dieser Ebene kann nur dann nachvollzogen werden, wenn diese Adressierungsarten korrekt interpretiert werden. C und BASIC verfügen ebenfalls über Funktionen, die den RAM-Speicher direkt adressierbar machen: CALL [Spital u.a. 1985:3.8] erlaubt in Locomotive-BASIC den Aufruf eines Maschinenspracheprogramms an einer spezifischen Speicheradresse von BASIC aus, POKE [vgl. ebd.:3.60] das Schreiben und PEEK() [vgl. ebd.:3.58] das Lesen von RAM-Speicherinhalten. OUT [vgl. ebd.:3.87] ermöglicht das Schreiben auf, INP [vgl. ebd.:3.27] das Lesen von Interfaceadressen, die (indirekt) über Speicheradressen lokalisiert sind. Über einen *Zeiger*, als „Variable, die die Adresse einer Variablen enthält“ [Kernighan/Ritchie 1990:91], lässt sich in C die vielfältige Manipulation von Speicherinhalten vollziehen [vgl. ebd.:91-122].

Schließlich wird durch die Präprozessor-Direktive #include in C und den Pseudo-Opcode .include im 6507-Assembler direkt (deiktisch) auf einen anderen Programmtext verweisen, dergestalt, dass dieser im Kompilier- bzw. Assemblierprozess in den aufrufenden Programmtext integriert wird. Damit greift diese Form der Kohäsion über den eigentlichen Programmtext paratextuell hinaus. Locomotive-BASIC verfügt mit den CHAIN- und CHAIN MERGE-Befehlen [Spital u.a. 1985:3.8f.] über eine ähnliche Funktion, bei denen das BASIC-Programm ‚sich selbst erweitern‘ kann [vgl. Höltgen 2018a].

Eine Sonderrolle nimmt in diesem Zusammenhang die Ball-im-Kasten-Implementierung ein, weil ihr ‚Programm‘ als Diagramm (Prinzipschaltung) vorliegt. Dieses enthält sowohl ikonische als auch symbolische Zeichen. Erstere sind durch Linien miteinander verknüpfte Schaltsymbole (Verstärker, Komparatoren, Potentiometer u.a.), letztere zumeist mathematische Symbole und Zahlen als ‚Beschriftungen‘ der einzelnen Recheneinheiten. Die in solchen Prinzipschaltplänen realisierte Kohäsion ist vor allem indexikalischer Natur:

- Zusammengehörende Elemente, sind über *Striche* (grafisch angedeutete elektrische Leiter) miteinander verbunden. Diese erfüllen für den Leser die Funktion von Junktionen.

- Zeitliche Strukturen (Temporalkonnektoren) realisieren sich vor allem durch die Tatsache, dass Rechenelemente durch die *Rückkopplung* eines Operationsverstärkerausgangs an seinen Eingang realisiert werden. (Je nach Bauteil, das auf dem Rückkopplungsweg eingebraucht wird, wird aus dem Operationsverstärker ein Summierer oder Integrierer). Die Zeit als einzige ‚Variable' im Analogrechner wird über diesen Rückkopplungsprozess auch als kohäsives Element verstehbar, da die Spannungsgrößen *über die Zeit* aufsummiert oder integriert werden. Temporale Zusammenhänge sind darüber hinaus über die *Indexzahlen an Parametern* (y_0, v_{0x} usw.) sowie über Summen- und Integralzeichen (als deren Schrankenangaben von ... bis ...) ablesbar, über welche (zeitlich) sukzessiv ablaufende Operationen gekennzeichnet werden.
- Bedingungsprüfungen finden über *Komparatoren* statt, die Spannungsgrößen miteinander vergleichen und bei bestimmten Größen die Stromflussrichtung (Polung) ändern.
- Eine Restriktion in Hinblick auf die mögliche Flussrichtung des Stroms besitzt die *Zener-Diode* in der Schaltung. Sie begrenzt Spannungen auf einen Höchst- und Tiefstwert, so dass der dargestellte Ball seine Größe behält.
- Durch die *Form einiger Schaltelemente* (Dreiecke für Verstärker und [Zener-]Dioden) und die Angabe der *Spannungspolung* (+ und -) sowie *Pfeilspitzen* am Ende von Linien ist die Lese- und spätere Signallaufrichtung als Deixis [Kunkel-Razum u.a. 2016:1120] aus dem Plan ablesbar.

Etliche der hier vorgestellten Kohäsionstypen und -marker finden sich in den Ballsprung-Programmtexten (vgl. Tab. 4.2.3). Solche Kohäsionsmarker geben dem Programmierer Auskunft über die internen Strukturen des Codes und erleichtern dem Leser das Verständnis des Programmcodes. Bei Sourcecodes liegen durch Metatexte (wie Kommentare des Programmierers), bedeutungsvolle Variablen- und Funktionsnamen sowie die Strukturierung von Codebestandteilen weitere Verständnishilfen vor. C verfügt über eine *struct*-Funktion [Kernighan/Ritchie 1990:123ff.], die strukturiertes Programmieren noch stärker unterstützt. C, BASIC und Assembler bieten überdies die Möglichkeit durch verschiedene Einrückungsebenen visuelle Strukturhinweise zu geben (vgl. BLITTER-Code in Kap. 4.1.3.2), die, wie geschrieben, kohäsive Verbindungen räumlich visualisieren.

Kohäsionsmarker	**BALL IM KASTEN**	**BOING!**	**BOING RECONSTRUCTION**	**BLITTER**	**VCS BOING**
Interpunktion	-	; ( ) , + -	; /* [ ] ( ), + ++ - * / =	: , ( ) ' + - * / =	; ( ) : @ . ,
Junktionen	Elektrische Kontakte	AND CMP	!= < > == &&	OR	AND EOR ORA CMP

Temporal-Konnektoren	Rückkopplungen in OPV	DB*cc*	for	FOR-TO-NEXT WHILE-WEND	Bcc
Konditionale Konnektoren	Komparator	Db*cc* Bcc	If else	IF-THEN-ELSE	If-then (PsOp) Bcc
Restriktive Konnektoren	Zener-Diode	CMP	If !=	keine	BNE und CMP/CPX/CPY
Deixis und Phorik	Pfeile, Stromlaufrichtung, Dreiecksymbole	JMP JSR B*cc* RTS	{} #include	GOTO GOSUB CALL OUT	B*cc* JMP JSR RTS RET .include .segment

Abb. 4.2.3: Kohäsionsmarker in den Ballsprung-Demonstrationen

Derartige Unterstützung des Programmverständnisses fehlt bei disassembliertem oder dekompiliertem Code jedoch ebenso wie die Kommentare und ‚sprechenden' Variablennamen. Sie müssen vom menschlichen Interpreten mühevoll rekonstruiert werden, während ein Computer mit der Interpretation unstrukturierten und unkommentierten Codes, der aus (für Menschen) schwer verständlichen Opcode-Daten-Zahlenkolonnen besteht, keinerlei Schwierigkeiten hat. Im Folgenden soll der Blick auf den *Computer als Code-Leser* gerichtet werden, um diese Prozesse den menschlichen Lektüren gegenüberzustellen und die Bedeutung der Ausführung von Code für dessen vollständiges Verständnis einmal mehr zu unterstreichen.

4.1.5 Computerphilologie II: Maschinen lesen Code

Die vom Computerphilologen untersuchten Texte unterscheiden sich in einem Punkt fundamental von Texten, mit denen sich Philologen ansonsten beschäftigen: Sie wurden ursprünglich nicht für den Nachvollzug durch menschliche Leser geschrieben, sondern für den Vollzug durch Maschinen. Ihr ‚Sinn' nach der erfolgten „maschinelle[n] Interpretation (eben durch ‚Compiler' und ‚Interpreter')" [Ernst 2016] muss vom menschlichen Leser mitvollzogen werden, wenn sich das Verständnis der Beziehungen zwischen Unterfläche und Oberfläche entfalten soll.

Zur Laufzeit entzieht sich diese Lektüre allerdings dem menschlichen Leser, welcher der Ausführungsgeschwindigkeit des Computers nicht mehr lesend folgen kann. Diese Diskrepanz hat sich in den oben dargestellten Programmen und ihren Analysen teilweise bereits gezeigt: Die Momentaufnahme der Ball im Kasten-Demonstration zeigte die spiralförmige Ablenkung des Oszilloskop-Kathodenstrahls (vgl. Abb. 4.1.2), die dem Betrachter-Auge während der laufenden Simulation verborgen bleibt, und der sichtbare schichtwei-

se Aufbau des Balls in der Blitter-Demo zeichnete ein ‚Stilleben' der späteren Farbrotation (vgl. Abb. 4.1.9), bei der die unterschiedlichen Farben als Bewegung perzipiert werden. Solche Momentaufnahmen besitzen daher analytische Qualitäten, die sich allerdings ebenfalls erst zur Laufzeit der Demos voll entfalten: Die Computer demonstrierten erst dann ihr simulatives Potenzial, wenn sie auf Basis ihrer Ausführungsgeschwindigkeiten gestaltförmige Oberflächeneffekte (hier die Anmutungen springender Bälle) erzeugen.

Computerphilologie muss daher, mit Ernst gesprochen, eine erweiterte Perspektive auf Programme einnehmen, denn „Medienphilologie und Medienarchäologie sind Zwillingsmethoden." [Ernst 2016] Erstere richtet sich ebenso wie zweitere nicht bloß auf Dokumente, sondern auch auf *Prozesse*, die sie *in ihrer Operativität also ihrer Eigenzeitlichkeit* zu beschreiben versucht:

> [...] Software-Philologie soll im Unterschied zur klassischen Textphilologie beide Existenzweisen ihres Objekts – als statischer Quellcode und als zeitkritischer Prozess – berücksichtigen, in der Dualität von Schaltplan und Medienvollzug. Daraus resultiert eine Unschärferelation von Objekt und Prozess. Den Textbegriff überschreitet Software, insofern ihre Textualität die operative Implementation in einer Maschine voraussetzt, als operative Diagrammatik, die – anders als ein Buch – selbst liest und schreibt. [Ernst 2016]

Dies erfordert also neben der Lektüre den gleichzeitigen Einsatz technischer Methoden, um Objekt und Prozess in Bezug aufeinander analysieren zu können. Auf diese Weise wird die Vergegenwärtigung eines historischen Artefakts (Code, Schaltung) im Zuge seiner Operativierung beschreibbar. Zugleich kann aber auch seine spezifische Historizität durch eine neu verstandene Art von ‚Quellenstudien' untersucht werden:

> Wie die akademische Archäologie und die Editionswissenschaft sind auch Medienarchäologie und -philologie mit lückenhafter und fragmentierter Überlieferung in den Sektoren von Datenträgern konfrontiert. Was dem Leser digitaler Dateien als kohärenter Text auf dem Bildschirm erscheint, liegt tatsächlich verstreut vor. Von daher resultiert das Primat der archivischen Verwahrung von Software auf Disketten und Festplatten, gleich archäologischen Scherben von den Sektorimages als Quelle auszugehen. [...] Quellcode stellt im Archiv der Gegenwart eine neue Quellengattung dar. [Ernst 2016]

Solche technischen Lektürehilfen können als Hardware- (Oszilloskope, Logikanalysatoren) oder Software-Tools (Debugger) Einsatz finden. Zudem lassen sich die Programme auch über die Manipulation ihrer Codes und die daraus resultierende Abweichung der Ausgaben zur Laufzeit untersuchen. Zu dieser Form der technischen Lektüre laden insbesondere die hier diskutierten Demoprogramme ein (und fordern deren Nutzer sogar explizit dazu auf). An den einzelnen Demoprogrammen sollen solche Lektüren vorgestellt werden.

4.1.5.1 Feedback-Historiografie: Ball im Kasten

Bereits die Programmierung von Analogcomputer beruht auf dem Prinzip der Referenzialität, denn aus einem Operationsverstärker – so er nicht bloß als Invertierer genutzt werden soll – wird erst durch Rückkopplung (Feedback) des Ausgangssignals auf einen seiner Eingänge ein Rechenelement. Je nachdem, ob über den Rückkopplungsweg weitere elektronische Elemente vom Signal passiert werden, kann der Operationsverstärker als Summierer (Widerstand im Rückkopplungsweg), Integrierer (Kondensator im Rückkopplungsweg) usw. genutzt werden. Da die Zeit (t) dabei die einzige Funktionsvariable ist, nach der die in den Operationsverstärker eingespeiste Spannungsänderung stattfindet, lässt sich vor dem Hintergrund der hier vorgestellten Computerphilologie der Analogcomputer als selbstreferenzielle ‚Zeitmaschine' sehen, der deshalb „Zeitobjekte" generiert, „die nicht nur Einheiten in der Zeit sind, sondern die Zeitextension auch in sich enthalten."[66] [Husserl 1928:18] Diese Zeitmaschine operiert allerdings auf der mikrozeitlichen Ebene der Signallaufprozesse. Eine prinzipielle Lektüre dieser Prozesse war über den Schaltplan möglich; operativ lassen sie sich jedoch nur messen – wobei der Output des Programms zugleich das Ergebnis dieser Messung ist, denn eine Kodierung der Signale findet hier nicht statt.

Das Oszilloskop wurde für die Demonstration Ball im Kasten von einem Messmedium zu einem Ausgabemedium umfunktioniert. Anders als beim Rasterbildschirm wird hier kein Bild für die Ausgabe generiert; das, was der Bildschirm zeigt, sind die Spannungsverläufe, die auf die x- und y-Ablenkung des Kathodenstrahls übertragen werden und auf den Betrachter gestaltförmig wirken [vgl. Wertheimer 1923]. Dadurch, dass sich dieser Prozess in Echtzeit vollzieht, hat eine Manipulation des auf dem Patchfeld gesteckten Programms eine sofortige Änderung der Oszilloskop-Ausgabe zur Folge. Eindrucksvoll lässt sich dies an drei Beispiel-Eingriffen demonstrieren:

- Die Entfernung der Zener-Diode aus der Schaltung hat zur Folge, dass die Kreisgrafik auf dem Oszilloskop ihre Stabilität (ihren festen Durchmesser) verliert. Durch das Aufschaukeln der Spannungsabweichungen des realen physikalischen Prozesses (den Spannungsflüssen im Analogcomputer) wächst der Ball stetig. Die Entfernung der Zenerdiode führt also die Physikalität des Computings mit all ihren Signalschwankungen deutlicher vor Augen, als dies der Prinzipschaltplan könnte [vgl. AEG/Telefunken o.J.a:3].

- Die Veränderung der Koeffizienten-Potentiometer-Einstellungen hat Einfluss auf ganz unterschiedliche Qualitäten und Quantitäten der Ausgaben. Zum einen können diese Änderungen zur Unter- oder Übersteuerung einzelner Operationsverstärker führen (wobei der ursprünglich gesteuerte Rückkopplungsprozess zusam-

66 Weiter heißt es dort zum Vergleich: „Wenn ein Ton erklingt, so kann meine objektivierende Auffassung sich den Ton, welcher da dauert und verklingt, zum Gegenstand machen, und doch nicht die Dauer des Tones oder den Ton in seiner Dauer. Dieser als solcher ist ein Zeitobjekt." [Husserl 1928:18]

menbricht und die Rückkopplung gegen Null abebbt oder gegen unendlich übersteuert). Zum anderen können dadurch die physikalischen Eigenschaften der Ballsprung-Demonstration in Echtzeit geändert werden. Über das Potentiometer 1 lässt sich beispielsweise die Gravitationskonstante verändern, Potentiometer 12 und 13 regulieren den Radius des Balls usw. [vgl. AEG/Telefunken o.J.a:8]

- Die Umprogrammierung der passiven Demonstration Ball im Kasten zum interaktiven Spiel Tennis für Drei innerhalb des o.g. Universitätsseminarkontextes: Hier wurden vor allem die Komparator-Elemente so manipuliert, dass die Vorzeichenänderung nicht mehr durch interne Signale, sondern durch ein extern eingespeistes Triggersignal herbeigeführt wird. Diese Änderung fand sukzessive statt, weil zunächst Wege gefunden werden mussten, um die Komparatoren und die ihnen zugehörigen Bauteile (Inverter und Potentiometer) in ihren Wirkungsweisen auf die Signaländerung studieren zu können (Abb. 4.1.15). Das Re-Enactment des Spiels Tennis for Two als Experiment in actu/nascendi lieferte hier also Erkenntnisse über die detaillierten Funktionsweisen der Demonstration Ball im Kasten, die so aus der Programmbeschreibung [AEG/Telefunken o.J.a] nicht hervorgingen.

- Die Signalprozesse, die beim Programmablauf innerhalb der Rechenschaltung stattfinden, vergegenwärtigen nicht nur die Historizität der Analogrechner-Demo aus den 1970er-Jahren, sie produzieren darüber hinaus eine eigene, „mikrohistoriographische Operation“ [Ernst 2013:193] – die ihrer eigenen Signalverläufe: Die Rückkopplungen in den Operationsverstärkern der Analogcomputer verursachen eine zeitbasierte Änderung der Signalquantitäten; das in den Operationsverstärker einfließende Signal zum Zeitpunkt t_n unterscheidet sich vom Signal t_{n+1}, das über die Rückkopplungsstrecke wieder in den Operationsverstärker eingespeist wird usw. Diese zeitbasierte Signaländerung ließe sich als ein Rezeptionsprozess des technischen Mediums bei sich selbst, oder als eine Form von ‚Schaltungskohäsion‘ über Temporalkonnektoren beschreiben.

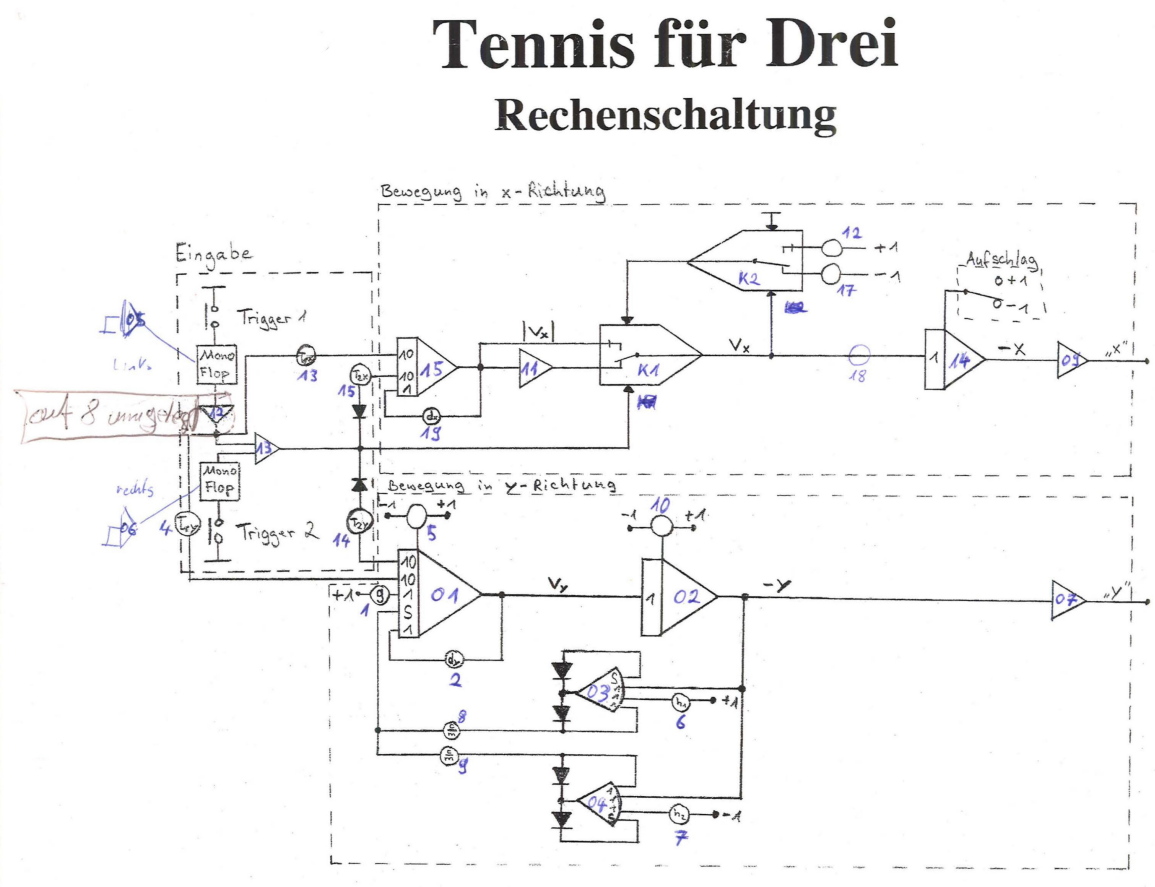


Abb. 4.1.15: Modifizierter Prinzipschaltplan für Tennis für Drei [Höltgen/Maibaum/Rech 2012]

4.1.5.2 Zeitkomplexität: LED Pong

Auch im Re-Enactment der IBM-704-Ballsprungdemo zeigen sich die mikrozeitlichen ‚Lektüreprozesse' des Systems selbst. Hier stellte die Anpassung der Systemzeit an die Reaktionszeit des menschlichen Spielers eine besondere Herausforderung dar. Im Unterprogramm in den Programmzeilen/-adressen 0300_{16}-0308_{16} wurde eine Warteschleife programmiert, die die Aufleuchtdauer einer Leuchtdiode bestimmte, bevor das Ball-Bit zur nächsten Leuchtdiode weiter-rotiert wurde:

```
; Warteschleife (0,5 Sekunden):
;
0300            06 71           LODI,R2 71      ; Dauer äußere Schleife
0302            07 C2           LODI,R3 C2              ; Dauer innere Schleife
0304            FA 7E           BDRR,R2 7E              ; innere Schleife
0306            FB 7C           BDRR,R3 7C              ; äußere Schleife
0308            17              RETC,UN                 ; RETURN
```

Die Quelle für die Warteschleifen entstammte der zeitgenössischen Sekundärliteratur zum System Instructor 50. Dort wurden die Werte für 0,5 Sekunden Wartezeit wie folgt begründet:

> Die beiden LODI-Instruktionen belegen je zwei und jede BDRR-Instruktion drei Taktzyklen zu je drei Taktperioden. Die LODI-Instruktionen werden nur einmal exe-

kutiert. Das Register R_2 wird einmal mit n_2 und (n_3-1)-mal mit 256, das Register R_3 hingegen nur einmal mit n_3 dekrementiert; hieraus ergibt sich für die Verzögerungszeit t, wenn T die Periodendauer der Taktimpulse und n_2 und n_3 die zugeordneten Register-Inhalte (dezimal) sind:

$$t=3\cdot\{4 + 3\cdot[n_2 + n_3 + 256-(n_3 - 1)]\}\cdot T$$

Für die verlangte Verzögerungszeit von 0,5 Sekunden für eine Halbperiode erhält man

$n_2 = 113_{dez.} = 71_{hex.}$ und $n_3 = 194_{dez.} = C2_{hex.}$

da die Taktfrequenz des INSTRUCTOR 50 eben 894,887 kHz beträgt. [Fischer 1982:121f.]

Im zum Programm veranstalteten Brückenkurs wurde diese Quelle zunächst nicht genannt, sondern die Warteschleife zuerst experimentell, dann mathematisch erprobt. Dabei hat sich den Kursteilnehmern insbesondere die verglichen zur menschlichen Wahrnehmungsfähigkeit immense Geschwindigkeit des Computerprozesses durch das durch-rotierende Leuchtdioden-Bit im Wortsinne vor Augen geführt. Dass ein Prozessortakt der 2650-CPU innerhalb von nur etwas weniger als einer Mikrosekunde vollzogen wird, zeigte sich hier ebenso eindrucksvoll wie der Vergleich mit heutigen Systemen, die mit bis zu 4 Gigahertz getaktet sind und damit pro Nanosekunden 4 Takte abarbeiten.

Das Programm sollte als Re-Enactment einer Ballsprung-Demonstration auf dem Großrechner IBM 704 verstanden werden. Über dieses System, das zwischen 1954 und 1960 gebaut und verkauft wurde, sind einige technische Daten bekannt – unter anderem über seine Geschwindigkeit:

> Speed: The Type 704 multiplies or divides in 240 microseconds, or approximately 4,000 operations per second. The doubling of speed in arithmetic, plus the use of index commands, makes the Type 704 a very versatile and capable data processing machine.[67]

Hierbei ist zu berücksichtigen, dass IBM-704-Operationen 36 Bit große Wörter darstellten, was zusammen mit anderen technischen Faktoren den Vergleich mit einer 8-Bit-CPU erschwert. Dennoch lässt sich festhalten, dass 4000 ausgeführten Instruktionen pro Sekunde auf dem IBM 704 zwischen 223721 und 447443 Operationen pro Sekunde auf dem Instructor 50 gegenüberstehen.[68] Letzterer arbeitet Prozessorinstruktionen also zwischen 67 und 111 mal schneller ab.

67 https://www-03.ibm.com/ibm/history/exhibits/mainframe/mainframe_PP704.html [letzter Abruf: 21.03.2018].

68 Die 1-3 Bytes großen Opcodes des 2650 verbrauchen 2-4 Takte für ihre Ausführung [vgl. Fischer 1982:174f.].

Ein weiterer experimenteller Erkenntniswert stellte sich bei der Frage nach der Bemessung der Schleifenzähler innerhalb der beiden (verschachtelten) Schleifen ein und auf welch dramatische Weise sich die Wartezeiten ändern, wenn der innere oder der äußere Schleifenzähler manipuliert oder gar eine dritte Schleife verschachtelt wird. Hier konfligierte die ‚Langsamkeit' der historischen Hardware mit den Hochgeschwindigkeiten mikrozeitlicher Operationen und offenbarten die Widersinnigkeiten solche Prozesse innerhalb historischer Zeitmaßstäbe diskursiv zu diskutieren. Sie können bestenfalls durch Demonstration am laufenden System sinnlich erfahren werden.

4.1.5.3 Waiting for Boing! to see

Über die Entstehung der Boing!-Demo für den Commodore Amiga existiert eine Vielzahl historiografischer und akademischer Beiträge und die Popularität des Programms hat es zu einer regelrechten Ikone der Computergeschichte gemacht. Das hat dazu geführt, dass das ausführbare Programm heute noch aus zahlreichen Quellen zu beziehen ist. Die Funktionsweise des Programms hatte sich Maher über ein Disassembly verfügbar gemacht. Dieses hat den ‚unsauberen' Programmierstil der Entwickler offenbart, der oben aus dem Zusammenhang mit der noch im Prototypen-Stadium befindlichen Hardware begründet wurde.

Dennoch läuft die Boing!-Demo ja auch noch auf heutigen Amiga-Computern (und deren Emulatoren) fehlerfrei ab – trotz der unsauber („nasty") programmierten und sogar fehlerhaften („BUG") Codeteile. Der Computer ‚liest' den Programmtext offenbar anders als Sintonen. Während der Mensch Computerprogramme mit seinem Kontextwissen (Geschichtskenntnis, Lehrbuchwissen, ...) abgleicht, sind die Gradmesser für die ‚gelingende Lektüre' eines Amigas sein Mikroprozessor und dessen Peripherie(bausteine). Sie müssen den Code ‚verstehen', um ihn ausführen zu können. Dies kann nur bei korrektem Code geschehen, dessen „Algorithmen die gewünschte Spezifikation erfüllen, sodass [sie] auf alle Eingabedaten mit den gewünschten Ausgabedaten reagieren." [Schöler 2018:42]. Wenngleich dies im vorliegenden Fall nicht geschehen ist, ließe sich die Korrektheit der Boing!-Algorithmen auch mathematisch beweisen [vgl. ebd.: FN 1]. Der fehlerfreie Ablauf der Demo stellt damit ein Indiz dafür dar, dass der Boing!-Code korrekt ist – aus der Sicht des Computers.

Hier zeigen sich also zwei verschiedene Sichtweisen auf Code und dessen Verständnis. Sintonen hat nur wenige Kommentare in sein Disassembly eingefügt, die den Schluss zulassen, dass er sich gut mit der bezogenen Hardware auskennt und selbst bereits für diese Plattform programmiert hat.[69] Von der Warte des Entwicklers, der vielleicht manchmal gezwungen ist, fremden Code zu verstehen und zu bearbeiten, ist Sintonens normative Bewertung der Programmierweise nachvollziehbar. Die „nasty" Programmierung der Boing!-Demo lässt jedoch auch Rückschlüsse über die bezogene Hardware – den Lorraine-

69 https://sintonen.fi/ [letzter Abruf: 21.03.2018].

Prototypen auf der Winter CES 1985 – zu. Und die Tatsache, dass Boing! Auch auf späteren, seriengefertigten Amiga-Computern fehlerfrei läuft, zeigt, wie ‚serienreif' das Konzept des Computers bereits bei diesem Prototypen gewesen ist.

Indes offenbart sich abermals anhand einer Verzögerung, dass dies nicht in allen Aspekten so gesehen werden kann. Maher notiert im Sourcecode seiner C-„reconstruction" von Boing!:

```
/*The original Boing demo drew the ball onto the screen programatically [sic], using a series
of complex floating point trigonometry functions. This is the main reason for the considerable
delay that follows the execution of that program. [...]
```

Dieser „delay" lässt sich sinnlich (optisch) wahrnehmen und mit dem verzögerungsfreien Start der Boing! reconstruction vergleichen. Abermals ließe er sich aus der Vorberechnung der trigonometrischen Daten für die Ballgenerierung im Disassembly mikrosekundengenau berechnen.

4.1.5.4. De-structured Boing! reconstruction

Das Programm Mahers nutzt mit der Programmiersprache C das elaborierteste Programmierparadigma der hier vorgestellten Demoprogramme: Zwar ist die erstmals 1972 publizierte Sprache C noch imperativ wie BASIC und die Assemblersprachen, doch bereits prozedural und strukturiert programmierbar. Gerade in letzterem zeigen sich die Vorteile von C, ermöglichen Sie doch einen auch für Menschen gut verständlichen Code zu schreiben.

Das Konzept des strukturierten Programmierens ist zur selben Zeit wie die Sprache C entstanden [vgl. Dijkstra 1972] und fordert einen Programmaufbau, der baumartig gegliedert ist. Insbesondere auf GOTO-Anweisungen muss bei dieser Art der Programmierung verzichtet werden [vgl. Dijkstra 1968]; anstelle dessen sollen entfernte Programmteile als Prozeduren aufgerufen werden. Damit steht die strukturierte Programmierung der prozeduralen bereits nahe: „Solche Sprachen bestehen aus einer Reihe von Prozeduren (bzw. Unterprogrammen oder Funktionen), die beim Aufruf ausgeführt werden. Jede Prozedur besteht aus einer Folge von Anweisungen, wobei jede Anweisung Daten manipuliert, die lokal sind und durch einen Parameter angegeben oder global definiert werden." [Pratt/Zelkowitz 1998:576]

Obschon C zwar die Möglichkeit besitzt über *Zeiger* maschinennahe Operationen zu implementieren und dies sogar zu Kritik und zu Vergleichen mit der GOTO-Programmierung geführt hat [vgl. Kernighan/Ritchie 1990:91; auch existiert in C ein GOTO-Sprungbefehl, vgl. ebd.: 91f.,221] und ebenso wie Assemblersprachen imperativ zu sein, sind das strukturierte und prozedurale Programmierparadigma vordergründig. Dies zeigt sich schon bei einem optischen Vergleich der Programmtexte. Einrückungen verdeutlichen Programmstrukturen auf den ersten Blick, können jedoch trügerisch sein, wie

Kernighan/Ritchie an einem Beispiel zeigen: „Einrücken zeigt zwar unmißverständlich, was wir möchten, läßt jedoch den Übersetzer kalt“ (Kernighan/Ritchie 1990:56].

Dieser Übersetzer (Compiler) erzeugt zuletzt aus jedem C-Programm ein ausführbares Maschinenspracheprogramm, bei dem keine der strukturierenden Möglichkeiten der Hochsprache mehr erkennbar sind. Auf der Maschinenebene residiert der Spaghetti-Code, der eigentlich für eine GOTO-orientierte Programmierung in Sprachen wie BASIC definiert wurde:

> So entstand der Begriff des Spaghetticodes für verächtlich belächelten Programmierstil. Er war weniger, wie man vielleicht glauben könnte, das Ergebnis fehlender Programmierkenntnisse, sondern der Wunsch, Speicherplatz zu sparen – auch Formatierung kostet Bytes. [Wieland 2011:169]

Auch wenn, wie Maher konstatiert, die 68000-CPU für Sprachen wie C designt wurde, sind in ihr keine Möglichkeiten strukturierter Programmierung implementiert. Sprünge an Adressen (absolut oder relativ zum Program Counter), bestenfalls definiert als Sprünge in Subroutinen mit Return-Option, sind hier die einzigen Verzweigungsmöglichkeiten. Auf dieser Ebene programmiert, lässt sich Speicherplatz sparen und hoch-performanter Code entwickeln.

An dieser Stelle zeigt sich, dass selbst die Betrachtung ein und desselben Programms als Sourcecode und als ausführbare Datei (und tatsächlich in Ausführung befindliches Programm) deutlich unterschieden werden muss: Der C-Compiler ‚liest‘ und ‚übersetzt‘ einen Sourcecode auf andere Weise und mit anderem Ergebnis als es ein Mensch dies täte [vgl. Sander 2019:46-54]. Die vergleichende Betrachtung dieser ‚Lektüren‘ würde markante Unterschiede in der Implementierung spezifischer Algorithmen und Routinen zwischen den Assembler-Programmversionen zeigen.

4.1.5.5 Blitter-Interpretation

Noch deutlicher als bei einem kompilierten Sourcecode lässt sich der maschinelle Lektüre-Prozess bei einer interpretierten Programmiersprache wie BASIC zeigen. Das Blitter-Programm in Locomotive-BASIC wird in mehreren Schritten schon während der Eingaben für den späteren Ablauf aufbereitet und mit im Speicher befindlichen Maschinensprache-Routinen verknüpft:

1. Eine eingegebene BASIC-Zeile wird zunächst als String gebuffert; Ist eine Zeilennummer vorhanden, wird diese als 16 Bit große Binärzahl gespeichert. Zudem wird die Länge der Zeile mitgespeichert.

Ist keine Zeilennummer vorhanden, wird der Befehl direkt ausgeführt. Hier zeigt sich bereits der Unterschied zwischen der BASIC-Eingabeshell und einem Texteditor.[70] Der Interpreter unterscheidet Eingaben im so genannten „Direktmodus“ [Spital u.a. 1985:5.18] von solchen innerhalb von Programmen. Letztere sind durch das Vorhandensein einer Zeilennummer am Zeilenanfang gekennzeichnet und werden erst ausgeführt, nachdem das Programm mit RUN gestartet wurde.

2. Dann wird die BASIC-Zeile geparst und Schlüsselwörter werden durch ein Token ersetzt.

3. Eingaben, die nicht als Befehle oder Funktionen erkannt wurden, werden als Variablennamen gespeichert.

4. Sprunganweisungen (GOTO/GOSUB) führen dann zu einem Durchsuchen des übrigen Programmtextes, um die Sprungziele zu lokalisieren. Beim Locomotive-BASIC wird diese Suche (im Gegensatz zu vielen anderen BASIC-Dialekten) nur einmal ausgeführt, da das Sprungziel beim Übersetzen durch die konkrete RAM-Adresse der BASIC-Zielzeile ersetzt wird.

5. Die Berechnung der Ausdrücke nimmt dann einen wichtigen Teil des Interpretationsprozesses ein, weil BASIC nur für wenige Variablentypen eine notwendige Deklaration benötigt (insb. Zahlen werden ausschließlich als Fließkommazahlen behandelt).

6. Schleifenkonstruktionen benötigen einen BASIC-Stack, auf dem die Laufvariablenwerte zwischengespeichert werden. Aufgrund des Last-in-first-out-Prinzip dieses Speichers ist eine Schachtelung von Schleifen möglich.

Brückmann u.a. fassen diesen Prozess wie folgt zusammen:

> Die Ausführung eines Statements durch den BASIC-Interpreter läßt sich vereinfacht folgendermaßen darstellen. Jede Programmzeile beginnt wie beschrieben mit der Programmlänge und der Zeilennummer. Danach kommt der eigentliche BASIC-Befehl. Der Interpreter prüft nun, ob es sich um ein Befehlstoken handelt, das durch einen Wert zwischen &80 und &E1 gekennzeichnet ist. Ist dies der Fall, so benutzt er dieses Token als Zeiger in eine Tabelle, die die Adressen sämtlicher BASIC-Befehle enthält. Der BASIC-Befehl wird als Unterprogramm ausgeführt. Danach wird wieder in die sogenannte Interpreterschleife zurückgekehrt. Begann die Anweisung jedoch nicht mit einem Befehlstoken, so wird zum LET-Befehl verzweigt. [vgl. Brückmann u.a. 1985:299]

70 Es ist anzumerken, dass der CPC mit dem „COPY-Cursor“ über eine Copy-and-Paste-Funktion verfügt, wodurch genuine Texteditor-Funktionen in die Shell implementiert wurde. [Vgl. Spital 1985:1.30f.].

Dieser technische Hintergrund[71] bleibt dem BASIC-Programmierer in der Regel verborgen. Zwar ermöglichen BASIC-Befehle wie TRON und TROFF [Spital u.a. 1985:3.88] ein Monitoring des BASIC-Programms (zum Beispiel zum Debugging), die Funktionsweisen des Interpreters bleiben jedoch unsichtbar und werden auch vom BASIC-Handbuch nicht näher erläutert [vgl. ebd.:9.2, Anhang 2.16]. An dieser Stelle klärt abermals die spezialisierte Sekundärliteratur auf [vgl. Brückmann u.a. 1985:295-304]. Auch Emulatoren können diesbezüglich einen didaktischen Surplus erzeugen, wenn sie es zulassen, die internen Abläufe des emulierten Systems in Echtzeit – etwa mit Hilfe eines Disassemblers oder Debuggers – darzustellen (vgl. Kap. 4.2).

Hier zeigt sich (vgl. Abb. 4.1.16), dass die maschinelle Interpretation des Blitter-Programms ein komplexer Prozess ist, bei welchem die hochsprachlichen BASIC-Befehle systematisch in einzelne Maschinensprache-Routinen übersetzt (bzw. mit diesen verlinkt) und direkt ausgeführt werden. Das kommentierte BASIC-ROM des Amstrad CPC erlaubt den Einblick in einige Maschinensprache-Routinen, die hinter den jeweiligen BASIC-Befehlen stehen. Wenn beispielsweise der PRINT-Befehl 50 weitere Maschinensprache-Subroutinen[72] aufruft [Janneck/Mossakowski 1985:554-562], offenbart sich eine noch vielfältigere ‚Vernetzung' der Anweisungen. Ließ sich der hochsprachliche BASIC-Spaghetticode nur durch didaktische Strenge (vgl. Abb. 4.1.17) für menschliche Leser verständlich programmieren, so entzieht er sich auf der maschinellen Ebene durch derartige CALL-Anweisungen und andere bedingte und unbedingte Sprünge (mit JP, JR, DJNZ u.a.) jedem intuitiven Verstehensprozess. Von der Hardware wird er hingegen fehlerfrei und vergleichsweise schnell ausgeführt.

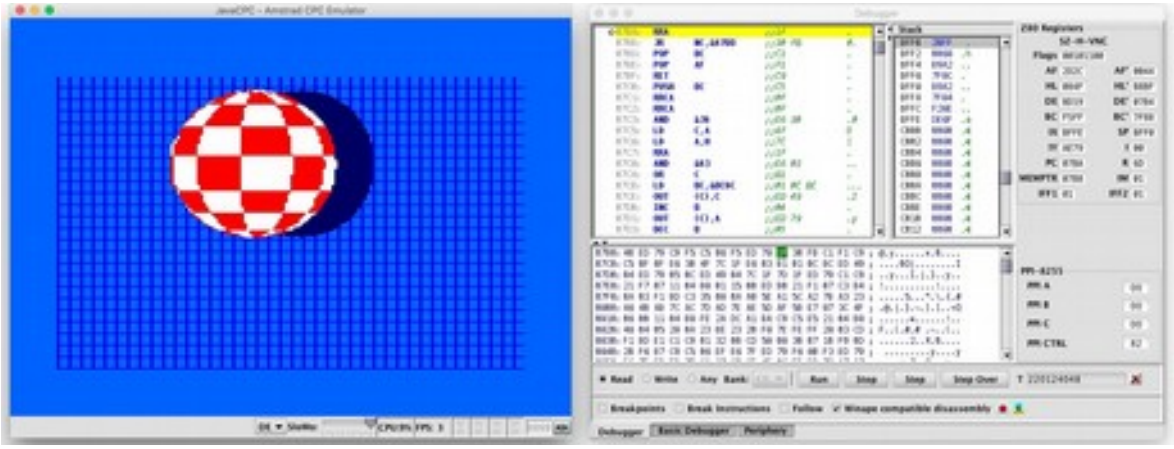


Abb. 4.1.16: Das laufende Blitter-Programm und der parallel dazu laufende Debugger im Emulator JavaCPC zeigen die Übersetzung des BASIC-Programms in Maschinensprache zur Laufzeit. Zum besseren Nachvollzug kann die CPU-Geschwindigkeit gedrosselt werden (hier auf 9 % der regulären Leistung). Das Debuggerfenster bietet zahlreiche Informationen über die internen Lektüreprozesse des Computers (Registerinhalte, abgelaufene Takte usw.).

71 http://cpctech.cpc-live.com/docs/bastech.html [letzter Abruf: 21.03.2018].

72 Hierunter sind alle PRINT-Routinen gefasst, etwa auch PRINT SPC, PRINT USING, die PRINT-Ausgabe von Strings, Zahlen und die Bearbeitung von Formatierungszeichen.

Abb. 4.1.17: Drei Lehrbücher zum „strukturierten Programmieren in BASIC“ [Nevison 1982; Baumann 1983; Schneider 1985].

4.1.5.6 (De)Bugging VCS Boing

Wie bereits erwähnt, zwangen und zwingen insbesondere die technischen Begrenzungen und Idiosynkrasien historischer Computer damalige und heutige Programmierer solcher Systeme zu stark hardwareorientierter Programmierung und den Rückgriff auf ‚Tricks‘. An den hier diskutierten Programmen zeigt sich das nirgends so deutlich wie an der Boing!-Adaption für *Ataris* Spielkonsole VCS/2600. Allein die Darstellung eines Balls auf dem Bildschirm erfordert exakt ‚getimten‘ Code, kann die Konsole doch mangels Videospeicher mit ihrem Grafikchip TIA stets nur ein Pixelzeilen hohes Bildelement darstellen (und dieses bestenfalls noch einmal gespiegelt in derselben Pixelzeile). Zur Generierung einer flächigen Grafik, wie der einer Kreisscheibe, muss diese also in die einzelnen Pixelzeilen aufgelöst werden und synchron mit dem Rasterstrahl des Bildschirms Zeile für Zeile gezeichnet werden. Die Geschwindigkeit, mit der dies geschieht, ist der alleinige Grund dafür, dass beim menschlichen Betrachter trotzdem der Seheindruck eines Kreises entsteht: Das Nachleuchten der CRT-Bildröhre führt dazu, dass das erste gezeichnete Bildelement noch nicht verloschen ist, wenn das letzte gerade gezeichnet wird. Verfolgt man diesen Prozess jedoch in Zeitlupe[73], wird der technische Trick sichtbar. Weil die Spielkonsole deshalb vor allem mit der zeitkritischen Ausgabe der Spielgrafik beschäftigt ist, können die übrigen Routinen erst dann abgearbeitet werden, wenn gerade keine Grafik auf den Bildschirm gebracht werden muss.

Dies geschieht zum einen am Ende einer jeden Bildschirmzeile (horizontal blank) und nachdem der Kathodenstrahl das Ende des Bildschirms erreicht hat (vertical blank); zum anderen, wenn der Kathodenstrahl in jenen Bereichen ist, die von der VCS nicht für die Bilddarstellung zur Verfügung stehen (Overscan) (vgl. Abb. 4.1.11). Da die 6507-CPU über

73 https://youtu.be/3BJU2drrtCM?t=105 [letzter Abruf: 21.03.2018] zeigt die Hochgeschwindigkeitsaufnahme (1600, 2500, 28500 und 188000 frames per second) des Bildschirmaufbaus einer Nintendo-NES-Spielkonsole.

keine Interrupt-Leitungen verfügt, können die Zeitpunkte, in denen der Kathodenstrahl nicht zu sehen ist, nicht auf diese Weise vom Grafikchip an die CPU übermittelt werden. Hierfür dienen jedoch unterschiedliche Timer (*waitblank, waitscreen, waitoverscan*, die vom Programmierer selbst zu verwalten sind [vgl. Moll 2014:74].) Mittels dieser Timer können die entsprechenden Register im TIA-Chip (VSYNC, VBLANK, WSYNC [vgl. Hugg 2015:35]) synchronisiert werden.

Bei dieser „Racing the Beam"-Programmierung ist der ‚racer' jedoch nicht etwa der Programmierer, sondern der Computer. Der Programmierer hat lediglich dafür Sorge zu tragen, dass seine Grafikausgabe- und übrigen Programmroutinen zur richtigen Zeit ausgeführt werden. Hierzu muss er in Kenntnis der Taktzyklen der CPU und der Größe seiner Programmanweisungen den Zeitverbrauch korrekt bestimmen. Rein rechnerisch lässt sich dies zwar aus dem Sourcecode des Programms herausarbeiten (ähnlich wie in der Warteschleifenkonstruktion in Kap. 4.1.5.2), die zeitkritische Diffizilität des Prozesses wird jedoch erst während der Ausführung der Demo deutlich. Dies lässt sich abermals über einen Emulator mit integriertem Debugger zeigen, der es überdies ermöglicht zu zeigen, was geschieht, wenn man (etwa durch Änderung von zeitkritischen Routinen) den Computer und den Bildschirm desynchronisiert (vgl. Abb. 4.1.18).

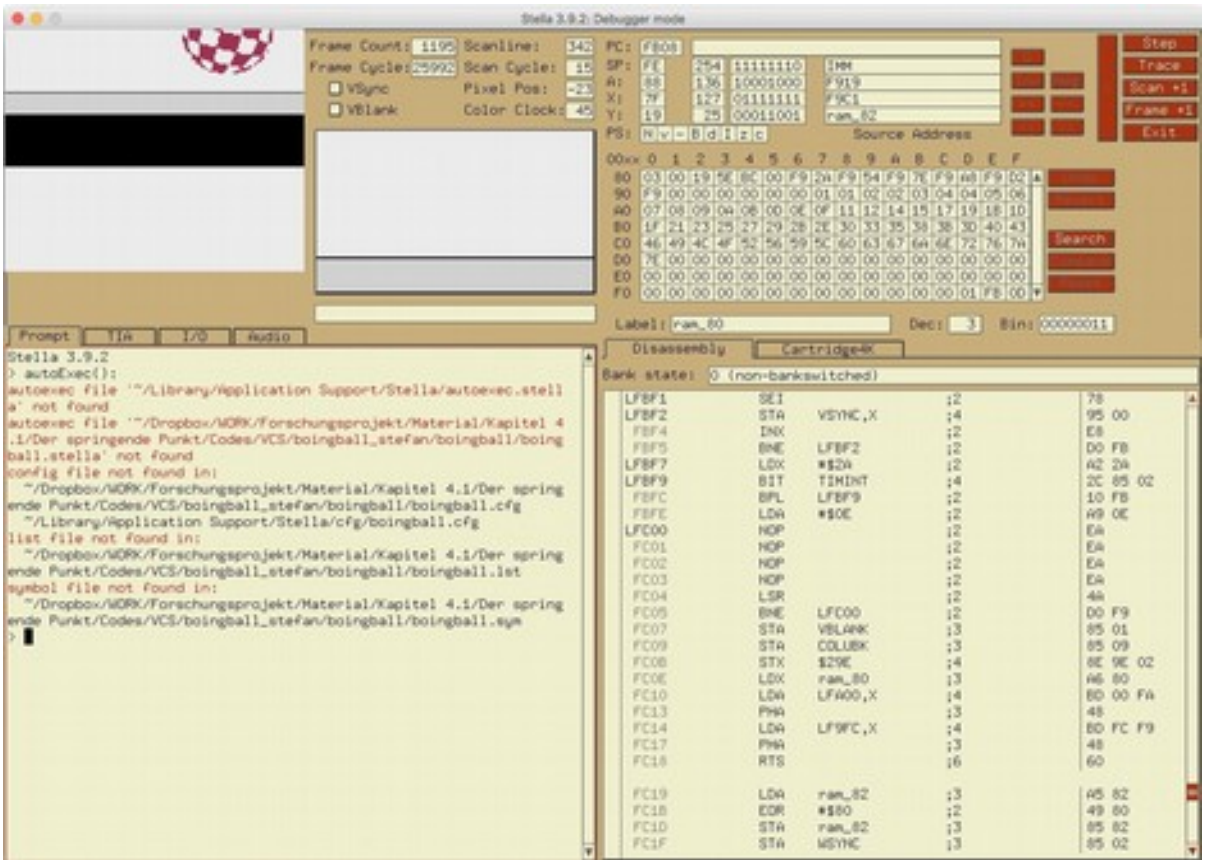


Abb. 4.1.18: VCS Boing Im Stella-Emulator. Im Debugger wurden die Adressen $FC00_{16}$ bis $FC02_{16}$ mit NOPs (EA_{16}) gefüllt und dadurch zwei Synchronisationsoperationen überschrieben. Die Grafikausgabe (oben links) zeigt ein Bild mit vertikalem Bildlaufen.

Der hier künstlich eingebrachte Fehler führt zu Erkenntnissen über die mikrozeitlichen Prozesse des Systems, die sich allerdings erst zur Laufzeit einstellen. Auch hier ist der Programmierfehler lediglich semantischer Natur; die CPU verarbeitete die NOP-Instruktionen fehlerfrei.

Die erneute Diskussion der Programmbeispiele, nun aus der Perspektive der jeweiligen Systeme betrachtet, sollte zeigen, wie sehr sich menschliches und maschinelles ‚Verständnis' von Programmcode unterscheiden. Lässt sich Code als Text noch mit klassisch-philologischen Methoden nachvollziehen, bedarf es für den Nachvollzug der maschinellen Perspektive anderer Bewertungskriterien und Instrumente. Der radikale, computerarchäologische Unterschied zwischen Literatur und Codes bleibt also irreduzibel in der Operativität der Computerprogramme zu suchen: Ihr ‚Sinn' vollzieht sich erst nach dem Start des Programms, das sie kodieren. Anders als die Rezeptionstheorie der Literatur, die das medientechnische Substrat der Literatur (Papier) und den technologischen Prozess der Texterstellung (Schrift, Drucktechnik, ...) ignoriert, ist ein Programm als bloßer ausgedruckter oder angezeigter Text nicht vollständig verstehbar. Hier muss das Substrat (die verschiedenen Speicher, die Leitungen/Busse und ihre Transportprozesse und -geschwindigkeiten und die Arbeitsweise der CPU) mitbedacht werden.

Die Ausführung findet allerdings erst *nach* dem Zusammenstellen/-schreiben der Codes statt, die sich im Prozess der Kompilation/Interpretation in einen für die Maschine ‚verständlichen' Text umschreiben. Und selbst dort, wo der Text bereits maschinenlesbar (etwa als Opcodes) eingegeben wurde, muss er noch in der Abarbeitung durch das Mikroprogramm in einzelne Mikroanweisungen übersetzt werden, die sich insbesondere bei diskret aufgebauten historischen Systemen, schließlich als Signale auf der Hardwareebene messen lassen. Daher findet die ‚eigentliche' Rezeption des Codes, verstanden als dessen *(Aus)Wirkung im Realen,* nicht durch einen Menschen, sondern immer schon erst durch den Computer statt. Aber selbst diese Prozesse lassen sich noch im Sinne einer Rezeptionstheorie deuten, die dann allerdings innerhalb einer ‚Mikro-Chronologie' wirkt, welche das Zeitbewusstsein des Menschen aufgrund ihrer Geschwindigkeit unterläuft und sich damit jedem emphatischen Geschichtsbegriff entzieht. Software schreibt sich hier nicht nur selbst in die Software-Geschichte ein, sie *interagiert* mit ihr, weil ein laufendes Programm mit den Zitaten (zitierten Routinen und Algorithmen) *arbeitet* – ein Re-Enactment im Sinne Collingwoods [1955:300].

Diese Form der automatisierten Geschichtsarbeit unterscheidet sich noch einmal grundlegend von aktuellen Ansätzen aus der Forschung zur Software-Evolution [Mens u.a. 2005; D'Ambros u.a. 2008]. Dort lesen Computer Code-Texte auf Basis maschinellen Lernens oder Big-Data-Auswertungen von Code-Repositories [Zimmermann u.a. 2004], um daraus ökonomisch verwertbare Erkenntnisse (etwa über die ‚Vererbung' von Programmierfehlern in unterschiedlichen Programmversionen [Antoniol u.a. 2008; Li/Zhou 2005] oder die Klärung der Urheberschaft einzelner Programme [Caliskan u.a. 2018]) zu gewin-

nen. Softwarearchive und Repositories können mit Hilfe von Computern durchsucht und analysiert werden. Tools der Digital Humanities ermöglichen neue quantitative Analysen (zum Beispiel für Anti-Plagiat-Software [vgl. Maurer et al. 2006; Lee 2011]), Codevergleiche und statistische Aussagen über solche Archive, die durch menschliches Tun allein aus zeitlichen Gründen kaum zu bewältigen wären.

4.1.6 Zusammenfassung

4.1.6.1 Programmierdidaktik und Linguistik

Dass Programme[74] in Programmier*sprachen geschrieben* werden, kennzeichnet bereits die Verwandtschaft des Programmierens mit dem Schreiben und der Schrift nicht-formaler Sprachen. Bereits Alan Turing hat auf diese Verwandtschaft hingewiesen:

> *Wenn man auf Englisch, wo nötig unter Zuhilfenahme mathematischer Symbole, vollkommen unzweideutig erklären kann, wie eine Rechnung ausgeführt werden muß, dann ist es stets auch möglich, einen beliebigen Digitalrechner zur Durchführung dieser Rechnung zu programmieren, vorausgesetzt, daß die Speicherkapazität ausreicht.* [Turing 1987b:121 – Hervorh. i. O.]

Algorithmen sind also natur- wie formalsprachlich formulierbar und ineinander übersetzbar. Die Übersetzung von natürlichen in formale Sprachen für Computer dient der „Visualisierung des Kognitiven" [Krämer 2005:52] – hier mathematischer Sachverhalte – und ist als Kulturtechnik historisch gewachsen, wie Hagen [1997:66] zeigt.

Diese Verwandtschaft zwischen formalen und natürlichen Sprachen ist sowohl beim Versuch der ‚Formalisierung' natürlicher Sprachen (Leibniz' *Lingua Universalis*) als auch in Hinblick auf die Programmiersprachenentwicklung bedacht worden. COBOL und die Vorläufersprache FLOW-MATIC wurden von Grace Hopper mit dem Zweck entwickelt Sprachen zur Verfügung zu stellen, die auch von Nicht-Informatikern gelesen werden können. BASIC wurde von Kurtz und Kemeny ursprünglich für Studenten von Fächern aus den Bereichen *arts* und *humanities* entwickelt. Obwohl in beiden Fällen die Nähe der natürlichen Sprache Englisch und der jeweiligen Programmiersprache das Entwicklungsparadigma bildete (Verständlichkeit auf der menschlichen Seite), ist verschiedentlich darauf hingewiesen worden, dass eine immer stärkere ‚Naturalisierung' von Programmiersprachen nicht zielführend sein kann [vgl. Dijkstra 1978]. Dennoch wird diese Idee stetig weiter verfolgt (etwa durch esoterische Programmiersprachen [vgl. Höltgen 2018a:120f. FN8]).

Die Tatsache allerdings, dass syntaktisch-grammatische Beschreibungen zur *Analyse* beider Sprachtypen angewendet werden können, eröffnet besondere didaktische Mög-

74 Das lat. *programma* heißt auf Deutsch übersetzt u.a. „schriftliche Bekanntmachung"; die Konstituente *-gramma* steht für „Schrift" (vgl. *Grammatik*).

lichkeiten. Die vorangegangenen Ausführungen haben gezeigt, wie sich Analysemethoden aus der Sprach- und Literaturwissenschaft (natürlicher Sprachen) auf Programmcodes und sogar nicht-symbolische Implementierungen für Analogcomputer anwenden lassen. Dieses Instrumentarium ließe sich sowohl für die Programmierlehre fruchtbar machen (etwa, indem Homologien zwischen Sprachen aufgezeigt werden) als auch für eine allenthalben eingeforderte *code literacy*, also „das Verständnis dafür, wie Computer und Programme arbeiten“ (Vgl. Fußnote 53). Neben der anzustrebenden *aktiven* Programmierausbildung auf für Informatik-ferne Gruppen, könnte sich eine Informatik-Didaktik in Kooperation mit linguistischen Disziplinen auch für eine solche *passive* Programmiersprachenkenntnis einsetzen. Ansätze für die Programmierlehre hierfür existieren bereits. Sich Sourcecode (zunächst) lesend zu nähern wird seit einigen Jahrzehnten propagiert [vgl. Raymond 1991] und selbst Thomas Kurtz, einer der Entwickler der Programmiersprache BASIC, favorisiert als ersten Lernschritt die (lesende) Konsultation des Programmierhandbuches [zit. n. Biancuzzi/Warden 2009:91].

Busjahn[75]/Schulte [2013] untersuchen die Rolle des Codelesens im Prozess des Programmierenlernens. Sie definieren: „By CR [Code Reading, S.H.] we do not mean, that reader‘s eyes move over the text. It rather denotes the first step in program comprehension, taking in the written elements and understanding them.“ [Ebd.:3] Dieser Lese- und Verstehensprozess seien untrennbar miteinander verbunden:

> only information considered as relevant during that moment will be processed and included in the process of chunking information into comprehensive forms of elements. So it is much more than just a translation that takes place in reading. Program reading and program comprehension thus cannot be separated, but are highly intertwined. In essence, therefore comprehension problems might be grounded in reading problems. [Ebd.:4]

Der lesende Zugang zum Verständnis des eigenen Codes findet nicht nur während des Programmierens statt, sondern auch während „debugging and maintenance“ [ebd.:6]. Das Schreiben, Lesen des Geschriebenen, Korrigieren des Geschriebenen aufgrund der (Re-)Lektüre stellt einen Kreislauf zwischen Computer, Programmtext und Programmierer dar, der im Sinne der Kybernetik als „Trial and Error“-Feedback-Schleife aufgefasst werden kann. Dabei tritt der menschliche Codeleser Busjahns/Schultes zufolge bereits aus ‚ausführendes Organ‘ („comprehension“) auf. Wodurch dieser Verstehensprozess im einzelnen getriggert und forciert wird, delegieren die Autoren an weitergehende Forschungen. (Das vorangegangene Kapitel hat Vorschläge hierfür als Computerphilologie

75 Teresa Busjahn, die Ko-Autorin des Beitrags, hat 2010 ihre Magisterarbeit am *Fachgebiet Medienwissenschaft* der *Humboldt-Universität zu Berlin* zum Thema „Perzeption von Quelltext als ein elementarer Prozess im Kontext von Medienkompetenz“ geschrieben. Hierin hat sie Eye-Tracking-Experimente bei der Codelektüre ausgewertet, um die Unterschiede und Ähnlichkeiten beim Lesen formal- und natürlichsprachlicher Texte zu markieren. Hierzu hat sie später auch publiziert [vgl. Busjahn u.a. 2015].

unterbreitet.) Busjahn/Schulte konzentrieren sich auf die Bedeutung der Codelesefähigkeit für das Programmierenlernen – also vor allem auf eine synchrone Perspektive. Hierzu wurde oben eine diachrone Perspektive ergänzt und die Bedeutung der Codelektüre für die Software-Geschichte hingewiesen.

4.1.6.2 Mediensprachen

Das vorangegangene Kapitel konnte aber ebenso zeigen, dass ein allein menschliches (lesendes) Nachvollziehen von Computerprogrammen deren letztlichen ‚Sinn' kaum vollständig zu offenbaren in der Lage ist. Der letztliche Zweck der Programmierung offenbart sich allein in der *Demonstration*, die deshalb oben bereits als eine Methode computerarchäologischer Analyse dargestellt wurde (vgl. Kap. 3.3.2). In der Demo-Programmierung stellen die Entwickler diese Prozesshaftigkeit von Code und Hardware auf die Bühne und erzeugen damit ein „Medientheater" [Pias 2002a:51; Hartmann 2017:250; Leeker 2001] (vgl. Kap. 3.3.2), in welchem nicht mehr Menschen sondern Maschinen Texte auf- bzw. ausführen. Aus diesem Grund erachtet Botz [2011] die Echtzeitausführung von Demos als konstitutiv für das Software-Genre und kontrastiert dies mit dem Unterschied von Theater- und Film-Vorführung:

> Das Konzept der Demoscene entspricht [..] der Szene als Theaterbühne, wie es sich vom griechischen ‚skene' ableitet. Auf einer Bühne agieren alle Beteiligten innerhalb von vereinbarten Rollen mit der Absicht, etwas darzustellen und ihre Fähigkeiten zu demonstrieren. [Botz 2011:18] Der Aspekt des Hervorbringens gegenüber der puren Reproduktion und das passive Teilhaben des Zuschauers an der Hervorbringung unter den gegebenen Bedingungen machen den Unterschied zwischen einer Filmvorführung und dem Ausführen einer Computerdemo aus. [...] Die Betrachtung einer Computerdemo unterscheidet sich also wesentlich von der eines Films im Kino, Fernsehen oder Internet. Die Kopplung der Wahrnehmung an die Vorstellung von der Flexibilität der Hardware, auf der die Demo ausgeführt wird, führt zu einer Situation, in der der Zuschauer die Demo nicht als zufälliger Empfänger, sonder grundsätzlich als ‚Eingeweihter' verstanden wird. [Ebd.:289f.]

Computerphilologie ist also in dieser doppelten Hinsicht zu verstehen als Lektüre von Code durch menschliche Leser und durch den Computer selbst, wobei sich die Lektüreergebnisse in unterschiedlichen Zeitverhältnissen zum Lektüreprozess ergeben: Performanz/Demonstration auf Seiten des Computers, Hermeneutik/Nachvollzug auf Seiten des Menschen. Diese Differenz wäre für eine Software Preservation fruchtbar zu machen, die nicht nur den Erhalt der Datenträger (und ihrer Speicherinhalte) berücksichtigt, sondern auch deren Beziehung zu einem realen Hardware-System als konstitutiv für das Verständnis der Softwaregeschichte versteht.

Die Lesbarkeit von historischem Code ist keine Frage, die allein Archive und Museen beschäftigt; es bestehen auch schon seit längerem [vgl. Schmitz 1991] wirtschaftliche Interessen, Programme in ‚alten' Programmiersprachen auf aktuelle Systeme zu portieren. Konzepte, wie die modulare oder objektorientierte Programmierung, die die Wiederverwendbarkeit von Code(bestandteilen) in ihre Paradigmen integriert haben, sind auf Software, die vor der Erfindung solcher Sprachen entstanden ist, nicht anwendbar. Automatische Übersetzungen von historischen in aktuelle Sprachen sind noch nicht in Sicht [Keller 2017] und zudem bei maschinennah programmierter Software kaum ohne dezidierte Kenntnisse der ursprünglich bezogenen Plattformen einsetzbar. Hier bleiben Behelfslösungen, wie die Emulation durch Wrapper[76]. Die Übersetzung durch menschliche Programmierer, die auf Basis von linguistischen Konzepten, wie in Kapitel 4.1.4 vorgestellt, hat bereits zu praktikablen Ergebnissen geführt, wie beispielsweise die Spielportierung zwischen verschiedenen Generationen von Computer(spiel)hardware oder die Praxis der Software Migration [vgl. Wagner 2014] zeigt.

Auch für eine andere Quelle von Software wird menschliche Übersetzungsarbeit notwendig bleiben: Viele computerhistorische Software-Artefakte schlummern in noch unerschlossenen Quellen, wie historischen Computerzeitschriften, Privatarchiven und urheberrechtlich geschützten Publikationen. Hier stößt automatisierte Archivarbeit wie bei Zimmermann [u.a. 2004] an ihre Grenzen und macht den Einsatz einer menschliche geleiteten Computerphilologie notwendig. Beide Sichtweisen auf Computerphilologie müssten also einander zu ergänzen.

Als ein wichtiges Tool hierfür haben sich Software-Emulatoren erwiesen. Mit ihnen ist es möglich mikrozeitliche Codeausführungen für die menschliche Wahrnehmung aufzubereiten, experimentelle Eingriffe in laufende Programme vorzunehmen und nicht zuletzt Software auf ihre Lauffähigkeit hin zu testen. Insbesondere dieser letzte Aspekt darf den Test (die Demonstration) von Programmen auf dedizierter Hardware jedoch nicht suspendieren, denn Softwareemulatoren implementieren weder die reale Physik noch überhaupt alle technischen Funktionen und Idiosynkrasien von Computern. Sie sind nicht nur Werkzeuge, sondern auch eine weitere Lektüreinstanz, die den Fremdcode in den Code des Hostsystems übersetzt und dabei weniger das zu emulierende System als ihre eigene ‚Interpretation' von dessen Funktionen vorführt.

Pias sieht Emulatoren daher als „maschinisierte Methode" der Code-Lektüre, die zu einer „Interpretation zweiter Ordnung" [Pias 2017:384] führt. Er sieht die Defizite von Emulation jedoch vor allem in einem technisch nicht kompensierbaren Mangel:

76 Wrapper (bzw. Adapter) werden beispielsweise unter dem Betriebssystem Linux benutzt, um hardwarenah programmierte Software, wie Gerätetreiber, die nicht für Linux entworfen wurden, zu nutzen. Hierzu werden die Originalprogramme in einen von Linux ausführbaren Code ‚eingebettet', welcher als Softwareschnittstelle nach dem Prinzip eines Emulators zwischen den Funktionen der Programme und dem Betriebssystem vermittelt [vgl. Eilebrecht/Starke 2010:103-105].

> Stattdessen könnte man [...] Emulatoren mit einem *gebauten kritischen Apparat* vergleichen: einem Apparat, ohne dessen Kritik der Überlieferung das wahrnehmbare, konzeptuelle Objekt nicht erreichbar wäre, der selbst wiederum einen kritischen Apparat mit sich führt und dessen Zukunft darin besteht, zu seinen Lebzeiten selbst immer wieder emuliert zu werden, um den Erhalt von Lesbarkeit zu gewährleisten. [Ebd.:381]

Vor dem Hintergrund der obigen Diskussion sollte Emulation daher vielleicht weniger als technisches Surrogat von Hardware, denn als *maschinelles Re-Enactment* zur Unterstützung von Wissensprozessen verstanden werden. Damit ließe sich dann auch die durchaus variable Qualität unterschiedlicher Emulatoren (im Sinne einer ‚authentischen' Emulation der Zielsysteme) epistemologisch fruchtbar machen. Dieser Aspekt von Emulation wird im folgenden Kapitel (4.2) diskutiert, bevor die Frage nach der technologischen Beziehung zwischen Original-Hardware und deren Software-Emulation (in Kapitel 4.3) aufgeworfen und beantwortet wird.

4.2 Computerspiele und Simulationen

Wie jedes historische Denken findet auch das Denken technikhistorischer Gegenständen als ein ‚Nachdenken' in der Gegenwart statt. Anders als Objekte der Zeitgeschichte, sind technische Artefakte jedoch nicht bloß menschlicher Deutung zu unterwerfen, sondern auch deren technische ‚Eigenheiten' (Zeitlichkeit, apparativer Idiosynkrasien, ...) zu berücksichtigen. Für eine adäquate technikhistorische Reflexion gilt es solche Aspekte ebenso nachzuvollziehen, wofür sich besser das Experiment als der Diskurs eignet. Wie ein solches Experiment vollzogen werden kann, soll in diesem Kapitel ebenso vorgestellt, wie die Problematik reflektiert werden, ob ein Artefakt, wenn es für ein solches Experiment nicht zur Verfügung steht, auch als Simulation untersucht werden kann. In welcher Beziehung steht dabei das Artefakt zu seiner Simulation und wie lassen sich Erkenntnisse vom einen auf das andere übertragen?

Im Zentrum der folgenden Überlegungen stehen *zelluläre Automaten*, die nicht nur geeignet sind diskrete Prozesse algorithmisch zu simulieren, sondern zugleich auch selbst ein historisch reichhaltiges Motiv der theoretischen und angewandten Informatik darstellen. Daher wird es hierbei zu einer terminologischen Konfrontation der Begriffe *Simulation* und *Emulation* kommen, wobei auch das seit einigen Jahrzehnten bestehende Arbeitsfeld der Computersimulation diskutiert wird. Die Reichweite computersimulatorischer Forschung wird hier beispielhaft durch das Arbeitsfeld des *Unconventional Computing* abgesteckt, welches um die computerarchäologische Facette simulativen *Toy Computings* ergänzt werden soll.

4.2.1 Simulation

Im Kapitel 4.1.1 wurde die Simulation physikalischer Prozesse mittels Analogcomputern bereits vorgestellt. Die Analogrechner-Technologie gilt als früheste Form der maschinellen Simulation [vgl. Ulmann 2010:13ff.]. Mechanische Analogrechner werden mindestens seit dem Jahr 150 v.u.Z. genutzt, elektromechanische Analogrechner seit 1923 [vgl. ebd.:25ff.]. Der praktische Vorteil von Analogrechnern liegt in ihrer nahezu verzögerungsfreien Ergebnisdarstellung; ihre Nachteile darin, dass ihre Ein- und Ausgaben nicht numerisch erfolgen, sie physikalisch-bedingte Rechenungenauigkeiten, insbesondere bei sehr kleinen Rechenwerten, aufweisen und in ihrer fehlenden Universalität. Digitalcomputer, die diese Nachteile nicht aufweisen bzw. technologisch kompensieren, wurden ab dem Moment zu Simulationszwecken eingesetzt, als ihre Rechengeschwindigkeiten soweit anstiegen, dass sie bei mindestens gleicher Genauigkeit (nach menschlichen Maßstäben) ebenso schnell wie Analogcomputer arbeiteten.

Am Unterschied zwischen Analog- und Digitalcomputer-Simulationen zeigt sich bereits ein Definitionskriterium des Begriffs „Simulation“: *kontinuierliche* versus *diskrete Simulation*:

> Bei der diskreten *Simulation* werden lediglich ausgewählte Werte zu bestimmten klar unterscheidbaren Zeitpunkten berechnet. Auf diese Weise erreicht man eine Einsicht in das Verhalten eines komplexen Systems, in dem nur wenige wichtige Ergebnisse berechnet werden. [...] Im Gegensatz zur *diskreten Simulation* werden sämtliche Werte des Systems in Abhängigkeit von der kontinuierlich fortschreitenden Zeit berechnet. Dadurch lassen sich am besten zeitliche und kausale Abhängigkeiten, insbesondere *Rückkopplungen* in Systemen, erfassen. Die *numerische* Berechnung von Differentialgleichungen, z.B. mit der Rechteckintegration, stellt eine Näherungslösung für die kontinuierliche *Simulation* dar. [Strübel 1986:534f. – Hervorh. i. O.]

Epistemologisch betrachtet liegt der Unterschied zwischen den beiden Simulationsvarianten in der Beschreibung des zu simulierenden Modells: Bei *kontinuierlichen Simulationen auf Analogcomputern* wird ein Analogie-Modell ohne Umweg über eine symbolische Kodifizierung *im Realen konstruiert* (bei dem etwa ein kinetischer Bewegungsablauf als elektrischer Spannungsverlauf dargestellt wird, vgl. Kap. 4.1.4.3), mathematisch in infinitesimal kleinen (Zeit)Einheiten beschrieben und non-numerisch verarbeitet. Die Simulationsqualität wird durch die Größe und technische Komplexität des Rechners bestimmt. Für *diskrete Simulationen* auf Digitalcomputern muss das Modell zunächst in ein *symbolisches Zeichensystem* (ein Programm) kodiert werden; die mathematische Beschreibung (über Differentialgleichungen) kann dann auch nicht kontinuierlich, sondern muss aufgrund der nummerischen und zeitdiskreten Abarbeitung von Programmen durch die CPU schrittweise erfolgen. Die Rechengeschwindigkeit begrenzt hierbei die An-

zahl dieser Simulationsschritte sowie der Parameter und damit die Qualität der Simulation.

Der Terminus Simulation wird auch in anderen Zusammenhängen[77] und nicht immer kongruent zur hier beschriebenen *technischen Simulation* verwendet. In der Theoretischen Informatik findet sich ein weiterer Simulationsbegriff. Hier beschreibt Simulation

> ein Verfahren, mit dem Programme für einen Rechnertyp durch Programme auf einem anderen Rechnertyp so nachgeahmt, d. h. simuliert werden können, daß sie dasselbe Eingaben-Ausgabe-Verhalten haben. Wenn sich also zwei Rechnertypen gegenseitig simulieren können, sind für sie dieselben Probleme lösbar und damit auch dieselben Probleme unlösbar. [Wegener 1997:100]

Adamatzky [2002b:V] spezifiziert dies als *Universalitätssimulation* von Systemen: „If a system simulates behavior of a universal machine, which universality has been already proved, it is called simulationally universal." Am Schluss dieses Kapitels und im nachfolgenden Kapitel 4.3 wird sich zeigen, dass diese Definition aus der Perspektive der *angewandten Informatik* Schnittmengen mit dem Begriff *Emulation* bildet.

Ein weiteres Definitionskriterium für Simulationen ist, ob bei der Beschreibung des zu simulierenden Systems *stochastische Elemente* einfließen sollen oder nicht. *Stochastische Simulationen* (z. B. Monte-Carlo-Simulationen) benötigen einen Zufallszahlengenerator, dessen Qualität die Adäquanz der Simulation mitbestimmt.

Qualität	diskret		kontinuierlich	
Modell	deterministisch	stochastisch		deterministisch
Universalität	universell	spezialisiert		
Beispiel	Game of Life	Monte-Carlo-Simulation	Maxwell-Brown-Simulation	Tennis for Two
Quantität	numerisch (Funktion)	Numerisch (Zufallszahlen)	non-numerisch (stochastische DGl)	non-numerisch (gewöhnliche DGl)

Tabelle 4.2.1: Klassifikation von Simulationen mit Anwendungsbeispielen

Da die oben bereits verwendeten Begriffe „System" und „Modell" im Zusammenhang mit Simulationen eine zentrale Rolle spielen, sollen auch sie noch definiert werden: „A system is a *construct or collection of different elements that together produce results not obtainable by the elements alone.* [...] Importantly, the value of the system as a whole is the relationship among the parts." [Banks 2009:7 – Hervorh. i. O.] Systeme können entweder

77 Die weitreichende und nicht unumstrittene Verwendung des Begriffs „Simulation" in den Geistes- und Kulturwissenschaften kann für die vorliegende Betrachtung ausgeklammert werden. [Vgl. Höltgen 2010:163f. sowie Höltgen 2003].

diskret oder kontinuierlich beschaffen sein [vgl. ebd.:7]. Der Modellbegriff wird aus dem Systembegriff abgeleitet:

> A *model* is a representation of an event and/or things that is real (a case study) or contrived (a use-case). It can be a representation of an actual system. It can be something used in lieu of the real thing to better understand a certain aspect about that thing. To produce a model you must abstract from reality a description of a vibrant system. The model can depict the system at some point of abstraction or at multiple levels of the abstraction with the goal of representing the system in a mathematically reliable fashion. [Ebd.:5]

Die Simulation stellt damit eine Methode zur Beschreibung von Systemverhalten auf Basis eines mathematischen oder anderweitig symbolischen Modells dar [vgl. ebd.]. Als solche wurde sie 1999 von der *National Science Foundation* als dritter Zweig (neben *Theorie* und *Experiment*) wissenschaftlicher Methodik deklariert [vgl. ebd.:3]. Die Simulation auf Basis von Modellen biete sich als Alternative für das Experiment an, denn „[d]as Modell kann in einer Weise manipuliert werden, die bei dem wirklichen System unmöglich [...], zu gefährlich [...] oder zu zeitraubend wäre [...]." [Mertens 2009:534]

Dies gilt insbesondere für die Simulation mittels Computern, weshalb sich der *Deutsche Wissenschaftsrat* in einem Positionspapier für die Etablierung einer Simulationswissenschaft als eigenständige akademische Disziplin und als Studienergänzung für verschiedene Fächer ausgesprochen hat [vgl. Deutscher Wissenschaftsrat 2014:16-26]. In dem Empfehlungsschreiben wird jedoch lediglich die diskrete Simulation und diese auch bloß vor ihrem praktischen Anwendungshintergrund diskutiert, was sich in der Behauptung zeigt, dass „rechnerbasierte mathematische Modellierungen zum Zwecke der Simulation" erstmals in „den fünfziger Jahren" eingesetzt wurden [ebd.:7, vgl. kontrastierend dazu Ulmann 2010:13ff. sowie Banks 2009:7-14] und darin, dass sowohl medientheoretische/epistemologische als auch ethische/gesellschaftliche Aspekte der Simulation unberücksichtigt bleiben.

Im Folgenden wird eine Form diskreter Simulation (der zelluläre Automat „Game of Life") in einem computerarchäologischen Projekt eingesetzt und in ihrer Theorie und Historizität vorgestellt.

4.2.1.1 Zelluläre Automaten

Zelluläre Automaten sind in mehrfacher Hinsicht hybride Objekte: Sie bilden ein vielseitiges Verfahren diskreter Simulation unterschiedlichster Prozesse aus Biologie, Soziologie, Kybernetik, Medizin, Militär, Spieltheorie und anderen Disziplinen und sie sind zugleich algorithmische Beschreibungen theoretisch-informatischer Sachverhalte (der Automatentheorie). Sie stellen ein sehr frühes und kontinuierlich weiter entwickeltes Verfahren diskreter Computersimulation mit einem beachtlichen kulturellen Einfluss

dar. An ihnen entzündeten und entzünden sich gleichermaßen wissenschaftstheoretische wie forschungspolitische Debatten [vgl. Mainzer/Chua 2012].

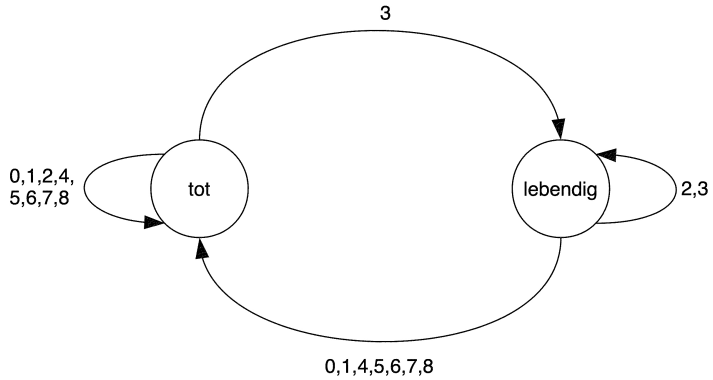


Abb. 4.2.1: „Game of Life“ als Diagramm eines Zustandsautomaten

Bei zellulären Automaten handelt es sich um Zustandsautomaten, die durch einzelne Zellen repräsentiert werden. Diese werden durch ihre möglichen *Zustände* und *Übergänge* beschrieben. Diese Zustände wechseln (oder wechseln nicht) auf Basis von *Übergangsregeln* nach diskreten Zeitabläufen (vgl. Abb. 4.2.1). Je nach Anzahl der Zustände und Beschaffenheit der Übergangsregeln können zelluläre Automaten verschiedenen Komplexitätsklassen der Automatentheorie zugeordnet werden.

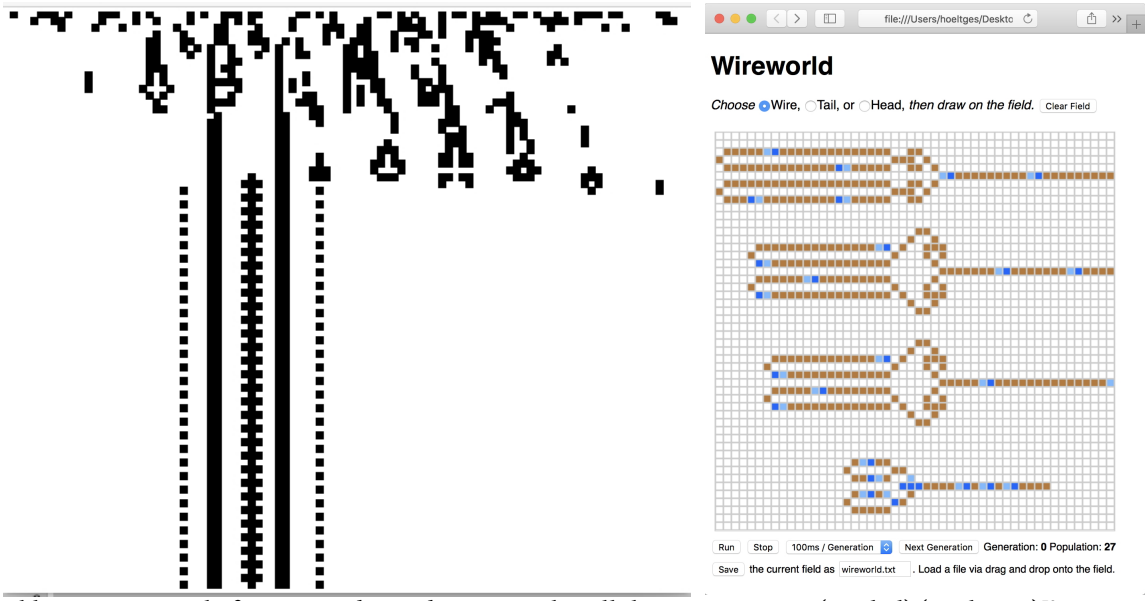


Abb. 4.2.2: Beispiele für ein- und zweidimensionale zelluläre Automaten: (Nückel) (Maibaum)[78]

Ein isolierter zellulärer Automat (A) lässt sich mathematisch beschreiben als:

$$A(Z_{t+1})=f_{ü}(A(Z_t))$$

Sein Zustand (Z) ist ein Funktionswert, der sich auf Basis von Übergangsfunktionen ($f_{ü}$) verändert, deren Variable den zeitlichen Übergang von einem Zustand (t) in den nächsten Zustand (t+1) repräsentiert. Die Zustände sind dabei determiniert von den Zu-

78 Beide: https://www.musikundmedien.hu-berlin.de/de/medienwissenschaft/medientheorien/signallabor/praxisarbeiten [letzter Abruf: 16.05.2018].

ständen der ‚Nachbarzellen'. Die Veränderung kann auf ein Raster aufgetragen werden, das mathematisch als Matrize mit beliebig vielen Dimensionen darstellbar ist. Hier für einen zweidimensionalen zellulären Automat:

$$\begin{vmatrix} A_1(Z_t) & A_2(Z_t) & A_3(Z_t) \ldots & \\ A_4(Z_t) & A_5(Z_t) & A_6(Z_t) \ldots & \\ A_7(Z_t) & \ldots & & \end{vmatrix} = F$$

Dieser Umstand legt die Implementierung zellulärer Automaten auf Digitalcomputern nahe.[79] Oder anders herum ausgedrückt: Je mehr Dimensionen das Raster besitzt, je mehr Zellen miteinander interagieren, je mehr mögliche Zustände eine Zelle einnehmen kann und je mehr Regeln die Übergänge zwischen den Zuständen beschreiben, desto schwieriger wird die Durchführung eines solchen Algorithmus ohne (computer)technische Unterstützung.

Zelluläre Automaten lassen sich historisch bis zur Erfindung der Digitalcomputertechnologie zurückverfolgen [vgl. Zhuang u.a. 2017]; aus der Architektur dieser Rechner scheint die Entwicklung zellulärer Automaten buchstäblich hervorzugehen. So hat beispielsweise der Mathematiker Stanley Ulam Ende der 1930er-Jahre [vgl. Adamatzky 2002b:VII] einen frühen zellulären Automaten entworfen, dessen Konzept von John von Neumann [1966] aufgegriffen und erweitert wurde. Alan Turing [1952] hat sich mit der Morphogenese beschäftigt und diese 1952 erstmals mittels eines zellulären Automaten erklärt. Und Konrad Zuse [1967; 1969] hat 1967 mit dem „Rechnenden Raum" einen kosmologische Theorie auf Basis zellulärer Automaten vorgestellt.

Zu den jüngeren, diskursmächtig gewordenen Arbeiten zählt Stephen Wolframs „A New Kind of Science" [Wolfram 2001], worin er den von ihm seit 1985 entwickelten eindimensionalen zellulären Automat mit zwei Zuständen als Grundlage für einen wissenschaftlichen Paradigmenwechsel diskutiert: Mit Hilfe zellulärer Automaten soll dabei theoriegeleitete Wissenschaft in Simulationswissenschaft überführt werden. Implementierungen wie Wolframs Programmiersprache Mathematica oder die künstliche Intelligenz seiner Suchmaschine Wolfram Alpha[80] basieren auf diesen Automaten. Dem damit einhergehenden wissenschaftstheoretischen und -politischen Anspruch ist im Zuge der Big-Data-Debatte häufig widersprochen worden [vgl. Mainzer/Chua 2012].

4.2.1.2 Game of Life

Für Wolframs zellulären Automat wurde 2004 Turingvollständigkeit bewiesen [vgl. Cook 2004], mit ihm lassen sich daher Turingmaschinen simulieren. Er ist „simulationally universal" im Sinne von Adamatzkys Definition. Für den zellulären Automaten „Game of

79 Es existieren allerdings auch derartige Anwendungen für Analogrechner, die diese jedoch implizit in diskret rechnende Maschinen transferieren [vgl. Ulmann 2010:287f.].

80 https://www.wolframalpha.com [letzter Abruf: 06.06.2019].

Life“, um den es im Folgenden geht, wurde dieser Beweis bereits 1974 erbracht [vgl. Wainwright 1974]. Conway entwickelte sein „Game of Life“ in den späten 1960er-Jahren auf Papier [vgl. Levy 1993:66; Gerhardt/Schuster 1995:33f.]. Vorgestellt wurde es vom Scientific-American-Autor Martin Gardener [1970] in dessen Kolumne „Mathematical Games“.

Der zelluläre Automat „Game of Life“ ist wie folgt beschreibbar:

1. Zwei Dimensionen: „‚LIFE‘ wird auf einem unendlichen Schachbrett gespielt.“ [ebd.]
2. Zwei Zustände: „Ein Spiel-Zustand ist dadurch gegeben, daß man sagt, welche Quadrate oder Zellen leben und welche tot sind.“ [ebd.]
3. Drei[81] Übergangsregeln:
    1. „Geburt: Eine zur Zeit t tote Zelle wird zum Zeitpunkt t+1 genau dann lebendig, wenn zur Zeit t genau 3 von ihren 8 Nachbarn[82] lebendig waren.“ [ebd.]
    2. „Tod durch Überbevölkerung: Eine Zelle, die zur Zeit t lebt, aber zugleich noch 4 oder mehr lebende Nachbarn hat, ist zur Zeit t+1 tot“ [ebd.]
    3. „Tod durch Einsamkeit: Eine Zelle, die zur Zeit t lebt, aber nur einen oder keinen lebendigen Nachbarn hat, ist zur zeit t+1 tot.“ [ebd.]
    4. „Überleben: Eine Zelle, die zur Zeit t lebt, ist auch zur Zeit t+1 am Leben, wenn sie 2 oder 3 zur Zeit t lebende Nachbarn hat.“ [ebd.]

Der „Game of Life“-Algorithmus kann danach notiert werden als:

$$z_{i,j}(t+1)= \begin{cases} 1, \text{wenn } \sum z_{kl}(t)=3 & \mid (k,l) \in N_{ij} \\ 1, \text{wenn } \sum z_{kl}(t)=4 \text{ und } z_{ij}(t)=1 & \mid (k,l) \in N_{ij} \\ 0, \text{sonst.}[83] \end{cases}$$

Auf Basis dieser Vorgaben kann das Spiel zwei mögliche Enden haben: Alle Zellen auf dem Raster sind tot (1) oder alle Zellen befinden sich in Stasis bzw. sich stetig wiederholenden Mustern (2). Dass am Schluss alle Zellen lebendig sind, kann aufgrund der zweiten Übergangsregel ausgeschlossen werden. Ob allerdings ein ‚Endzustand‘ (3) existiert, bei dem sich endlos neue Muster ergeben, konnte zunächst nicht bestätigt werden. Für den-

81 Da die vierte Regel implizit aus den ersten drei Regeln hervorgeht, muss sie eigentlich nicht eigens definiert werden.

82 Bereits die Beschreibung „Schachbrett“ kennzeichnet maximal 8 mögliche Nachbarn eines Feldes. Diese Nachbarschaft wird als „Moore-Nachbarschaft“ bezeichnet. Werden die dem bezogenen Feld diagonal benachbarten Felder nicht mit in die Nachbarschaft einbezogen, handelt es sich um eine „Von-Neumann-Nachbarschaft“ (hergeleitet aus dessen zellulärem Automat).

83 Vgl. [Gerhardt/Schuster 1995:32]. k und l sowie i und j sind die Koordinaten einer Zelle z(i,j) und ihrer Nachbarzellen z(kl) zum Zeitpunkt t.

jenigen, der den Beweis für diese dritte Möglichkeit erbringt, schrieb Conway ein Preisgeld aus [Levy 1993:75]. William Gosper, einer der ‚early hackers of MIT', gewann dieses Preisgeld durch die Konstruktion einer Zellstruktur, die stetig neue Zellen produziert, welche sich über das Raster bewegen [Levy 1984:120-144]. Diese nach ihm benannte „Gosper Glider Gun" [Gosper 1984] wurde zur Grundlage für die Entwicklung der Computer-Simulation [Conway u.a. 1985a:136ff.] mit „Game of Life", ermöglichte sie es doch diskrete Signalströme und damit logische Gatternetze auf dem Spielraster zu erzeugen.

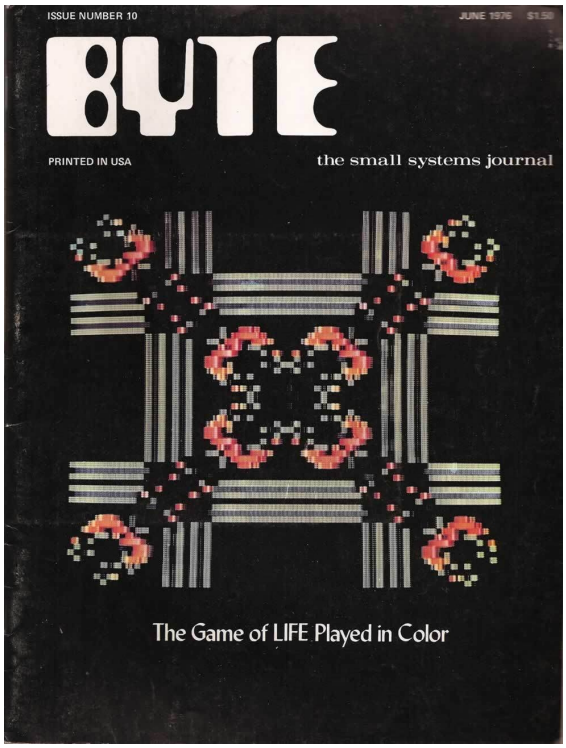


Abb. 4.2.3: Cover *Byte Magazine* (10/1976) mit „Game of Life"-Titel

Zunächst wegen dieser von Conway öffentlich ausgeschriebenen Aufgabe, später aber aufgrund seiner leichten Implementierbarkeit selbst mit einfachen Programmiersprachen (vgl. [Ahl 1982:103-105]) wurde das „Game of Life" in den 1970er- und -80er-Jahren sehr populär (Abb. 4.2.3). Es gab Implementierungen für zahlreiche Homecomputer und in unterschiedlichen Programmiersprachen und -dialekten. Über das Programm, seine Geschichte, Anwendungsmöglichkeiten und kulturelle Bedeutung sind etliche Publikationen erschienen. Insbesondere in Hacker-Szenen hat der Algorithmus eine intensive Auseinandersetzung und eine regelrechte ‚Zoologie' an verschiedenen Konstruktionen provoziert [vgl. Conway u.a. 1985a:125-133], mit deren Hilfe komplexe Konstruktionen von Logik-Gattern (vgl. [Rennard 2002]) bis hin zu funktionierenden Turing-Maschinen ([vgl. Rendell 2016:45ff.] sowie Abb. 4.2.4) und „Game of Life in Game of Life"-Implementierungen[84] (Abb. 4.2.4) entwickelt wurden. In der angewandten Informatik wird „Game of Life" (aber auch andere zelluläre Automaten) heute als unkonventionelles Rechnermodell erforscht (vgl. Kap. 4.2.5).

84 Vgl. https://www.youtube.com/watch?v=xP5-iIeKXE8 [letzter Abruf: 14.08.2018].

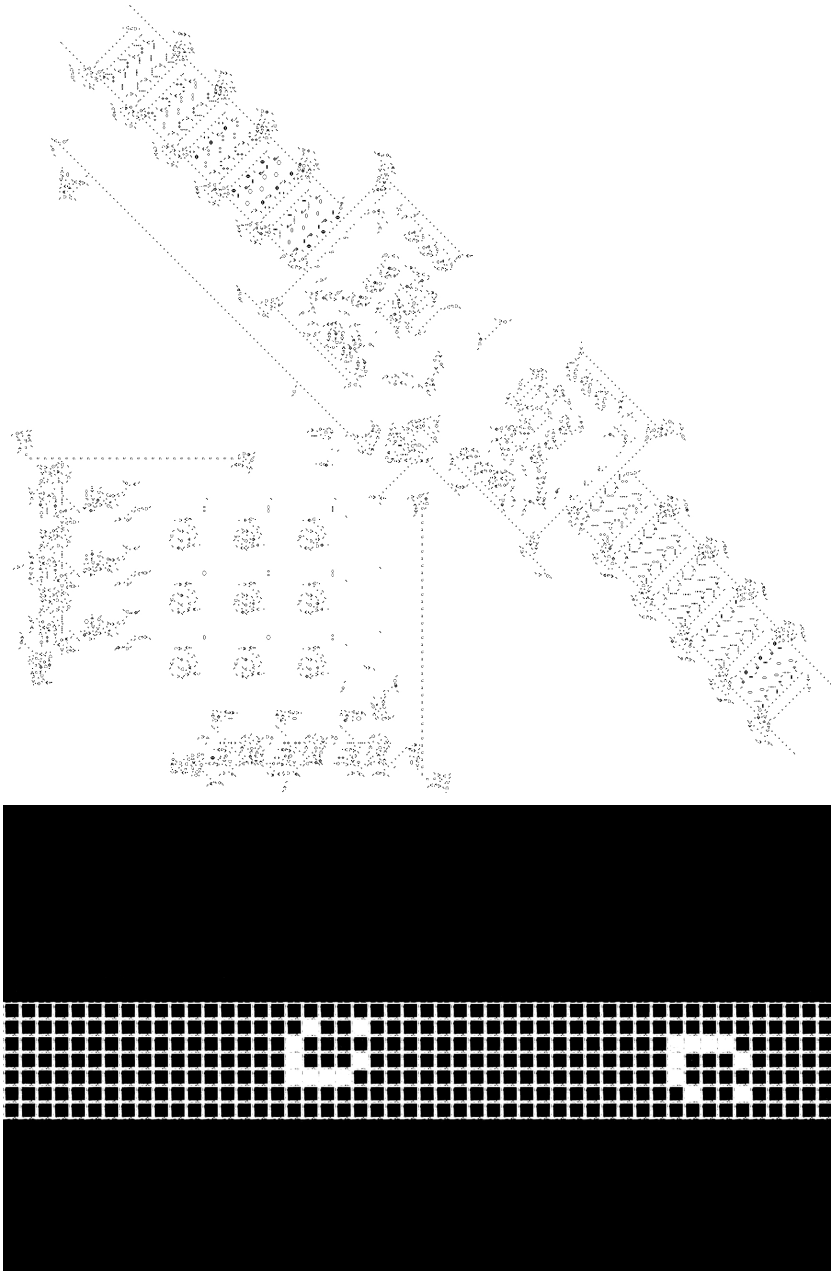


Abb. 4.2.4: Rendells Turing-Maschine (oben) und Bradburys „Life in Life“ (unten)

4.2.2 Games of Life and Death

„Game of Life“ wurde aus diesen Gründen als Algorithmus für die Implementierung einer diskreten Simulation im Rahmen eines Master-Seminars[85] in der Medienwissenschaft ausgewählt. Am Programm sollten dabei sowohl die Geschichte und Theorie von Simulation im Allgemeinen als auch die Rezeption von „Game of Life“ als ein Fall von Mediengeschichte im Besonderen diskutiert werden. Die Studieren erlernten hierzu während des

85 https://agnes.hu-berlin.de/lupo/rds?state=verpublish&status=init&vmfile=no&publishid=88822&moduleCall=webInfo&publishConfFile=webInfo&publishSubDir=veranstaltung [letzter Abruf: 16.08.2018]. Als Semesterarbeit zum Erwerb eines Leistungsscheins sollten eigene Zellulare Automaten in einer frei gewählten Programmiersprache entwickelt werden.

Semesters (flankierend zu einer textbasierten, inhaltlichen Diskussion) die Programmiersprache TrueBASIC, mit der sukzessive das folgende „Game of Life“-Programm implementiert wurde:

```
10 CLS: REM CLEAR SCREEN
20 DIM S(11,11):DIM A(11,11)
30 GOSUB 500
40 GOSUB 900
50 GOSUB 600
60 GOSUB 700
80 GOTO 40
500 REM SEED
510     FOR I=1 TO 25
520             X=INT(RND(1)*10)+1:Y=INT(RND(1)*10)+1
530             S(X,Y)=1:A(X,Y)=1
540     NEXT I
550 RETURN
600 REM ARBEIT2SPIEL
610     FOR Y=1 TO 10
620             FOR X=1 TO 10
630                     S(X,Y)=A(X,Y)
640             NEXT X
650     NEXT Y
700 REM SPIELREGEL ANWENDEN
710     FOR Y=1 TO 10
720             FOR X=1 TO 10
730                     Q=S(X,Y-1)+S(X-1,Y-1)+S(X-1,Y)+S(X-1,Y+1)+S(X,Y+1)+S(X+1,Y+1)+S(X+1,Y)
+S(X+1,Y-1)
740                     IF S(X,Y)=0 AND Q=0 THEN GOTO 860
750                     IF X=1 AND Y=1 THEN Q=S(2,1)+S(2,2)+S(1,2):GOTO 830
760                     IF X=1 AND Y=10 THEN Q=S(1,9)+S(2,9)+S(2,10):GOTO 830
770                     IF X=10 AND Y=1 THEN Q=S(9,1)+S(9,2)+S(10,2):GOTO 830
780                     IF X=1 AND Y=10 THEN Q=S(9,9)+S(10,9)+S(9,10):GOTO 830
790                     IF X=1 AND (Y>1 OR Y<9) THEN Q=S(1,Y-1)+S(2,Y-1)+S(2,Y)
+S(2,Y+1)+S(1,Y+1):GOTO 830
800                     IF X=10 AND (Y>1 OR Y<9) THEN Q=S(10,Y-1)+S(9,Y-1)+S(9,Y)
+S(9,Y+1)+S(10,Y+1):GOTO                        830
810                     IF Y=1 AND (X>1 OR X<9) THEN Q=S(X-1,1)+S(X-
1,2)+S(X,2)+S(X+1,2)+S(X+1,1):GOTO 830
820                     IF Y=10 AND (X>1 OR X<9) THEN Q=S(X-1,10)+S(X-
1,9)+S(X,9)+S(X+1,9)+S(X+1,10):GOTO                  830
830                     IF S(X,Y)=0 AND Q=3 THEN A(X,Y)=1
840                     IF S(X,Y)=1 AND Q<2 THEN A(X,Y)=0
850                     IF S(X,Y)=1 AND Q>3 THEN A(X,Y)=0
860             NEXT X
870     NEXT Y
880 RETURN
900 REM ANZEIGEN
910     LOCATE 1,1: REM HOME
920     FOR Y=1 TO 10
930             FOR X=1 TO 10
940                     IF S(X,Y)=1 THEN PRINT CHR$(42);
950                     IF S(X,Y)=0 THEN PRINT CHR$(32);
960             NEXT X
970             PRINT
980     NEXT Y
990 RETURN
```

Das Programm implementiert den „Game of Life“-Algorithmus auf einem zehn mal zehn Felder großen Raster mit ASCII-Zeichen (* als lebendige, das Leerzeichen als tote Zelle). Neben dem Spielfeldraster (das Array S(X,Y)) nutzt es ein Arbeitsfeldraster (im Array A(X,Y)). Die Spielregeln sind in den Zeilen 700-880 implementiert. Als Ausgangssituation werden pseudozufällig 25 der 100 Felder mit lebenden Zellen gefüllt (Zeile 500-550).

Neben informatischen Fragestellungen (Automatentheorie, Simulation) wurde auch das Thema der biologischen Terminologie erörtert: Konzepte wie „Leben“ [Conway u.a.

1985a] aus „Zellen“, die sich „evolutionär“ [Zhuang 2017:475f.] entwickeln, Algorithmen als „Gen-Technologie“ [ebd.:154], „Software der Natur“ [Gerhardt/Schuster 1995:16] usw. wurden im Seminar vor dem Hintergrund eines möglichen Biologismus, seiner Metaphorik [vgl. Rennard 2002:491] und der diskurshistorischen „Camouflage der Kybernetik“ [Hagen 2004] diskutiert. Einen roten Faden bildete allerdings die Fragestellung der Geschichte von Simulationen biologischer Prozesse durch Digitalcomputer.

Sie beginnt nicht erst bei John von Neumanns eigener Arbeit über zelluläre Automaten („Comparisons between computing machines and the nervous systems“ [von Neumann 1966:64]), sondern lässt sich bis zum *Draft* der nach ihm benannten Rechnerarchitektur zurück verfolgen: Dort entwirft er ein Digitalcomputersystem aus Komponenten, die er als „organs“ [vgl. von Neumann 1993:33-36] bezeichnet. Schon kurze Zeit später beschreibt Alan Turing erstmals das Prinzip der Morphogenese tierischer Zellstrukturen mittels zellulärer Automaten [Turing 1952]. In der Theorie und den Anwendungen [z.B. Schönfisch 1993] zellulärer Automaten im Allgemeinen sowie „Game of *Life*“ expressis verbis wird das Potenzial dieser Simulation für biologische und medizinische Prozesse weitergeführt.

Eine dritte theoretische Perspektive warf die Frage der Darstellbarkeit emergenter Computerprozesse auf. Die Abstraktion von der einzelnen Zelle und ihrer Umgebung hin zum größeren Kontext konnte dabei das simulative Potenzial des Zellulären Automaten vor Augen führen. Experimente mit verschiedenen Strukturen des Spiels zur Entwicklung komplexerer Elemente (z.B. logischer Gatter bis hin zu binären Halbaddierern, vgl. [Rennart 2002:408ff.]) zeigten den didaktischen und epistemologischen Mehrwert des Spiels. Im Rahmen eines spezifischen Projektes wurde „Game of Life“ überdies als Inhalt für eine technikhistorische Frage genommen: Wie lässt sich das Spiel auf einem technisch weitestmöglich reduzierten System implementieren? Als Bezugssystem wurde der Signetics Instructor 50 ausgewählt, dessen Programmierung von einigen Studierenden bereits im Rahmen eines Assembler-Projektseminars erlernt worden war.

4.2.2.1 Brainware, Paperware, Hardware(s) und/oder Software(s)

Zur Beantwortung der Frage wurden mehrere Etappen der Entwicklung als Re-Enactment historischer Programmiermethoden kleinschrittig nachvollzogen:

- Conways Spiel und seine eigene Erstimplementierung [Gerhardt/Schuster 1994:33f.] wurden mit Hilfe eines auf Papier aufgezeichnetes Rasters und Münzen als lebendige Zellen nachvollzogen, um der schnell anwachsenden Komplexität der Simulation gewahr zu werden.
- Das Spiel wurde dann in der Sprache TrueBASIC (siehe oben) implementiert. Die Grundlage bildete dafür der in [Gerhardt/Schuster 1995:17-32] beschriebene Algo-

rithmus. Die theoretischen und praktischen Erkenntnisse sollten danach in die Implementierung für das Minimalsystem einfließen.

- Hierzu wurde zunächst die Architektur des ausgewählten Mikroprozessors Signetics 2650, der Aufbau seines Befehlssatzes sowie seine Kommunikationsmöglichkeiten mit Peripheriegeräten anhand historischer Manuals [Signetics 1978a] und Lehrbücher [Glagla/Feiler 1984:307-356; Fischer 1982:174-176] studiert.
- Danach wurden verschiedene historische und zeitgenössische Implementierungen mit diesem Prozessor vorgestellt (vgl. Kap. 3.3). Der Fokus lag dabei auf dem Lerncomputer Signetics Instructor 50, dessen Aufbau und Eigenschaften mithilfe der Literatur [Signetics 1978b; Fischer 1982] und der darin enthaltenen Abtipp-Programme exploriert wurde.
- Mit Hilfe von Programming Sheets (vgl. Abb. 4.2.5) wurden dann erste Assembler-Programme von Hand auf Papier entworfen. Da der Instructor 50 nicht in Assembler programmierbar ist, sondern lediglich über ein Monitorprogramm verfügt, das Opcodes nur in hexadezimaler Form entgegennimmt, welche über die eingebaute Hex-Tastatur eingegeben werden müssen, oblag es den Programmierern, ihre Programme zunächst selbst mithilfe der genannten Literatur zu assemblieren. Diese sehr archaische Form der Programmentwicklung und -eingabe war bis zur Entwicklung erster Assemblersprachen und Assemblierer die Standard-Vorgehensweise bei der Programmierung. Der didaktische Mehrwert dieser Form der Programmierung ergibt sich daraus, dass alle Register-Bezüge und alle Adressierungsarten in die jeweiligen Opcodes manuell enkodiert werden müssen. Damit offenbaren sich mnemonische Befehle als veritable Software-Blackboxes.[86] Bei der manuellen Assemblierung, während der die binär anzugebenden Opcodes durch Register- und Adressierungsart-Bits ergänzt werden müssen, werden hingegen viele Funktionsweisen des Mikroprogramms nachvollziehbar.

Einige Teilnehmer des Kurses hatten in den Semestern zuvor Projektkurse zur Programmierung in Signetics-2650-Assembler besucht. Im Wintersemester 2013/14[87] wurde hierfür der Signetics Instructor 50 als Lehrplattform für diesen CPU und ihre Maschinensprache eingesetzt. In Rahmen dieses Projektseminars fanden auch erste Experimente mit zellulären Automaten statt, wovon einer im Folgenden vorgestellt wird.

86 Diesem Umstand ist es beispielsweise zuzurechnen, dass für den MOV-Befehl der x86-Architekturen Turingvollständigkeit bewiesen werden konnte [vgl. Dolan 2013]. Tatsächlich gibt es gar nicht einen MOV-Befehl, sondern das Mnemonic MOV steht für zahlreiche Lade- und Speicher-Opcodes mit unterschiedlichen Quellen, Senken und Adressierungsarten, die teilweise ganz unterschiedliche Mikroprogramm-Bestandteile integrieren.

87 https://agnes.hu-berlin.de/lupo/rds?state=verpublish&status=init&vmfile=no&publishid=76427&moduleCall=webInfo&publishConfFile=webInfo&publishSubDir=veranstaltung [letzter Abruf: 21.08.2018].

SEED
(0120-013F)

ADRESSE	HEX-BEFEHL 1	2	3	4	SYMBOLISCHER MARKE	OPCODE	OPERAND	KOMMENTARE
0120	75	02			Seed	CPSL	02	COM auf arithm. Schalter
0122	06	FF				LODI,R2	FF	R2 = FF
0124	05	FF				LODI,R1	FF	Beginne Schleife (255 x)
0126	54	73			LOOP	REDE,R0	73	Zufallswert holen
0128	D0					RRL,R0		und rechts rotieren
0129	35	04				TPSL	04	teste Vorzeichen-Bit
012B	9C	01	31			BCFA	NOERR	Bei 0 springen
012E	57	73				REDE,R3	73	weiteren Zufallswert holen
0130	22					EORZ,R3	~~22~~	R0 ⊻ R3 → R0
0131	C3				NOERR	STRZ,R3		Zufallszahl in R3
0132	04	15				LODI,R0	15	"0" ...
0134	CF	6E	00			STRA,R3	0900	... in Feld schreiben
0137	FD	01	28			BDRA,R1	LOOP	Schleifenende
013A	FE	01	24			BCTA,UN	0124	wiederhole Seed
013D	1F	17	86			BCTA,UN	INIT	zurück und Init

Abb. 4.2.5: Manuell erstelltes Programming Sheet für 2650-Assembler mit SEED-Routine aus Game of Memories

4.2.2.2 Game of Death

Der damalige Student Thomas Nückel entwickelte als Projektarbeit das Programm Game of Death.[88] Dabei handelt es sich um einen eindimensionalen zellulären Automaten, dessen Ausgaben auf den acht Leuchtdioden des Instructor 50 stattfinden. Die Zellen von Game of Death verfügen über zwei Zustände: ‚tot' (Leuchtdiode abgeschaltet) und ‚lebendig' (Leuchtdiode eingeschaltet). Der Regelsatz lautet wie folgt:

1. „Spontane Entstehung: Sind vier zusammenhängende Zellen leer, entsteht an der zweiten Position von rechts eine neue lebende Zelle."
2. „Zellteilung: Hat eine lebende Zelle keine Nachbarn, so teilt sich sich. Besonderheit: Hier affizieren nicht nur die Nachbarfelder die bearbeitete Zelle, sondern auch die aktive Zelle ihre Nachbarfelder."
3. „Erstickungstod: Hat eine lebende Zelle auf jeder Seite einen Nachbarn, so stirbt sie."

Der Anfangszustand des Automaten wird mit den Kippschaltern des Computers als 8 Bit große Binärzahl eingegeben. Über die Intention hinter der Programmentwicklung schreibt der Autor:

88 https://www.musikundmedien.hu-berlin.de/de/medienwissenschaft/medientheorien/signallabor/game_of_death.zip [letzter Abruf: 21.08.2018]. Die zip-Datei enthält den Sourcecode des Programms, ein Programm-Image für den Emulator WinArcadia, eine Präsentation und eine kurze Programmbeschreibung als PDF. Letzteren beiden sind die folgenden Zitate entnommen.

„Um die Regeln zu entwickeln, wurden zuvor Tests auf dem Papier durchgeführt. Da die Zellen bei den meisten Versuchen in kurzer Zeit ausstarben, blieb der daraus resultierende Name bis in die finale Version haften: Game of Death.

Die Regeln hatten einigen Kriterien genügen. Sie sollten

a) optisch den Eindruck erwecken, es mit Wachsen und Vergehen zu tun zu haben,

b) geringe Tendenz aufweisen in kurze Endlosschleifen oder Stillstand zu verfallen und

c) eine gewisse Ästhetik an den Tag legen."

Das Programm ist 146 Byte groß und verwendet 3 Byte des RAM-Speichers als Scratchpad sowie zur Sicherung der Zustände des Automaten. Es ließ sich damit vollständig im RAM des Signetics Instructor 50 unterbringen und darauf ausführen.

Thomas Nückel hat, aufbauend auf den Erfahrungen, die er mit der Entwicklung seines Game of Death gesammelt hatte, weitere zelluläre Automaten programmiert. Einer davon war eine Weiterentwicklung der vom Seminarleiter in einer unfertigen Version zur Diskussion gestellten „Game of Life" für den Signetics Instructor 50. Dieses wurde unter dem Titel Game of Memories[89] von Nückel fertiggestellt und wird im folgenden diskutiert.

4.2.3 Game of Memories

Hier werden ausgewählte Elemente des Sourcecodes (vgl. Anhang C) vorgestellt, an denen sich insbesondere Spezifika des bezogenen Systems, computerhistorische Aspekte und Kodierungsprobleme markieren lassen.

4.2.3.1 Der Code

Der Programmcode umfasst 510 Zeilen (inklusive Kommentarzeilen) und belegt 644 Bytes RAM. Die Struktur stellt sich wie folgt dar:

89 Der Sourcecode, eine Image-Datei für den WinArcadia-Emulator sowie ein Kommentar finden sich https://www.musikundmedien.hu-berlin.de/de/medienwissenschaft/medientheorien/signallabor/prasixarbeiten [letzter Abruf: 16.08.2018]. Das Programmlisting mit >500 Zeilen zu umfangreich, um es hier vollständig zu publizieren. Anstelle dessen wird auf die entsprechende Zeile im Sourcecode-Datei (Anhang C) verwiesen.

```
; Programmstruktur (mit Labeln und Adressen):
;
; INIT (1780h): Initialisierung von Spiel- & Arbeitsfeld
;       CPA2B   (178Bh): Kopieren ...
;       17PROOF (179Dh): ... des Spielfeldes ...
;       A2BEND  (17A2h): ... ins Arbeitsfeld.
; MAIN (17A5h): Hauptschleife
;       ANAANF (17ACh): Analyse des Spielfeldes
;               SOND (0180h): Test der Spielfeldecken und -ränder
;                       LOE (02D0h): Sonderfall: Linke obere Ecke
;                       ROE (02F0h): Sonderfall: Rechte obere Ecke
;                       LUE (0310h): Sonderfall: Linke untere Ecke
;                       RUE (0330h): Sonderfall: Rechte untere Ecke
;                       RO  (0350h): Sonderfall: Oberer Rand
;                       RU  (0380h): Sonderfall: Unterer Rand
;                       RR  (03E0h): Sonderfall: Rechter Rand
;                       LR  (03B0h): Sonderfall: Linker Rand
;               NORM (0260h): Adress-Arithmetik (Analyse der jeweiligen Nachbarzellen)
;       REG (0410h): Regeln anwenden
;               REG1 (0413h): Regel 1
;               REG2 (0425h): Regel 2
;               REG3 (0432h): Regel 3
;       REGEND (043C): Ende der Regelprüfung
; LOOPEND (043Fh): Ende der Hauptschleife
; Subroutinen:
;       ADDFLD (02B0h): Subroutine: Addition der Umfelder
;       CLRART (0150h): Subroutine: Löschen des Arithmetikfeldes
;       CLRFLD (0102h): Löschen von Arbeits- und Spielfeld.
;       SEED   (0120h): Spielfeld ...
;               LOOP  (0126h): ... zufällig ...
;               NOERR (0131h): ... mit Zellen füllen.
```

[Höltgen/Nückel 2015:Z. 37-67]

Eine Besonderheit dieses „Game of Life“-Programms ist, dass es über keinerlei Ausgabe des Spielfeldes verfügt. Sämtliche Operationen finden ‚unsichtbar‘ im Speicher des Signetics Instructor 50 statt: Das Spielfeld residiert dort im Adressbereich 0000_{16}-$00FF_{16}$, das Arbeitsfeld unter $0E00_{16}$-$0EFF_{16}$. Weitere 9 Bytes RAM ($0F00_{16}$-$0F08_{16}$, vgl. Höltgen/Nückel 2015:Z. 476ff.]) werden als Zwischenspeicher für arithmetische Operationen verwendet. Um dennoch eine Kontrollmöglichkeit über den Ablauf des Programms zu haben, werden die acht Leuchtdioden der Binäranzeige des Systems als Fortschrittsanzeige genutzt, deren Stand an Adresse $0F09_{16}$-$0F0A_{16}$ hinterlegt wird:

```
[Adr.   Label          Mnemonics/Argumente    Opcodes/Argumente             Kommentare]
17AC    ANAANF  WRTD,R0                F1                              ; Analysefortschritt auf
LEDs
        ausgeben
17AD                   BCTA,UN SOND           1F 01 7E                      ; Springe zur
Analyse der Felder
```

[Höltgen/Nückel 2015:Z. 107f.]

Diese binäre Repräsentation des Spielfortschritts orientiert sich an einer historischen Vorlage: In frühen Digitalcomputern wurden Lautsprecher verbaut, die mit einer Datenleitung des Prozessors verbunden waren [vgl. Levy 1984:20f.]. Beim Ablauf von Programmen kam dadurch ein stetig wechselnder Ton zur Ausgabe, der den Programmierer darüber informierte, ob das Programm noch lief, beendet war oder an einer Stelle ‚hing‘ (abgestürzt war). Letzterer Fall wurde dadurch erkennbar, dass über den Lautsprecher repetitive (rhythmische) Tonfolgen ausgeben wurden [Miyazaki 2013:78ff.]. In Ermangelung eines Lautsprechers wurde im vorliegenden Programm daher die LED-Anzeige als Kontrollleuchte verwendet, deren pulsierende ‚Lichterkette‘ über den Fortschritt des

Programms informiert. Um dennoch auch dieser computerhistorischen Spur zu folgenden implementierten die damaligen Studenten Thomas Nückel und Christoph Borbach eine Variante von Game of Memories mit akustischen Ausgaben – allerdings in einer anderen Programmiersprache und für ein anderes System [Nückel/Borbach 2016].

Bei Game of Death und moderneren Versionen vom „Game of Life“[90] kann eine Anzahl von Zellen bei Programmstart als ‚tot‘ oder ‚lebendig‘ definiert werden. Damit wird es erst möglich solche zellulären Automaten für gezielte Experimente zu verwenden. In Ermangelung an einer hierfür notwendigen intuitiven Eingabeschnittstelle, wurde in Game of Memories nach dem Programmstart eine zufällige Besetzung des Spielfeldes mit ‚lebenden‘ Zellen erzeugt:

```
0120    SEED          CPSL 02        75 02                    ; COM arithmetisch
0122                  LODI,R2 A0             06 FF                    ; R2=FF
0124                  LODI,R1 FF             05 FF                    ; Beginn Schleife
(255 mal)
0126    LOOP          REDE,R0 73             54 73                    ; Zufallswert holen
0128                  RRL,R0         D0                       ; nach rechts rotieren
0129                  TPSL 04        B5 04                    ; testet Vorzeichenbit
012B                  BCFA NOERR             9C 01 31                 ; Bei 0 springen
012E                  REDE,R3 73             57 73                    ; noch einen
Zufallswert holen
0130                  EORZ,R3        22                       ; R0 XOR R3 -> R0
0131    NOERR         STRZ,R3        C3                       ; Zufallszahl in R3
ablegen
0132                  LODI,R0 15             04 15                    ; "o" ...
0134                  STRA,R3 0900   CF 6E 00                 ; ... in Feld schreiben
0137                  BDRA,R1 LOOP   FD 01 28                 ; Schleifenende
013A                  BCTA,UN 0124   FE 01 24                 ; Wiederhole Seed
013D                  BCTA,UN INIT   1F 17 86                 ; zurück nach INIT
```

[Höltgen/Nückel 2015:Z. 459-473, vgl. Abb. 4.2.5]

Aus dem Inhalt der Adresse 73_{16} wird hierzu ein Zufallswert ermittelt. In dieser Adresse befindet sich eine Kopie der unteren 8 Bit des Clock-Cycle Counters des Systems, das mit dem Start des Rechners von 0 ab aufwärts zählt. In der obigen Programm-Routine wird diese Adresse zwei mal ausgelesen. Die ausgelesenen Zahlen werden auf Binärebene durch Exklusiv-Oder verknüpft. Hieraus entsteht eine Zahl zwischen 0 und 255 (FF_{16}), die als Zellenadresse der Speicher-Seite (siehe unten) genutzt wird, um eine ‚lebende‘ Zelle darin zu generieren. Bei diesem Zufallswert handelt es sich um eine Pseudozufallszahl [Kneusel 2018:50f.], die aufgrund ihrer Generierung leicht rückrechenbar wäre.[91] Dennoch bekommt Game of Memories damit Aspekte einer stochastischen Simulation.

Abschließend soll noch ein Blick auf das Spielfeld-Raster geworfen werden. Auch hierin unterscheidet sich Game of Memories notwendigerweise von hochsprachlichen Implementierungen (z.B. [Höltgen 2017a:27] bzw. Kap. 4.2.2), die für Spiel- und Arbeitsfeld je einen zweidimensionalen Daten-Array anlegen (je nach vorhandenen Datentypen sogar für

90 Zum Beispiel das Programm Golly (http://golly.sourceforge.net/ [letzter Abruf: 16.08.2018]).

91 Da das Monitorprogramm des Instructor 50 (und auch dessen Emulator) den nach dem Einschalten zufällig gefüllten SRAM-Inhalt mit 0 initialisiert, konnte dieser, anders als bei anderen 2650-Systemen, nicht genutzt werden (vgl. Kap. 4.3.2).

Boole'sche Werte, bei denen 0 für ‚tot' und 1 für ‚lebendig' stehen kann). Im 2650-Assembler existiert keine Möglichkeit mathematische Matrizen als einen Datentyp ‚Array' zu definieren. Hierfür können allerdings Speicher-Seiten (*pages*) genutzt werden.

Die page eines 8-Bit-Systems besteht für gewöhnlich aus 256 Speicherzellen. Der RAM-Speicher kann auf diese Weise vorteilhaft hexadezimal in page-basierte Adressbereiche unterteilt werden (vgl. Abb. 4.2.6): $nn00_{16}$-$nnFF_{16}$ (wobei nn die page-Nummern 00_{16}-$7F_{16}$ enthalten). Bei Aufteilung einer Page in 16 mal 16 Felder ergibt sich ein weiterer Adressierungsvorteil (vgl. Tabelle 4.2.2). Die Speicherzellen lassen sich pro Spalte mit der ersten und pro Zeile mit der zweiten Hex-Ziffer adressieren (in der Tabelle ist die Zelle B6 mit „X" markiert). Die elektronische Speicherorganisation verfährt ebenso: RAM-Speicher wird mit Hilfe von Multiplexer-Bausteinen adressiert und die Adressen in Zeilen und Spalten kodiert. Mit dieser Technologie lassen sich auch größere Adress-Pages und mehr Speicher adressieren. Die Verwaltung dieser Speicher geschieht in der Randelektronik der Speicherkarten [vgl. Völz 2017:227-231].

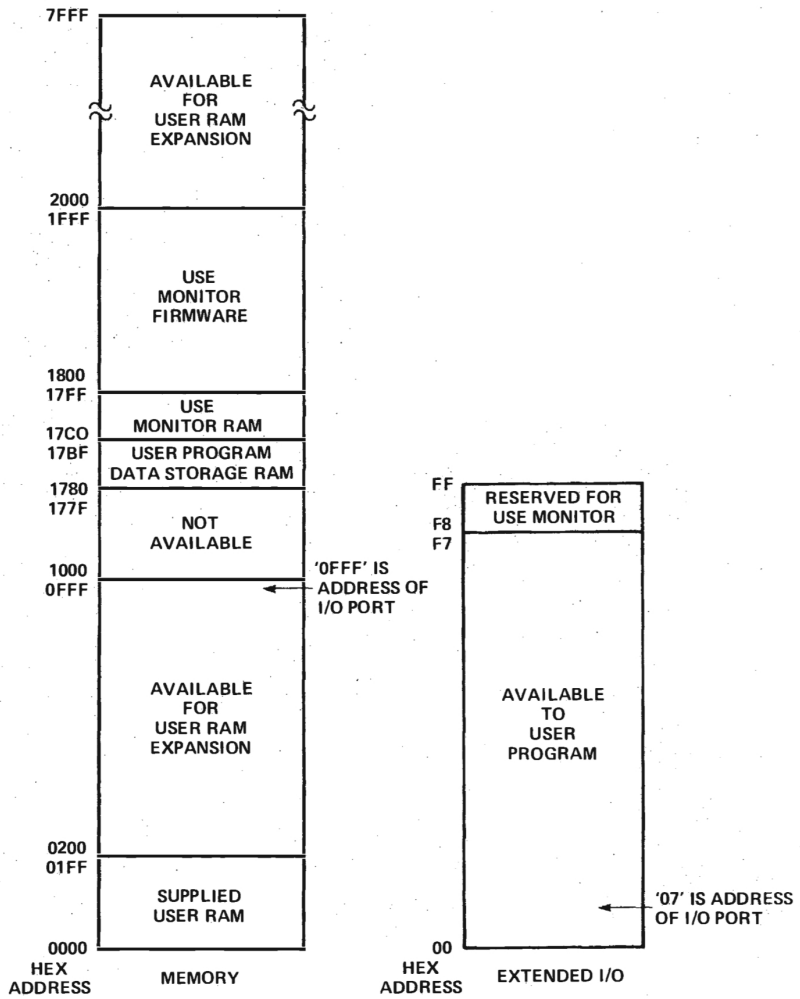


Abb. 4.2.6: Memory Map des Signetics Instructor 50 [Signetics 1978b:3-8]. Der vom Programmierer nutzbare Speicher liegt in den Pages 0 (0000_{16}-$00FF_{16}$) und 1 (0100_{16}-$01FF_{16}$) sowie in Teilen der Page 23 (1700_{16}-$17FF_{16}$).

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0																
1																
2																
3																
4																
5																
6																
7																
8																
9																
A																
B							X									
C																
D																
E																
F																

Tabelle 4.2.2: Darstellung einer Speicher-Page als 16×16-Raster mit markierter Zelle $B6_{16}$ und ihrer Nachbarzellen.

Computerspiel-Programmierern [vgl. Höltgen 2015b:120-125] kommt eine solche zweidimensionale Deklaration des Speichers entgegen. Mit ihr wird die bereits in der Schachprogrammierung verwendete Technologie des Arbeitens mit „Bit Boards“ [Reinefeld 2006:265][92] einsetzbar: Speichermatrizen repräsentieren hierbei Spielfeldraster; jedes Spielfeld entspricht einer Speicherzelle und kann bis zu 256 verschiedene Zustände (Spielfiguren, Werte etc.) speichern. Dieses Prinzip wird (notwendigerweise) auch beim Spiel Game of Memories genutzt und damit eine historische Technologie der Spielentwicklung nachvollzogen. Dass der Computer das Spiel allein im Speicher zwischen zwei pages (Arbeitsfeld und Spielfeld) spielt, ist einer der Gründe für dessen Titel Game of Memories.

Um den Inhalt eines durch eine Speicherzelle („X“) repräsentierten Spielfeldes unter Berücksichtigung seiner Nachbarfelder zu bearbeiten, muss eine spezifische Adressarithmetik verwendet werden:

92 https://chessprogramming.wikispaces.com/Bitboards [letzter Abruf: 16.08.2016].

```
; ADRESS-ARITHMETIK:
;
; Die arithmetischen Beziehungen zwischen der gerade untersuchten Zelle X und ihren
Nachbarzellen
; stellen sich wie folgt dar (innerhalb der Nachbarzellen ist der relative Abstand zu X
angegeben;
; außerhalb der relative Abstand der Nachbarzellen zueinander):
;
;             +1/1h  +1/1h
;             ----->  ---->
;        +------+------+------+
;        | -17/ | -16/ | -15/ |
; -16/ ^ | -11h | -10h | -Fh  | | +16/
; -10h | +------+------+------+ | +10h
;      | | -1/  |  X   |  +1/ | v
; -16/ ^ | -1h  |      |  +1h | | +16/
; -10h | +------+------+------+ | +10h
;      | | +15/ | +16/ | +17/ | v
;        | +Fh  | +10h | +11h |
;        +------+------+------+
;            <----   <----
;            -1/-1h  -1/-1h
;
; Für die Sonderfälle der Eck- und Randzellen gelten die selben Abstände, nur dass diese
; entsprechend weniger Nachbarn aufweisen. Im Falle der vier Eckzellen jeweils drei Nachbarn,
im
; Falle der 56 Randzellen fünf Nachbarn und acht bei den normalen Zellen im Spielfeld.
```

[Höltgen/Nückel 2015:Z. 159-178]

Die horizontalen und vertikalen Pfeile in diesem Kommentar zeigen die Richtung an, in der die Nachbarfelder des mit „X“ markierten Zentralfeldes untersucht werden. Hierbei muss lediglich die Adresse dieses Zentralfeldes als Summand verwendet werden, der mit der relativen Distanz zum jeweiligen Nachbarfeld (als zweiter Summand) addiert wird. Auf diese Weise lässt sich in acht Schritten (bei Feldern am Spielfeldrand entsprechend weniger) die für die Regelanwendung entscheidende Anzahl ‚lebender‘ Nachbarzellen ermitteln. Das Ergebnis dieser Prüfung wird auf einem zweiten Bit Board (dem Arbeitsfeld) gespeichert.

In dieser Vorgehensweise liegt ein zweiter Grund für die Betitelung des Spiels. In Code heißt es dazu:

```
; Dieses Programm ist ein Game of Life nach dem Regelsatz von John Horton Conway.
; Es wurde für den Emulator des Signetics Instructor 50 geschrieben. Die dabei
; verwendete Simulationssoftware ist Winarcadia 22.74. Da der Instructor 50 nur über
; eine rudimentäre Ausgabe verfügt, wurde das Spielfeld für das Game of Life
; in einen 16x16 Byte großen Speicherbereich gelegt, den der Emulator graphisch
; darstellen und so dem Betrachter zugänglich machen kann - wodurch sich der Name des
; Programms erklärt: Game of Memories.
```

[Höltgen/Nückel 2015:Z. 7-13]

4.2.3.2 Emulation of Memories

Aus mehreren Gründen wurde das Spiel nicht auf und für der Originalplattform Instructor 50 implementiert:

1. Der Spielverlauf ist dort nicht (oder nur indirekt über die Fortschrittsanzeige) sichtbar.

2. Der Rechner ist umständlich zu programmieren: Die spezifischen Idiosynkrasien des verwendeten (historischen) Lerncomputers, insbesondere seine prellende Tastatur und die notwendige Eingabe des Programms als hexadezimale Zahlenkolonne (vgl. Abb. 4.1.7) erwiesen sich als sehr programmierfehleranfällig.
3. Die stark reduzierten Möglichkeiten des Gerätes (vgl. Kap. 4.1.5.2) lassen es nicht zu ein Programm derartiger Größe (644 Bytes) und Speicheranforderungen (523 Bytes für Arbeits-, Spielfeld, Arithmetik-Zwischenspeicher und Spielstand-Speicher) darauf zu implementieren.
4. Im Signetics Instructor 50 hätte eine Zufallszahl nicht aus der Adresse 73_{16} bezogen werden können, denn das System ist nicht in der Lage zur Laufzeit seinen eigenen Clock-Cycle Counter zu notieren. (Die Zufallszahlenfunktion wurde auf den Wunsch der Kursteilnehmer vom Entwickler als Zusatz-Feature seinem Emulator hinzugefügt. vgl. Kap. 4.3.2.)

Daher wurde für die Programmentwicklung auf das Verfahren des „Cross Platform Development“ ausgewichen, wofür ein moderner PC[93], der Emulator WinArcadia in der Version 22.74[94] und zum Assemblieren des Sourcecodes der Crossassembler VACS[95] verwendet wurde.

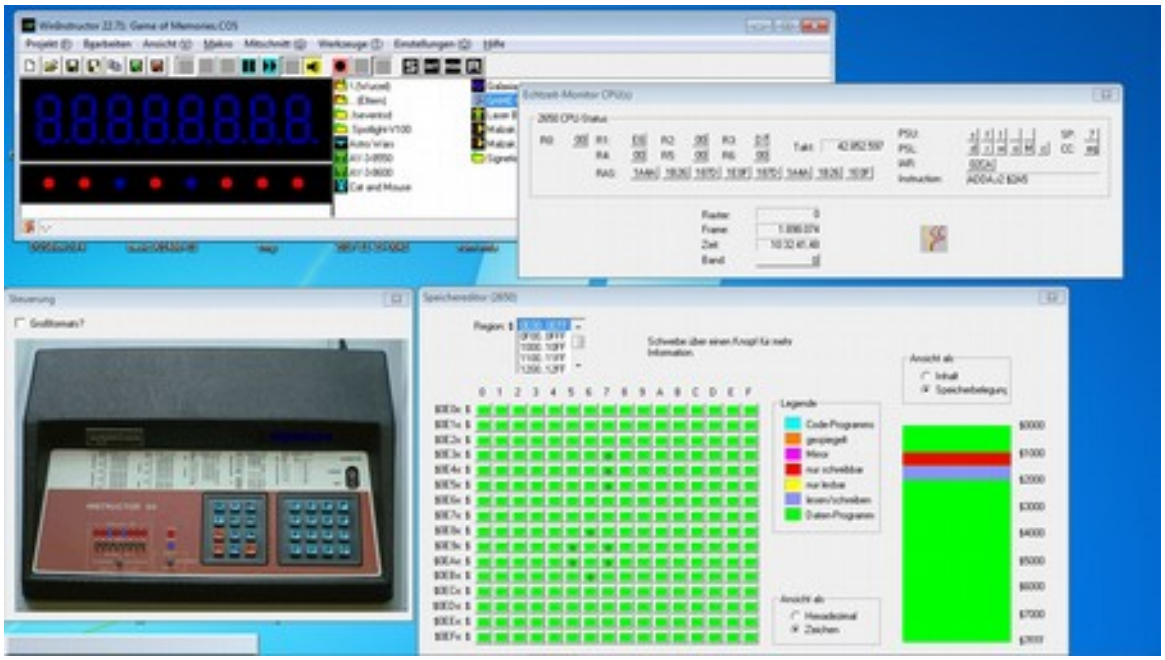


Abb. 4.2.7: WinArcadia mit geladenem Instructor-50-Emulator und laufendem Programm Game of Memories

93 Samsung Netbook Modell NF310, Intel Atom N550, Intel Graphics Media Accelerator 3150, 2 GB RAM, Windows 7.

94 http://amigan.1emu.net/releases/ [letzter Abruf: 16.08.2018] Der Emulator wurde u.a. im Zuge unserer Nutzung und auf Basis von Hinweisen der Seminarteilnehmer fehlerkorrigiert und erweitert (vgl. https://www.facebook.com/groups/392123604238327/ [letzter Abruf: 16.08.2018].

95 https://amigan.yatho.com/vacs124h.zip [letzter Abruf: 16.08.2018].

Abb. 4.2.8: Präparierte Netbook-Tastatur

WinArcadia ist ein so genannter *Multi-Plattform-Emulator*, welcher zahlreiche Systeme, die auf dem Prozessor Signetics 2650 basieren, emuliert. Die Instructor-50-Emulation enthält neben dem Hauptfenster (Abb. 4.2.7, links oben) ein interaktives Foto des Rechners (Abb. 4.2.7, links unten), auf dem die Tasten mit der Maus anklickbar sind und dessen Anzeigen je nach Programmausgabe wechseln. Sämtliche Tasten des emulierten System lassen sich überdies auf die Tastatur des Hostsystems mappen. Hierfür wurden kleine Aufkleber auf die Tastatur der Netbooks geklebt (vgl. Abb. 4.2.8). Neben der Nachahmung der Ein- und Ausgaben des Originalsystems verfügt der Emulator über Features, mit denen Prozessor-Zustände und Register-Inhalte in Echtzeit angezeigt werden können (Abb. 4.2.7, rechts oben) sowie einen Speichereditor (Abb. 4.2.7, rechts unten), der den Inhalt jeweils einer page und die gesamte Speicherbelegung des Systems anzeigt.

Letzterer macht die echtzeitige Anzeige und Veränderung von Speicherinhalten möglich. Gezeigt wird hierbei jeweils eine page des Speichers. Der Inhalt kann wahlweise nummerisch („Hexadezimal“) oder als „Zeichen“ angezeigt/eingegeben werden. Letztere Option wurde für Game of Memories genutzt und das Zeichen „o“ (Wert 15_{16}) für eine ‚lebendige‘ Zelle und ein Leerzeichen (Wert 17_{16}) für eine ‚tote‘ Zelle gewählt [vgl. Höltgen/Nückel 2015:Z. 496ff.]. In der Abb. 4.2.7 ist eine Momentaufnahme am Ende (Option 2: Stasis, vgl. Kap. 4.2.1.2) des Spiels zu sehen. Die Zellen (hier des Arbeitsfeldes bei $0E00_{16}$-$0EFF_{16}$) sind bis auf wenige ‚tot‘.

Die Implementierung des Programms war allerdings von vornherein nicht für die Original-Plattform Instructor 50 geplant gewesen: Das System verfügt über zu wenig RAM-Speicher, um den Anforderungen (1167 Bytes) zu genügen. Die im Programm angegebenen Bereiche für die Daten adressieren eine (mögliche, vgl. Abb. 4.2.6) Speichererweiterung, die im vorliegenden Original-System nicht implementiert war. Wie die Abbildung des Speichereditors zeigt, unterstützt der Emulator bis zu 32 Kilobyte RAM. Diese wurden für das Game of Memories genutzt, denn ein Ziel der Implementierung war es mit dem Emu-

lator ein Experiment durchzuführen, für das die Originalhardware nicht geeignet gewesen wäre.

4.2.4 Pixelspiele

Wie bereits die Programmierung des IBM-704-Ballspiels auf dem Instructor 50 (vgl. Kap. 4.1.2) gezeigt hatte, können historische Mikrocomputer aufgrund ihrer oft ähnlichen Architektur gut für Re-Enactments technikhistorischer Prozesse genutzt werden. Dass beispielsweise zahlreiche Einplatinencomputer der 1970er- und -80er-Jahre sowie die ersten Homecomputer dieser Zeit nur Binär- oder Hex-Tastaturen zur Eingabe besitzen und den Programmierer damit zum händischen Assemblieren nötigen, deutet an, dass sich mit der Erfindung des Mikrocomputers zunächst noch einmal die Frühgeschichte des Digitalcomputers selbst wiederholt hat. Diese Analogie zeigt sich in einigen technischen Details früher Mikrocomputer (vgl. Kap. 5.2.3).

Der Praxis des computerhistorischen Re-Enactments sah sich das Assembler-Programmierprojekt verpflichtet. Die Leuchtdiodenreihe gemahnte dabei ebenso an historische Vorbilder der Mainframe- und MinicomputerComputer-Ära, wie die optionale bitweise Programmierung über Kippschalter). Erst mit dem Emulator des Systems konnte allerdings ein weiteres Historem der Computergeschichte vor Augen geführt werden: Die Entwicklung der Pixelgrafik aus der Speichertechnologie.

In den 1940er-Jahren konstruierten die Ingenieure Frederic C. Williams und Tom Kilburn einen Speichertypus auf Basis der Röhrentechnologie. Im Unterschied zu Flip-Flops aus Elektronenröhren-Verschaltungen, verwendeten sie eine Kathodenstrahlröhre als Verzögerungsspeicher (delay storage), welche durch eine besondere Vorrichtung als Arbeitsspeicher nutzbar wurde: Vor den Bildschirm platzieren sie eine „pick-up plate“ [Williams 1960:3] als photoelektrischen Sensor. Sobald ein Kathodenstrahl das Innere der Mattscheibe erreichte und es zu einem Lichtblitz kam, wurde dieser von der pick-up plate registriert (gemessen) und als Bit-Impuls entweder zum Prozessor des Systems weiter- und/oder wieder in das Röhrenspeicher-System zurückgeleitet, um dort gespeichert zu bleiben. Der Nachleuchteffekt des Bildschirms sowie der regelmäßig notwendige ‚refresh‘ erinnern an den sukzessiven Spannungsverlust und das dadurch notwendige wiederholte Aufladen der Kondensatoren in modernen DRAM-Speicherbausteinen. Der Bit-Zustand „1“ erfordert in beiden Fällen ein definiert starkes Signal [vgl. Berz 2009].

Diese nach ihren Erfindern Williams-Kilburn-Tube benannte Technologie wurde in verschiedenen zeitgenössischen Computern eingesetzt – unter anderem 1950 im AN/FSQ-7 (der zahlreiche Innovationen in die Computertechnologie einführte [vgl. Ulmann 2014:36-39]), 1948 im Manchester Mark I und im selben Jahr im EDSAC (Electronic *Delay Storage* Automatic Calculator). Bei letzterem führte sie zu einer unerwarteten Innovation: Die Bits in der Williams-Kilburn-Tube wurde als Leuchtpunkte gespeichert, die für den Anwender jedoch wegen der „pick-up plate“ unsichtbar blieben. Beim Entfernen der

Platte zeigten sie sich als Punktraster auf dem Bildschirm (vgl. Abb. 4.2.9). Zu diesem Zweck wurde die Speichermatrix auf einer zweiten Kathodenstrahlröhre ‚gespiegelt' und sichtbar gemacht. Diese Speicherdarstellung brachte 1952 den Ingenieur Alexander Douglas auf die Idee, das Spiel OXO (Tic-Tac-Toe) auf dem EDSAC zu programmieren. Die Ausgaben des Spiels sollten dabei auf dem Bildschirm der verwendeten Williams-Kilburn-Tube angezeigt werden (Abb. 4.2.9). Dieser ‚Hack' stellt nicht nur ein Initialelement in der Geschichte der Computerspiele[96], sondern auch der Pixelgrafik auf CRT-Monitoren dar [vgl. Smith 2016; Link 2006].

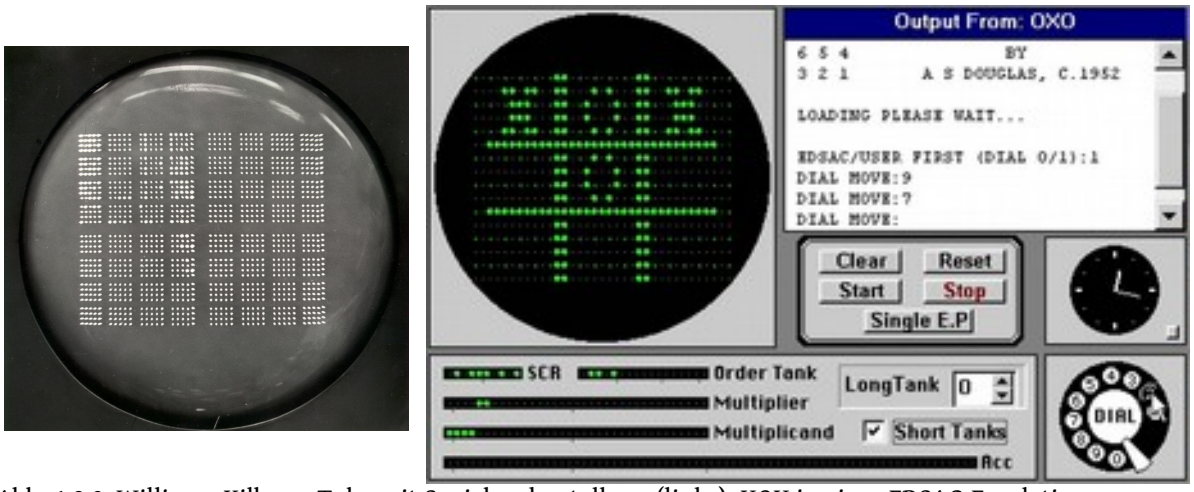


Abb. 4.2.9: Williams-Kilburn-Tube mit Speicherdarstellung (links), XOX in einer EDSAC-Emulation

Das OXO-Spiel wurde bereits mehrfach mit computerspielhistorischer Intention[97] nachprogrammiert [vgl. Wieland 2011:357-359]. Da es ein 1- bzw. 2-Personen-Spiel ist und die Interaktion eines Mitspielers erfordert, um fortzuschreiten, eignet es sich für eine automatisch ablaufende Simulation weniger als „Game of Life". Dieses ermöglicht allerdings beim Ablauf die Untersuchung der entstehenden emergenten Strukturen. Die historischen Diskurse zum „Game of Life" haben sich im Laufe der Geschichte der zellulären Automaten immer stärker auf solche Strukturen konzentriert, die vom Betrachter als ‚Gestalten' (Gleiter, Raumschiff, Pulsar, Uhr, Kröte, ... [vgl. Conway u.a. 1985a]) beobachtet werden können. Dieser Gestalteffekt [vgl. Wertheimer 1923] sollte mit einer Williams-Kilburn-Tube-artigen Einrichtung vorgeführt werden. Die Zell-Zustände ‚tot'/'lebendig' transformierten dabei in die Pixelzustände an/aus der schwarzweißen Darstellung der Röhre. Die dritte Bedeutung des Titels Game of Memories verweist auf diese archäologische Intention. Es handelt sich damit also um ein Spiel mit Speichern und mit (historischen) Erinnerungen an ‚untergegangene'[98] Medien-/Speichertechnologien.

96 https://videogamehistorian.wordpress.com/2014/01/22/the-priesthood-at-play-computer-games-in-the-1950s/ [letzter Abruf: 06.06.2019].

97 Ein Exponat in der Dauerausstellung des Berliner Computerspiele-Museums zeigt eine Emulation des Spiels nebst nachgebildeten Input- und Output-Devices.

Über das Experimentieren mit der Adressierung einzelner RAM-Speicherzellen konnte so die Entstehung der Pixelgrafik-Technologie nachvollzogen werden. Pixelbilder lassen sich vor diesem Hintergrund als ‚Fenster in den Speicher' sehen. Diese im doppelten Sinne zu verstehenden „Bildschirmspeicher" [Spital u.a. 1985:1-50] besitzen bei bei frühen Mikrocomputern dedizierte Adressbereiche, die allein für grafische Inhalte genutzt werden sollen.[99] Das jeweilige Betriebssystem stellt dem Nutzer deshalb zumeist komfortable Routinen zur Positionierung von Zeichen mit symbolischen X/Y-Koordinaten und Farbangaben zur Verfügung. Intern werden diese Informationen in RAM-Adressen des Bildschirmspeichers umgerechnet. Zusammen mit der Farbinformation, die einen Teil der 8 Bit großen RAM-Inhalte ausmacht, werden sie dann in den Speicher geschrieben, über die I/O-Hardware von dort zum Bildschirmport umgeleitet und auf dem Bildschirm angezeigt.

Diese Technologie, die mit der Nutzung von Pixel-basierten Bildschirmen (LCD-, Plasma- und jüngst LED-Bildschirmen) wieder sichtbar wird, hat sich im *physical computing* als leicht zu programmierende Ausgabe-Hardware erwiesen. Kleincomputer, wie BBC-Mircro-Bit und Calliope verfügen über eingebaute LED-Matrix-Bildschirme; für andere Systeme (Arduino, RaspberryPi, ...) sind solche in zahlreichen Varianten als Shields erhältlich (vgl. Abb. 4.2.10). Diese neuen Einplatinencomputer rufen damit auch die Anzeigen- und Programmiermöglichkeiten der frühen Mikrocomputer und ihrer Bildschirmspeicher in Erinnerung.

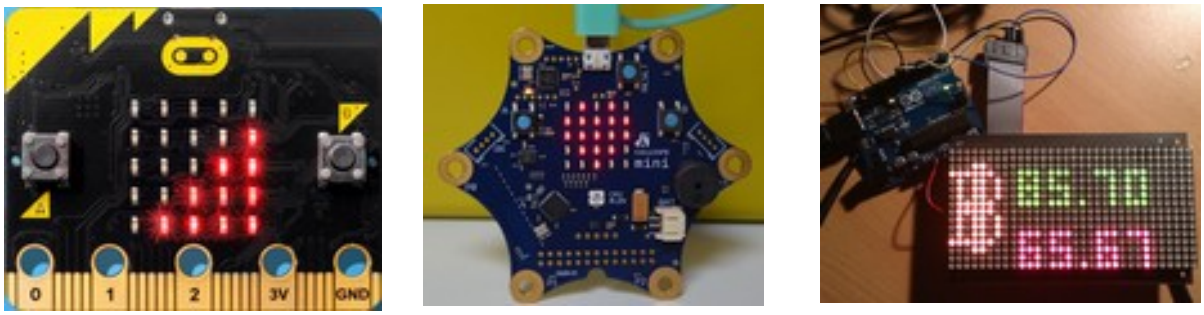


Abb. 4.2.10: BBC-Micro-Bit (links), Calliope Mini (Mitte) und LED-Matrix-Shield für Arduino Uno (rechts)

Der programmierende Nachvollzug der diskreten Simulation „Game of Life" in historischen Programmiersprachen und auf historischen Plattformen sowie sein computerarchäologischer Einsatz als Pixeldisplay-Inhalt haben sich als didaktisch sehr vorteilhaft erwiesen. Bevor dieser Umstand dargestellt und ausgewertet wird, soll ein vergleichswei-

98 In den nachfolgenden Magnetkernspeichern wurden die Bits ebenfalls in einem – nun aber materiellen – Raster gespeichert. Die Entwicklung von Röhren ging konsequenterweise in Bildschirmröhren über [vgl. Ulmann 2014:39].

99 Dies trifft für Systeme mit memory-mapped I/O zu, wie sie die meisten frühen Mikrocomputer nutzten. Bei einigen dieser Systeme ist der RAM-Speicher so knapp bemessen, dass – zum Beispiel beim Laden und Entpacken von Programmen – der Bildschirmspeicher mitgenutzt wird, was kurzzeitige „Einblicke" in die Software ermöglicht. (Vgl. https://youtu.be/jzxWLnHvp44?t=135 [letzter Abruf: 17.08.2018].)

se neues Arbeitsfeld der Informatik vorgestellt werden, in welchem zelluläre Automaten regelmäßig thematisiert und untersucht werden.

4.2.5 Unconventional Computing

Das Arbeitsfeld des *Unconventional Computing* (zeitweise auch A*lternative Computing*) wurde 1998 im Zuge der Tagung „Unconventional Models of Computation" (Aukland, Neuseeland)[100] gegründet. Es ist

> an interdisciplinary research area with the main goal to enrich or go beyond the standard models, such as the von Neumann computer architecture and the Turing machine, which have dominated computer science for more than half a century.[101]

Hierzu untersuchen Wissenschaftler unterschiedlichster Fachgebiete [vgl. Callude 2017:861f.] Computing mit *elektronischen* (Memristoren), *physikalischen* (Kugelstöße), *chemischen* (Aktin, Belousov-Zhabotinsky-Reaktion), *biologischen* (Schleimpilze, Ameisen, Pflanzenwurzeln, DNA-Sequenzen, Nervenzellen) und *virtuellen* (Zellulare Automaten) Komponenten. Ziel ist es dabei sowohl herkömmliche Computerstrukturen (logische Gatter, Speicher) und -architekturen mit unkonventionellen Mitteln nachzubilden als auch neue Wege der Signalverarbeitung zu ergründen [vgl. ebd.:862]. Unconventional Computing widmet sich solchen Verfahren und Materialen vor dem Hintergrund der Beschränkungen und Probleme des ‚conventional computing': Das P-NP-Problem, das Moore'sche Gesetz, die nach Turing und Church definierten Grenzen der Berechenbarkeit [vgl. ebd.:855f.] aber auch Probleme der Energieversorgung, Ressourcenknappheit, Nachhaltigkeit und Materialeigenschaften (etwa im Bereich der Raumfahrt) bilden Anreize alternative Wege der Computertechnologie zu ergründen.

Innerhalb des computerarchäologischen Diskurses ist das Unconventional Computing relevant, weil es nicht lediglich neue Verfahren und Technologien entwickelt, sondern sich auch bewusst oder unbewusst auf historische Aspekte und Elemente rückbesinnt, um sie innerhalb der eigenen Arbeiten zu aktualisieren. Dies erscheint bei einem Blick in die Computergeschichte plausibel: Aus der Perspektive des elektronischen Digitalcomputers betrachtet, beginnt Computergeschichte mit Unconventional Computing: mechanische Rechenmaschinen und Analogrechner, Computerarchitekturen auf Basis von Fluidik, elektronische Analogcomputer oder logische Maschinen bildeten bereits vor der Erfindung des Digitalcomputers alternative Möglichkeiten der Signalverarbeitung und wurden parallel zur ‚Computergeschichte' stetig weiterentwickelt. Aber auch schon zuvor wurden Phänomene unterschiedlicher Disziplinen für Berechnungen ‚zweckentfremdet':

100 https://www.cs.auckland.ac.nz/research/groups/CDMTCS/conferences/umc98/ [letzter Abruf: 21.08.2018].

101 https://cnls.lanl.gov/uc07/ [letzter Abruf: 21.08.2018].

> We wanted to bring to light working prototypes of unconventional computers akin to Riemann's experiments on the flow of electricity in thin metallic sheet, akin to Lord Kelvin's analog differential analyzer, and akin to Plateau's experiments on the calculation of the surface of the smallest area bounded by a given closed contour. [Adamatzky/Teuscher 2006:V]

Im Zusammenhang mit der Diskussion dieses Kapitels sind die Arbeiten zu zellulären Automaten als Elemente unconventional computings hervorzuheben. Zahlreiche Publikationen [vgl. Zhuang u.a. 2017] beschäftigen sich mit unterschiedlichen Typen zellulärer Automaten im Allgemeinen und „Game of Life" im Besonderen. Bei letzterem steht regelmäßig die Implementierung standardisierter logischer Gatter[102] [vgl. Bellos 2014:278ff.] und anderer Elemente von Digitalcomputern im Fokus. Vor allem die massive Parallelität, mit der Signalströme innerhalb der „Game of Life"-Simulation erzeugt und verarbeitet werden können, macht das Spiel zu einem Experimentierfeld unkonventioneller Rechnertechnologien.

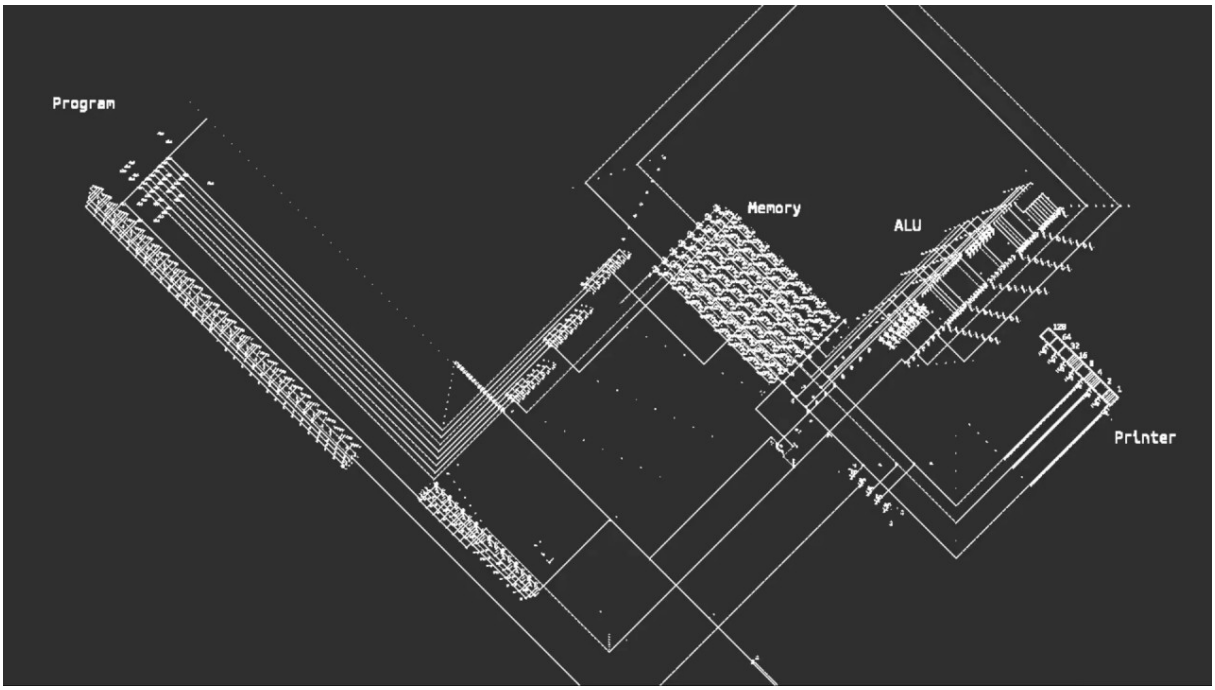


Abb. 4.2.11: „Game of Life"-Computer

Die Tatsache, dass es sich bei „Game of Life" – wie der Name bereits ausdrückt – um ein (Computer)*Spiel* handelt (spieltheoretisch extensiv beschreibbar als „0-Personen-Spiel" [Conway u.a. 1984:123]] mit vollständiger Informationen), das zudem in zahlreichen historischen und aktuellen Implementierungen genutzt werden konnte und kann, öffnet diesen Forschungsdiskurs für eine bereitere Öffentlichkeit. Die Einfachheit des Spiels (sein zweidimensionales Spielraster, der limitierte Satz von Übergangsregeln und lediglich zwei Zellzuständen) erleichtern sowohl eigene Implementierungen des Algorithmus als auch das spielerische Experimentieren mit Game of Life-Strukturen. Aus dem frühen

102 https://www.youtube.com/watch?v=vGWGeund3eA [letzter Abruf: 22.08.2018].

Beweis der Turingvollständigkeit des Spiels [Wainwright 1974] sind zudem Implementierungen unterschiedlicher Algorithmen *innerhalb von* GAME OF LIFE entstanden – etwa ein programmierbarer Computer[103] (Abb. 4.2.11) oder ein TETRIS-Spiel[104].

Adamatzky [2002b] zählt GAME OF LIFE zu den Anwendungen des „Collision based computing“:

> how to do computation in a structureless medium populated with mobile objects. No wires, no valves, nothing is there: just compact patterns wandering in space smashing one to another and calculating [Adamatzky 2002b:V]

In diese Kategorie fallen neben Mikrosystemen (wie Moleküle vgl. [Adamatzky 2002c]) und zelluläre Automaten auch Makrosysteme (wie Billard-Kugel-Computer vgl. [Fredkin/Toffoli 2002:64ff.; Jérôme 2002]). Die komputativen Eigenschaften solcher Makrosysteme sind schon von Leibniz [vgl. Leibniz 1679 sowie Stein 2016] genutzt worden und finden in Spielzeugen seit den 1960er-Jahren einen didaktischen Einsatz. Auch „Game of Life“ lässt sich aufgrund seiner Universalität und der öffentlichen Beschäftigung mit ihm (im Sinne der *Gamification* nach dem hier diskutierten Verständnis) als didaktisches Spielzeug sehen.

4.2.6 Toy Computing

Dieser Blickwinkel soll hier mit dem Konzept des *Toy Computing* das Arbeitsfeld des Unconventional Computing ergänzen. Dabei steht nicht so sehr die Überwindung von ‚Grenzen der Informatik‘ im oben genannten Sinn im Zentrum als eine *didaktische Reduktion* von Komplexitäten unterschiedlicher Provenienzen der Informatik. Dies kann mit konkreten Lernspielzeuge erreicht werden, welche als Simulationen von Computerprozessen aufzufassen wären. Diese reichen von Implementierungen im Realen bis hin zu virtuellen Rechnern mit zellulären Automaten.

Der Einsatz sowohl der genannten Methoden als auch der Lernspielzeuge verknüpft einerseits historische autodidaktische Zugänge zur Computertechnologie mit aktuellen Hacker-Initiativen; andererseits öffnet er Themengebiete der Informatik auch fachfremden Gruppen und Personen. Hier sollen abschließend einige solcher Spielzeuge vorgestellt werden. Um die Vergleichbarkeit zu gewährleisten, handelt es sich bei allen um Implementierungen des „Nim“-Spiels.

103 https://github.com/nicolasloizeau/gol-computer [letzter Abruf: 21.08.2018].

104 https://codegolf.stackexchange.com/questions/11880/build-a-working-game-of-tetris-in-conways-game-of-life [letzter Abruf: 21.08.2018].

4.2.6.1 Das „Nim“-Spiel

Abb. 4.2.12: „Nim“-Spiel, Variante 1 (links) und Variante 2 (rechts)

Bei „Nim“ handelt es sich um ein Zwei-Personen-Nullsummenspiel mit perfekter Information. Es kann von zwei menschlichen oder einem menschlichen und einem maschinellen Antagonisten gespielt werden. Seine historischen Ursprünge sind unbekannt – wahrscheinlich stammt es aus China [Redheffer 1948:343; Rougetet 2014]. Das Spiel besteht aus einer Anzahl von Spielsteinen, die in einer Version (1) in einer Reihe angeordnet werden, in einer anderen Version (2) als ‚Haufen‘ (vgl. Abb. 4.2.12). In Version 1 kann jeder Spieler eine maximal definierte Menge (aber mindestens einen) Spielsteine aus der Reihe entfernen. In Version 2 kann jeder Spieler beliebig viele Spielsteine (aber mindestens einen) aus einer Reihe des Haufens entfernen. Es gibt für beide Versionen überdies zwei Gewinnvarianten: Es gewinnt oder verliert derjenige, der den letzten Stein nimmt. Für beide Versionen des Spiels existiert eine Gewinnstrategie; für Version 1 bei [Dunietz 2015], für Version 2 in [Bouton 1901:37f.]. Letztere wurde durch [Moore 1910] verallgemeinert.

Die Gewinnstrategie lässt sich mittels binärer Modulo-Arithmetik beschreiben und auf die folgende logische Form bringen:

$$x_1 \text{ xor } x_2 \text{ xor } \ldots \text{ xor } x_n = 0$$

(n steht für die Anzahl der Reihen, x_i ist die Anzahl der Objekte in der i-ten Reihe; das Exklusiv-Oder fungiert als binärer Modulo-2-Operator. [Oltean o.J.:3])

Das „Nim“-Spiel ist als Beispiel für Toy Computing interessant, weil es einige didaktische Aspekte jenseits seiner Implementierungsproblematik beinhaltet: „[it] permits the

learning of strategy techniques, logic methods, and mathematical systems.“ [Morris 1972:1] Im einzelnen (und darüber hinaus) heißt dies:

- Es ist vollständig durch binäre Logik beschreibbar und damit sowohl über Logikgatter in unterschiedlich komplexen Systemen (bis zu zellulären Automaten) implementierbar. Es existieren zahlreiche Implementierungen in unterschiedlichen Programmiersprachen – auch für frühe Mikrocomputer [Ahl 1982:122-124].
- Es kann nach Kenntnis des binären Zahlensystems mit einer Gewinnstrategie gespielt werden [vgl. Gardner 1997], motiviert die Spieler also Binärzahlen und -arithmetik zu erlernen.
- Es bietet einen guten Einstiegspunkt für eine Beschäftigung mit der mathematischen Spieltheorie [Conway u.a. 1985b:42ff.; Rougetet 2016:2] und deren Erweiterung zur kombinatorischen Spieltheorie [Conway 1983:101ff.].
- In technischen Implementierungen kann ein künstlicher Gegner realisiert werden, der wiederum ein spezifisches Konzept vom künstlicher Intelligenz und deren Kritik verdeutlicht. [Rougetet 2016:5; Weizenbaum 2018:25]
- Es gilt als das früheste elektronische Spiel auf Basis von digitaler Logik [verfügt über eine lange Geschichte technischer Implementierungen, von denen einige technikhistorisch bedeutsam wurden [Redheffer 1948] (Abb. 4.2.13).
- „Nim“ besitzt neben seiner Technik- auch eine markante kulturhistorische Bedeutung. Rougetet [2016:3] merkt an, dass das Spiel erst durch solche technischen Implementierungen öffentlich bekannt wurde. Alan Turing stellt seine nachfolgende Bedeutung mit einer Anekdote heraus: „Nach dem Ende des Festival of Britain wurde Nimrod nach Berlin geflogen und auf der Handelsmesse ausgestellt. Die Deutschen hatten nie etwas derartiges gesehen und kamen zu Tausenden, um ihn zu betrachten, tatsächlich waren es so viele, daß sie am ersten Tag der Messe einen Ausschank, an dem es Freigetränke gab, gänzlich ignorierten und es nötig war, eigens Polizei zu rufen, um die Massen unter Kontrolle zu halten. Die Maschine wurde sogar noch populärer, nachdem sie den Wirtschaftsminister, Dr. Erhardt [sic], in drei aufeinanderfolgenden Spielen geschlagen hatte.“ [Turing 1987b:118]

Abseits professioneller oder wissenschaftlich motivierter Implementierungen ist das „Nim“-Spiel ab 1955 [Rougetet 2016:7-10] in zahlreichen Varianten als dediziertes (Lern-)Spielzeug produziert worden, um es didaktisch einzusetzen – „especially to explain basis of computer science“ [ebd.:3]. Es ist algorithmisch leicht beschreibbar [ebd.:5] und wird daher bis heute im schulischen Informatikunterricht verwendet: „Nim game is set for a comeback in French educational program in September 2016 as a recreational application to tackle algorithmic and programing in mathematic high school classes“ [ebd.:12].

Nach Bekanntwerden seiner Gewinnstrategie fanden sich auch zeitnah Eigenimplementierungen. Drei „Nim“-Spiele der Variante 1 für programmierbare Spiel(zeug)e sollen hier kurz vorgestellt werden. Ausgewählt wurden hierfür Systeme, die auf Ansätzen[105] der Technologie „collision-based computing“ basieren.

Abb. 4.2.13: Ferrantis NIMROD (1951)[106] (links) und Claude Shannons Nimwit (1950) [Roch 2009:23] (rechts) als Beispiele früher Implementierungen

4.2.6.2 The Amazing Dr. Nim

Das Spiel The Amazing Dr. NIM (Abb. 4.2.14) erschien im Jahre 1966 von der US-amerikanischen Spielzeugfirma *E.S.R. Inc.*[107] Es ist ein mechanisches Spiel, das eine Person gegen „Dr. Nim“ (einen maschinellen Gegner) spielt: „the game could be played in normal or in misére convention (the last player to move loses) and the initial number of marbles could vary between 9 and 20“ [Rougetet 2016:10]. Für den mechanischen Gegenspieler nimmt der menschliche Spieler in jeder Runde einen initialen Zug vor.

Die sehr einfache Mechanik des Spiels scheint im Kontrast zu der Tatsache zu stehen, dass die Maschine das Spiel gegen einen menschlichen Spieler gewinnen kann:

> [...] how did Dr. NIM know how many marbles were left? How could it calculate how many to release? Could the old bastard count? How does plastic compute? Just like silicon! [Dunietz 2015]

105 Lediglich Minecraft (Kapitel 4.2.6.3) erfüllt die Bedingung „structureless“ [Adamatzky 2002:v]. Dadurch, dass die Murmeln in The Amazing Dr. Nim (Kapitel 4.2.6.1) strukturverändernd wirken und die Struktur in Turing Tumble (Kapitel 4.2.6.2) selbst Gegenstand des „computing“ ist, können diese Spiele zumindest unter den Begriff „unconventional computing“ gefasst werden.

106 https://videogamehistorian.wordpress.com/2014/01/22/the-priesthood-at-play-computer-games-in-the-1950s/ [letzter Abruf: 27.08.2018].

107 Vgl. http://www.computerhistory.org/collections/catalog/102688881 [letzter Abruf: 27.08.2018]. Ein Spielverlauf kann hier verfolgt werden: https://www.youtube.com/watch?v=oxBghtQ8McA [letzter Abruf: 27.08.2018].

Abb. 4.2.14: The Amazing Dr. NIM (1966) und Digi-Comp II (ca. 1960)

Es ist die Kenntnis der Gewinnstrategie, deren Algorithmus hier als mechanische Struktur implementiert wurde [E.S.R. 1966]. Dadurch, dass die „Flip-Flops" [ebd.] über lediglich zwei Zustände verfügen, können sie als Speicher eines mechanischen Binär-Zählerwerks verwendet werden. Dieses muss lediglich bis zur vier (100_2) Zählen, um den Zug für eine Gewinnposition in seiner Struktur abzuspeichern [Dunietz 2015]. Die Murmeln bilden hierbei gleichzeitig die zu verarbeitenden Datensignale (Eingabe/Ausgabe) und die Steuerungssignale (Umschaltung der Flip-Flops).

Im Patent [Godfrey 1968] und in der Anleitung [E.S.R. 1966] wird das Spiel demzufolge als „Binary Digital Computer" bezeichnet. Die Anleitung begründet die künstliche Intelligenz des Systems mit einer Anspielung auf Alan Turings Frage „Können Maschinen denken?" [Turing 1987:149]:

> By now you have played against DR. NIM enough to respect and appreciate his ability. Does he really think? [...] You will probably say that DR. NIM does not ‚think' despite the fact hat he plays a clever game of NIM. [...] So, let us leave this subject of ‚Can Machines Think?' for the moment, and consider DR. NIM first from the computer machine point of view and then the computer programming point of view. [...] DR. NIM is a ‚binary digital computer' specially designed to play the game of NIM. [...] Counting, logical functions, altering internal states, memory, and gating impulses are all typical characteristics of a computer. They are combined in such a manner that the machine appears to play an ‚intelligent' game of NIM. [E.S.R. 1966]

Im Patent fasst der Erfinder seine didaktischen Intentionen zusammen:

> Inclusive among the concepts which may be explained and understood by this computer invention are the following: the binary number system; the simplicity for machine design using binary arithmetic; [...] the rule for binary counting and addition; modular arithmetic; the use of two's complement arithmetic to achieve subtraction with only its add capability; [...] binary multiplication. [Godfrey 1968 zit. n. Rougetet 2016:11]

Nicht nur dieser didaktische Paratext, auch die Mechanik basieren auf dem Vorgängerspiel von The Amazing Dr. NIM: Digi-Comp II[108] (Abb. 4.2.13). Es erschienen in den 1960er-Jahren bei *E.S.R. Inc.* Bei Digi-Comp II, der dem Digi-Comp I und dem Think-a-Dot (beide ebenfalls *E.S.R. Inc.*) nachfolgt, handelt es sich um einen murmelbasierten Computer, der jedoch programmierbar und turingvollständig ist [Aaronson 2014]. Digi-Comp II ist in der Lage binärarithmetische Berechnungen anzustellen. Hierfür stellt das Handbuch sogar eine eigene Programmiersprache (Digi-Tran) zur Verfügung [E.R.S. o.J.].

4.2.6.2 Turing Tumble

Digi-Comp II ist ein ‚Software-Spielzeug', bei dem es darum geht, mit der vorhandenen Struktur Programmieraufgaben zu lösen. Die Hardware von Digi-Comp II lässt sich nicht modifizieren. Ebenso erlaubt es The Amazing Dr. Nim nicht, ein anderes Spiel als Nim darauf zu spielen. Dieses Defizit gleicht ein jüngerer Murmelcomputer mit dem Titel Turing Tumble[109] aus. Turing Tumble ist 2017 im Rahmen einer Kickstarter-Kampagne entwickelt worden. Bei diesem Spiel müssen mit Hilfe vorhandener Elemente (Rampen, Register, Speicher, Zahnräder) Strukturen auf einem Steckbrett entwickelt werden, um Aufgaben zu lösen. Diese Aufgaben werden (mit ansteigendem Schwierigkeitsgrad) im Handbuch als Rätsel präsentiert, die wiederum selbst auf unterschiedlichen mathematischen Problemen basieren.

Turing Tumble (Abb. 4.2.15) wird als mechanischer Computer und Lernspielzeug gehandelt – durch seine Dokumentation und Präsentation auf der Internetseite auch für den Schuleinsatz. Das Handbuch [Boswell/Boswell 2017] stellt explizite Bezüge zur Hardware eines Mikrocomputers her [ebd.:vi] und in einigen Experimenten auch Automaten vor (einen Zustandsautomaten im Experiment „Regular Expression" [ebd.:47]). Mit Turing Tumble können Hardware-Elemente von Digitalcomputern (ein Addierer [ebd.:62], ein Dekodierer/Kodierer [ebd.:84,87] usw.) hergestellt werden. Am Ende stellt die Anleitung die Implementierung eines „Dr. Nim"-Spiels [ebd.:91] zur Aufgabe:

> Objective: Win the game! Take turns with Magnus. Whoever takes the last ball on their turn loses! On your turn, release 1, 2, or 3 balls. Then turn the gear bits to the left once you have finished your turn. Press the lever down once more and Magnus will take 1, 2, or 3 balls, and the gear bits back to the right when it is your turn again. You take the first Move! [ebd.:91]

Hier besteht nun die Aufgabe nicht in der Konstruktion des Nim-Spiels, sondern darin, es gegen die Künstliche Intelligenz („Magnus") zu gewinnen. Die Implementierung von „Dr.

108 https://en.wikipedia.org/wiki/Digi-Comp_II [letzter Abruf: 27.08.2018]. Das Spiel wurde 2014 nachgebaut: http://www.scottaaronson.com/blog/?p=1902 [letzter Abruf: 27.08.2018]. Eine frühere Implementierung in Lego existiert seit 2007: http://www.brickshelf.com/cgi-bin/gallery.cgi?f=288822 [letzter Abruf: 27.08.2018].

109 https://www.turingtumble.com/ [letzter Abruf: 27.08.2018].

Nim“ zeigt nicht nur die Mächtigkeit von Turing Tumble auch komplexere als nur arithmetische Probleme lösen zu können, sondern stellt zugleich auch einen historischen Rückgriff auf The Amazing Dr. Nim dar.

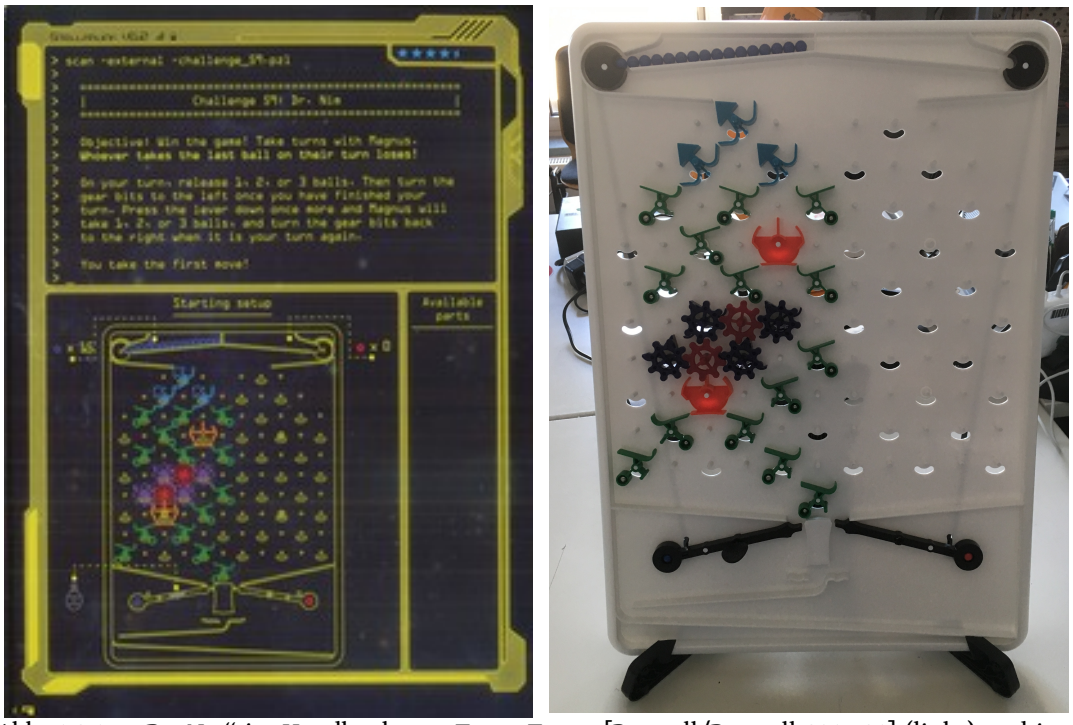


Abb. 4.2.15. „Dr. Nim“ im Handbuch von Turing Tumble [Boswell/Boswell 2017:91] (links) und implementiert (rechts)

Der Übergang vom Software-Spiel (Digi-Comp II) zum Hardware-Spiel (Turing-Tumble) eskaliert in experimentellen Spiel-Implementierungen, bei denen die Hardware-Elemente zunächst selbst entworfen werden müssen.

4.2.6.3 Minecraft

Im Zuge der Entdeckung der Turing-Vollständigkeit von Game of Life hat sich bereits gezeigt, dass Nutzer für solche Systeme auch komplexe Implementierungen vorgenommen haben. Die Tatsache, dass viele Computerspiele mit Bit-Board-ähnlichen Strukturen arbeiten, hat Experimente provoziert, ob sie sich als zelluläre Automaten einsetzen lassen. Ein populäres Beispiel hierfür ist das Spiel Minecraft, das 2009 für unterschiedliche Computerplattformen erschienen ist.

Minecraft kann in mehreren Modi gespielt werden. Von zentraler Bedeutung für das Spiel ist die Notwendigkeit aus Rohstoff-Elementen des Spiels komplexere Gegenstände zu konstruieren, um diese im weiteren Spielverlauf nutzen zu können. Um mit dieser Spielfacette experimentieren zu können, existiert ein „Kreativmodus“, in welchem dem Spieler alle Ressourcen zur Verfügung stehen. Dieser Modus zeigt das Spiel Minecraft als turingvollständigen zelluläre Automaten: Mittels der so genannten „Red Stone“-Technologie können Strukturen in Minecraft konstruiert werden, die diskrete Signale generieren, übertragen, speichern und verarbeiten. (Darüber hinaus lässt es die API des Spiels zu mit

einem Python-Interpreter zu kommunizieren, so dass Minecraft auch im Rahmen von Programmiereinführungen eingesetzt wird [vgl. Richardson 2016]).

Für die „Red Stone"-Technologie, auf der alle genannten Implementierungen basieren, existieren inzwischen Manuale (z. B. des Spielentwicklers [vgl. Mojang 2014]), die die Programmierung in Minecraft lehren. Dies hat zu umfangreichen Experimenten geführt, in deren Rahmen logische Gatter[I], größere logische Strukturen (wie ALUs[II]), Rechenmaschinen[III], komplexe Anwendungen wie Computerspiele (Tetris[IV], OXO[V], Pong[VI], „Game of Life"[VII], ...) und sogar Software-Emulatoren für Spielhardware (Nintendos GameBoy[VIII]) in Minecraft realisiert worden sind. 2017 ist in diesem Rahmen auch eine „Dr. Nim"-Implementierung veröffentlicht worden[IX].[110] (Abb. 4.1.15)

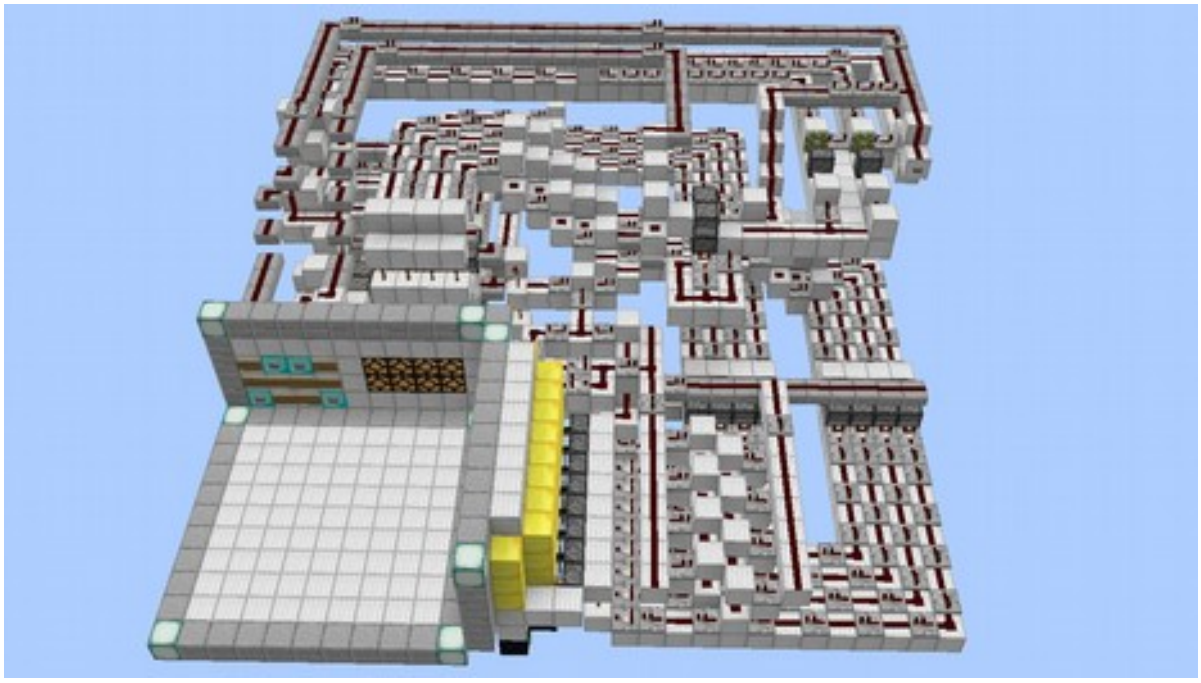


Abb. 4.2.16: „Dr. Nim" in Minecraft Frontend (links unten) und Backend (rechts und oben)

Die Physikalität der Murmeln aus The Amazing Dr. Nim oder Turing Tumble wird hier durch virtuelle diskrete Signale aus den „Red Stones" simuliert. Für die Ein- und Ausgaben des Nim-Spiels werden Minecraft-spezifische Spielelemente (Tasten, Rampen, Spielfiguren) benutzt. Zur Eingabe stehen zwei Tasten (eine für den Spieler, eine zur Aktivierung des Spielzugs der Künstlichen Intelligenz „Dr. NIM" zur Verfügung. Die Unterfläche des Spiels lässt sich bei einem „Rundflug" um die Struktur erkunden (Abb. 4.2.16). Der Code

110 Im Folgenden Links zu Videos und Blog-Einträgen der entsprechenden Minecraft-Implementierungen: I. https://www.youtube.com/watch?v=9EY_XoEImjM; II. https://www.youtube.com/watch?v=HUVxbAh44XE; III. https://www.youtube.com/watch?v=BEXtiwXHTfk; IV. https://www.youtube.com/watch?v=IZk6mp90Zi4; V. https://www.youtube.com/watch?v=HfTLndv3VQg; VI. https://www.youtube.com/watch?v=20HQOwDCbWs; VII. https://www.youtube.com/watch?v=TKnjH2OYSYU; VIII. https://www.youtube.com/watch?v=IeUsoJQg_Ho; IX. https://www.planetminecraft.com/blog/dr-nim-in-minecraft/ [alle letzter Abruf: 27.08.2018].

des ‚Spiels im Spiel' ist offen, so dass es möglich ist, diesen zu modifizieren und zu erweitern.

	The Amazing Dr. Nim	Turing Tumble	Minecraft
Implementierung	Original	Hardware-Emulation	Software-Emulation
Substrat	Hardware (dediziert)	‚Firmware' (mikroprogrammierbar)	Software (programmierbar)
Signale	15 Murmeln	15 Kugeln	Zelluläre Automaten
Struktur der Implementierung	3 Flip-Flops, 1 Equalizer	14 Ramps, 2 Crossovers, 3 Bits, 2 Gears, 4 Gear-Bits	Diverse Logikgatter aus Red Stones
Mächtigkeit	‚Nim complete'	‚circuit complete'	Turing complete

Tabelle 4.2.2: Vergleich der drei „Nim"-Implementierungen

4.2.7 Zusammenfassung

Das vorangegangene Kapitel zeigte, ausgehend von einer Begriffsbestimmung von „Simulation" die ‚spielerischen' Facetten innerhalb diskret-simulierender Computerprogramme – hier zellulärer Automaten und Game of Life. Über die Auseinandersetzung mit konkreten Implementierungen für ein historisches Computersystem wurde gezeigt, zu welchen Zwecken ein solcher zellulärer Automat eingesetzt werden kann: angefangen bei der *Simulation von Hardware* (hier funktionellen Eigenheiten der Williams-Kilburn-Tube) über die *Simulation von informatischer Theorie* (hier die Turing-Maschine) bis hin zur *Simulation von Software* (hier: Game of Life in Game of Life).

Mit dem „Nim"-Spiel wurde dieser Aspekt vor einen informatikdidaktischen Hintergrund gestellt: Mittels komputierender Spielzeuge findet seit Jahrzehnten eine spielerische Auseinandersetzung mit den Themen Komplexität, Automatentheorie und Programmierung statt, der sich von Fragen zur Funktion (The Amazing Dr. Nim), zur Erzeugung von Rechnerstrukturen (Turing Tumble) bis hin zur Konstruktion von Elementarstrukturen unterhalb des Tanenbaum'schen Layermodells (Dr. Nim in Minecraft-"Red Stone"-Technologie) erstreckt. Die Erforschung dieser Aspekte vollzog sich größtenteils aus ‚Willen zum Wissen' innerhalb hobbyistischer Projekte.

Hieraus ließe sich eine *didaktische Methode* entwickeln, das als *Toy Computing* Elemente der Informatik, der Mikroelektronik, der Logik und der Mathematik (Spieltheorie, Komplexität und Berechenbarkeit u.a.) auf niedrigschwelligem Niveau und mit spielerischen Mitteln erprobt. Als Teil des Unconventional Computing könnte eine solche Didaktik historische und zeitgenössische alternative Rechnermodelle, -architekturen und -technologien nutzen und diese zurück in den Diskurs bringen. Der Einsatz von Spielzeugen zu didaktischen Zwecken setzt erst zu Beginn des 20. Jahrhunderts ein [Rougetet 2016:7]. Für

eine informatik-didaktische Verwendung sind hier vor allem dedizierte Computerspielzeuge (zu einzelnen Spielzeugen: Roboter [Frei u.a. 2000], Spielzeug-Computer [Nixon 2011:130], Lernbausätze [Resnik u.a.: 1996; 1998] oder Lernspiel-Software [Squire 2005]) untersucht worden. Unter den Begriff *Toy Computing* wurden in der Literatur bislang spezifische Spielzeuge gefasst: „Toy computing incorporates the physical component of a traditional toy combined with networking and sensory capabilities of mobile devices." [Rafferty/Hung 2015:1] Hier soll eine alternative bzw. erweiterte Sicht auf den Begriff und das Konzept vorgeschlagen werden: *als Verwendung von Spielzeugen unterschiedlicher Art/Komplexität/Epochen zu informatikdidaktischen Zwecken*. Hierunter fällt der ‚Missbrauch' (im Sinne des *Hacking*) von Spielzeugen zu Zwecken des Computing. Die daraus resultierenden didaktischen Möglichkeiten sind bislang nicht systematisch untersucht worden. Hier eröffnen sich Möglichkeiten einer vertiefenden theoretischen und experimentellen Forschung.

Das Verständnis von Simulation als ‚Mimikry-Spiel(zeug)' und die Erweiterung von *Toy Computing* hin zu *Toy Computation* (zur Differenzierung im Rahmen des „uconventional computation" vgl. [Calude 2017:855f.]) fügt der informatischen auch eine medienwissenschaftliche Komponente hinzu, die als Frage nach der Unterscheidung von Simulation und Emulation kulminiert. Für die Implementierung von Game of Memories ist bereits ein Emulator zum Einsatz gekommen. Welche technikhistorischen und (vor dem Hintergrund der Überlegungen aus Kapitel 4.1) zeichentheoretischen Erwägungen lassen sich zur Differenzierung dieser Begriffe und Techniken ansetzen? Hierbei stünde die Unterscheidung von Hardware und Software zur Diskussion (wie Kittler [1993b] bereits angemerkt hat), wird Software doch schon längst (etwa in maskenprogrammierten ROMs) als Hardware realisiert und lassen sich Hardwarestrukturen (in EPROMs, EEPROMs, FPGAs usw.) symbolisch programmieren/manifestieren.

Computerarchäologisch offenbart sich hier zudem eine *Präservationsproblematik*: Die zuvor dargestellten Möglichkeiten von Software-Emulationen historischer Hardware-Plattformen als epistemologische Werkzeuge widersprechen ihrem intendierten Zweck – der Suspendierung von Hardware. Sie können zugleich mehr und weniger als ihre Vorbilder. Als derartige Surrogate bergen sie Probleme, die auf ganz unterschiedliche Gründe zurückzuführen sind. Das folgende Kapitel wird solche Probleme der oben bereits vorgestellte Emulationssoftware benennen und eine Alternative zur Software-Emulation vorstellen.

4.3 Software Preservation und Emulation

Im Zuge des durch die Mikrocomputertechnologie ausgelösten soziokulturellen Wandlungsprozesses, der gemeinhin als ‚Digitalisierung' bezeichnet wird, rücken deren obsolet gewordene Objekte auch als zu bewahrende Artefakte ins Zentrum musealer und archivalischer Bemühungen: Zur Technikgeschichte der Hardware, der Nutzungs- und damit

verbundenen Sozialgeschichte, tritt die *Software* als ein Kulturgut mit zahlreichen technologischen und historischen Implikationen hinzu. Die Bewahrung von Software konfrontiert Museen und Archive allerdings mit neuartigen Problemen, die von der Frage nach dem grundsätzlichen Wesen von Software [vgl. Kittler 1993b; Kittler 2007] über die materiellen Substrate, auf denen sie gespeichert/überliefert werden kann, bis hin zur möglichen und erwünschten Ausführbarkeit von Programmen und Lesbarkeit von Daten auf historischen oder zeitgenössischen Hardware-Plattformen reicht.

Software Preservation ist vor allem mit den letzten beiden Problemen der Datenträgerkonservierung und *Lauffähigerhaltung von Programmen* sowie der *Lesbarkeit von Daten* beschäftigt und versucht Möglichkeiten der Bewahrung zu etablieren, die zwischen dem *vollständigen Erhalt einer Hardware-Datenträger-Software-Infrastruktur* und der *vollständigen Virtualisierung* all dieser Objekte situiert sind [vgl. Douglas 2000; Mortensen/Kapper 2015; Murlasits/Resinger 2012; Borghoff/Krebs/Röding 2014]. Die für diese Arbeit zentralen Fragen zur möglichen/unmöglichen Historiografisierung von Software sowie zu den sich aus den verschiedenen Bewahrungsstrategien ergebenden epistemologischen Mehrwerten, waren bisher von lediglich peripherem Interesse. Die *Emulation von Computerhardware* als Mittel zur Lauffähighaltung von Software steht derzeit im Fokus sowohl des hobbyistischen [Wüthrich 2007; Sieg 2013] als auch des museologischen [Lange 2016] als auch des informatischen [Loebel 2014] Diskurses über Software Preservation. Im folgenden Kapitel sollen diese Diskurse miteinander konfrontiert und um eine computerarchäologische Perspektive erweitert werden.

Nachdem das Konzept der Emulation in seiner historischen Genese und seiner technischen Praxis vorgestellt wurde, wird die Anwendung (und die Probleme) von Emulatoren demonstriert. Deren Bestimmungszwecks wird als „rückwärtsgerichtet“ [Loebel 2014:114] bestimmt: „Es soll eine abgeschlossene Menge existierender Software erhalten werden. Der Blick ist also nicht auf die Entwicklung und das Testen neu zu erstellende[r] Software gerichtet.“ [ebd.] Eben dieser Aspekt der *Neuentwicklungen* innerhalb der Retrocomputing-Szenen ist für eine computerarchäologische Betrachtung allerdings von besonderem Interesse und soll deshalb an einem konkreten Projekt dargestellt werden, welches aus folgenden Teilen besteht:

1. der Entwicklung eines neuen Computerspiels für eine historische Spielkonsole,
2. dem Einsatz von Emulatoren für diesen Zweck und
3. einer Hardware-Entwicklung, die diesen Zweck unterstützt.

Dieses Projekt fand im Rahmen einer (unter Kap. 4.1.2 und Kap. 4.2.2 bereits teilweise vorgestellten) Lehrveranstaltungsreihe statt. An deren Ende sollte die Entwicklung einer eigenen Erweiterung stehen: eine kombinierte Hardware-Software-Lösung zur Verwendung auf der Originalplattform VC-4000 von *Interton* aus dem Jahr 1978. Das von den Ent-

wicklern Johannes Maibaum und Mario Keller VC4000MULTIROM betitelte Modul sollte es erstmals ermöglichen auf dieser Plattform virtualisierte ROM-Dateien über eine SD-Speicherkarte in die Spielkonsole zu laden und damit nicht nur das Spielen historischer Spiele, sondern auch die umstandslose Entwicklung neuer Programme auf dem System ermöglichen. Entstanden ist das VC4000MULTIROM im letzten Teil des 4-semestrigen Assembler-Workshops für BA- und MA-Studenten am *Fachgebiet Medienwissenschaft*.[111]

In diesem Projektseminar sollten die Studierenden den Mikroprozessor Signetics 2650, spezifische darauf basierende Lernplattformen, die Programmierung der VC-4000 sowie die Entwicklung eines Flash-ROM-Steckmoduls erlernen. Dies lies sich angesichts der Komplexität der Aufgaben, der unterschiedlichen akademischen Herkunft der Teilnehmer und der Tatsache, dass es sich um benotete Lehrveranstaltungen[112] handelte, nicht als rein autodidaktische Aufgabe realisieren. Die von der Studienordnung vorgeschlagene und anstelle dessen zur Anwendung gebrachte Lehrmethode wird als *Blended Learning* (vgl. Kap. 5.2.3.3) bezeichnet: Hierbei findet eine Kombination aus Präsenzlehre, Eigenarbeit und E-Learning statt. Die Präsenzlehre wurde im Semesterwechsel nach verschiedenen Lern-Strategien vollzogen: *Hacking* (experimentell), *Reverse Engineering* (induktiv), *steigende Programmkomplexität* (induktiv), *problemorientierte Programmentwicklung* (gemischt) und durch *Erarbeiten des Befehlssatzes* (deduktiv) [vgl. Höltgen 2014a]. Im ersten Semester arbeiteten die Studenten an individuellen Programmierprojekten (vgl. Kap. 4.2), im zweiten Semester sollte in Kleingruppenarbeit ein Spiel entwickelt werden (vgl. 4.3.2.2). Im dritten und vierten Semester stand die gemeinsame Entwicklung der Hard- und Software für das Flash-ROM auf dem Plan.

Die Teilnehmerzahl war zuletzt allerdings stark zurückgegangen und von den verbliebenen Studenten waren nur zwei Teilnehmer gewillt, das Projekt plangemäß abzuschließen. Dies vollzogen sie dann sowohl im oben definierten Sinne eines autodidaktischen E-Learnings im Dialog mit den Computern als auch im landläufigen Sinne durch Nutzung von Lehrplattformen wie Moodle (hier waren die Dokumentation und zahlreiche Lehrwerke für die Studierenden hinterlegt[113]), Online-Dialogsysteme (E-Mail, Chat) und Versioning-Verwaltungssysteme (GitHub[114]). Neben einigen abgeschlossenen Programmierprojekten der ersten Semester entstand ein vollständig funktionsfähiger Prototyp

111 https://agnes.hu-berlin.de/lupo/rds?state=verpublish&status=init&vmfile=no&publishid=88820&moduleCall=webInfo&publishConfFile=webInfo&publishSubDir=veranstaltung [letzter Abruf: 15.03.2019].

112 Projektkurse im Fach Medienwissenschaft (BA und MA) laufen über ein oder mehrere Semester und werden von den Studierenden weitgehend autonom (Workload: 225 Stunden/Semester) absolviert. Der Dozent trifft sich zusätzlich wöchentliche zu einer Konsultationsstunde (14 Stunden/Semester) mit ihnen, um die Projekte zu planen, Fragen zu beantworten und Hilfestellungen zu leisten. Diese Konsultationsstunden wurden hier gebündelt zur Einführung in die Assembler-Programmierung verwendet. Die Programmierprojekte der Studierenden wurden danach autonom absolviert.

113 https://moodle.hu-berlin.de/course/view.php?id=55020 [letzter Abruf: 15.03.2019].

des VC4000MultiROM. In diesen flossen die in der Präsenzlehre gewonnenen Kenntnisse in Assembler-Programmierung und zum Hardware-Aufbau der VC-4000 autodidaktisch erarbeitete Kenntnisse in der Mikroelektronik, des Schaltungsdesigns sowie der Programmiersprache C (das Hostsystem des VC4000MultiROM ist ein in C programmierter Arduino-Nano-Computer) ein.

Ein intrinsisches Motiv zur Entwicklung des Moduls hatte zudem die Programmierung des im zweiten Semester geplanten Spiels Flappy Bird für die VC-4000 gegeben: Aufgrund zu starker Abweichungen zwischen der Spielkonsole und ihrer Emulation in WinArcadia wurde die Entwicklung unvollendet abgebrochen. Mithilfe des VC4000MultiROM sollte diese wieder fortgesetzt werden. Allerdings war zwischenzeitlich ein kursexterner Hobbyist, angeregt durch die Vorüberlegungen, tätig geworden und hatte Flappy Bird für die VC-4000 fertig gestellt.[115]

4.3.1 Emulation

Die bei der Entwicklung des Spiels entdeckten Defizite des im Kurs verwendeten WinArcadia-Emulators waren/sind „prinzipbedingt“ [Loebel 2014:146], weil er selbst zur Gruppe der hobbyistischen Retrocomputing-Projekte gehört und damit zur Veröffentlichung keinen Qualitätsstandards genügen muss. Bevor diese Probleme eingehender dokumentiert und interpretiert werden, soll ein kurzer systematischer und historischer Überblick über die Softwaregattung *Emulator* gegeben werden. Dieser bezieht sich im Wesentlichen auf Loebel [2014]. Dort wird der Begriff Emulation wie folgt definiert:

> Im Kontext der Informatik meint Emulation die Nachahmung von bestimmten Teilaspekten eines Computersystems mittels Soft- oder Hardware, welche als Emulator bezeichnet werden. Ein solcher Emulator ist in der Lage, die Software eines Systems A [im Folgenden *Originalsystem*, S. H.] (beispielsweise ein obsoleter Heimcomputer, wie der Commodore 64) auf einem System B ([im Folgenden *Hostsystem*, S. H.] z.B. ein aktueller Intel-PC) auszuführen und erzielt im Idealfall bei gleichen Eingabedaten die gleichen Ergebnisse. [...] Dabei wird der Begriff in Abgrenzung zur Simulation verwendet, bei der eine Reduktion auf spezifische Eigenschaften im Vordergrund steht und der in der Regel ein abstraktes oder abstrahiertes Modell zugrunde liegt. Demgegenüber verfolgt die Emulation einen ergebnisorientierten Ansatz: Gleiche Eingaben sollen zu gleichen Ausgaben führen, wobei die internen Zustände des Systems je nach Grad der Abstraktion weniger von Interesse sind. [41]

Die hier vorgenommene definitorische Abgrenzung zum Konzept der Simulation aus der Perspektive der angewandten Informatik entspricht der oben (Kap. 4.2.1) vorgenommenen epistemologischen Einordnung des Begriffs: Simulationen bilden die internen Zu-

114 https://github.com/mkeller0815/VC4000MultiROM.git [letzter Abruf: 11.07.2016].

115 http://tempect.de/senil/progra/FLAPPY2.ASM [letzter Abruf: 01.11.2016].

stände eines Systems auf Basis einer formalen (z.B. mathematischen) Beschreibung ab, um auf diese Weise Erkenntnisse über Prozessabläufe zu erhalten. Demgegenüber werden Emulatoren „ergebnisorientiert programmiert“ [100], was bedeutet, dass hier vor allem auf eine hohe „Authentizität der Reproduktion“ [104][116] von Ausgaben, Zeitverhalten und I/O-Prozessen geachtet wird, um eine möglichst genaue Mimikry des Originalsystems zu erzielen. Dabei können die in der Emulator-Software kodierten Prozesse und Strukturen stark von denen dieses Originalsystems abweichen.

4.3.1.1 Aufbau und Arbeitsweise von Software-Emulatoren

Zur Erreichung dieses Ziels können Emulatoren rein in Software oder als Kombination aus Hardware und Software implementiert werden, was davon abhängt, „welche Teile des Originalsystems durch den Emulator nachgebildet werden“ [61] sollen. Emulation kann zudem an verschiedenen Schichten des Originalsystems ansetzen:

1. auf der Applikationsebene, wobei das Programm unter Beibehaltung der Spieläs-thetiken für ein neues System programmiert wird (vgl. La Mulana, Tab. 4.3.1)
2. als Emulation der Programmierschnittstelle (API) (ein Beispiel hierfür ist die Z-Machine) [vgl. 62],
3. auf der Betriebssystemebene, wobei das Betriebssystem, nicht jedoch die zugrun-de liegende Hardware (insb. nicht der Prozessor) emuliert wird [vgl. 64],
4. als „Full System“-Emulation, bei der „eine vollständige Emulation aller Hardware-komponenten“ angestrebt wird [vgl. 68-71].

Varianten 1 und 2 werden für die Emulation von historischen Computern zu Zwecken der Software Preservation von Loebel als „nicht sinnvoll“ [62] erachtet oder stellen „keine langfristige Alternative“ [64] zur Hardware Preservation oder zu Variante 3 dar: Die Emulation der API ist zu aufwendig angesichts der Tatsache, dass für jede Applikation eine neue Emulation entwickelt werden müsste; die Emulation des Betriebssystems ist zu eingeschränkt auf ein spezifisches (Betriebs)System, so dass auf ihr nicht alle möglichen (historischen) Applikationen für eine Hardwareplattform ablaufen können.

Die „Full System“-Emulation wird daher als einzige adäquate Alternative beschrieben. Aber auch diese kann noch „5 global levels of accuracy“ [Tijms 2000] aufweisen:

1. „Datapath accuracy: [...] modeling the internal structure of components which can not be seen from the outside. This level is mostly used by hardware designers to test new architectures and is mostly referred to as simulation.“ [ebd.]

116 Der durchaus polyseme Begriff der Authentizität wurde an anderer Stelle definiert und diskutiert [vgl. Höltgen 2009:28-56].

2. „Cycle accuracy: [...] modeling the system (or some of its parts) in such a way that instructions complete in the same relative amount of time compared to each other and to other components in the system. This level is often needed for accurate emulation although usually not strictly required.“ [ebd.]

3. „Instruction level accuracy: [...] models everything a genuine instruction can see and touch but without the strict timing constrains. E.g., a complicated divide instruction could possibly complete just as fast as a simple add in terms of the virtual machine timing.“ [ebd.]

4. „Basic block accuracy: [...] used in combination with dynamic translation [...]. The idea is to replace a basic block with one in native code having the same outputs for a given input.“ [ebd.]

5. „(Very/ultra) high level emulation accuracy [...]: instead of replacing basic blocks it replaces larger blocks that have some specific semantics. These blocks contain high-level functionality provided by the original system, which could be anything, but in practice it's usually restricted to audio or video output.“ [ebd.]

Welcher Grad an „accuracy“ erreicht werden kann, hängt nicht nur von der Software (etwa den Kenntnissen des Emulator-Entwicklers) ab, sondern auch von den Möglichkeiten der Host-Hardware. Zudem existiert eine *kategorische Schranke* der Emulierbarkeit.

4.3.1.2 Grenzen und Trade-Offs von Software-Emulatoren

Die Abbildung einer Hardwarestruktur in Software ist mit ‚Übersetzungsproblemen' (siehe 4.3.1.3) verbunden. Selbst die „datapath accuracy“ unterliegt noch einer Abstraktion: „Nicht emuliert werden das analoge Verhalten der Widerstände, die Kapazität und Ableitung von Kondensatoren sowie das zeitliche Schaltverhalten der Transistoren.“ [Loebel 2014:89] Dieser Umstand hat für die Software Preservation Einschränkungen zur Folge. Diese betreffen auch die Emulation von Spezialbausteinen (z.B. für Sound und Grafik) sowie das „outside environment“ [112] (etwa spezifische Ausgabe- und Eingabegeräte usw.) des Systems. Sollen diese mit-emuliert werden, wir die zusätzliche Implementierung von „Skeumorphismen“ notwendig,

> die funktional bzw. strukturell inhärente Eigenschaften von zur Funktion notwendigen intrinsischen Komponenten des Originalsystems in der Emulationsumgebung nachbilden. Dabei verlieren diese ihren originären Zweck und nehmen einen schmückenden Charakter an. [127]

Hierzu zählen beispielsweise optische Filter, die Bildrauschen, Scanlines, Colour Bleeding und andere Effekte von CRT-Monitoren oder Bewegungsgeräusche mechanischer Peri-

pheriegeräte[117] nachahmen. Auch die Implementierung solcher Skeumorphismen und anderer Artefakte unterliegt einem Trade-Off: „wegen der aufwendigen Berechnungen [muss aber] eine Abwägung zwischen Emulationsgenauigkeit und Ausführungsgeschwindigkeit“ [120] stattfinden.

Je detaillierter die Emulationssoftware das Originalsystem im Hostsystem abbildet, desto größer werden solche Trade-off-Effekte. Die Ausführungsgeschwindigkeit des Hostsystems sinkt mit steigender Emulationskomplexität des Originalsystems: „Bei der Datenbus-Genauigkeit kann die notwendige Rechenleistung [das] mehrere 1000-fache der Originalleistung betragen, um die gleiche Ausführungsgeschwindigkeit zu erhalten.“ [90f.] Dies kann dazu führen, dass das Zeitverhalten der Emulation erheblich von dem der Originalplattform abweicht. Um eine detailgetreue und komplexe Emulation historischer Systeme auf Datenbus-Ebene zu gewährleisten, muss das Hostsystem mit Fortschreiten der Computerentwicklung (und daraus hervorgehenden ‚neuen alten Computern‘ mit höherer Komplexität und Taktrate) ebenfalls sukzessive seinen Geschwindigkeitsvorsprung behalten oder ausbauen. (Loebel macht darauf aufmerksam, dass die Kontinuität des Moore‘schen Gesetzes damit eine notwendige Bedingung für die Nachhaltigkeit von Software-Emulation darstellt [60].)

4.3.1.3 Die Geschichte und (Weiter)Entwicklung von Emulatoren

Die theoretische Grundannahme der Emulation ist, dass Computer symbolische Maschinen im Sinne Turings sind und als solche von (anderen) Turing-Maschinen simuliert werden können [vgl. Turing 1947:112]. Diese Simulierbarkeit findet ihre Grenzen dort, wo Computer nicht mehr bloß als theoretische Konstrukte, sondern als reale Apparate verstanden werden, deren materielle Idiosynkrasien ebenso wie ihre internen Zeitverhältnisse mit-simuliert werden müssten – sie mithin also keine Objekte der theoretischen Informatik mehr sind. Die Diskurse zwischen theoretischer Beschreibung von Turingmaschinen und technischer Implementierung von Computern zeigen sich bereits in ihren Historien als getrennt.

Die praktischen Anfänge der Emulation mit Computern können bis 1941 zurückverfolgt werden: Bereits der Colossus-Computer kann als Emulator der Enigma-Chiffriermaschinen betrachtet werden, lag seine Aufgabe doch darin die Abläufe mehrer Enigma-Maschinen in hoher Geschwindigkeit nachzuvollziehen.[118] Da diese Emulation nicht interaktiv sein musste, sondern im Gegenteil, die Enigma-Funktionen im Brute-Force-Verfahren so schnell wie möglich nachahmen sollte, war Timing-Genauigkeit keine Anforderung an den Colossus. Erste intendierte Ansätze zur Emulation *obsoleszenter* Rechnerplattformen

117 Der Entwickler des Emulators JavaCPC versucht sukzessive sämtliche dieser Skeumorphismen in seiner Software zu Berücksichtigen: https://sourceforge.net/p/javacpc/news/ [letzter Abruf: 15.03.2019].

118 https://kaluszka.com/vt/emulation/history.html [letzter Abruf: 15.03.2019].

für neue Systeme beginnen Anfang der 1960er-Jahre[119], als IBM von den 70XX-Architekturen zur /360-Architektur wechselt und den Nutzern ersterer ermöglichen möchte die bereits angeschaffte oder sogar speziell für sie entwickelte Software weiter verwenden zu können.

Mikrocomputer wurden zunächst vor allem zur Software-Entwicklung emuliert, wobei Mini- oder Mainframe-Plattformen als Hostsysteme dienten. So entstand beispielsweise das Altair BASIC von *Micro-Soft* 1975 auf der Basis einer Intel-8080-CPU-Emulation mit einem PDP-10-Computer [vgl. James/Erickson 1992:81-83]. Diese musste allerdings weder zeitkritisch operieren noch alle maßgeblichen Bestandteile des Altair-8800-Computers emulieren (sie benötigte damit lediglich eine „instruction level accuracy"). Die Software-Emulationen von 8-Bit-Mikrocomputern in Hinblick auf „Ergebnisorientierung" beginnt Mitte der 1980er-Jahre auf den damals publizierten 16-Bit-Plattformen (Commodore Amiga, Atari ST, Apple Macintosh, IBM x86-Kompatible). Weil der Performance-Unterschied zwischen Host- und Originalplattform allerdings immer noch vergleichsweise gering war, wurden 8-Bit-Computer zumeist auf OS-Ebene emuliert, wie z.B. beim BBC Environment Emulator (1985)[120] oder verschiedenen MS-DOS-Emulatoren (bspw. für Amiga [vgl. Wüthrich 2007:17]). Dort, wo die CPU emuliert wird (STXformer emuliert Atari-8-Bit-Systeme [Mihocka 1987; 1988]), musste aus Performance-Gründen noch auf eine vollständige Emulation der Grafik- und Soundhardware verzichtet werden [1988:72]. Diese Phase der 8-Bit-Computer-Emulatoren ließe sich also als ‚Proof of Concept' verstehen, bei der es auch um den Erwerb von notwendigen Kenntnissen zur Emulatorprogrammierung ging.

8-Bit-Originalsysteme ohne derartige Zusatzprozessoren konnten auf Hostsystemen mit 32-Bit-Architektur bereits kurze Zeit später auch pin- und datenbusgenau emuliert werden. Dies geschah jedoch offenbar noch nicht aus Gründen der Preservation, denn die ‚Lebenszyklen' der Originalsysteme überlappten sich mit dem Beginn ihrer Emulation: Der ZX-Spectrum-Emulator SPECTRUM[121] für MS-DOS erschien 1989, der ZX-Spectrum-Computer selbst wurde allerdings noch bis 1992 produziert. Die Emulation des Systems sollte also vielleicht die Migration erleichtern (wie beim genannten IBM-Beispiel). Bereits zuvor wurde dieses System zu diesem Zweck in der Hardware seiner Nachfolgemodelle

119 In der Wikipedia findet sich eine umfangreiche Liste von Emulatoren mit Entstehungsdaten Host- und Originalsystemen: https://en.wikipedia.org/wiki/List_of_computer_system_emulators [letzter Abruf: 28.03.2019].

120 http://archive.retro-kit.co.uk/bbc.nvg.org/emulators.php3.html#Environment [letzter Abruf: 27.03.2019].

121 http://www.formauri.es/personal/pgimeno/spec/spec.html [letzter Abruf: 18.03.2019]. In der Readme-Datei spricht der Entwickler von noch früheren Protoversionen: „First official public version after VGASPEC was illegally spread (noone knew about EGASPEC which also existed initially until I joined them both in a unique program). Basic changes were: change in file format, a few bugs corrected and a proper menu screen with my name in it (this made me learn!)." [http://www.formauri.es/personal/pgimeno/spec/spec099f.zip [letzter Abruf: 18.03.2019].

emuliert: Von *Sinclair* erschienen zwischen 1984 und 1987 mehrere unterschiedliche Spectrum-Nachfolger, die jeweils einen optionalen Kompatibilitätsmodus beinhalten, der sich als Hardware-Emulator des Originalsystems bezeichnen ließe. Ein ähnliches Verfahren fand bei der Entwicklung des Homecomputers Commodore 128 (1985) statt, der eine Hardware-Emulation des Commodore 64 (1982) beinhaltete. Anders als Spectrum, das ein hobbyistisches Projekt war, waren diese Hardware-Emulationen professionellen Ursprungs.

Loebel weist in einer empirischen Bestandsaufnahme [129-143] nach, dass die meisten Emulatoren nicht aus professionellen, sondern hobbyistischen Entwicklungen stammen. Insbesondere Programme zur Homecomputer-Emulation, wie Spectrum, entstanden und entstehen zumeist in privaten Kontexten. Dies hat sowohl Einfluss auf ihre Qualität als auch ihren Status als Wissensobjekte (siehe Kap. 4.3.4). Da Emulatoren von Hobbyisten entwickelt werden, entfallen zumeist „wohldefinierte Standards und Arbeitsmethoden" [146] des Software Engineering. Programmiert werden Emulatoren zumeist in den Hochsprachen C und C++. Es finden sich zudem einigen jüngere Projekte, die in Java oder (speziell zum Betrieb in Webbrowsern) in JavaScript implementiert sind. In solchen Sprachen programmierte Emulatoren lassen sich zumeist ohne großen Aufwand für neue Hostsysteme kompilieren bzw. auf diesem ausführen. Dort, wo aus zeitkritischen Gründen Bestandteile von Emulatoren jedoch in Assembler programmiert wurden, führt dies bei der Portierung zu Problemen, sobald die Hostsysteme unterschiedliche Prozessoren und Peripherie-Bausteine besitzen.

Nicht erst die Implementierung in stark hardwareabstrahierenden Hochsprachen, sondern bereits die Kodierung von Hardwarefunktionen eines Originalsystems mittels symbolischer Programmiersprachen (Hoch- oder Maschinensprache) führt dazu, „dass diese [Emulatoren, S.H.] allein aufgrund ihrer technischen Beschaffenheit nicht in der Lage sind, jedwede theoretisch mögliche Software des Originalsystems abspielen zu können." [Loebel 2014:99] Dies zeigt sich umso mehr bei der Nutzung historischer Software, die spezifische Elemente der Hardware des Originalsystems (zum Beispiel analoge Komponenten) ausnutzt.

4.3.1.4. Fehler in Emulatoren

Die Tatsache, dass Emulatoren zumeist aus nicht-professionellen Kontexten stammen und damit nicht den üblichen Qualitätsstandards unterliegen, erweist sich als Problem für die Software Preservation. Programmierfehler in Emulatoren können nach zwei Arten klassifiziert werden [vgl. Herrmann 1999:25]: *errors of omission*, die sich zeigen, wenn es unterlassen wurde, bestimmte Elemente des Originalsystems zu implementieren, was beispielsweise auf „fehlende Systemspezifikationen sowie Missverständnissen bei der Anforderungsanalyse" [Loebel 2014:146] zurückgeführt werden kann, und *errors of commission*, welche Programmierfehler im engeren Sinne darstellen. Errors of omission

entstehen bei der Emulatorentwicklung zumeist aus Wissensdefiziten [vgl. Höltgen 2014c:297-300]; errors of commission verbleiben in Emulatoren aufgrund nicht ausreichender Programmierkenntnisse und fehlender Qualitätssicherung.

Führen die Fehler nicht zum Einfrieren oder Absturz des Emulators, erzeugen sie einen Output auf dem Hostsystem, der vom Output des Originalsystems auf unerwünschte Weise abweicht. In dem Moment, wo Emulatoren als ‚historische Werkzeuge' (z.B. im Museum) eingesetzt werden sollen, kann hierdurch ein ‚falscher Eindruck' von Geschichte entstehen. Aber nicht nur Fehler im Emulator, auch nicht berücksichtigte Fehler des Originalsystems führen zu derartigen Problemen: Insbesondere dort, wo die Ungenauigkeiten des Originalsystems nicht mit emuliert werden, können daraus anachronistische Annahmen über die Qualität (z.B. von Grafik oder Sound) historischer Plattformen resultieren. Darüber hinaus können noch unentdeckte Fehler im Originalsystem nur von Datenbus- und Pin-genauen Emulatoren mit emuliert werden – dann allerdings sogar, wenn sie vom Emulator-Entwickler unintendiert oder diesem sogar unbekannt sind [vgl. Loebel 2014:89f.].

Zwischen errors of omission und *intendierten Abweichungen* zwischen Emulatoren und Originalsystemen existiert eine fließende Grenze. Emulatoren werden häufig mit Funktionserweiterungen versehen. Solche Features, die im Originalsystem nicht vorhanden sind, reichen von der Möglichkeit, den operativen Status des emulierten Systems (etwa während eines laufenden Spiels) ohne Konsequenzen zu beschleunigen, zu pausieren oder als RAM-Image abzuspeichern, bis hin zur Erweiterung der historischen um moderne Schnittstellen, etwa um virtualisierte Datenträger via USB in den Emulator laden zu können. [Fernandez-Vara/Montfort 2013:5] sehen solche Abweichungen als möglichen Gewinn insbesondere für die computer(spiel)historische Lehre, da sie einen analytischen Zugang zur Spieloberfläche erlauben. Sie besitzen, wie sich weiter unten zeigen soll, jedoch auch einen epistemologischen Mehrwert.

Loebel diskutiert übersetzungsbedingte Abweichungen zwischen Original- und Hostsystem als „translation gap": Damit definiert er ein

> Maß für die Authentizität der Reproduktion eines dynamischen Objektes den Grad bzw. die Anzahl der Abweichungen." [104] „Das Translationsproblem ist jedoch eine systemimmanente Komponente der Emulationslösung. Je weiter sich die Schnittstellen bzw. Hardware des Hostsystems in ihren intrinsischen Eigenschaften von denen des Originalsystems unterscheiden, desto größer werden Abweichungen und eventuelle Verluste der Reproduktion. *Skeumorphismen* können diese nur bis zu einem gewissen Grad abmildern. [128 – Hervorh. i. O.]

Vor dem Hintergrund einer Software Preservation, die um (historische) Authentizität bemüht ist, muss *jede* Abweichung der Ausgaben des Hostsystems gegenüber dem Originalsystem als Fehler oder „Gap" gewertet und möglichst vermieden werden – insbeson-

dere dann, wenn nur ein einziger Emulator für ein Originalsystem existiert, so dass keine Vergleichsmöglichkeiten vorhanden sind. Hier deutet sich bereits an, dass die computerarchäologische Perspektive auf das Phänomen Emulation anders ausfallen wird: Reale historische Computer lassen sich aus einer grundsätzlichen Erwägung nicht authentisch emulieren: Denn bei der „Übersetzung“ [41] von Hardware in Software vollzieht sich ein *kategorialer* Zeichenwechsel [vgl. Holenstein 1992:16f.], der die jeweiligen Eigenschaften des *Realen* (im Sinne Peirces‘ [1983:64ff.] indexikalischem Zeichen) und des *Symbolischen* als unmöglich verlustfrei ineinander übersetzbar kennzeichnet.

Diese Lücke ist jedoch einzig für virtuelle Plattformen überbrückbar, wie beispielsweise Computer-Implementierungen in Minecraft oder „Game of Life“ (vgl. Kap. 4.2.6.3). Wie sich diese Lücke in der Praxis darstellt, wird sich am Beispiel der Neuentwicklung eines Retro-Computerspiels verdeutlichen.

4.3.2 Retrogame Development

Loebels Erhebung zufolge wird der

> größte Anteil verwendeter Emulatoren von einer am Thema ‚retro-computing‘ interessieren Gemeinde von technisch versierten Hobbyisten entwickelt. Für diese Gruppe dienen Emulatoren dazu, obsolete Video- und Computerspiel-Plattformen auf modernen Computersystemen wieder erlebbar zu machen. [...] Dabei stehen der nostalgische Aspekt und das Interesse an alten Computerplattformen im Vordergrund. [135].

Aus diesem Grund seien Emulatoren, wie bereits erwähnt, zumeist rückwärtsgerichtet, um historische Spiele für historische Computersysteme spielbar zu machen. Das Phänomen Retro-Gaming ist allerdings facettenreicher, wie Tabelle 4.3.1 zeigt. So werden nicht nur historische Spiele für neue Plattformen portiert und neue Spiele, die sich einer spezifischen Retro-Ästhetik bedienen für aktuellen Systeme entwickelt [vgl. Beil 2013]; es finden auch Neuentwicklungen von Spielen für alte Computer statt. Hierbei werden nicht selten aktuelle Spieltitel einem ‚Downgrade‘ unterzogen, um sie auf 8- und 16-Bit-Homecomputern lauffähig zu machen.

	Aktuelle Systeme	**Historische Systeme**
Aktuelle Retro-Spiele	La-Mulana[122]	Flappy Bird[123]
Historische Spiele	Giana Sisters[124]	Prince of Persia[125]

Tabelle 4.3.1: Retrogaming

122 https://la-mulana.com/en/introduction/system [letzter Abruf: 18.03.2019].

123 https://hackaday.com/tag/flappy-bird/ [letzter Abruf: 18.03.2019].

124 https://de.wikipedia.org/wiki/Giana_Sisters:_Twisted_Dreams [letzter Abruf: 18.03.2019].

Hierzu zählt auch der Flappy-Bird-Clone für die VC-4000-Spielkonsole. Das Original des Spiels wurde am 23./24.05.2013 von *dotGears* für Mobilsysteme auf Basis von iOS und Android vorgestellt. Obwohl das Spiel vom Entwickler wenig später (09.02.2014) wieder aus den Online-Stores entfernt wurde, ist dieser Titel seither für zahlreiche aktuelle und historische Plattformen portiert worden.

4.3.2.1 Emulation vs. Originalsystem bei der Spielentwicklung

Der Programmierer Martin Wendt (Szene-Pseudonym „enthusi“) entwickelt seit Anfang der 1990er-Jahre Spiele für (damals noch nicht) historische Heimcomputer. Hierzu zählen neben PC-Plattformen der C64, Atari VCS/2600, Atari Lynx, Watara Supervision, Sinclair ZX Spectrum und Nintendo Gameboy Advance – mit Ausnahme der PCs und des Sinclair-Computers also für MOS-6502-basierte Systeme in Assembler.

Wendt beschreibt den Entwicklungsprozess, wie er für die meisten Retrocomputing-Homebrew-Projekte typisch sein dürfte:

- Entwicklung der Source Codes (für 8-Bit-Systeme zumeist in Assembler) in einem Texteditor unter einem modernen Betriebssystem
    - Falls Vorhanden, Nutzung einer IDE für die Zielplattform (vgl. Abb. 4.3.1)
- Assemblierung mit Hilfe eines Cross-Assemblers für den jeweiligen Prozessor
- Test der Objektdatei auf syntaktische und semantische Programmierfehler in einem Emulator
- Übertragung der Objektdatei auf einen modernen Massenspeicher (zumeist Flashspeicher)
    - Seltener (falls moderne Speicherlösung nicht vorhanden ist) Brennen des Objektcodes auf EPROMs oder ähnliches
- Test auf Timinggenauigkeit und korrekte/intendierte Hardware-Features auf dem Originalsystem
- Debugging in der Programmierumgebung (oder im Emulator, falls vorhanden)

Im Interview (vgl. Anhang D) zeigt sich, dass der Entwicklungsprozess eines Spiels aus der Retrocomputing-Szene deutlich von den Maßstäben modernen Software Engineerings [vgl. Sommerville 2012] abweicht. Am ehesten entspricht die von Wendt dokumentierte Entwicklungsarbeit dem historischen „Wasserfallmodell“ [ebd.:56f.]:

1. Anforderungsanalyse (Programmierwettbewerb für den C64),

125 https://popc64.blogspot.com/2011/10/prince-of-persia-for-commodore-64128.html [letzter Abruf: 18.03.2019].

2. System- und Softwareentwurf (Idee für einen Hack und Definition von künstlichen Limitierungen),
3. Implementierung der Module (Rollenverteilung für die und Umsetzung der Engine-, Grafik- und Soundprogrammierung),
4. Testphase (zum Wettbewerbsende testen die Entwickler und „ein professioneller Spieletester“ das Spiel) und schließlich
5. Auslieferung, Betrieb und Wartung (Versioning nach Nutzerfeedback, das in eine Tank you!-Edition mündete.

In der Planungsphase stand aber nicht ein konkretes Problem, zu dessen Lösung ein Programm entwickelt werden sollte, im Vordergrund, *sondern eine spezifische Hacking-Idee*, die dann durch eine Programm-Idee umgesetzt wurde (welche ein Spiel oder eine Demo sein kann). Ähnlich, wie in den Anfangszeiten der Heimcomputer-Ära, als „einzelne Programmierer ihre Spiele oftmals in Heimarbeit entwickelten und sie sich [erst] einige Jahre später zu kleinen Teams von maximal drei Mitgliedern zusammenschlossen“ [Keichel 2017:129-131], entstehen auch Spiele und Demos in den Retrocomputing-Szenen. Bereits damals wurde auf bekannte professionelle Entwicklungsmethoden (im Sinne des damaligen Software Engineerings: diagrammatische Vorentwürfe, strukturierte Programmierung, Projekt-Management usw. [vgl. Hering 1992; Sommerville 2012:88]) nicht zurück gegriffen. Diese Vorgehensweisen waren damals durch die gerade im entstehen befindliche Computerspielindustrie (deren Firmen nicht selten selbst von Homecompter-Hackern gegründet wurden [vgl. Wiltshire 2015:29-112]) und heute durch die hobbyistische Unabhängigkeit der Entwickler begründet. Die Entwicklung von Retro-Spielen verfolgt nur in Ausnahmefällen von Beginn an kommerzielle Ziele; die meisten dieser Programme sind nach der Veröffentlichung gratis aus dem Internet zu beziehen.[126]

Im Interview zeigt sich, dass Emulatoren die bezogenen Originalplattformen beim Test neuer Software kaum ersetzen können. Oft sind Hardware-Features in ihnen nicht oder nicht adäquat abgebildet und erzeugen dann bei hardwarenaher Programmierung Timingprobleme. Wendt bestätigt die rückwärts gerichtete Tendenz von Emulatoren: Am Beispiel der Atari-Lynx-Emulatoren zeige sich, dass „der Fokus ganz klar darauf [lag], den vorhandenen Softwarekatalog zu emulieren und weniger die Hardware selbst vollständig abzubilden.“ [Anhang D]

Die Tatsache, dass Wendt nicht nur spezifische selbstentwickelte Speicherlösungen nutzt, sondern sogar eigene Emulatoren entwickelt, um sie für die Spielprogrammierung einzusetzen, zeigt wie ‚eng‘ diese Softwaregattungen (Emulatoren und Retro-Games) miteinander verbunden sind. Die Emulatorentwicklug zehrt gar von den errors of omission, die

126 Das von Wendt im Interview erwähnte Caren and the Tangled Tentacles ist gratis unter diesem Link zu beziehen: https://csdb.dk/release/?id=156948 [letzter Abruf: 04.07.2019].

von Retrospiel-Programmierer entdeckt werden: „Die Emulationsentwickler sind sehr dankbar für Software, die Schwächen aufzeigt oder überprüfbar macht. [... Es ist ein] konstruktiver Wettlauf zwischen Demo-Programmierern und Emulator-Entwicklern“ [Anhang D]. Dieser Eindruck konnte bei der Entwicklung von Flappy Bird bestätigt werden.

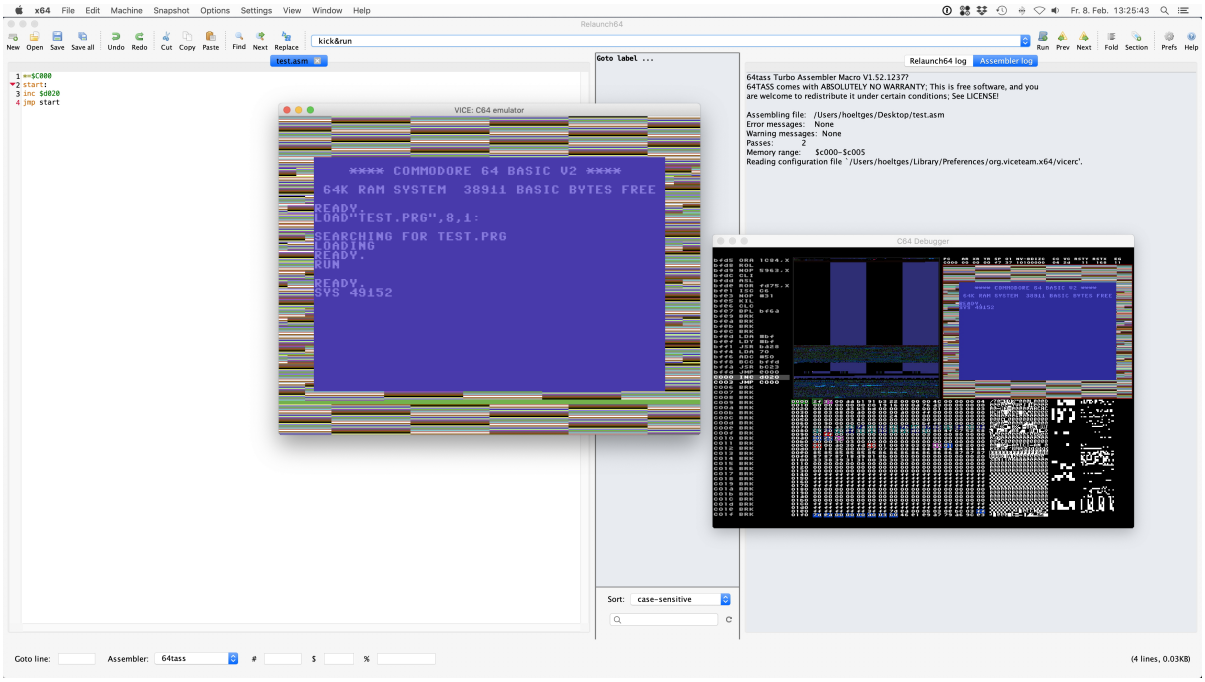


Abb. 4.3.1: Entwicklungsumgebung Relaunch64 und C64Debugger für den Commodore 64

4.3.2.2 Flappy Bird

Das Casual Game Flappy Bird hat sehr früh Reimplementierungen für historische Homecomputer provoziert.[127] Dies mag neben seiner Grafik, die im Amiga-Stil bereits als Retro-Ästhetik [vgl. Camper 2009] bezeichnet werden kann, vor allem am für historische Action-Games typischen Gameplay liegen: Eine von links nach rechts scrollende Szenerie, durch welche die Spielfigur (ein Vogel) in Seitenansicht gesteuert werden muss. In unregelmäßigen Abständen versperren vertikale Hindernisse den Weg, die jedoch mehr oder weniger enge Durchlässe enthalten, durch welche der Vogel hindurch gesteuert werden kann. Die Spielfigur muss durch Antippen des Bildschirms mit Auftrieb versorgt werden, denn die Gravitation zieht sie beständig in Richtung Boden. Eine Kollision mit dem Boden oder mit einem der vertikalen Hindernisse führt zum Verlust einer Spielfigur. Die Aufgabe des Spielers besteht also darin, den Vogel sowohl in der Luft zu halten als auch ihn dabei durch die Durchlässe in den Hindernissen zu manövrieren. Der sukzessive verringerte Abstand der aufeinanderfolgenden Hindernisse (mit jeweils unterschiedlich darin positionierten Durchlässen) sowie die steigende Geschwindigkeit, mit der diese von rechts nach links über den Bildschirm scrollen, bildet die Schwierigkeitskurve von Flappy

127 http://www.retrocollect.com/News/complete-list-of-retro-gaming-ports-of-flappy-bird.html [letzter Abruf: 22.03.2019].

Bird. Der Fortschritt wird anhand der durchflogenen Hindernisse gezählt und gewertet. (Vgl. Abb. 4.3.2)

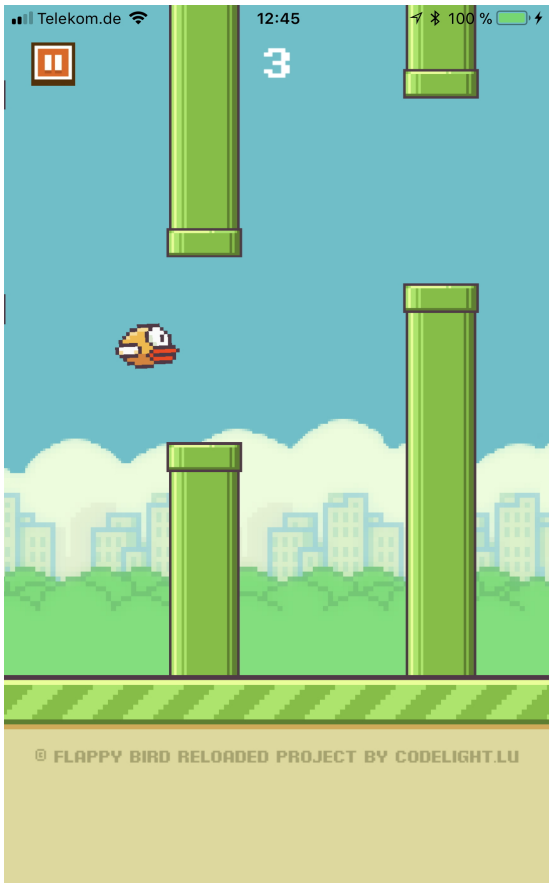


Abb. 4.3.2: Screenshot Flappy Bird (iPhone-Version)

Die möglichst genaue Beschreibung des Spiels (seiner Grafik- und Soundausgaben, seiner Steuerung, der Spielphysik und des Gameplays) standen am Anfang des historisch ‚inversen' Re-Enactments (aktuelles Programm für eine historische Plattform). Da der Source Code nicht verfügbar war, musste das Spiel auf Basis äußerlicher Beschreibungen sowie durch Analysen der (quelloffenen) Codes bereits erfolgter Adaptionen für andere 8-Bit-Systeme programmiert werden. Die spezifischen technischen und ästhetischen Rahmen/Grenzen der Zielplattform stellten dabei die besondere Herausforderung dar:

- Der Objektcode des Spiels sollte maximal 4 KiB[128] groß sein, damit das Spiel auf ein EPROM gebrannt und dieser gegen das ROM eines vorhandenen Spiels ausgetauscht werden konnte (vgl. Abb. 4.3.3).
- Das Spiel durfte keinen zusätzlichen RAM-Speicher benötigen, der über die 37 Bytes, die im Programmable Video Interface (PVI) verfügbar sind [Signetics 1981:1], hinausgeht.

128 Dies richtet sich nach den Speichergrößen der EPROM-Bausteine, die in Zweierpotenzen (KiB) und nicht in Zehnerpotenzen (KB) angegeben sind. 4 KiB = 4,096 KB stellt die Standardgröße der kommerziellen ROM-Module dar. Lediglich eines dieser Module besitzt 6 KiB ROM. Dieses und drei weitere haben noch zusätzlichen RAM-Speicher integriert [vgl. Maibaum 2015:2,8,24-30].

- Die Farb- und Soundausgaben des Spiels mussten an die vorhandenen Farb- und Frequenzumfänge des Systems angepasst werden. [Jacobs 2007]
- Die Steuerung sollte über einen der der Konsole beigefügten Controller erfolgen können. (Dieser Controller besitzt neben den diskret-elektronisch verwalteten Tastern einen analogen Steuerungshebel.)

Die Kursteilnehmer wurden in Teams aufgeteilt, die mit (a) der Grafikprogrammierung, (b) der Soundprogrammierung, (c) der Controller-Abfrage, (d) der Implementierung der Spielphysik und (e) der Koordination der main loop, in der die Initialisierung vorgenommen, die Interrupts verwaltet und die Ergebnisse der vier anderen Gruppen integriert werden sollten. Auf diesem Weg konnte das Spiel nach dem Paradigma der modularen Programmierung [Grechenig u.a. 2010:198] auch dezentral (außerhalb der Lehrveranstaltung) weiterentwickelt werden. Als Informationsgrundlage wurden unterschiedliche Quellen genutzt:

1. Unterlagen zur Architektur und Programmierung von Signetics-2650-Systemen [Signetics 1978a] sowie zum PVI-Chip Signetics 2636 [Signetics 1981].
2. Unterlagen zu Systemspezifikationen der VC-4000-Spielkonsole [Jacobs 2007]
3. Source Codes und Codefragmente von VC-4000-Spielen (zur Verfügung gestellt vom Entwickler Hans-Heinz Bieling[129])

Zur Programmierung wurde folgendes Setting eingesetzt:

- Der Source Code wurde in einem Texteditor (Notepad++ unter Windows 7) eingegeben.
- Als Assembler wurde der VACS 2650 Cross Assembler (Version 1.2x[130]) verwendet
- Als Emulator (der auch einen Assembler sowie Elemente eines Debuggers enthält) zum Testen der Software diente WinArcadia (in der Version 24.21[131])
- Zum Testen der Software auf der Originalplattform wurde eine Steckmodul-Platine, von der der Original-ROM-Baustein entlötet und durch einen IC-Sockel ersetzt wurde, benutzt. Die Software wurde mittels eines EPROM-Brenners auf einen passenden ROM-Baustein (M2716 mit 16 KiB maximaler Kapazität) geschrieben und

129 Diese Dokumente sind gescannt im Moodle-Kurs zum Projektseminar hinterlegt: https://moodle.hu-berlin.de/course/view.php?id=55020 [letzter Abruf: 18.03.2019].

130 https://amigan.yatho.com/vacs124b.zip [letzter Abruf: 18.03.2019].

131 https://web.archive.org/web/20150909081016/http://amigan.1emu.net:80/releases#amiarcadia [letzter Abruf: 15.03.2019]. Hier sind sowohl das Binary-File als auch die Source Codes abrufbar. Auf diese Version bezieht sich der folgende Text.

konnte danach mit einem UV-Löschgerät wieder davon entfernt werden. (Abb. 4.3.3)

Abb. 4.3.3: Setup zum Schreiben/Löschen der Spiel-Software auf EPROM

Bei der Implementierung wurde in einigen Aspekten vom Originalkonzept Flappy Birds abgewichen: Die Farben wurden an die Farbpalette des PVI-Chips angepasst (mit dunkelblauem Himmel und den drei Farben Hell-Gelb, Hell-Rot und Hell-Blau für den Vogel), es wurde eine timergesteuerte Hintergrundmusik hinzugefügt (nach der Vorlage Close to you der Band *The Carpenters* aus dem Jahr 1971) und als Score wurden die verstrichenen Spielsekunden gezählt. Die Grafik der Hindernisse sollte anders als im Originalspiel nicht aus flächig/farbig gefüllten Elementen bestehen, sondern die spezifischen eingefärbten „Grids" [Jacobs 2007] des PVI-Chips verwenden. Zur Steuerung (dem Auf- und Abflug des Vogels) sollte eine beliebige Taste auf dem VC-4000-Controller verwendet werden.

Am Ende des Kurses waren die Programmteile für die Vogel-Grafik und ihre Animation[132], die Steuerung des Vogels über den Controller und die Musik für eine Hintergrundmelodie sowie die Punktezählung[133] fertig programmiert. Es fehlte vor allem noch die Hindernis-Grafik (die Grids). Die Entwicklung war ins Stocken geraten, weil sich insbesondere die Grafik- und Soundausgaben im Emulator und auf der Originalplattform maßgeblich voneinander unterschieden: Der Sound wurde im Emulator stark verrauscht ausgegeben, war auf der VC-4000 jedoch klar zu hören; die Grafik, insbesondere die Farbgebung des Hintergrundes (intendiert war „dark blue" [Jacobs 2007], auf der Konsole wurde jedoch „black" angezeigt) unterschieden sich voneinander. Überdies erschienen in der Bildschirmmitte Elemente des Grafik-Grid, die gar nicht programmiert worden waren. (Vgl. Abb. 4.3.4)

132 https://vimeo.com/99324414 [letzter Abruf: 18.03.2019].

133 https://vimeo.com/106671850 [letzter Abruf: 18.03.2019].

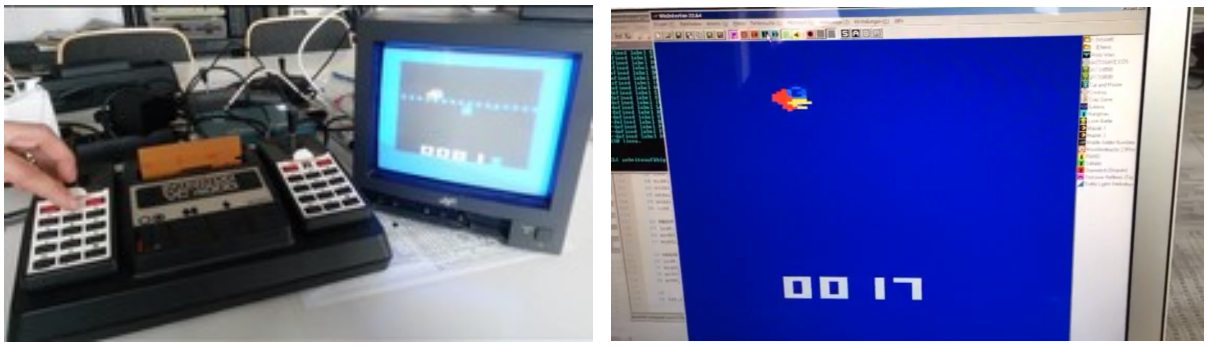


Abb. 4.3.4: Fehlerhaftes Flappy Bird auf der VC-4000[134] (links) und im Emulator (rechts)

Befürchtend, dass bei der weiteren Programmierung insbesondere der zeitkritischen Grafik-Elemente zusätzliche Fehler des Emulators zutage treten könnten, und weil jeder Softwaretest auf der Originalplattform circa 30 Minuten Vor- und Nachbereitungszeit in Anspruch nahm (EPROM brennen, einbauen und für den nächsten Test löschen), wurde die Spielprogrammierung ausgesetzt und die Entwicklung des Flash-ROM-Moduls vorgezogen. Dessen Verwendung sollte dem Problem der langwierigen Testvorbereitung entgegenwirken. Unterdessen bestand ein fortgesetzter E-Mail- und Facebook[135]-Kontakt mit dem Emulator-Entwickler James Jacobs [vgl. Anhang E]. Dieser hatte bereits Anregungen zur Erweiterung seiner Software aus vorangegangenen Kursen erhalten (vgl. Kap. 4.2.3.2). Zur Vorbereitung der VC-4000-Programmierung wurden ihm vom Kursleiter eine VC-4000-Konsole sowie gescannte Materialien aus deutschen Archiven zur Verfügung gestellt. Auf das Fehlerprotokoll reagierte Jacobs in der Facebook-Gruppe wie folgt:

> 1. The sound issue will be fixed for the next version.
> 2. The console is expecting SCORELT/SCORERT to be written during the frame, not during vertical blank; the emulator is more tolerant of when it is written.
> 3. Memory at $1F80..$1FAC holds the background grid definitions; the game needs to clear this at boot.[136]

Hans-Heinz Bieling, der Anfang der 1980er-Jahre die Spiele Space Laser und Motocross für die VC-4000 entwickelt hatte [vgl. Othmer 2013] und ebenfalls Mitglied der Facebook-Gruppe war, trug zu einem Workaround des zweiten Problems folgendes bei: „Der Timer muss während des Bildaufbaus geschrieben werden. Je nach Timing kann man so oben und unten einen Timer mit verschiedenen Inhalten laufen lassen.“[137] Dies gab bereits einen Hinweis darauf, dass hier eine fehlerhafte Implementierung des Bildaufbaus vor-

134 https://www.facebook.com/jmaibaum/videos/764307840274237/?comment_id=643746732409345 [letzter Abruf: 18.03.2019].

135 Unter https://www.facebook.com/groups/392123604238327/ [letzter Abruf: 18.03.2019] wurde eine geschlossene Seite zur Kommunikation der Kursteilnehmer und Interessierten (zu denen auch der Emulator-Entwickler und ein zeitgenössischer VC-4000-Software-Entwickler gehörten) etabliert.

136 https://www.facebook.com/groups/392123604238327/permalink/643611902422828/?comment_id=643746732409345 [letzter Abruf: 15.03.2019].

lag, aus der die Fehlanzeige des Spiel-Scores resultierte. Der Emulator generierte das Bild nicht Rasterzeilen-genau, sondern Frame für Frame und konnte dadurch weniger timing-genau programmiert werden als der PVI in der Originalplattform.

Indem Emulator-Entwickler Jacobs in seinem Posting Hinweise für die Umgehung der Probleme 2 und 3 gab, wies er implizit darauf hin, dass er sich dieser errors of omission bewusst war. Maibaum ergänzt hierzu: „Mehr noch, im Falle von Punkt 3 verzichtet er bewusst darauf, den Implementierungsfehler – im Sinne einer genaueren oder *vollständigeren* Emulation – zu beheben, die beobachtete ‚Translation Gap' bleibt in diesen Fällen also bestehen." [Maibaum 2015:4f – Hervorh. i. O.]

Der Fehler lässt sich wie folgt erklären: Die VC-4000 verfügt lediglich über 37 Byte RAM-Speicher, welcher in Form von Flip-Flops (SRAM) innerhalb des PVI-Chips verbaut ist. Hier dient er an den Adresse $1F0E_{16}$, $1F0F_{16}$, $1F1E_{16}$, $1F1F_{16}$, $1F4E_{16}$-$1F6D_{16}$, $1FAD_{16}$ vor allem zum Hinterlegen der Werte für die Grid- und Hintergrundgrafik, kann aber auch als Scratchpad (User-RAM) genutzt werden, wie der „Programming Guide" anmerkt:

> Vertical and horizontal grid registers are of course usable as ordinary user RAM by games which do not use the grid (such games do not set the ‚grid/background enable' flag in BGCOLOUR, or set the grid colour to be the same as the screen (background) colour). [Jacobs 2007]

Wird das BGCOLOUR-Flag gesetzt, dann wird der Inhalt des PVI-RAM ausgelesen und für die Grid-Darstellung genutzt. Da sich Flip-Flops nach dem Einschalten des Systems in einem nicht-definierten Zustand befinden [Holcomb 2009] (vgl. Kap. 4.4.3.2), kann es daher zur ungewollten Anzeige von Grid-Elementen kommen. Dies wird verhindert, indem die einzelnen Zellen des PVI-RAM durch eine kleine Routine *innerhalb des Spiel-Programmcodes* auf den Wert 0 initialisiert werden.

Maibaum analysiert in seiner Arbeit [5], dass Jacobs diesen Software-Prozess *in seinem Emulator* als Hardwarefunktion der Plattform selbst vorwegnimmt, indem er folgende Routine[138] einfügt:

```
for (i = 0; i <= 0x7FFF; i++)
{       memory[i] = 0;
}
```

Diese Schleife initialisiert gleich nach dem Start des Emulatorprogramms (also quasi nach dem ‚Einschalten' der virtuellen Spielkonsole) den gesamten adressierbaren Speicher mit Nullen. Dies führt dazu, dass es den Programmierern im Kurs nicht auffallen konnte, wenn das BGCOLOUR-Flag gesetzt war, weil das PVI-RAM durch die Initialisierung keine zufälligen Grafikinformationen mehr enthielt. Erst auf der VC-4000-Konsole

137 https://www.facebook.com/groups/392123604238327/permalink/643611902422828/?comment_id=649120608538624 [letzter Abruf: 15.03.2019].

138 Vgl. [Maibaum 2015:5] sowie im Sourcecode interton.c, Zeilen 48-50.

mit ihrem nicht-initialisierten PVI-RAM konnte dieser Fehler sichtbar werden. Bei Originalsoftware wird die Initialisierung der Adressen $1F00_{16}$-$1FFF_{16}$ meist zu Beginn des Gamecodes vorgenommen, wie das Codebeispiel aus dem Disassembly des Spiels Labyrinth[139] zeigt:

```
[…]
;$0004:
        strz    r3              ;r3 = r0;                                ;2,1 $0004:        C3
;$0005:
        stra,r0 $1F00,r3-       ;*($1F00 + --r3) = r0;                   ;4,3 $0005..$0007: CF 5F
00
        brnr,r3 $0005           ;if (r3 != 0) goto $0005; // ROM         ;3,2 $0008..$0009: 5B 7B
[…]
```

Inzwischen wurde dieser Aspekt des PVI-RAM im Emulator berücksichtigt; in der Version 25.12 (vom 13.03.2019) von WinArcadia ist der folgende Code für die Initialisierung des RAM-Speichers enthalten, der nun Zufallswerte zwischen 0 und 255 in die Adressen schreibt:

```
for (i = 0; i <= 0x7FFF; i++)
{   memory[i] = rand() & 0xFF; // ie. rand() % 256
}
```

Die `rand()`-Funktion von C erzeugt allerdings lediglich Pseudozufallszahlen von weniger guter Qualität [vgl. Kernighan/Ritchie 1990:161f.] mittels eines *linearen Feedback-Bit-Shifts (LFBS)* [vgl. Braguinski 2018:196-213], wohingegen sich die Initialwerte einer SRAM-Speicherzelle entropisch sind [vgl. Holcomb 2009:1204f.]. Der (Weiter-)Entwickler von Flappy Bird hat sich dieser Zufallszahlen bedient, indem er nach dem Start des Programms[140] die PVI-RAM-Adressen $1F0E_{16}$ und $1F0F_{16}$ ausliest:

```
;User Variables
;ok if these are unreliable, then there's not 2 bytes RAM.
;bool crash ;kann man ins MSB von ifc tun
;bool egg; kann man ein Bit drunter tun
RANDOMLSB       equ $1F0E
RANDOMMSB       equ $1F0F
```

Der Programmierer deutet (in der zweiten Kommentarzeile) an, dass sein Zufallszahlengenerator als Indikator dafür benutzt werden kann, ob das Programm auf einer realen VC-4000 oder einem Emulator läuft. Die Nutzung dieser echten Zufallswerte als Seed für unterschiedliche Zwecke innerhalb des Spiels ist überdies platzsparender als ein programmierter LFBS-Algorithmus. Die Emulation solcher echter Zufallszahlen gehört zu jenen *Übersetzungsunmöglichkeiten* zwischen realem Originalsystem, welches aufgrund seiner Hardwareidiosynkrasien nicht-deterministische Elemente enthalten kann, und dem symbolisch kodiertem Emulator, der als Software immer schon eine deterministische Turing-Maschine darstellt:

139 Das Disassembly labyrinth.asm ist enthalten im Verzeichnis autodis/interton/ des Pakets http://amigan.yatho.com/autodis.rar [letzter Abruf: 20.03.2019].

140 Der Sourcecode findet sich unter: http://tempect.de/senil/progra/FLAPPY2.ASM [letzter Abruf: 21.03.2019].

> Wenn ‚Zufallszahlen' nach einem bestimmten Algorithmus ermittelt werden, können sie nicht mehr dem Zufall unterliegen, denn sie sind berechenbar und somit voraussagbar. Die Besonderheit der vom Computer erzeugten Zufallszahlen liegt nun darin, daß der Benutzer der Zufallszahlen, der den Algorithmus nicht kennt, die erzeugten Pseudo-Zufallszahlen als wirkliche gleichwahrscheinlich verteilte Zufallszahlen auffassen kann. Dies liegt daran, daß die Perioden, mit denen die Pseudo-Zufallszahlen wiederholt werden, sehr groß und die Korrelation zwischen den erzeugten Pseudo-Zufallszahlen genügend klein ist. [Schneider 1982:83]

4.3.3 Das VC4000MultiROM

Der Einsatz eines Flash-ROM-Moduls sollte die Tests der Spielsoftware erheblich verkürzen, weil die assemblierten Programm(teil)e nun am Windows-Computer auf einen Flashspeicher (SD-Karte oder USB-Stick) geschrieben und über das Modul direkt in die VC-4000 geladen werden konnten. Das zeitaufwändige Brennen und Löschen von EPROM-Bausteinen würde damit entfallen.

Derartige Hardware-Lösungen existieren für zahlreiche historische Computer- und Computerspiel-Plattformen. Sie zählen zu den populärsten Projekten der Retrocomputing-Szenen, weil sie es ermöglichen im Internet archivierte virtualisierte Software (vor allem Spiele) historischer Systeme nutzbar zu machen.[141] Zunächst wurden hierfür so genannte *Image*-Formate für Kassetten, Disketten, Programmdateien u. ä. entwickelt, um historische Software in Emulatorprogramme laden zu können. Bereits hierzu wurden spezielle Adapter-Peripherien konstruiert, welche historische externe Massenspeicher an moderne Computer anschlussfähig machten.

Geräte wie zum Beispiel ZoomFloppy[142] (Abb. 4.3.5, rechts) ermöglichen den Anschluss eines historischen Diskettenlaufwerks an die USB-Schnittstelle eines modernen Computers, um so Inhalte physikalischer Datenträger virtualisieren und auf Massenspeichern des modernen Systems in Form einer Image-Datei speichern zu können.[143] In entgegengesetzter Richtung lassen sich mit solchen Geräten aber auch Inhalte von Datenträger-Images wieder auf Datenträger zurückschreiben, um sie so auf den historischen Plattformen über deren zeitgenössische Massenspeicher laden zu können. Mit Hilfe von ZoomFloppy können unterschiedliche Diskettenlaufwerke für Commodore-Heimcomputer an den USB-Anschluss eines Windows- oder Macintosh-Systems angeschlossen werden. Zur Kommu-

141 Ein solches Archiv für VC-4000-Spiele findet sich unter https://amigan.yatho.com/games.rar [letzter Abruf: 18.03.2019].

142 https://www.c64-wiki.de/wiki/ZoomFloppy [letzter Abruf: 18.03.2019].

143 Von besonderer Bedeutung für die Software Preservation sind hier Systeme wie KryoFlux (https://www.kryoflux.com/ [letzter Abruf: 11.01.2019]), die ungeachtet des vorliegenden Datenträgerformates und Filesystems Daten bitweise vom Datenträger lesen und in eine Datei schreiben.

nikation zwischen Computer und Laufwerk existieren spezifische Tools, mit denen Datenträger gelesen, geschrieben, formatiert und geprüft werden können.[144] Für zahlreiche historische Computersysteme bzw. ihre Massenspeicher (Disketten, Kassetten, ROM-Module, Mikrokassetten, Lochbänder, Lochkarten etc.) sind solche Adapter verfügbar und werden unter anderem auch zur Software Preservation eingesetzt.

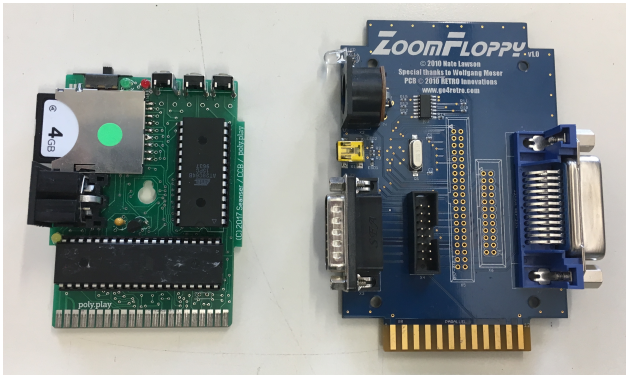


Abb. 4.3.5: MSD2IEC (links) und ZoomFloppy (rechts)

Den entgegengesetzten Weg gehen Erweiterungen, die die Verwendung *moderner Massenspeicherlösungen (zumeist als USB-Sticks, SD- oder CF-Karten) an historischen Computern* erlauben, wie zum Beispiel das MSD2IEC[145] (Abb. 4.3.5, links). Solche Peripherie-Geräte vereinfachen nicht nur die umweglose Nutzung von im Internet verfügbaren Datenträger-Images am Originalgerät, sondern auch die Entwicklung von Homebrew-Software im Cross-Platform-Development-Verfahren, weil sie die Verfügbarkeit und Vorzüge von heutigen Massenspeichern für historische Systeme eröffnen [vgl. Anhang D]. Wurden diese Konverter zunächst als Module für Spielkonsolen entwickelt (um alte und neue Spiele auch massenweise auf einem Datenträger verfügbar zu machen), so hat sich die Entwicklung bald auch auf Computersysteme ausgedehnt.

Für die Interton VC-4000 existierte zum Zeitpunkt des Projektseminars noch kein solches Modul. Die Entwicklung eines solchen wurde im Frühjahr/Sommer 2015 von den Kursteilnehmern Mario Keller und Johannes Maibaum abgeschlossen. (Abb. 4.3.6) Letzterer dokumentierte den technischen Entwicklungsprozess sowie die damit verbundenen medienwissenschaftlichen Implikationen in einer umfangreichen Projektarbeit [Maibaum 2015], die er im Rahmen seines Masterstudiums ablegte. Das Design des fertigen Moduls (Platinenlayout, Bestückung) sowie die zugehörige Software sind seitdem online verfügbar[146] und haben der Community als Grundlage für Weiterentwicklungen[147] und Spieleprogrammierung[148] gedient.

144 https://opencbm.trikaliotis.net/opencbm.html [letzter Abruf: 19.03.2019].

145 https://www.c64-wiki.de/wiki/MSD2IEC [letzter Abruf: 18.03.2019].
146 https://github.com/mkeller0815/VC4000MultiROM [letzter Abruf: 18.03.2019].

147 Inzwischen ist ein Multi-ROM-Modul, das zahlreiche Spiele für die Konsole auf einem Baustein enthält, erhältlich: http://atariage.com/forums/topic/259234-multi-cartridge-for-interton-vc4000-video-computer/?p=3707455 [letzter Abruf: 18.03.2019].

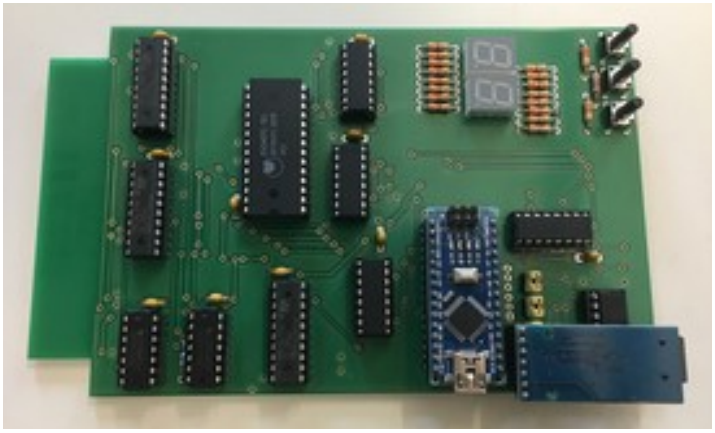


Abb. 4.3.6: VC4000MULTIROM

Die Projektarbeit erbrachte (neben der Entwicklung der Hard- und Software) Erkenntnisse über den Nutzen praktischer Tätigkeiten innerhalb eines (medien)theoretisch ausgerichteten Studiums. Erstens ergab sich hier die Möglichkeit zuvor erworbenes Theorie-Wissen in seiner praktischen Anwendung zu validieren. Hierbei ging es Maibaum etwa um die Frage, ob der Anspruch der vor drei Jahrzehnten inaugurierten Medienarchäologie gültig geblieben ist: dass sich durch die praktische Beschäftigung mit technischen Artefakten Wissen erwerben lässt, welches über die bloß diskursive Beschreibung von Technik hinausgeht – mit Friedrich Kittler gesprochen: dass „eine Maschine [...] also zugleich weniger und mehr [kann], als ihre Datenblätter zugeben“ [Kittler 1993a:221, zit. n. Maibaum 2015:40]. Zweitens konnten beim Entwurf und der Konstruktion des Moduls neue „[m]edientheoretische Erkenntnisse durch aktives Retrocomputing“ [ebd.:41] gewonnen werden – etwa in Hinblick auf die Frage nach der Motivation für derartige Projekte, die Maibaum auch jenseits der „Nostalgie der Beteiligten“ [41] feststellte: Einige elektronische Komponenten des Flash-ROM-Moduls verkörpern bereits selbst ein „aktives Retrocomputing der Halbleiterindustrie“ [42], so etwa der Pegelwandler-Baustein SN74LCV125A von *Texas Instruments*, der zwischen der ‚historischen‘ TTL-Spannung von 5 Volt und der modernen von 3,3 Volt vermittelt und so die unterschiedlichen Generationen von TTL-Bauteilen, die sich auf dem VC4000MULTIROM und in der VC-4000 befinden, miteinander kombinierbar macht.

Derartige Retrocomputing-Projekte berühren darüber hinaus mehrere Wissensfelder der *Technischen* und *Praktischen Informatik*:

- *Technische Informatik:* Architektur von Mikrocomputern und Programmierung auf der ISA-Ebene
- *Praktische Informatik:* Modulare Software-Entwicklung, inklusive Projekt- und Qualitätsmanagement, Anforderungserhebung, Konzeption, Design, Implementierung und Test [vgl. Grechenig u.a. 2010:197-348]

148 http://tempect.de/senil/games.html [letzter Abruf: 18.03.2019]. Der Entwickler hat Teile des im Projektseminar begonnenen FLAPPY BIRD-Codes übernommen und das Spiel zu Ende programmiert (s.o.).

- *Digitaltechnik/Elektronik:* Erwerb von Kenntnissen im Schaltungsdesign (zuzüglich der Nutzung von Tools zum Schaltungsentwurf), in der Elektronik und Digitaltechnik
- *Hacking/Reverse Engineering:* Evaluation fremder Software

Kenntnisse auf diesen Themengebieten wurden sukzessive aus unterschiedlichen Quellen erarbeitet: über Dialoge mit professionellen und Hobbyentwicklern, Lektüre von Fachliteratur sowie Analyse fremder Codes und Schaltpläne. Diese Quellen waren sowohl historischer als auch aktueller Art und erforderten damit ein ständiges ‚diachrones' Denken (im Sinne der Rezeptionstheorie) zwischen unterschiedlichen technischen Epochen. Die Tatsache, dass das Projekt nicht notwendigerweise erfolgreich und termingerecht abgeschlossen werden musste (auch das Scheitern eines Projektes kann, sofern es begründet dokumentiert wird, einen erfolgreichen Abschluss des Kurses ermöglichen), führte dazu, dass die Fertigstellung des Moduls erst circa ein Jahr nach dem Ende des Kurses vollzogen war. Auch hierin spiegelt sich ein modus operandi hobbyistischen Retrocomputings wider, dass Projekte außerhalb professioneller Zwänge in der Freizeit realisiert werden. Dies ließ insbesondere dem für das Selbstlernen notwendigen Trial-and-Error-Verfahren genug zeitlichen Spielraum.

4.3.4 Epistemologische Aspekte der Emulation

Maibaums Projektarbeit stellte bereits einige medienwissenschaftliche Erkenntnisse vor, die aus dem Projekt und seiner Realisierung gezogen wurden. Als ein erstes Fazit sollen diesen hier noch weitere Aspekte hinzugefügt werden. Hierzu zählen die bereits angesprochenen Mehrwerte von Emulatoren, die Frage, welche ‚Rolle' Emulatoren in Hinblick auf die computerarchäologische Konfrontation von Vergangenheit und Gegenwart des Computings einnehmen, und wie sich jenseits der informatischen Definition Emulation als ein computerphilologisches Verfahren beschreiben lässt.

4.3.4.1 Emulatoren als epistemische Objekte

Epistemische Objekte sind nach Rheinberger [2000] Forschungswerkzeuge, die selbst zum Gegenstand der Forschung werden. Dies geschieht beispielsweise dadurch, dass sie sich als defizitär oder defekt erweisen (vgl. Kap. 4.4.5). In dieser Hinsicht können die errors of omission Emulatoren in epistemische Objekte transferieren. Loebel beschreibt die *accuracy* von Software-Emulatoren mit einem Schichtenmodell [Loebel 2014:61-63]. Je näher der Emulator dabei der Hardware des Originalsystems kommt, desto größer seine Authentizität (mit den oben diskutierten Trade-Offs). Die „Full System"-Emulation, zumal mit Datenbus- und Pin-Genauigkeit, stellt dabei die authentischste Nachbildung des Originalsystems in Software dar. Diese kann sogar derartig genau sein, dass Fehler des Originalsystems implizit mit emuliert werden. WinArcadia gehört zu der Klasse von

„Full System“-Emulatoren, die von ihrer Detailgenauigkeit „[b]etween levels 2 (Cycle accuracy) and 3 (Instruction level accuracy)“ [Anhang E] liegen. Daher reproduziert der Emulator keine solchen dem Entwickler unbekannten Hardware-Designfehler. Dennoch haben die während des Kursverlauf entdeckten Fehler im Emulator einen epistemischen Mehrwert produziert, wie Maibaum schreibt:

> Tatsächlich findet sich auf den gesamten 14 Seiten [der technischen Dokumentation des PVI, S. H.] kein expliziter Hinweis darauf, dass irgendein Speicherbereich des 2636 zur korrekten Inbetriebnahme initialisiert werden müsste. Vermutlich gingen die Autoren davon aus, dass ihre primäre Leserschaft ausgebildete Ingenieure und nicht Geisteswissenschaftler wären, sodass sie das im Falle unseres fehlerhaften Spielcodes entscheidende technische Detail, dass eine elektronische Speicherzelle nach dem Anlegen ihrer Versorgungsspannung einen undefinierten, nicht vorhersagbaren Zustand einnehmen kann, schlicht wegließen. [Maibaum 2015:5]

Diese Erkenntnis ergab sich allerdings erst nach der eingehenden Analyse des Emulatorcodes, eines disassemblierten historischen Spielprogramms sowie der gezielten Manipulation der betreffenden 37 SRAM-Zellen. Der Fehler hat somit eine intensive historische und kontemporäre technische Analyse provoziert.

Zu den von [Tijms 2000] genannten Fehlerklassen könnte noch eine dritte Klasse hinzugefügt werden: *errors of addition.* Hierzu würden sich *intendierte Abweichungen* des Emulators vom Originalsystem zählen lassen, die Loebel als „Funktionserweiterungen durch Emulation“ [Loebel 2014:58ff.] bezeichnet. In WinArcadia wurde auf Bitte der Kursteilnehmer von Jacobs ein Zufallszahlengenerator in die Instructor-50-Emulation integriert.[149] Dieser kann so nur im emulierten System existieren, denn nur dieses kann aus der ‚Meta-Perspektive‘ des Hostsystems heraus die Taktrate des Zielsystems in Echtzeit mitzählen. Die Zufallswerte werden in Adresse 73_{16} abgelegt und errechnen sich aus den Clock Cycles (dividiert durch 3 und als 8-Bit-Zahl maskiert):

```
acase 115: // $73
t = (UBYTE) ((cycles_2650 / 3) % 256);[150]
```

Bei derartigen Ergänzungen handelt es sich aber nicht nur um analytische Tools, wie den „Echtzeit-Monitor CPU(s)“ und den „Speichereditor (2650)“ (vgl. Abb. 4.2.7), sondern auch um Funktionen, „um das Spielerlebnis zu verbessern“ [58], indem das Zeitverhalten des Emulators geändert wird (Pause, Verlangsamungs- und Beschleunigungsfunktionen, „undo function“ [Shibata/Hiraki 2006]) sowie solche, die das emulierte System um moderne Features ergänzen (Zugriff auf das Filesystem und die Schnittstellen des Hostsys-

149 Dieser wurde für die Erzeugung von Zufallswerten im Game of Memories genutzt (vgl. Kap. 4.2.3).

150 Die Routine dazu findet sich im Sourcecode in der Quelldatei engine1.c in der Zeile 7457f.: (http://amigan.1emu.net/releases/WinArcadia-src.rar [letzter Abruf: 23.08.2018]).

tems, Speichererweiterungen, skalierbare Grafikausgaben, ...) Solche Features könnten im Betrachter ebenfalls den Eindruck von Anachronismus erzeugen.

Von der Warte der Computerarchäologie aus betrachtet, durchbrechen solche Ergänzungen die Dissimulation der Emulation und verweisen darauf, dass das Hostsystem eben nicht identisch mit, sondern sich lediglich (teilweise) „im Zustand" [Ernst 2012:365] des Originalsystems befindet. Dieser Zustand ist jedoch nicht real, sondern lediglich virtuell: Die im Hintergrund laufenden Systemprozesse (Uhrzeit, Mauszeigerbewegung, Fensterbegrenzungen des Emulators usw.) oder die im Multitasking verwalteten, simultan zum Emulator ablaufenden Programme (Internetbrowser, MP3-Player, Chatprogram, ...) des Hostcomputer-Betriebssystems erinnern beständig daran, dass der Emulator-Nutzer Computergeschichte in einer virtuellen Umgebung nachvollzieht. Dieser Emergenzeffekt [vgl. Höltgen 2009:118,325] der Sandbox durchbricht den Anschein historischer Authentizität und wird im musealen Kontext daher oft als Blackbox kaschiert.

So werden beispielsweise im Berliner *Computerspielemuseum* Emulatoren als historische Exponate ausgestellt, ohne dass der Museumsbesucher darüber informiert wird.[151] Insbesondere die Verwendung von Flachbildschirmen ist hier nicht nur ein Indikator dafür, dass dahinter ein moderner PC das historische System emuliert, sondern auch die Oberfläche eines historischen ‚Trugbildes'. Dies zeigt sich beispielsweise im Falle des dort ausgestellten Arcadeautomaten Poly-Play (vgl. Abb. 4.2.7): Das Gehäuse enthält einen modernen PC sowie einen TFT-Flachbildschirm. Das Spielsystem bedürfte zur Erreichung historischer Authentizität jedoch eines (DDR-)Farbmonitors mit all seinen gewollten und ungewollten Bilddefiziten, wie color bleeding, Rasterung, Flimmern etc.[152]

151 Doron Swade, der Kurator des *Science Museums* in London, hinterfragt die Ausstellung von Nachbildungen kritisch mit dem Beispiel von Ersatzmöglichkeiten für den fehlenden Knopf an Napoleons Uniform. Zeitgenössische oder aktuelle Replika mögen im Besucher zwar den Eindruck von Authentizität evozieren, verfehlen jedoch weitergehende Fragen zu historischen Artefakten (Swade nennt hier beispielsweise forensische Untersuchungen [vgl. Swade 2000:140]). Einen Emulator im Museum auszustellen, ist in etwa vergleichbar damit, vermag er die Idiosynkrasien der Spielhardware prinzipbedingt nie erfahrbar zu machen. [vgl. Schweibenz 2012:51-60; Swalwell 2013; Swade 2000:139-141].

152 Einige dieser Skeumorphismen lassen sich durch einen zwischengeschaltete „Scan Line Generator"-Platine (http://arcadeforge.net/SLG3000/Scanline-Generator-SLG3000-v2::12.html [letzter Abruf: 19.03.2019]) künstlich erzeugen, um so den dissimulativen Effekt von Emulatoren zu forcieren.

Abb. 4.3.7: Der dissimulierte Poly-Play-Emulator im Berliner Computerspielemuseum

4.3.4.2 Symbolische Computer

Durch die computerphilologische vergleichende Analyse unterschiedlicher Ballsprung-Algorithmen (vgl. Kap. 4.1.4f.) sensibilisiert, fällt die Verwendung sprachwissenschaftlicher Terminologie im Diskurs um Emulatoren besonders auf. Thijms [2000] definiert Emulation als *Übersetzung*: „*Translation* is the art of transforming a binary in such a way that the *semantics* of the result equal those of the original exactly and precise." [Hervorh. S. H.] Diese Übersetzung kann, wie schon Loebel anmerkt, nie lückenlos sein. Dabei auftretende „Translation gaps" [104ff.] sind demnach mit sprachlichen Übersetzungslücken vergleichbar:

> In der Sprachwissenschaft wird mit dem Begriff Translation Gap die semantische Lücke bei der Übersetzung von Texten einer natürlichen Sprache in eine andere bezeichnet. Diese entsteht beispielsweise durch Spracheigentümlichkeiten oder Redewendungen für die es keine (exakte) Entsprechung in der Zielsprache gibt. Vielmehr müssen zur Übersetzung sinngleiche, semantisch ähnliche Konzepte in der Zielsprache gefunden sowie in den jeweiligen kulturellen Kontext des Rezipienten-

> kreises eingebettet werden. Bei historischen Texten ist zudem die Anpassung von nicht mehr gebräuchlichen Idiomen notwendig. [104f.]

Anstatt dieses Konzept, wie Loebel nun vorschlägt, auf „Abweichungen [..], die bei der Rezeption eines komplexen digitalen Artefakts innerhalb einer Emulationsumgebung“ [105] entstehen, zu übertragen, soll hier versucht werden die Übersetzung und die dabei entstehenden Fehler als linguistische Phänomene zu werten und von dieser Warte aus zu fragen, warum Emulationen an einer bestimmten Stelle „prinzipbedingt“ scheitern.

Grundsätzlich übersetzen Software-Emulatoren physikalische in semiotische Zeichenkategorien [Holenstein 1992:17]: Ihr Ziel ist es die *Hardwarestruktur* des Originalsystems in einer *Software* abzubilden, die auf dem Hostsystem ausgeführt wird. Hierbei werden mehrere Übersetzungsgrenzen überschritten:

1. Die *Hardware des Originalsystems* wird zunächst soweit abstrahiert, bis ihre physikalischen und elektronischen Idiosynkrasien verschwunden sind und sich das Originalsystem als Von-Neumann-Architektur [Schöler 2018:33f.] darstellen lässt.
2. Diese Von-Neumann-Architektur wird in einer formalen (Programmier)Sprache kodiert.
3. Ein Compiler oder Assembler übersetzt diesen Programmcode dann in die Maschinensprache des Hostsystems.
4. Die *Software des Originalsystems* wird von ihrem Datenträger über eine Retrocomputing-Schnittstelle geladen, als *Image* virtualisiert und als File im lesbaren Format und mit den notwendigen Ergänzungen (File Header etc.) auf dem Hostsystem gespeichert.
5. Diese Software wird in den Emulator geladen und darin ausgeführt. Das bedeutet, dass das bereits dreifach transkodierte System (1-3) nun selbst eine Übersetzungsleistung vornimmt, indem es die Programmdaten vom Dateiimage lädt und von der Sprache (Assembler, BASIC, ...) des Originalsystems in die Maschinensprache des Hostsystems übersetzt.
6. Das Hostsystem überträgt die Resultate des Programmablaufs nun auf seine eigene Peripherie. Dies und die zuvor vorgenommene Interpretation der Programmdaten treten aus dem symbolischen System der (Programmier)Sprachen hinaus und affizieren die (reale) Hardware und Peripherie des Hostsystems.

Diese Beschreibung ließe sich sicherlich noch weiter verfeinern, betrachtete man neben den informatischen auch noch die logischen, elektronischen und physikalischen Prozesse, die bei diesen Übersetzungen stattfinden. Es zeigt sich jedoch bereits hier, dass bei der Emulation eines Originalsystems auf einem Hostsystem zahlreiche Übersetzungen zwischen unterschiedlichen Zeichensystemen stattfinden. Allein aber schon die Überset-

zungen vom Realen (des Originalsystems) ins Symbolische (der Programmiersprache, in der der Emulator entwickelt wird) und später wieder zurück ins Reale (der Hardwareereignisse des Hostsystems) müssen „prinzipbedingt“ Verluste in Form von „translation gaps“ zeitigen.

Das Prinzip dahinter ist bereits 1944 durch John von Neumann [von Neumann 1993] begründet worden: Um eine allgemeingültige Architektur für Digitalcomputer formulieren zu können, war es zunächst nötig, von jenen logischen, elektronischen und physikalischen Elementen materieller technischer Objekte so weit zu abstrahieren, dass sie als standardisierte „organs“ [33-35] zusammengefasst werden konnten. Auf diese Weise kam nicht nur der Begriff vom „Computer“ im Kollektivsingular (erleichtert um alle Idiosynkrasien spezifischer Computer) in den Diskurs, sondern so wurde auch erst die theoretische Vorgabe Alan Turings und Alonzo Churchs, dass eine Turing-Maschine jede andere Turing-Maschine simulieren kann, technisch realisierbar. Daher bleiben bei Tanenbaum [2006:3f] die materiellen Substrate der Computertechnologie in seinem Layer/Level-Modell als außerhalb der Informatik stehende Phänomene unerwähnt und in Loebels Schichten-Modell „eines typischen Rechnersystems“ [Loebel 2015:62] werden sie als Phänomene unterhalb der „Systemhardware und Peripherie“ unbeschreib- und damit unemulierbar.

Die obigen Übersetzungsprozesse ereignen sich nicht nur zwischen zwei Zeichenkategorien, sondern auch zwischen zwei ‚Übersetzern‘. *Menschliche Übersetzer* treten hierbei als die Programmierer (der Software des Emulators und der später zu emulierenden Software des Zielsystems, bspw. eines Spiels) auf. *Maschinelle Übersetzer* sind dann neben den Compilern (die den Emulator-Sourcecode für das Hostsysteme ausführbar machen) die Emulatoren selbst, die als *Interpreter* die historische Software in Echtzeit auf dem Hostsystem ausführen, indem sie diese als mehr oder weniger akkurate „translation“ [Tijms 2000] von der Sprache des Original- in die des Zielsystems übertragen.

Betrachtet man diese Übersetzungstätigkeit nicht nur als *synchrone* (zwischen zwei Systemen/Sprachen), sondern auch als *diachrone* (zwischen zwei computerhistorischen Zeiten), so treten die Emulatoren hier auch als nicht-menschliche Historiografen auf. In Hinblick auf das „Retrocomputing der Halbleiterindustrie“ merkt Maibaum an: „Technische Medien betreiben [...] ihre eigene Archäologie, als eine Medienarchäologie im Wortsinne“ [Maibaum 2015:40]. Diese Archäologie tritt in Erscheinung durch „asynchrone *Kommunikation*“ [17f.], „*Orchestrierung* der Signale“ [46] und als „*translator* in a mixed 3.3-V/5-V system environment“ [Texas Instruments 2004 zit. n. Maibaum 2015:11] (Hervorhebungen S. H.). Was für die Elektronik des VC4000MultiROM-Moduls gilt, ist auch für Emulatoren wie WinArcadia richtig, wenngleich diese beim Lesen von virtualisierten Dateiimages keine ‚alten‘ in ‚neue‘ Signalpegel mehr übersetzen müssen: Im Hinblick auf ihre synchronen wie diachronen Übersetzungsleistungen ist Emulation bereits Computerarchäologie (mit dem Hostsystem als Archäologen).

Als epistemologische Tools verhelfen Emulatoren ihren Nutzern – insbesondere wenn sie über errors of addition verfügen – so eine „alien phenomenology“ zu betreiben. Dieses Konzept des US-amerikanischen Medienwissenschaftlers Ian Bogost versucht eine Perspektive auf Technologie vorzuschlagen, die jenseits einer anthropozentrischen Perspektive liegt. Einen technischen Prozess aus der Perspektive der Objekte zu sehen [vgl. Bogost 2012:101], könnte eine Umschreibung dieser Methode lauten.[153] Auf diese Weise können fremdartige Betrachtungswinkel und -weisen vorgestellt werden. Notwendig sei dazu u. a. ein konstruktives Vorgehen, bei welchem Wissen nicht mehr bloß über die Lektüre von Texten, sondern durch „carpentry“ [85-112] erworben und vermittelt wird. Einen Emulator als Werkzeug zur synchronen und diachronen Übersetzung von Computerprozessen zu nutzen, mehr aber noch ihn zu programmieren, ermöglicht eine solche fremdartige (Computer-)Perspektive auf Hardware und Software des Originalsystems. Die Konstruktion des VC4000MULTIROM-Moduls im Sinne des carpentry hatte zusätzliche Innenperspektiven des Originalsystems ermöglicht.

Von dieser Warte aus betrachtet erscheint das von Loebel abschließend diskutierte Thema des „Emulator[s] als dynamisches Objekt“ [Loebel 2014:161-164] nicht nur als Eskalation des Übersetzungsprozesses, sondern auch als Selbstreflexion des Emulators als Tool: Emulatoren sind als Computerprogramme ebenfalls von der Obsoleszenz ihrer Hostsysteme bedroht. Als „Bewahrungsstrategie[] für Emulatoren“ [162] schlägt Loebel die „Emulation der Emulationsplattform“ [ebd.] vor: „Emulator-Schachtelung [...] kann theoretisch unendlich fortgeführt werden.“ [165] *Theoretisch* – sowohl im Sinne der Originalplattform als Von-Neumann-Architektur als auch ohne Anspruch auf Authentizität der Ausführungsgeschwindigkeit – ist Emulationsschachtelung beliebig tief denkbar, stößt jedoch an praktische Grenzen. Mit der Schachtelungstiefe multipliziert sich auch die Übersetzungsarbeit der unterschiedlichen Host- und Originalsysteme zwischen dem Realen und dem Symbolischen (vgl. Abb. 4.2.8). Dadurch „würden sich in jedem Programm vorhandene Softwarefehler [...] potenzieren und zu nicht vorhersagbaren Seiteneffekten führen.“ [ebd.] Dies führe „ab einer gewissen Tiefe“ wahrscheinlich zum Zusammenbruch der „Emulatorkette“ und stelle damit „eine harte Schranke für die Dauer der Bewahrung“ dar [165f.]. Im Hinblick auf die Frage, wie ein Emulator die Fehlerhaftigkeit des jeweils implementierten Sub-Emulators darstellt, ergäbe sich hieraus eine Weiterung für die archäologische Leistung des Hostsystems und die didaktische Nutzung der Emulation. In den ineinander verschachtelten Sandboxes könnten Fehler, ihre Ursachen und Auswirkungen vom Nutzer empirisch studiert werden.

153 Zur Exemplifikation sowohl der Carpentry-Methode als auch der „Alien Pheononology“ hat Bogost „I am TIA“ entworfen: ein Software-Tool, das die ‚Sichtweise‘ des Atari-VCS-Grafikchips bei der Generierung von Bildschirmausgaben ermöglicht [vgl. Bogost 2012:103f.].

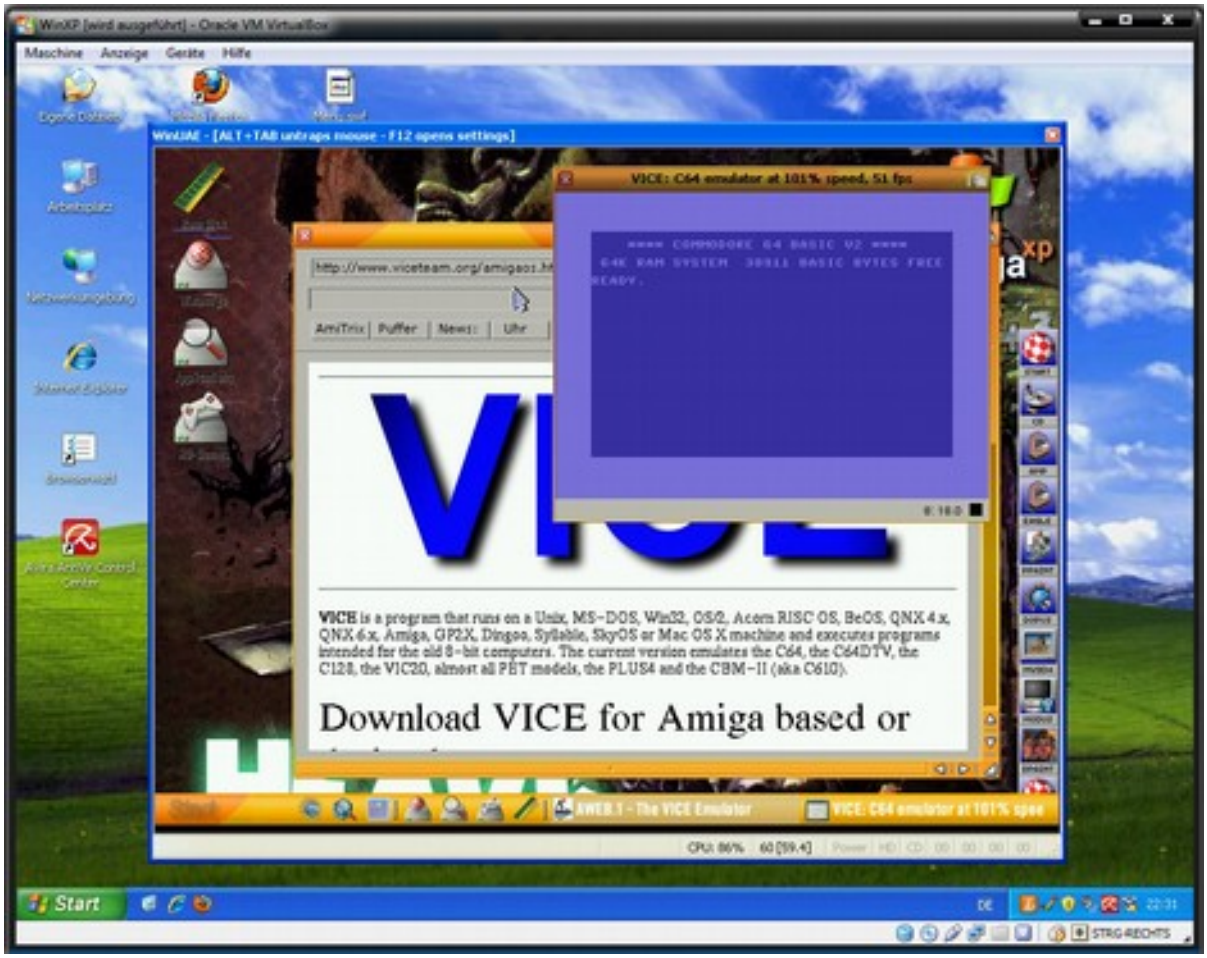


Abb. 4.3.8: Verschachtelte Emulatoren: Host (Windows 7) emuliert Windows XP emuliert Commodore Amiga emuliert Commodore 64

4.3.4.3 Operatives Wissen

Emulatoren gehören zu den populärsten Retrocomputing-Softwareprojekten – nicht, weil sie für Zwecke der Software Preservation eingesetzt werden, sondern auch, weil sie jenen ersten nostalgischen Rückbezug ihrer Nutzer ermöglichen [Wüthrich 2007:16; Muñoz-Cremers 1999]. Wie Jacobs [vgl. Anhang E] bestätigt, reicht Nostalgie eines Nutzers kaum aus, um ein so umfangreiches Softwareprojekt zu realisieren, denn „[z]ur Erstellung eines Emulators sind aber in jedem Fall umfangreiche und fundierte Kenntnisse über das zugrunde liegende System vonnöten.“ [Loebel 2014:42] Diese werden vor allem aus Online-Archiven und dem Austausch innerhalb der Retrocomputing-Community rekrutiert. Je nachdem, an welcher „Schicht“ der Emulationsprozess ansetzt, lässt sich auch die ‚Tiefe‘ des Wissens über das System ablesen.

Für einen Hobbyisten stellt die Emulator-Entwicklung ein überaus komplexes und elaboriertes Programmierprojekt dar, bei dem neben Programmierkenntnissen computerhistorisches Wissen erworben und zur ‚Anwendung‘ gebracht, was auch heißt: in ein operatives System umgesetzt werden muss. Es hängt maßgeblich von der Tiefe der Program-

mierkenntnisse wie den historischen Kenntnisse ab, welche Qualität das Resultat letztlich besitzt. Der Mangel an ersterem führt zu errors of commission, der an zweiterem zu errors of omission. Zu wissen, wer mit welchem Vorwissen und unter Zuhilfenahme welcher Quellen einen Emulator entwickelt hat, lässt Schlüsse über Abweichungen zwischen Originalsystem und dessen Emulation zu [vgl. Loebel 2014:106]. Ein Blick in die Versionsübersichten zu Emulatoren wie WinArcadia zeigt daher auch das anwachsende technische Verständnis und computerhistorische Wissen der Entwickler. So findet sich beispielsweise der in Kapitel 4.3.2 monierte Fehler der SRAM-Initialisierung in späteren Versionen des Emulators nicht mehr.

Neben den zu synchronisierenden historischen Zeiten des Original- und Hostsystems muss der Programmier auch die unterschiedlichen zeitkritischen Prozesse innerhalb der Systeme aufeinander abstimmen. Emulationsentwicklung ist ‚Zeitarbeit', die an den Gegenwarten operativer Original- und Hostsystemen laboriert. An der sukzessiven Weiterentwicklung und Verbesserung von Emulatorprogrammen zeigt sich in dieser Hinsicht ein weiteres mal, dass Retrocomputing kein (historisch) rückwärts gerichteter Prozess ist, sondern am Progress des Wissens über die Geschichte zur Herstellung technischer Gegenwarten arbeitet. Dies wird von der angewandten Informatik auch durch Verfahren des *Retrofitting* bestätigt.

4.3.5 Retrofitting

Unter dem Begriff „Retrofit" werden Maßnahmen verstanden, mit denen bestehende Systeme an neue Anforderungen angepasst werden. Der Begriff taucht häufig in Architektur und Bauwesen auf, wenn bestehende Bauwerke nachträglich modifiziert werden, um erdbebensicher (Reinforcment), energieeffizient u.a. zu werden. Retrofitting lässt sich also als eine *rückwärtskompatible Einpassung neuer Technologien in alte* umschreiben. Maibaums These vom „aktive[n] Retrocomputing der Halbleiterindustrie" [Maibaum 2015:42] (in Hinblick auf den TTL-Pegelwandler SN74LCV125A) ist auch in der Informatik seit Jahrzehnten gängige Praxis. Dort ist das Konzept namentlich seit Beginn der Vermarktung von Mikrocomputern bekannt. Seit Ende der 1970er-Jahre werden Hardware- und Softwarelösungen unter der Bezeichnung „Retrofit Kit" angeboten, die die Obsoleszenz einer Plattform aufschieben sollen. Zwei historische Beispiele seien hierfür angeführt.

4.3.5.1 Geschichte des Konzepts

Der bereits 1974 von *National Semiconductor* veröffentlichte 8-Bit-Mikroprozessor SC/MP wurde im PMOS-Verfahren hergestellt, woraus sich bestimmte nachteilige Eigenschaften für Systeme, in denen er im Einsatz war, ergeben. So benötigt er beispielsweise zwei unterschiedliche Betriebsspannungen (-7 Volt und +5 Volt), was zu komplexeren Anforderungen an die Netzteile der Systeme führt. Außerdem lässt sich die CPU lediglich mit

maximal 1 MHz takten. *National Semiconductor* brachten daher 1977 einen im NMOS-Verfahren konstruierten Nachfolger, SC/MP-II, auf den Markt, der lediglich eine +5-Volt-Spannung benötigt und mit maximal 4 MHz betrieben werden kann ([vgl. National Semiconductor 1977:1] – dort werden weitere Unterschiede aufgeführt.)

Aufgrund derselben Bauform (DIP-40) lässt sich der SC/MP leicht durch den SC/MP-II austauschen. Zum Betrieb mit dem neuen Mikroprozessor muss das System allerdings elektronisch modifiziert werden. Hierzu veröffentlichte *National Semiconductor* 1977 das SC/MP-II Microprocessor Retrofit Kit, welches neben der CPU und dem Schwingquarz aus einigen elektronischen Bauteilen bestand, die für einen solchen Umbau benötigt wurden. Diese elektronischen Umbaumaßnahmen waren für SC/MP-Systeme aufgrund ihrer geringen Komplexität (zumeist handelte es sich um Einplatinen-Lehr-/Lerncomputer wie das SC/MP INTROKIT) noch vom Anwender durchführbar. Dort allerdings, wo komplexere Systeme, die auch im professionellen Einsatz waren, Retrofitting unterzogen wurden, gab es Fertiglösungen.

So zum Beispiel beim Retrofit Kit für den Victor-9000-Computer (Abb. 4.2.9). Dessen Aufgabe wie folgt beschrieben wird: „The Victor PlusPC retrofit kit was a way of providing IBM PC ‚compatability' to the Act Sirius 1 / Victor 9000. The retrofit kit allowed users to either use the computer as a Sirius 1 / Victor 9000 or as a PC compatible computer."[154] Die 1982 veröffentlichte Plattform basierte auf einem Intel-8088-Mikroprozessor, der intern zwar über eine 16-Bit-Architektur verfügt, jedoch nur mit einem 8-Bit breiten externen Datenbus ausgestattet war und im Unterschied zum zeitgleich erschienenen 8086 kein DMA (Direct Memory Access) ermöglichte. Auf diesem System lief das Betriebssystem CP/M mit den zugehörigen Applikationen. Mit Hilfe des Retrofit Kits, das aus drei Erweiterungskarten bestand, konnte der Computer so modifiziert werden, dass er wie ein IBM-PC-kompatibles System nutzbar war:

1. In den Sockel der 8088-CPU wurde ein „Mapper Board" gesteckt, das ebenfalls einen 8088 enthielt aber noch zusätzliche Bauteile, die das System um DMA erweiterten.
2. Ein „Disc Controller Board" ersetzte das im Rechner verbaute Controller-Board und ermöglichte den Anschluss an ein DMA-Board.
3. Ein „DMA Board" wurde direkt neben das „Mapper Board" und das „Disc Controller Board" installiert und mit diesen verbunden.

154 https://web.archive.org/web/20130722164808/http://actsirius1.co.uk/pages/pluspc.htm [letzter Abruf: 19.03.2019].

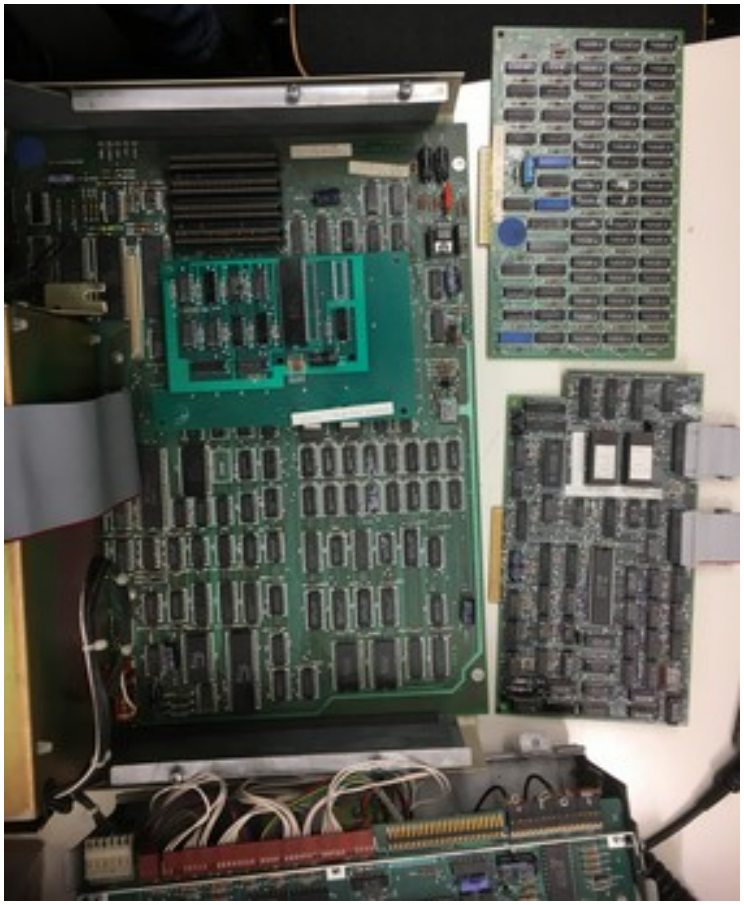


Abb. 4.3.9: Retrofit Kit für den Victor 9000: Links „Mapper Board“ (eingesteckt), rechts oben: „DMS-Board“, rechts unten: Disc Controller Board

Damit wurde das System in einem „Multi-Mode-Computer“ überführt:

> The installation of PlusPC allows for a multi-mode computer. There were four different ways or operational modes to exploit the PlusPC environment. With the first two you have access to disks with either Victor or PC format: with the second two you have access only to disks formatted for the mode in which the computer boots up in. Included in the PlusPC operating system are the Victor MS-DOS 2.1 and an operating system compatible with PC-DOS 2.1.[155]

Der so umgebaute Victor-Computer ähnelt damit dem Commodore 128 (der durch Hardware-Implementierungen C64-kompatibel gemacht wurde) oder den Nachfolge-Systemen des Sinclair ZX Spectrum, die softwareseitig über einen Kompatibilitätsmodus zum Ursprungsgerät verfügten. Der entscheidende Unterschied zwischen diesen Konzepten und Retrofitting ist jedoch, dass letzteres *nachträglich* passiert, um bereits in Betrieb befindliche Systeme ‚vorwärtskompatibel‘ (‚zukunftssicher‘) zu machen.

155 https://web.archive.org/web/20130722164808/http://actsirius1.co.uk/pages/pluspc.htm [letzter Abruf: 19.03.2019].

4.3.5.2 Retrofitting in der Gegenwart

Derartige Verfahren sind auch heute in der Angewandten Informatik im Einsatz, wenn sich entweder Sicherheitsanforderungen ändern (oder neue ergeben, wie nach der Entdeckung des Y2k-Bugs, [vgl. Yang/Paradi 2004]) oder wenn sich Anforderungen an Software und Software-Entwicklungen ändern, wie mit der Popularisierung von Apps für Mobile Devices bei der APIs HTML/XML-Entwicklung auf JSON/REST umgestellt werden musste.[156] [Pöhl/Peitek 2016].

Besondere Aufmerksamkeit wird dem Retrofitting im aktuellen Diskurs um die ‚Industrie 4.0' zuteil, weil hier eine deutliche Umwälzung für Produktion und Distribution stattfindet, die neue Technologien erfordert und zum Einsatz bringt:

> Der Kern der Umwälzung besteht in der vollständigen Durchdringung der Industrie, ihrer Produkte und ihrer Dienstleistungen mit Software bei gleichzeitiger Vernetzung der Produkte und Dienste über das Internet und andere Netze. Diese Veränderung führt zu neuen Produkten und Diensten, die das Leben und Arbeiten aller Menschen verändern, und natürlich erst recht auch ihren Umgang mit Produkten, Technik und Technologien. Sie verlangt aber auch eine grundlegende Veränderung und Anpassung der industriellen Produktentwicklung und Produktion, um die neuen Technologien qualitativ hochwertig einsetzen und wirtschaftlich nutzbringend umzusetzen. [Sendler 2013:1]

Die sich aus dieser Umwälzung ergebenden Veränderungen (sowie die zusätzlich zu berücksichtigenden stetig beschleunigten Innovationszyklen von Hardware und Software [vgl. Russwurm 2013:23; Günther u.a. 2017:173]) führen dazu, dass im Betrieb befindliche Informatiksysteme immer früher nach ihrer Anschaffung veralten; ein Umstand, der insbesondere für kleine und mittelständische Unternehmen eine Belastung darstellt,

> weil man eine Maschine, die vielleicht eine Million Euro gekostet hat, nicht einfach wegwerfen kann wie ein altes Smartphone. ‚Das ist einfach anders als im Consumerbereich, wo Sie typischer Weise Ihr Handy dann nach drei oder vier Jahren letztlich gegen ein neues austauschen. Was durchaus machbar ist in der Produktion, dass Sie ein so genanntes Retrofit durchführen. Das heißt, Sie verwenden weiterhin die alte Hardware, die in vielen Fällen noch gut ist, und tauschen die Steuerung und damit die Software letztlich aus.'[157]

Zur eingangs erwähnten anwachsenden kulturellen Bedeutung von (operativer) Computergeschichte kommen eine wirtschaftliche und nicht zuletzt auch eine ökologische Seite (nachhaltiger Ressourcennutzung) hinzu. Neben *Preservation* erfordert *Retrofitting* not-

156 https://square.github.io/retrofit/ [letzter Abruf: 20.03.2019].

157 https://www.deutschlandfunk.de/industrie-4-0-vernetzte-anlagen-vor-schad-software-sichern.684.de.html?dram:article_id=428796 [letzter Abruf: 20.03.2019].

wendigerweise eine Sensibilisierung der technischen, praktischen und angewandten Informatik für die Computergeschichte, was auch innerhalb der Curricula der Informatikausbildung berücksichtigt werden müsste, etwa, indem Methoden-, Fakten- und Quellenwissen zur Computergeschichte gelehrt werden. Das Thema der „Medienkonservierung“, für das im museologischen, kultur- und kunsthistorischen Diskurs bereits didaktische Ansätze formuliert wurden [vgl. Gfeller 2013; Ruiz/Thomas u.a. 2013], fordert „[e]ine enge Zusammenarbeit [...] mit dem Ingenieurwesen“ [Gfeller 2013:618], wozu neben der Elektronik und der Chemie auch die Informatik gefordert wäre. Operative Computergeschichte öffnet darüber hinaus weitere lukrative Einsatzgebiete für Informatikerinnen und Informatiker. Bereits in der Vergangenheit hat die Weiternutzung scheinbar obsoleter Technologien (bei der z. B. ein Administrator für ein PDP-11-System in einem Atomkraftwerk[158] oder COBOL-Entwickler[159] in unterschiedlichsten Bereichen Einsatz finden) spezielle Aufgabengebiete ergeben. Die vorgenannten Retrofitting-Prozesse könnten diesen Bedarf ausweiten.

4.3.6 Zusammenfassung

Emulation und Retrofitting wurden hier als Formen der diachronen Kommunikation zwischen unterschiedlichen Geräten und Geräte-Epochen diskutiert. Beide bedingen unterschiedliche Strategien: Virtualisierungsbestrebungen auf der einen Seite, hard- und softwaretechnische Anpassungen auf der andere Seite. Die Unterschiede zwischen hobbyistischer und professioneller Emulation zeigen sich vor allem in der Qualitätssicherung. Während ein hobbyistisches Projekt mit Fehlern (sowohl formalen Fehlern als auch ‚Verständnisfehlern‘) veröffentlicht wird, wäre dies bei einem professionellen Emulator nicht tolerierbar. Wenn etwa im Cross-Platform-Development-Verfahren Software/Apps für mobile Systeme auf PCs entwickelt werden, muss die Emulation des Zielsystems exakt sein. Diese Exaktheit ist aber nur möglich, weil solche Emulatoren lediglich auf der Applikationsebene (vgl. Kap. 4.3.1.1) ansetzen und zudem das Wissen über das Originalsystem vollständig vorliegt und berücksichtigt werden kann.

Die fehlerhaften Emulatoren aus nichtkommerziellen Zusammenhängen haben sich jedoch gerade deshalb als interessante epistemische Objekte erwiesen, weil sie dokumentieren, wie computerhistorisches Wissen durch Hobbyisten im Selbstlernprozess und durch Hinzuziehen neuer Quellen „grauer Literatur“ [Kornwachs/Berndes 1999:23] ergänzt und operativiert wird. Dies zeigt sich bei der Verwendung von Emulatoren für historische Software allerdings seltener als, wenn diese für Neuentwicklungen eingesetzt werden. Diese Erfahrung konnte beim Versuch der Realisierung eines Softwareprojektes

158 https://www.theregister.co.uk/2013/06/19/nuke_plants_to_keep_pdp11_until_2050/ [letzter Abruf: 21.03.2019].

159 https://www.glassdoor.de/Job/germany-cobol-programmer-jobs-SRCH_IL.0,7_IN96_KO8,24.htm [letzter Abruf: 21.03.2019].

(Flappy Bird für Interton VC-4000) gemacht werden. Bei der daran anschließenden Konstruktion eines Flash-ROM-Moduls (VC4000MultiROM) für das Spielsystem VC-4000 konnte der fließende Übergang von Retrocomputing zu Retrofitting bis hinab auf die Bauteilebene verfolgt werden.

Wird der Emulationsprozesses als computerphilologisches Phänomen betrachtet (bei dem zwei in letzter Konsequenz virtuelle/reale, zeitgenössische/historische Computer/Emulatoren miteinander kommunizieren), ließen sich die Überlegungen zur interdisziplinären Betrachtung des Phänomens (aus Kapiteln 4.1.4-5) übertragen. Aus dieser Perspektive ermöglicht der ‚Lektüreprozess' zwischen unterschiedlichen Systemen/Generationen eine detaillierte Beschreibung und Analyse der Prozesse. Diese könnte sowohl die universitäre Informatikausbildung (in Hinblick auf Retrofitting) bereichern als auch die hobbyistische Entwicklung von Emulatoren. Dass Software-Emulation aufgrund ihrer notwendigen Abstrahierung realer Computer in symbolische Von-Neumann-Architekturen defizitär bleiben muss, führt dazu, dass sich Software Preservation und Retrocomputing eben nicht auf „Emulation [... als] Grundidee des Erhalts historischer Hardware durch Virtualisierung" [Lange 2016:314] verlassen dürfen. Eine möglichst dauerhafte Erhaltung der Hardware wäre hierfür erforderlich. Das letzte Projektkapitel stellt diese exemplarisch vor und diskutiert ihre Möglichkeiten, Grenzen und epistemologischen Implikationen.

4.4 Knowledge und Hardware Preservation

Die im Vorangegangenen geschilderte Konstruktion neuer Hardware offenbart sich bereits als ein Zugriff des Nutzers auf den Computer, der diesen jenseits des reinen Werkzeugs auch als epistemisches Objekt verortet. Diese Umdeutung verfolgt auch die Reparatur eines defekten Computers, die sich als ‚Arbeit am Wissen' verstehen lässt. Das nachfolgende Kapitel protokolliert und analysiert den Reparaturprozess eines historischen Mikrocomputers und zeigt dabei, welchen Ertrag dieser Prozess über die intendierte *Hardware Preservation* hinaus erbringt. Dabei wird der Akzent vor allem auf den autodidaktischen Erwerb von Reparaturwissen, die unkonventionellen Vorgehensweisen (Hacking) und die Dialogizität zwischen dem zu reparierenden Computer und dem Reparateur gelegt. Neben der Aktualisierung historischen Wissens aus verschiedenen Teilgebieten der Informatik (Hardware-Design, Schnittstellenprogrammierung, Datenformate, Didaktik) und Elektrotechnik (Messtechnik, Digitaltechnik) liegt der Akzent vor allem auf der epistemologischen Durchdringung des Reparaturprozesses und einen *Gamification*-Prozess zwischen Mensch und Maschine sowie der in Kapitel 3.3.3 geforderte Aspekt der Demonstration als Evidenzproduzent. Hierzu wird in das Kapitel ein Reparaturbericht, verfasst vom Reparateur (Marius Groth) eingefügt, der darauffolgend in Hinblick auf eine *Epistemologie des Reparierens* analysiert wird.

Computer als Gebrauchsgegenstände sind zuvorderst *Werkzeuge* zur Speicherung, Übertragung und Verarbeitung von Informationen. Um diese Funktionen realisieren zu können,

bedarf es *zweier Voraussetzungen*: *Erstens* müssen Computer im *Blackboxing* ihre technische Komplexität verbergen [vgl. Becker 2012:15] und auf ein menschliches Maß reduzieren, so dass sie über ihre Schnittstellen (sofort oder nach kurzer Zeit) intuitiv nutzbar sind. Dieses Verbergen hat eine historischen und epistemologischen Grund, der auf John von Neumanns „First Draft of a Report on the EDVAC" (1944) verweist. Dort ‚verbirgt' von Neumann die Komplexität der elektronischen Logikgatter hinter dem Begriff „organ", damit der Ingenieur *den Computer* (nunmehr im Kollektivsingular!) aus einer diagrammatischen Struktur/Architektur entwickeln kann [vgl. von Neumann 1993:33-36]. Aus dieser Sichtweise auf Computer erwächst zudem die Wissenschaft der Computer Science (Informatik), in deren Folge immer weitere Abstraktionsebenen („abstraction levels" [Tanenbaum 2006:22]) in den Computer ‚eingezogen' werden, um jede von ihnen für Spezialisten differenzierbar und bearbeitbar zu machen, indem die übrigen ‚ausgeblendet' werden. Die letzte Eskalationsstufe dieses Prozesses offenbart sich für den Anwender im „obiquitous computing" [Weiser 1991] – der nicht mehr wahrgenommenen Allgegenwart von Computertechnologie in Alltagsgegenständen. Wie in der Fortschrittsgeschichte üblich [vgl. Luhmann 1977], erbringt der Gewinn solcher Spezialisierung und funktionaler Differenzierung einen Verlust an allgemeiner Zugänglichkeit und Verstehbarkeit des Ganzen.

Die *zweite Voraussetzung* zum Werkzeuggebrauch von Computern liegt in ihrer *Operativität*. Nur, wenn ein Computer funktioniert (und wenn er eingeschaltet ist), ist er auch als Werkzeug in seinem Sinne verwendbar. Was zunächst wie eine Trivialität anmutet, soll im Folgenden von epistemologischer Seite aus diskutiert werden: Was wäre ein Computer, der nicht funktioniert? Was bedarf es, um einen defekten Computer zu reparieren? Welche Konsequenzen ergeben sich aus der Reparatur von Computern jenseits ihrer damit (wieder) erreichten Operativität? Und schließlich: In welchem Zusammenhang stehen die beiden genannten Voraussetzungen des Werkzeugcharakters von Computern zueinander?

4.4.1 Computer als historische Objekte

Computergeschichte wird, außer in Historiografien, vor allem in Technik- und Computermuseen vermittelt. Solche Museen widmen sich entweder der Geschichte der Computer im allgemeinen (Beispiel: *Heinz-Nixdorf-MuseumsForum*, Paderborn), fokussieren bestimmte Technologien (Beispiel: *Analogrechner-Museum*, Bad Schwalbach), Nationalitäten (Beispiel: *Rechenwerk*, Halle), Epochen (Beispiel: *Oldenburger Computermuseum*, Oldenburg), Firmen (Beispiel: *Zuse-Museum*, Hoyerswerda), Ingenieuren (Beispiel: *National Museum of Computing*, England) oder Anwendungen (Beispiel: *Computerspielemuseum*, Berlin). Neben dieser thematischen Differenzierung lassen sich Computermuseen auch nach dem Grad der Operativität ihrer Exponate unterscheiden. Die Skala reicht hier von der vollständig dysfunktionalen Hardware-Ausstellung (Beispiel: *Computermuseum,* Kiel) bis zum Hands-on-Museum (Beispiel: *Oldenburger Computermuseum*, Oldenburg oder *Living Computer Museum*, Seattle). Letztere akzentuieren Computer in ihren Ausstellungen als Apparate im

oben genannten Sinne (zum Speichern, Übertragen und Verarbeiten von Informationen) und vermitteln ihren Besuchern genau diese Aspekte ihrer historischen Genese. Dabei verursachen sie eine produktive Spannung zwischen Computern als historischen Artefakten und operativen Medien, die notwendig gegenwärtig sein müssen: Erst eingeschaltet kann ein Computer seine technische ‚Historizität' unter Beweis stellen: Verarbeitungszeiten, Ladezeiten, Funktionsbeschränkungen, spezifischen Schnittstellen etc. früherer Computertechnologie im Vergleich zu heutiger lassen sich nur auf diese Weise erfahrbar machen.

Neben Museen existieren noch weitere ‚Orte', in denen Computergeschichte erfahrbar wird. Dies sind vor allem private Sammlungen, Vereine und Szene-Treffs wie Börsen, Flohmärkte und Festivals. Hier ist das Verhältnis von dysfunktionalen zu funktionierenden historischen Computern zugunsten letzterer besonders deutlich: Oftmals ist der ‚Grund' einer Sammlung, eines Verkaufs oder einer Ausstellung die Erfahrbarmachung von Computergeschichte am operativen Gerät – etwa, um damit historische Computerspiele spielbar zu machen, kreative Wettbewerbe zu inszenieren oder Hardware- und Software-Neuentwicklungen für alte Plattformen vorzuführen. Da es im Wesen des ‚privaten' Computersammlers liegt, aus seinem Hobby eben keine Profession gemacht zu haben (oder anders herum: zumeist nicht zugleich professioneller Sammler/Kurator *und* Hobbyist zu sein), steht dieser oftmals vor dem Anspruch, seine Sammlungsgegenstände lauffähig zu halten oder wieder lauffähig zu bekommen, ohne sich hierzu einer professionellen Infrastruktur bedienen zu können [vgl. Takhteyev/DuPont 2013:432]. ‚Bewahrung' findet hier in der Regel auf Basis privater Mittel und mit autodidaktisch erworbenem Wissen statt, das insbesondere an den konkreten Plattformen, die zur Sammlung gehören, spezialisiert wird. Die dafür erlangten Kenntnisse und Praktiken gehören als „knowledgde preservation" [Agrifoglio 2015] zum stetig wachsenden Inventar des privaten Mikrocomputer-Nutzers. [Vgl. Zaks 1981]

Das *Signallabor* des *Instituts für Musikwissenschaft und Medienwissenschaft* an der *Berliner Humboldt-Universität* ist diesbezüglich ein ‚hybrider' Ort, dessen Ausstattung und Funktion zwischen diesen beiden Sphären situiert ist. *Professionell* ist es insofern, dass es sich bei den dort gesammelten Mikrocomputern um Objekte einer universitären Sammlung handelt, die Gegenstand eines medienwissenschaftlichen Forschungsprojektes sind. *Privat* ist diese Sammlung insofern, als die Computer nahezu sämtlich aus privaten Mitteln des Projektinhabers angeschafft wurden und dadurch, dass dieser, wie die oben beschriebenen Hobbyisten, als Medienwissenschaftler keine professionellen/akademischen Kenntnisse in Elektronik besitzt, um diese bei der Instandhaltung und -setzung anwenden zu können, jedoch innerhalb der Retrocomputing-Szenen gut vernetzt ist, um Wissen ‚rekrutieren' zu können. Dies stellt jedoch kein Dilemma dar, sondern bildet sogar einen der Forschungsaspekte ab, in dem es darum geht zu ermitteln, auf welche Weise autodidaktisches Wissen über Computertechnik und -geschichte in den so genannten Retrocomputing-Szenen erworben, eingesetzt und proliferiert wird.

4.4.2 Computer als Hardware

Wie eingangs angedeutet, stellt die Reparatur eines historischen medientechnischen Objektes für die Computerarchäologie mehr als bloß dessen Instandsetzung zur Wiedererlangung seiner Operativität dar. Der Prozess des Reparierens (Beseitigen von Defekten) ist eine intellektuelle und manuelle Form der Auseinandersetzung mit dem Objekt, die gänzlich andere Methoden und Werkzeuge als dessen historische und diskursive Beschreibung erfordert. In der Auseinandersetzung mit dem konkreten technischen Objekt, dem ‚hands-on', begegnet der Nutzer dem Objekt auf seiner ihm eigenen phänomenalen Ebene. Die ‚Unmenschlichkeit' des Technischen zeigt sich hier in all ihrer Deutlichkeit: Werden Computer auf diskursiver Ebene zum Beispiel als ‚Digitalmedien' dargestellt, die „nur 0 und 1 kennen", die als ‚Denkmaschinen' genutzt werden, eine fiskalisch messbaren ‚Wert' besitzen, mit denen gehackt oder Computerkunst geschaffen wird usw. – die also eine Geschichte, Soziologie, Ästhetik, Politik, Philosophie haben und dadurch Bezüge zur menschlichen Kultur besitzen, so zeigen sie sich auf dieser Ebene ausschließlich als Schaltungen, die auf Basis von physikalischen und elektronischen Gesetzmäßigkeiten Signale transportieren und variieren. Alle vorgenannten kulturellen Effekte finden sich hier (noch) nicht, sondern ergeben sich erst als Emergenz- und Nutzungseffekte aus dieser technischen Verfasstheit. In dieser Tatsache gründet letztlich der spezifische Charakter der Computerkultur. Während Museen Computer(hardware) als sichtbare Indizes solcher historischer, kultureller, ästhetischer und anderer Diskurse sammeln und zeigen, fokussiert Computerarchäologie Computer als Apparate, die losgelöst von diesen übergeordneten Sichtweisen zunächst ‚für sich' Forschungsgegenstände darstellen.

Abb. 4.4.1: Coverabbildung der Zeitschrift Popular Electronics, in der der Sol-20 im Juli 1976 als Bausatz vorgestellt wurde.[160]

160 http://www.swtpc.com/mholley/PopularElectronics/Jul1976/PE_Jul1976.htm [letzter Abruf: 11.01.2019].

Der nachfolgende Werkstattbericht beschreibt die Applikation dieser Theorie und die Anwendung der Methoden anhand der Reparatur eines defekten Mikrocomputers. Dabei handelt es sich um den Sol-20, der einer der ersten Homecomputer ist – also ein frei verkäuflicher Mikrocomputer, bei dem alle zum Betrieb benötigten Technologien (Computer, Tastatur, I/O-Schnittstellen) *in einem Gehäuse* verbaut sind, und der Anschlussmöglichkeiten für in privaten Ambientes verfügbare Medientechnologien (Kassettenrecorder, Fernseher, Schreibmaschine, Modem/Telefon) besitzt. (Diesem Design-Prinzip folgen ab Anfang der 1980er-Jahre die meisten Hersteller von Computern für den Heimgebrauch.) Der Sol-20 wurde 1976/77 publiziert und ist selbst das Ergebnis eines Projektes, das in die oben genannte Retrocomputing-Szenen passen würde: Er entstand als Bausatz für frühe Computerhobbyisten (Abb. 4.2.1) und als Erweiterung der Mitte der 1970er-Jahre populären TV-Typewriter (mit denen man selbst eingegebene Schrift auf den heimischen Fernseher darstellen konnte). [Levy 1984:237-239.] Lee Felsenstein, der federführende Ingenieur hinter dem Sol-20, stammt aus einer Tradition von Hackern, die sich bereits in den späten 1960er-Jahren die Popularisierung und Demokratisierung von Computertechnologie auf die Agenda geschrieben hatte. [Höltgen 2014a] Der Finanzier des Projektes, Les Solomon, gab etwa zur Zeit der Entstehung des Sol-20 eine Zeitschrift für Computerhobbyisten heraus („Creative Computing"), in der die ersten Computerbausätze vorgestellt wurden und die als Wegbereiter der privaten Computerkultur gesehen wurde. [Roberts/Yates 1975]

Abb. 4.4.2: Der Sol-20

Vom Sol-20 (Abb. 4.4.2) sind mutmaßlich 10.000 Exemplare [Battle 2006] gebaut und entweder als Bausatz oder Fertiggerät vertrieben worden. Aufgrund der schon damals rasanten Entwicklung der Mikrocomputertechnologie sowie der spezifischen Fehleranfälligkeiten des Sol-20, hat diese vergleichsweise geringe Zahl dazu geführt, dass heute nur noch wenige Exemplare existieren dürften. Sein technischer Aufbau ist typisch für Mikrocomputer seiner Entstehungszeit. Er basiert auf dem 8-Bit-Mikroprozessor 8080 der Firma *Intel*, der mit 2 Megahertz getaktet ist. Als 8-Bit-System kann er maximal 64 Kilobyte RAM verwalten; ausgeliefert wurde er zumeist mit 16 Kilobyte, die über Steckkarten erweitert werden konnten. Das Bus-System, das der Rechner zur Erweiterung einsetzt, ist das S-100-System (Abb. 4.4.3

und 4.4.4), das bei vielen zeitgenössischen Mikrocomputern zum Einsatz kam und damit einen frühen Standard[161] bildete. Für dieses Bus-System gab es zahlreiche Erweiterungskarten zu kaufen und es ließen sich durch die Anwender zudem leicht eigene Steckkarten entwickeln, womit der Sol-20 einen ‚Bastel-Computer' darstellte, der den Bedürfnissen seiner damaligen Nutzerklientel (Hobby-Elektroniker) sehr entgegen kam. Wie geschildert, integrierte der Sol-20 allerdings bereits Elemente, die für andere damalige Systeme erst dazu gekauft oder gebaut werden mussten: Eine Tastatur, eine Videobild-Erzeugung und Ein-/Ausgabeschnittstellen für Massenspeicher (Disketten- und Kassettenlaufwerk, Drucker, serielle Peripherie). Damit genügte er ebenso einer Nutzergruppe, die weniger auf Basteln, als auf Anwendungen orientiert war. Für letztere entstand sukzessive ein Angebot an Amateur- und professioneller Software verschiedener Gattungen. Durch die Möglichkeit auf dem Sol-20 das seinerzeit weit verbreitete Betriebssystem CP/M zu nutzen, erweiterte sich der Einsatzbereich für professionelle Anwendungen immens: Büroanwendungen, Programmiersprachen, Lernsoftware und vieles andere kann über CP/M auf dem Computer genutzt werden. Der Sol-20 zeigte sich also bereits von Beginn an als System, das sowohl zum Experimentieren mit Computertechnik einlud als auch als Werkzeug für die Erledigung von Arbeitsaufgaben genutzt wurde.

Abb. 4.4.3: Das S-100-Bus-System im Sol-20 voll bestückt mit Erweiterungskarten

Abb. 4.4.4: Das S-100-Bus-System im Sol-20 ohne eingesteckte Karten

161 Der Standard des S-100-Bussystems ist unter IEEE 696-1983 definiert: http://www.imsai.net/download/IEEE_696_1983.pdf [letzter Abruf: 11.01.2019]

Der Computer stellt heute einen vergleichsweise seltenes Objekt dar. Dies liegt zum einen in der zur Zeit seiner Entstehung hohen Innovationsfrequenz: Immer neue, verbesserte, schnellere und günstigere Computer erreichten den Markt; für alte Geräte gab es kaum Verwendung, weshalb sie oft entsorgt oder demontiert wurden. (Letzteres ist bei einem auf einem Standard-Bussystem wie dem des S-100 basierenden System sehr wahrscheinlich, weil viele Erweiterungen kompatibel zu nachfolgenden S-100-Computern waren.) Seine Verarbeitung, insbesondere die Verwendung von organischen Materialien (Holz für die Seitenleisten und organischen Schaumstoffe für die Tastatur – siehe unten) ließen den Computer zudem schneller an Zerfallsprozessen leiden. Aufgrund dieser Beschaffenheit dürfte heute kein funktionierender Sol-20 mit allen ursprünglichen Materialien mehr existieren. Seine Reparatur stellt mithin die einzige Möglichkeit dar, ihn als Computer (im hier verstandenen Sinne) erfahrbar zu machen.

4.4.3 Werkstattbericht

Empfangen haben wir[162] den Sol-20 im Mai 2015 als Dauerleihgabe auf dem *Vintage Computer Festival Europa* von Hans Franke, einem Sammler aus München. Die Prämisse war, dass der Rechner repariert und im *Signallabor* für das Forschungsprojekt genutzt werden sollte. Ursprünglich war geplant ihn auf dem *Vintage Computing Festival Berlin* im Herbst 2015 von seinem Erbauer (Lee Felsenstein) vor Ort reparieren zu lassen, was dann leider wegen Terminüberschneidungen nicht möglich war.

Abb. 4.4.5: Der geöffnete Sol-20

Aufgrund der Aussage des Besitzers, dass der Computer generell funktioniere, verlief eine Vorabkontrolle vor einer ersten Inbetriebnahme kursorisch: Das Gehäuse wurde geöffnet und das ‚Innenleben' begutachtet (vgl. Abb. 4.4.5). Zur Überraschung waren alle ICs gesockelt, so dass etwaige Fehler im digitalen Bereich mit geringem Aufwand zu beheben sein dürften (die ICs können leicht aus den Sockeln gezogen werden, anstatt sie zeitaufwändig von der Platine entlöten zu müssen, was immer mit dem Risiko der Beschädigung der Leiterplatte

162 Dieses Unterkapitel entstand in Zusammenarbeit mit Marius Groth (Berlin). Der „Wir"-Modus ist dem Laborprotokoll geschuldet und meint M. Groth und S. Höltgen. Das Kapitel 4.4.3 basiert auf einem gemeinschaftlich erstellten Text.

einhergeht). So sehr dies eine spätere Erleichterung darstellt, so stellen diese Sockel allerdings im Laufe der Zeit auch eine potentielle Fehlerquelle dar, da in ihren Fassungen durch Oxidation Kontaktschwierigkeiten entstehen können. Außerdem sind große Teile des Rechners auf Zusatzkarten ausgeführt (2 mal 16 Kilobyte RAM, 2 Schnittstellenkarten, „Personality Module“ mit Betriebssystem); auch diese können von Oxidation betroffen sein. Also wurden alle Steckkarten gezogen und die Kontakte vorsorglich mit Isopropanol gereinigt. Danach wurden mehrere Steckzyklen durchgeführt. Im Anschluss wurden alle ICs auf dem Mainboard von Hand nachgedrückt. Zusätzlich wurde mit einem Pinsel der Staub vom Mainboard gebürstet. Daraufhin erfolgte eine erste Messung mit einem Multimeter, um auszuschließen, dass auf der Platine Kurzschlüsse vorhanden sind. Der Test ergab keine derartigen Probleme. Staub und Verschmutzungen wurden oberflächlich von den Platinen entfernt.[163]

Es sprach also nichts gegen eine erste Inbetriebnahme. Da der Computer ausschließlich in den USA produziert und vertrieben wurde, war es notwendig die hiesige 230-Volt-Netzspannung mittels Step-Down-Converter auf 110 Volt anzupassen. Zu diesem Zeitpunkt bestand die Hoffnung, dass der Rechner aus der Netzfrequenz keinen Takt ableitet, denn dieser unterscheidet sich bei deutscher (50 Hertz) und US-amerikanischer (60 Hertz) Wechselspannung. Als Monitor stand ein Gerät zur Verfügung, das das NTSC-Bildformat akzeptiert (vgl. Abb. 4.4.16). Das erste Einschalten ergab lediglich wirre Zeichen auf dem Bildschirm, wobei es sich vielleicht nur um ‚Anlaufschwierigkeiten‘ handelte. Ein Powercycle brachte dann tatsächlich den zu erwartenden Cursor auf den Bildschirm. Eingaben via Tastatur waren allerdings nicht möglich. Im eingeschalteten Zustand wurden die Spannungen im Rechner kontrolliert; diese lagen alle im Soll-Bereich.

4.4.3.1 Reparatur der Tastatur

Abb. 4.4.6: Die Tastatur des Sol-20 (ausgebaut)

Der nächste Schritt war die Suche nach den Schaltplänen für den Sol-20. Der erste Treffer einer Google-Suche führte zur Internetseite von Jim Battle[164], wo nicht nur Schaltpläne sondern auch Anleitungen, Programme, typische Fehler und Reparaturen, ein Emulator und andere In-

163 Unter zwei Tastenkappen fanden sich tote/leere, verpuppte Insektenkokons, die entfernt wurden, nicht ohne dabei an den berüchtigten Labortagebuch-Eintrag Grace Hoppers „First Actual Case of Bug being found“ zu denken. [vgl. Hopper 1947]

formationen zum System abrufbar waren. Aus der Lektüre der Seiten ergab sich, dass ein typisches Problem, welches inzwischen beinahe jeden Sol-20 betrifft, die Tastatur ist. (Abb. 4.4.6) Diese ist in ihrer Bauart besonders: Einerseits verwendet sie nämlich keine Mikrotaster, welche einen Schaltkreis schließen, sondern mit einem leitfähigen Material bedampfte Plastikscheiben, die mittels Schaumstoff an den Tastenkappen angebracht sind und somit die Kapazität des Schaltkreises verändern, sobald man sie senkt oder hebt – eine Technik von der wir bis zu diesem Punkt nicht wussten, dass sie im Computerbereich existiert. Diese Bauart wurde recht schnell vom Markt verdrängt, erleichterte aber den weiteren Reparaturverlauf sehr. Als Fehlerquelle konnten mit sehr hoher Wahrscheinlichkeit die Schaumstoffpads identifiziert werden, welche im Laufe der Jahre spröde wurden und sich dann zersetzten (Abb. 4.4.7), wonach die Plastikscheiben dauerhaft auf der Tastaturplatine aufliegen und die Tasten damit aus elektronischer Sicht für den Rechner ‚klemmen'. Die exotische Technik der Kapazitätsänderung zum Registrieren von Tastendrucken stellte sich, wie geschrieben, an diesem Punkt als vorteilhaft heraus: Die ‚nackte' Tastaturplatine ohne Tastenmechanik lässt sich deswegen nämlich prinzipiell auch mit den Fingern bedienen (wenn auch nicht ganz ohne Probleme – wie weiter unten gezeigt wird).

Abb. 4.4.7: Die Tastatur mit abgezogenen Tastenkappen zeigt die zerfallenen Schaumstoffpads. Die weiße Substanz zwischen den Tasten (Bildmitte) ist ein verlassener Insektenkokon.

Zusätzlich zu dieser Fehlerbeschreibung fand sich noch die Website von Herb Johnson[165], auf welcher die Anleitung für den Bau eines Tastatur-Testers zu finden ist. So konnte die Funktion der Tastatur überprüft werden – unabhängig vom Rechner und ohne sie komplett reparieren zu müssen. Dieser Adapter (Abb. 4.4.8) wurde mit vorhandenen Bauteilen schnell auf einer Lochrasterplatine nachgebaut. Daraufhin wurde die Tastatur von ihren Tasten befreit. Ein Blick auf die Schaumstoffpads ergab das, was Battle auf seiner Webseite schildert: Diese hatten sich im Laufe der Zeit nahezu vollständig zersetzt und zusätzlich die beschichteten Plastikplättchen angegriffen. Mit dem Tastatur-Adapter ließ sich recht schnell feststellen, ob die Elektronik der Tastatur funktioniert. ‚Tastendrucke', welche mit den nack-

164 http://www.sol20.org [letzter Abruf: 28.06.2017].

165 http://www.retrotechnology.com/restore/sol_keys.html#display [letzter Abruf: 28.06.2017].

ten Fingern ausgelöst werden konnten, wurden korrekt auf dem Test-Adapter angezeigt (dieser zeigte die unterschiedlichen 8-Bit-Muster für die Zeichen sowie das Strobe-Signal zum Einlesen des Musters mittels Leuchtdioden an). In der Hoffnung, dass sich der Rechner mit der ‚nackten' Tastaturplatine (Abb. 4.4.9) bedienen lässt, wurde diese wieder angeschlossen – jedoch vergebens: Es wurden zwar Zeichen entgegengenommen, allerdings prellten[166] diese so stark, dass anstatt einzelner Symbole quasi zufällige Zeichenketten von der Tastatur an den Rechner übertragen wurden. Damit ließ sich allerdings bereits ein großer Teil der Reparatur abhaken – elektronisch schien der Rechner das zu tun, was er sollte. Lediglich die Auslösemechanik der Tastatur war reparaturbedürftig.

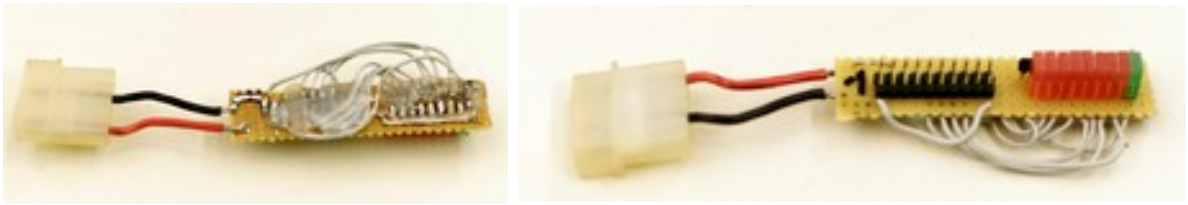


Abb. 4.4.8: selbst gebauter Adapter zum Testen der Tastatur-Funktionen

Auf den beiden vorher benannten Internetseiten wurde dieses Thema ausführlich behandelt und konnte so von uns als Reparatur-Information genutzt werden. Die originalen Schaumstoffpads und zugehörigen Plättchen werden seit circa 15 Jahren nicht mehr kommerziell vertrieben. Es gibt aber Möglichkeiten diese nachzubauen. Dazu wird Schaumstoff mit doppelseitigem Klebeband auf beiden Seiten ausgestanzt, auf der oberen Seite ein Plastikplättchen aufgeklebt, welches von der Tastenmechanik gehalten wird, und die Unterseite, die Kontakt mit der Tastaturplatine hat, mit Mylarfolie versehen. Vorgefertigte Schaumstoffpads gab es auf eBay von einem Dritthersteller zu kaufen[167], so dass ein Ausstanzen durch uns entfiel. Für den Ersatz der Mylarfolie gab es mehrere Optionen: Innenseiten von Kartoffelchipstüten, Heliumballons aus Geschenkshops oder Erste-Hilfe-Decken wurden als potentielle Quellen benannt. Die Entscheidung fiel auf Mylarfolie für Gewächshäuser.

166 Entgegen der technischen Definition von Prellen als mechanischer Erschütterung des Schalters [vgl. Baumann/Lanz 1998:166] wurde hier wahrscheinlich das Zittern des Fingers von den Sensoren als mehrfache Eingabe gewertet.

167 https://www.ebay.de/itm/121266887970 [letzter Abruf: 06.06.2019].

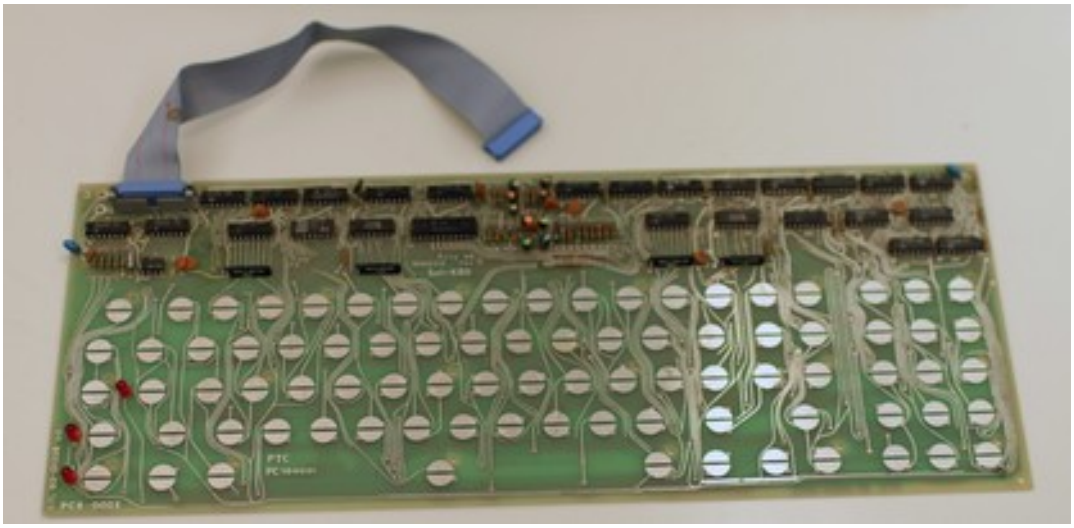


Abb. 4.4.9: Gereinigtes Tastaturfeld (vor dem Bekleben mit neuen Schaumstoffpads)

Nach dem Eintreffen der Schaumstoffpads und der Mylarfolie sollte die Reparatur fortgesetzt werden; die Mylarfolie erwies sich jedoch als unbrauchbar. Sie war nämlich mit einer zusätzlichen Plastikschicht versehen, so dass sie nicht leitfähig war, wie ein Test mit dem Multimeter ergab. Ein Entfernen dieser Schicht war nicht möglich. Zum Test wurde dennoch eine einzelne Taste damit versehen. Dazu wurden die alten Schaumstoffreste, Plättchen und die Folie aus einer Taste entfernt, das Plättchen von Klebe- und Schaumstoffresten befreit und ein neues ‚Sandwich' aus Plastikplättchen, Schaumstoff und Mylarfolie gebaut – ein Arbeitsschritt, der pro Taste circa drei bis fünf Minuten in Anspruch nahm, da einerseits das Entfernen der Plastikplättchen kompliziert war und zum anderen die Mylarfolie manuell auf das richtige Format zurechtgeschnitten werden musste. Die gesamte Tastatur des SOL-20 besitzt 85 Tasten. Daher fand der Test zunächst mit nur einer Taste statt. Wie erwartet, erbrachte er keine Funktion. Daher wurde nun nach anderen, vorhandenen Materialien Ausschau gehalten. Ein erster Test mit Alufolie erwies sich als nicht zufriedenstellend, da die Tasten ähnlich wie mit den nackten Fingern, stark prellten (also nicht korrekt auslösten und somit Zufallszeichen produzierten). Die Wahl fiel schließlich auf etwas, was per Zufall in den Blick kam aber gute Ergebnisse lieferte: elektrosensitives Metallschicht-Papier aus einem Drucker für einen anderen historischen Computer des Signallabors, welches in ausreichenden Mengen verfügbar war. Dieses Papier war überdies auch leichter zuschneidbar als Mylar- und Alufolie. Die komplette Reparatur der Tastaturstempel dauerte circa vier Stunden. Als Ergebnis (Abb. 4.4.10) besaß der Sol-20 danach eine „recht gut" funktionierenden Tastatur. „Recht gut" deshalb, weil einige wenige Tasten immer noch prellten. Diese prellenden Tastenstempel wurden dann auf den Zehner-Ziffernblock umgelegt, da dessen Tasten redundant sind (alle Ziffern befinden sich auch oberhalb der Buchstabenreihe).

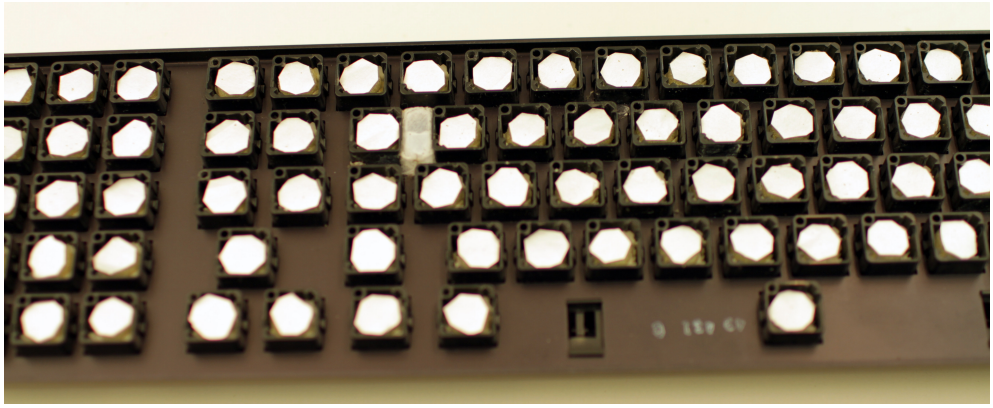


Abb. 4.4.10: Die mit neuen Schaumstoffpads und Kontaktfolie versehene Tastatur

4.4.3.2 RAM

Der Sol-20 war nun funktionstüchtig und konnte wieder betrieben werden. Ein Blick in das sehr detaillierte Handbuch, welches auch als Aufbauanleitung diente (der Rechner wurde, wie geschrieben, sowohl als Fertiggerät als auch als Bausatz vertrieben), offenbarte die Möglichkeit einzelne Bytes in das RAM zu schreiben und wieder auszulesen, was auch funktionierte. Der nächste Schritt war nun Software in den Rechner zu laden.

Dazu waren ab Werk drei Möglichkeiten vorgesehen: Programme manuell über das implementierte Monitorprogramm einzugeben, was sehr mühselig (übliche Programme sind circa zwei DIN-A4-Seiten lang) und auch sehr fehleranfällig (es handelt sich dabei um reine Zahlenfolgen ohne automatisierte Korrektheitskontrolle) ist. Alternativ kann man Programme, welche sich als Listing auf einem heute üblichen PC befinden, über die serielle Schnittstelle in den Sol-20 übertragen. Leider ist aber diese Schnittstelle heute kaum noch in Computern vorhanden[168] und hat obendrein mehrere Standards durchlaufen (Übertragungsraten, Verdrahtungsschema, Spannungen), so dass dies im Fehlerfall den Sol-20 sogar beschädigen könnte. Die Entscheidung fiel auf die dritte, typischste ‚damalige' Variante Software über den Kassettenport des Computers einzuspielen (wenn auch mit modernen Hilfsmitteln). Ein Sol-20-User hatte bereits einen Großteil der verfügbaren Software mit den vorher erwähnten Methoden in seinen Sol-20 eingetippt bzw. überspielt, die Programme als .wav-Sounddateien abgespeichert und auf seiner Internetseite zur Verfügung gestellt.[169]

Anstatt eines Kassettenrekorders wurde ein Smartphone (und später ein MP3-Recorder mit SD-Karte [Abb. 4.4.16]) als Audio-Abspielgerät verwendet, da auf Kassetten ohnehin lediglich analoge Töne abgespeichert sind, welche erst im Rechner digitalisiert werden. Dieses Prinzip der Übertragung von Software über das Smartphone konnte bereits mehrfach bei anderen Computern aus den 1980er-Jahren mehrfach erfolgreich angewendet werden – wie nun auch beim Sol-20. Der Großteil der so in den Computer geladenen Programme funktionierte

168 Es existieren Adapter-Lösungen, um den USB-Port als RS-232-kompatible Schnittstelle zu nutzen.

169 http://www.neoncluster.com/projects-sol20/sol20-wav.html [letzter Abruf: 25.11.2016].

– lediglich der BASIC-Interpreter von *Microsoft* führte zu der Fehlermeldung, dass zu wenig Speicher verfügbar sei – obwohl mit 32 Kilobyte eigentlich ausreichend RAM eingebaut war. Es lag also noch ein weiterer Fehler vor. Eine kurze Verifikation mittels Emulator[170] bestätigte, dass der BASIC-Interpreter mit 32 Kilobyte RAM funktionieren sollte. Der Sol-20 lag nun zunächst in einer funktionierenden 16-Kilobyte-Variante vor. Die zweite 16-Kilobyte-RAM-Karte hatten wir zur Sicherheit aus dem Slot gezogen, um so möglichen Konflikten vorzubeugen. Einschränkungen waren dadurch nicht feststellbar. (Abb. 4.4.11)

Durch das eingebaute Monitorprogramm war es zwar möglich Werte in den RAM-Bereich der Speichererweiterung zu schreiben und auch wieder erfolgreich auszulesen. Die Karte schien also prinzipiell zu funktionieren. Ein provisorischer ‚Fingertest' (fühlen, ob sich einzelne RAM-Chips unverhältnismäßig stark erhitzen) ergab keine Auffälligkeiten. Ein Auslesen des kompletten RAM-Bereichs erbrachte ein regelmäßiges Muster, was dem normalen Initialzustand von dynamischem RAM entspricht (32 mal 00, 32 mal FF im Wechsel). Es war also davon auszugehen, dass kein Defekt auf der Steckkarte vorliegt, sondern dass das Problem woanders zu suchen ist.

Abb. 4.4.11: Zu Testzwecken kann eine Karte von oben in den S-100-Bus gesteckt werden. Hier wird die scheinbar defekte RAM-Karte getestet, aus der ein RAM-Baustein (2. Reihe von unten, 6. IC von links) gezogen wurde.

Ein Blick auf die Karte selbst, sowie in die Anleitungen zum Sol-20 und zur Karte offenbarte ein ‚triviales' Problem: RAM auf Zusatzkarten muss im Sol-20 in einem durchgehenden Block liegen. Eingestellt war, dass Karte 1 die Adressen 0 bis 16383 enthält und Karte 2 Adressen 32768 bis 49151. Ein kompletter 16-Kilobyte-Block zwischen diesen beiden Karten war also gar nicht zugewiesen. Da uns der Computer vom Eigentümer mit dem Hinweis „generell funktionsfähig" überlassen wurde, gingen wir davon aus, dass dieser auch korrekt konfiguriert sei. Denkbar ist, dass der Sol-20 zuvor einmal mit drei Karten (also mit 48 Kilobyte RAM) betrieben worden war und eine davon irgendwann aus dem Rechner entfernt wurde. Die Adressierung war zwar schon vorher auffällig, allerdings war zunächst nicht klar,

170 http://www.sol20.org/solace.html [letzter Abruf: 11.01.2019].

dass diese ‚Speicherlücke' (englisch: Memory Gap) ein Problem darstellen könnte.[171] Nachdem die zweite Karte so umadressiert worden war, so dass sie direkt im Anschluss an Karte 1 liegt (die Karte besitzt dafür einen praktischen Jumperblock, so dass lediglich 4 Schalter umgestellt werden müssen, statt an der Karte zu löten), lief schließlich auch das MICROSOFT-BASIC.

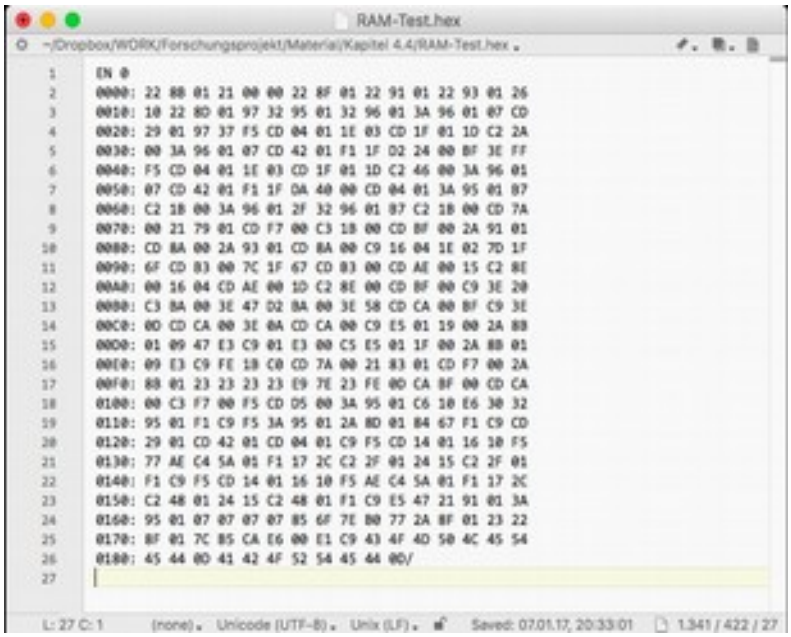


Abb. 4.4.12: Das Hex-Listing des Maschinensprache-Testprogramms, das in den Sol-20 eigetippt werden musste

Zur vollständigen Endkontrolle war in der Anleitung der Erweiterung ein Testprogramm als Assembler-Listing angegeben. [Vgl. Processor Technology 1978] Dieses enthält die Opcodes und deren Argumente ebenfalls als Hexadezimalzahlen, welche über das im ROM integrierte Monitorprogramm in den Rechner eingegeben werden können (so dass doch noch ein Hex-Listings einzutippen war. [Abb. 4.4.12]) Nach 30 Minuten und mehreren Kontrollen lief dann auch dieses Programm und meldete keine Speicher-Fehler. Zur Verifikation wurde das Programm mit einem Kassettenrekorder auf einer handelsüblichen Compact Cassette abgespeichert (das zuvor verwendete Smartphone unterstützt keine Aufnahmen über den Klinkensteckerport) und nach dem Einbauen eines Fehlers in der Speicherkarte (entfernen eines Speicherchips [Abb. 4.4.13]) wieder in den Sol-20 eingelesen und erneut ausgeführt, wobei der fingierte Fehler korrekt gefunden und angezeigt wurde. Der Sol-20 war damit wieder in vollem Umfang funktional.

171 In Computern späterer Zeit ist dieses Problem nicht mehr vorhanden. Bei genauerer Betrachtung scheint aber sinnvoll für ‚simple' Computer, dass RAM entweder an festen Adressen liegt oder, wenn, wie in diesem Fall, der Rechner ab Werk erweiterbar ist, in einem Block liegen muss, da dann die Adressen von der ersten RAM-Adresse durchgezählt werden können, bis kein RAM mehr vorhanden ist.

Abb. 4.4.13: Zu Testzwecken wurde ein RAM-Baustein aus der Karte (Stelle rot markiert) gezogen und ein Speichertest durchgeführt. Das „XG“ (Stelle rot markiert) im oberen linken Buchstabenblock zeigt den fehlenden Baustein.

4.4.3.3 Zusammenfassung

Der Reparaturbericht beschreibt detailliert die Informationsquellen, Methoden, Werkzeuge und Materialien, die dem Reparateur Marius Groth für die Reparatur des Computers im Rahmen hobbyistischer Möglichkeiten zur Verfügung standen und angewandt wurden. Diese seien hier noch einmal stichpunktartig zusammengefasst:

Werkzeuge:

- Multimeter zur Spannungs- und Widerstandsmessung
- Pinsel zur Staubentfernung
- Step-Down-Konverter zur Anpassung der Netzspannung
- Lötkolben und Heißklebepistole zur Herstellung des Tastatur-Testers
- Mobiltelefon als Audiowiedergabegerät und für Internetrecherchen
- 3,5-mm-Klinkenstecker-Kabel zum Anschluss des Computers an Smartphone und Kassettenrecorder
- Sol-20-Emulator zur Verifikation von Programmfunktionen
- Finger zum Andrücken der ICs, zur Herstellung von elektrischem Kontakt (Circuit Bending [vgl. Braguinski 2014]), zum Wärmetest der dRAM-Bausteine undnzur Programmeingabe

- Schraubenzieher zum Entfernen der Schaumstoffpad-Reste (der Sol-20 ist mit handlösbaren Schrauben verschlossen)
- Schere zum Ausschneiden der Kontaktfolie
- Audiokassettenrecorder und MP3-Recorder zum Speichern und Laden des abgetippten Programms

Arbeitsmittel:

- Isopropanol zur Reinigung von Kontakten und zur Entfernung von Klebstoffresten
- Mylarfolie, Aluminiumfolie und Metallschichtpapier als Ersatz für die Kontaktfolie
- Schaumstoffpads als Ersatz für die defekten originalen Schaumstoffpads
- doppelseitiges Klebeband zur Befestigung der Kontaktfolie auf den Schaumstoffpads und an den Tasten
- Diverse elektronische Bauteile und Lochraster-Platine zum Aufbau des Tastatur-Testers
- Audiokassette
- Papier zum Ausdrucken des Hex-Codes des Speichertestprogramms

Reparatur-Methoden:

- Sichtprüfung auf mechanische Beschädigungen und Korrosionen
- Spannungsmessung der korrekten Spannungen am Netzteil und der internen Elektronik
- Widerstandsmessung zur Prüfung von Kontakten und zum Test der Metallfolie
- Software-Test der Hardwarefunktionen

Informationsquellen:

- Internet: zur Ermittlung von Reparaturratschlägen, als Quelle für Schaltpläne und .wav-Dateien
- Reparatur-Erfahrung: vergleichbare Probleme bei vorherigen Computern

Bezugsquellen:

- eigene Material- und Bauteilsammlung: Isopropanol, elektronische Bauteile für Tastatur-Tester
- Elektronikbedarfshandel (Conrad Elektronik, Segor Elektronik, Reichelt)
- Internet (eBay): Mylarfolie, Tastaturstempel

Markant ist insbesondere der Einsatz von Materialien und Werkzeugen, die im ‚Hausgebrauch' (Isopropanol, Pinsel, …) zur Verfügung stehen oder von Materialien, die für andere Zwecke bestimmt sind (Musikkassettenrecorder, Handy, Audiokassetten, Mylarfolie, Klebeband, …) aber hier ‚umgenutzt' werden. Ebenso ist die Verwendung der Finger zu Reparatur- und Testzwecken hervorzuheben; die Verwendung des Attributs „hands-on" hat angesichts dieser Praktiken durchaus ihre Berechtigung.

Als Informationsquellen diente hier ausschließlich das Internet (und sekundär die von dort bezogenen Manuale, Schaltpläne und Reparaturtipps). Die dort archivierten Informationen über den Sol-20 wurden sämtlich von Hobbyisten gesammelt oder erstellt und sind dort mit dem Ziel hinterlegt worden, die Erinnerung und die dazugehörigen Informationen über diesen seltenen Homecomputer zu bewahren. Anders als in einem professionellen Archiv oder Museum ist der Bestand dieser Informationen weder fachlich strukturiert noch katalogisiert oder gar gesichert; während der Reparaturarbeiten ‚verschwand' die Seite mit den Sol-20-Programmen im .wav-Format, um dann kurze Zeit später unter einem anderen Domainnamen wieder verfügbar zu sein. Das beständige herunterladen, verarbeiten, erweitern und wieder hochladen (spiegeln) von Informationen gehört zum modus operandi der Retrocomputing-Szenen und sichert den Fortbestand, die Erweiterung und Proliferation von Informationen (als *knowledge preservation*) – wenngleich unter für akademische Zwecke erschwerten Bedingungen.

4.4.4 Reparieren als Spiel

Das Test-Programm [Processor Technology 1978:A2-4ff.], das zum Auffinden des Speicherfehlers genutzt wurde, kann vor dem Hintergrund der hier auch didaktisch motivierten Diskussion mehr als nur als Werkzeug zum „Hardware Test" [vgl. Anhang A2-1] verstanden werden. Das Verständnis des Fehlers, der sich letztlich gar nicht als Defekt, sondern als Falschkonfiguration offenbarte, basiert ganz wesentlich auf der *dialogischen Arbeit zwischen dem Reparateur und dem Reparatur-Objekt*, womit sich die Aspekte von *Trial-and-Error* und *E-Learning* als Lernmethoden verdeutlichen lassen. Hierzu sollen Teile der Test-Software auf genau diesen Aspekt hin interpretiert werden.

Die Programme zum Test der „16KRA"-Speichererweiterung waren seinerzeit auf Kassette verfügbar: „Your Processor Technology dealer may have this program on a tape which you may copy, to avoid having to key in such a long program. As an owner of the l6KRA, you have a right to copy this program without violation of the copyright." [A2-4]. Die Testprogramme sollen standardmäßig nach der Erstinstallation der Erweiterung [3-1] oder bei Problemen mit dieser ausgeführt werden. Liegt keine Kassette mit der Software vor, sind die Programme im Appendix des „User's Manual" [Processor Technology 1978] als kommentierte Assembler-Listings abgedruckt. Diese Listings erläutern die Funktionen des Maschinencodes mit Kommentaren, so dass der Nutzer ihn verstehen und gegebenenfalls nach seinen Bedürfnissen konfigurieren kann (etwa, wenn ein anderes Betriebssystem genutzt wird, das andere I/O-Routinen und -Adressen verwendet. [vgl. A2-5]). Enthalten sind zwei Test-Programme: 16KRA Short Memory Test [A2-2f.] und 16KRA Long Memory Test Program [A2-4–A2-9]; hiervon sollte das *zweite* für den oben genannten Speichertest genutzt werden, weil es Analysen einzelner Speicherbausteine ermöglicht. Eine Kassette mit den Programmen lag nicht vor, daher musste das Maschinensprache-Programm aus dem „User's Manual" in den Computer eingetippt werden. Der Reparateur verfügte nach eigenen Angaben über keine As-

sembler-Kenntnisse und konnte das Programm, dessen hexadezimale Opcodes und Daten er eingab, in seinen Details daher nicht verstehen, entwickelte beim Eingeben aber ein Grundverständnis der Zusammenhänge einiger Opcodes mit ihren Argumenten (vgl. Anhang B). Es soll zunächst vorgestellt werden.

Das Programm verfügt über folgende Funktionen: Es testet eine 16 Kilobyte große Speicherbank, indem es deren Adressen sukzessive beschreibt und wieder ausliest. Nach dem Start des Programms hat der Nutzer die Wahl, ob er den gesamten Speicher einmal oder mehrfach/kontinuierlich testen lassen möchte. Das Programm erstellt dann eine Bildschirmausgabe nach folgendem Muster [A2-4]:

```
GG  GG  GG  GG          GG  GG  GG  GG
GG  XG  GG  GG          GG  GG  GG  GG
```

Ein Buchstabe steht dabei für einen 512-Byte großen Bereich der 16-Kilobyte-Speichererweiterung, der jeweils durch einen dRAM-Baustein (Typ: TMS 40L44) repräsentiert ist:

> Each character in the map represents one of the 32 memory ICs: Ul-8, Ull-18, U2l-28, or U32-39. If the l6KRA board is viewed so that the assembly legend is in normal reading position, each character in the map is in a position which corresponds to an IC on the board. The upper left-hand ‚G' in the display represents Ul, while the bottom right-hand ‚G' represents U39, etc. ‚G' means the IC is good, ‚X' means bad. [A2-4]

Der Nutzer erhält durch den Ablauf des Programms also bereits Kenntnisse über den prinzipiellen Aufbau der Speicherkarte. Die im hier reparierten Sol-20 verbaute Karte weicht in ihrem Aufbau jedoch von der Beschreibung ab: Die ICs sind nicht in zwei Zeilen und 16 Spalten, sondern in vier Zeilen und 8 Spalten angeordnet. Über die Platinenbeschriftung (U1-U39) ließen sie sich jedoch mit der Beschreibung im „User's Manual" identifizieren.

Einige der im diskutierten Zusammenhang relevanten Programmstellen werden nun analysiert. Dabei wird die Zeilennummer des Listings als Referenz angegeben (vgl. Anhang A). Der systematische Aufbau ist wie folgt:

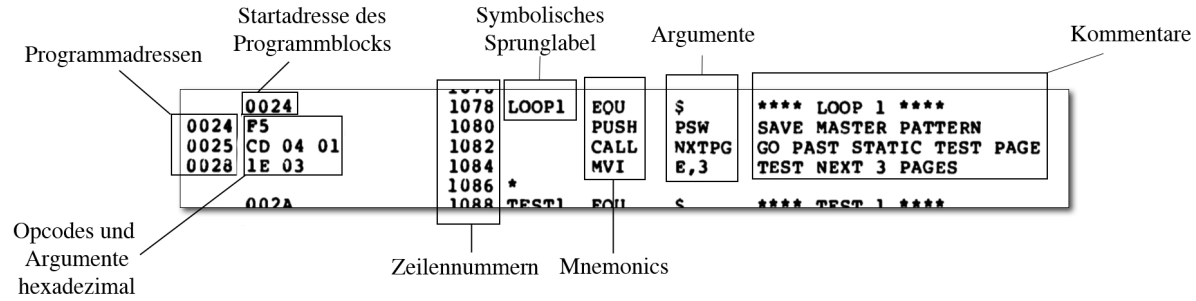


Abb. 4.4.14: Der Aufbau des Assembler-Listings

Das Listing ist horizontal in zwei große Bereiche geteilt (vgl. Abb. 4.4.14). Auf der linken Seite befinden sich die *Programmadressen* ab 0000_{16}, in die der Code eingegeben werden soll

und direkt rechts daneben die Hexadezimalzahlen der *Opcodes* und *Argumente* (Daten, Adressen, Register). Allein diese Zahlen können über das Monitorprogramm in den Sol-20 eingegeben werden. Über diesen Hex-Zahlen stehen zeitweilig die *Startadressen* des jeweiligen Programmblocks, die auf der rechten Seite den Adressen der Labels entsprechen. Im rechten Block stehen ganz links die *Zeilennummern* des Programms. Diese Nummerierung dient dem Programmierer/Leser als Ortungshilfe und ist von Programmiersprachen wie BASIC oder COMAL inspiriert hier jedoch ohne Bedeutung für die Programmfunktion. Die Zweierabstände lassen nachträgliches Einfügen von Zeilen zu. (Im Listing wurden offenbar die Zeilen 1069, 1099 und 1139 nachträglich eingefügt, was editionsphilologische Hinweise zur Programmentstehung geben könnte.) Direkt rechts daneben sind symbolische *Labelnamen* angegeben. Diese können von Assemblierern interpretiert (also in konkrete Adressen übersetzt) werden, dienen hier aber vor allen der Strukturierung des Codes und damit der Leseerleichterung. Selbiges gilt für die rechts daneben stehenden *Mnemonics*, welche Merkhilfen für die Opcodes des Prozessors sind. Rechts daneben finden sich die *Argumente* der Opcodes (Daten, Register und Adressen), die hier ebenfalls oft symbolisch angegeben sind, etwa, wenn nicht zu einer konkreten Adresse, sondern zu einem Label gesprungen wird, wie hier bei 1082 „NXTPG“, das für die reale Adresse 1376_{16} steht. Die Verwendung von symbolischen Labels und Sprungzielangaben erleichtert auch die dynamische Positionierung des Codes im Speicher. Dieser Vorteil ergibt sich allerdings nur, wenn das Programm mit einem Assemblierer übersetzt wird. Bei der manuellen Eingabe der Opcodes müssen die Label-Angaben vom Typisten vor der Eingabe selbst in konkrete Adressen umgerechnet werden. Können Label, Mnemonics und symbolische Argumente noch von Assemblierern in Maschinencode übersetzt werden, so gilt das für die ganz rechte Spalte mit den *Kommentaren* nicht. Sie sind allein für den Leser des Programmcodes bestimmt und informieren ihn über die Funktion der jeweiligen Programmzeile im Gesamtzusammenhang des Programms oder Algorithmus.

4.4.4.1 Software-Test, Hardware-Test, User-Test

Das 16KRA Long Memory Test Program testet den Speicher des Computers, auf dem es selbst ausgeführt wird. Deshalb ist es nötig, dass das Programm ein Speicher-Management verwendet, dass zwischen verschiedenen ‚Kategorien‘ von Speichern unterscheidet. Ein Blick in das Listing des Programms zeigt, dass für diesen Zweck drei verschiedene dieser ‚Kategorien‘ verwendet werden:

1. Der Adressbereich 0000_{16}-0196_{16}: Hierin ist das Programm selbst gespeichert; dieser RAM-Bereich gehört zum werkseitig verbauten Speicher (also nicht zur 16KRA-Karte) und kann/darf nicht getestet werden, weil sich das Programm sonst selbst überschreiben würde.

2. Ein WORKING STORAGE ab 1000_{16} (Programmzeile 1044 usw.): Das Registerpaar H und L dient als der Prozessorinterne Speicher, der die RAM-Adressen der zu lesenden Daten zur Weiterverarbeitung enthält.

3. Der auszulesende/zu testende Speicherbereich 4000_{16}-$7FFF_{16}$. Dieser ist im Programm durch die Angabe der Page und der Zeilen-Bits vorgegeben (Programmzeilen 1578-1584).

Die 16KRA-Speicherkarte enthält 32 ICs. Diese sind als 4 Pages mit jeweils 4 Kilobyte RAM organisiert. Jedes der 8 ICs einer Page wird mit einem spezifischen Bit (0-7) adressiert [vgl. ebd.: 2-3]. Mit Hilfe der Adressarithmetik (Programmzeilen 1376-1394) werden die Page-Nummern der jeweiligen Bänke nacheinander getestet. Dieser Test besteht darin, die Speicherzellen eines jeden ICs mit dem Wert 00000001_2 (Programmzeile 1586) zu beschreiben (Programmzeile 1426-1464) und danach wieder auszulesen (Programmzeilen 1466-1502).

Ergibt sich hierbei ein Unterschied zwischen dem geschriebenen und dem gelesenen Wert, deutet dies auf einen Fehler des Bausteins hin. Dieser Fall wird registriert (Programmzeilen 1504-1532) und in einer Tabelle, die sich in den RAM-Adressen ab 1000_{16} (Programmzeile 1054: $H=10_{16}$ ist das Highbyte der Tabellen-Adresse, $L=00_{16}$ ist das Lowbyte) befindet, protokolliert (Programmzeilen 1522-1530; das Pseudoregister M bezieht sich auf den 16-Bit-Wert, der in den Registern H und L gespeichert ist. Diese Register enthalten den Adresszähler für den RAM-Speicher, in welchem die Informationen über korrekte oder fehlerhafte Lese-/Schreibvorgänge gesichert werden.)

Bei diesem Prüfprozess wird sukzessive eine ‚Karte' des geprüften Speicherbereichs angelegt, in der die Information, ob die jeweilige Adresse funktioniert oder defekt ist, mit dem Wert 1 oder 0 eingetragen wird. Die Speicherzellen dieser Karte dürfen nun nicht selbst so groß wie das „Territorium" (vgl. Borges 1975:131) sein, das sie kartographieren, weil die 1:1-Kartengröße sonst 16 Kilobyte überschreiten würde. Im selben Speicherbereich liegt zwischen 0000_{16}-0196_{16} das Prüfprogramm, weshalb der WORKING STORAGE erst ab Adresse 1000_{16} beginnen kann. Eine Karte, die also die zu testenden 16 Kilobyte byteweise abbilden würde, würde den untersten 16-Kilobyte-Bereich verlassen und in den zweiten, den zu testenden 16-Kilobyte-Bereich hineinschreiben. Die Programmierer nutzten deshalb für die binäre Information „defekt/nicht defekt" ein so genanntes *Bitboard*: Die Informationen werden als jeweils 1 Bit in eine Speicherzelle geschrieben (Programmzeilen 1072 – hier wird das Masterpattern „11111111" geschrieben, also alle Bits auf „nicht defekt" gesetzt). Die Bits dieser Speicherzelle werden dann durch Rotieroperationen weitergeschaltet (Programmzeilen 1099, 1139 usw.). Ergibt ein Prüfprozess „defekt", wird aus einer „1" im Bitpattern eine „0". Auf diese Weise können 8 Informationen in einer Speicherzelle hinterlegt werden; die nächsten Informationen werden dann in der darauffolgenden Speicherzelle gesichert usw. Auf diese Weise kann die ‚Karte' in 2048 Byte hinterlegt werden. (Der WORKING STORAGE umfasst daher die Adressen 1000_{16}-1800_{16}.

Solche Bitboards besaßen gerade bei Computern mit wenig verfügbarem RAM-Speicher große Bedeutung in der Programmierung. Zum einen konnte mit ihrer Hilfe Binär-Information in kleinstmöglichen Speicherbereichen hinterlegt werden; zum anderen ließen sich damit Matrizen für Spielfelder im Speicher duplizieren und dort mithilfe logischer und Bit-Operationen verarbeiten. In Kapitel 4.2 wurde dies am Beispiel eines „Game of Life"-Programms demonstriert.

Das Auslesen der Bitboard-Karte erfolgt dann nach dem umgekehrten Prinzip: Die Bausteine, in denen die Karte abgelegt ist, werden sukzessive ausgelesen, indem ihre Inhalte durch das Carry-Register rotiert werden. Kommt dort eine 0 an (Programmzeilen 1208-1216), dann wird der Standardfall, dass das IC die Ausgabe „G" erzeugt (Programmzeile 1262) korrigiert (Programmzeilen 1270-1274) und die Ausgabe „X" erzeugt (Programmzeilen 1264-1266). So wird die unsichtbare Karte (das Bitboard) als sichtbare „MAP" (Programmzeilen 1184-1290) auf dem Bildschirm ausgeheben.

Die Ausgabe des ersten Programmdurchlaufs ergab, dass sämtliche ICs der Karte defekt sind – also in sie geschriebene Werte nicht wieder korrekt ausgelesen werden konnten. Dieser unwahrscheinliche Fall führte auf den richtigen Fehler: Der Speicherbereich der Karte (4000_{16}-$7FFF_{16}$) konnte gar nicht beschrieben werden; wohl aber der Speicherbereich danach (8000_{16}-$BFFF_{16}$). Der 8080-Mikroprozessor kann aber Speicherbereiche nur dann nutzen, wenn sie kontinuierlich aufeinander folgen. Die im Computer werkseitig verbaute 16-Kilobyte-SRAM-Karte belegt den Adressbereich 0000_{16}-$3FFF_{16}$. Die 16KRA-Karte hätte demzufolge für den Adressbereich 4000_{16}-$7FFF_{16}$ konfiguriert werden müssen. Die hierfür notwendige Einstellung der DIP-Schalter (vgl. Abb. 4.4.15 sowie [Processor Technology 1978:4-3]) adressierte die 16 Kilobyte der Karte jedoch in den Bereich 8000_{16}-$BFFF_{16}$. Nachdem diese Einstellung korrigiert worden war, konnte der Software-Test erfolgreich absolviert werden.

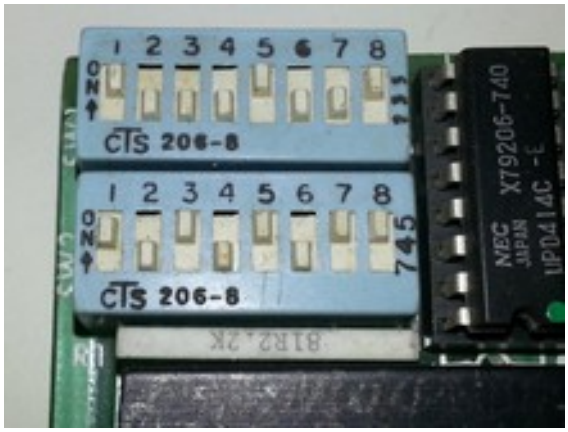


Abb. 4.4.15: Die fehlerhafte Einstellung der DIP-Schalter: Die Schalterpaare 1,2 (oben und unten) und 5,6 (oben und unten) stehen auf „ON-OFF", womit die komplette RAM-Karte die Adresse 10_2 kodiert, die für den Speicherbereich 8000_{16}-$BFFF_{16}$ steht. (Die übrigen Schalterpaare 3, 4 und 7, 8 [oben und unten] adressieren die jeweiligen Speicherbänke der Karte: 00_2, 01_2, 10_2 und 11_2.)

Um nun auszuschließen, dass die Karte überdies Fehler enthält, die die Software aufgrund eines Fehlers beim Abtippen bloß nicht anzeigt, wurde folgendes Experiment durchgeführt: Der Speicherbaustein U33 wurde aus seinem Sockel auf der Karte gezogen. Da die Ausgabe des Programms nicht differenziert, ob eine oder mehrere Adressen eines ICs nicht korrekt funktionieren oder der komplette IC fehlt, wird beides als „X" angezeigt. Eine phänomenolo-

gische Prüfung, die zwischen *Fehler* und *Fehlen* unterscheidet, kann von (der) Software nicht durchgeführt werden: Vom „Assembly language level“ [Tanenbaum 2006:5] kann der Computer nicht bis zum „Digital logic level“ [ebd.] hinab/hindurch blicken; diesen Durchblick muss sich der Reparateur durch Interpretation der Hardwareeffekte, die sich auf der Oberfläche (den Bildschirmausgaben) zeigen, erarbeiten. Dieser Durchblick ereignet sich auf spielerische Weise im Experiment: Der Reparateur testet die korrekte Funktionalität der Software, indem er diese die vermeintliche Fehlerhaftigkeit der Hardware erkennen lässt. Die Korrekte Funktion sowohl der Software als auch der Hardware ergibt sich damit als Beweis aus der Falschheit ihres Gegenteils.

4.4.4.2 Der Computer als Werkzeug und Werkobjekt

Die Einleitung zum 16KRA Long Memory Test Program enthält einen Widerspruch zum darauffolgenden Programm, den ein angehängtes Erratum des „User‘s Manual“ wie folgt beschreibt:

> Please note that the instructions for the 16KRA Long Memory Test on page A2-4 of the fourth printing of the 16KRA manual ask you to load the test into memory at address C900H, but the program listing which follows shows an assembly at address 0000. The program can run in built-in system RAM at C900 if the source code for the program is reassembled with this origin. If program is used as listed, it can run in a 4K memory board addressed at 0000, to test a 16K memory board addressed at 1000 as stated in the listing. [Processor Technology 1978:o.S.]

Dieser Widerspruch würde dann zu einem akuten Problem führen, wenn eine Speicherkarte, die im vierten 16-Kilobyte-Bereich ($C000_{16}$-$FFFF_{16}$) läge, selbst auf Fehler getestet werden müsste. Das Programm darf nämlich selbstverständlich nicht in dem Speicherbereich abgelegt werden, den es testen soll, sondern sollte vorzugsweise in einem Adressbereich residieren, der zum „built-in system memory“ [A2-4] gehört. Nur so lässt sich die Unterscheidung zwischen *Werkzeug* (der Speicherbereich des Programms) und *Werkobjekt* (zu testender Speicherbereich einer RAM-Erweiterung) aufrecht erhalten. Falls der Werkzeug-Speicherbereich selbst getestet werden muss, schlägt das Manual vor, das Programm zu „reassemblieren“ [vgl. ebd.], wobei dann sowohl die Adressen des Programmcodes als auch die in diesem verarbeiteten absoluten Adressdaten modifiziert werden müssen.

Diese Problematik, die aus der prinzipiellen Architektur von Von-Neumann-Maschinen (welche einen einzigen Speicher für Programme und Daten benutzen) herrührt, wirft eine Frage auf, die im folgenden Kapitel als eine didaktische Konstituente markiert wird: *E-Learning* bedeutet auch, dass das Werkzeug des Lernens gleichzeitig der Gegenstand des Lernens ist. Mit dem Computer lernt man etwas über den (selben) Computer. Diesen Prozess transponiert ein Test-Programm, das auf einem Computer läuft, dessen Speicher durch dieses Programm getestet werden soll, auf die technische Ebene. Der Reparateur analysiert das System, während

es sich selbst analysiert und zieht Schlüsse über dessen Funktionalität auf Basis der Ergebnisse des Selbsttests – er verhält sich damit zum System, wie durch die Kybernetik zweiter Ordnung beschrieben, als *Beobachter*. Er kann durch Eingriffe in die Hardware die Testergebnisse beeinflussen, um daraus Schlüsse über die Hardware und die Test-Software zu ziehen. Dieser Prozess soll im Folgenden in eine Überlegung des *Computers als ‚epistemisches Ding'* einfließen.

4.4.5 Historische Computer als epistemische Dinge

Im Vorausgegangenen zeigt sich bereits der eingangs genannte Hybridcharakter des zu reparierenden Computers als *Objekt* und zugleich *Werkzeug des Wissens*. Die Funktionalität des *Objektes* Sol-20 ließ sich nur erproben und bestätigen, indem man ihn darauf hin geprüft hat, ob er wieder als *Werkzeug* fungieren kann. Dies war aber nur möglich, indem man ihn, wie im abschließenden Experiment, mit einem Programm versehen hat, das seine Funktionalität bestätigt, indem der Computer ‚sich selbst prüft'. Im Folgenden soll dieser Doppelcharakter zu einer *Epistemologie des Defekts* umgedeutet werden, aus der sich – über zwei theoretische Ansätze – ein Mehrwert der *Reparatur als Wissensarbeit* (Knowledge Preservation) begründen lässt. *Diese epistemologische Perspektive stellt zudem die zentrale Motivation dar*, aus der heraus Computerarchäologie sich für die Reparatur historischer technischer Objekte interessiert. Es geht ihr nicht vorrangig darum, ein Objekt zurück in seinen ursprünglichen Funktionszusammenhang zu bringen, es wieder nutzbar zu machen, um so vielleicht der Obsoleszenzlogik (dem „moralischen Verschleiß" [vgl. Nake 2008:123]) von Industrieprodukten zu widersprechen. Dies wäre angesichts der Leistungsmerkmale des Sol-20 (verglichen mit heutigen Computern) auch keine sinnvolle Verwendungsweise. Es geht vielmehr darum, einen Computer zu einem Gegenstand des Wissens zu machen. Ein historischer Computer kann auf diese Weise über seine elektronische und informatische Beschreibung und über seine Reparatur Element einer Auseinandersetzung mit der (Computer)Geschichte werden.

Hierzu ist es jedoch zunächst notwendig, die Objektkategorien genauer zu definieren. Eine Differenzierung von *Werkobjekt* und *Werkzeug* ließe sich mit Martin Heideggers *Theorie der Zuhandenheit* vornehmen. In „Sein und Zeit" [Heidegger 1967] definiert er zunächst die *Zuhandenheit* (den Werkzeug-Charakter) am Beispiel eines Hammers [vgl. ebd.:68f.]: Ein Werkzeug muss, um als solches nutzbar zu sein, ‚unsichtbar' bleiben und sich in seinem Gebrauch erschöpfen. Problematisch wird dies jedoch dann, wenn das Werkzeug ‚in den Blick gerät' und damit wieder *vorhanden* wird, etwa, weil es defekt ist [vgl. ebd.:74]. Eine solche Störung macht das Werkzeug (wieder) sichtbar. Das defekte Werkzeug zeigt eine spezifische Eigenart in seiner (durch den Defekt verursachten) Vorhandenheit, die es von anderen vorhandenen Dingen (etwa den Werkstücken) unterschiedet: Es provoziert dazu, seine Zuhandenheit zu imaginieren und bestenfalls zu restituieren (es zu reparieren). Heidegger leitet aus diesem Doppelcharakter eine Unterscheidung von Praxis und Theorie ab:

> Der nur „theoretisch“ hinsehende Blick auf Dinge entbehrt des Verstehens von Zuhandenheit. Der gebrauchend-hantierende Umgang ist aber nicht blind, er hat seine eigene Sichtart, die das Hantieren führt und ihm seine spezifische Dinghaftigkeit verleiht. […] Das ‚praktische‘ Verhalten ist nicht „atheoretisch“ im Sinne der Sichtlosigkeit, und sein Unterschied gegen das theoretische Verhalten liegt nicht nur darin, daß hier betrachtet und dort gehandelt wird, und daß das Handeln, um nicht blind zu bleiben, theoretisches Erkennen anwendet, sondern das Betrachten ist so ursprünglich ein Besorgen, wie das Handeln seine Sicht hat. Das theoretische Verhalten ist unumsichtiges Nur-hinsehen. Das Hinsehen ist, weil unumsichtig, nicht regellos, seinen Kanon bildet es sich in der Methode. [Ebd.:69]

Demzufolge stellt also die Nutzung eines Werkzeuges eine Art ‚applizierte Theorie‘ und die theoretische Beschreibung eine Form von ‚imaginativer Invollzugsetzung‘ dar. Beides, *Theorien über die Dinge* und *Praktiken ihrer Verwendung*, vereinen sich in Heideggers Hammer-Beispiel. Diese Differenzierung und gleichzeitige Verschmelzung ist wissenschaftstheoretisch später durch Hans-Jörg Rheinberger wieder aufgegriffen und auf die Vorgehensweise wissenschaftlicher Untersuchungen, Untersuchungsobjekte und Untersuchungsapparate übertragen worden. Rheinberger unterscheidet dabei zwischen „epistemischen Dingen“ („Forschungsgegenstand, Wissenschaftsobjekt“) [Rheinberger 2000:53] und „technischen Dinge[n]“ („Experimentalbedingungen“) [ebd.:53] in Experimentalsystemen:

> Im Gegensatz zu den epistemischen Objekten müssen die Experimentalbedingungen innerhalb der jeweils gültigen Reinheits- und Präzisionsstandards von charakteristischer Bestimmtheit sein. Die Wissenschaftsobjekte werden von den Experimentalbedingungen im doppelten Sinne „eingefaßt“: Sie werden eingebettet und durch die Umfassung gleichzeitig begrenzt. [Ebd.:53]

Epistemische Dinge können durch „black boxing“ zu technischen Dingen werden [Ebd.:54]:

> Allerdings reflektiert dieser Ausdruck nur die eine Seite des Übergangs von einem epistemischen zu einem technischen Ding: den nachmaligen Routine-Charakter des transformierten Objekts. Mindestens ebenso wichtig sind jedoch die Auswirkungen dieses Vorgangs auf die neue Generation epistemischer Dinge, die gerade im Entstehen begriffen sind, die Eröffnung neuer Möglichkeiten der Untersuchung. [Rheinberger 2001:26]

Technische Dinge können andererseits allerdings auch zu epistemischen Dingen werden, wenn sie Fragen aufzuwerfen beginnen:

> Technische Gegenstände haben die Zwecke zu erfüllen, für die sie gebaut worden sind, es sind Maschinen, die definierte Antworten geben. Ein epistemisches Objekt hingegen ist in erster Linie eine Maschine, die Fragen aufwirft. Es ist in und aus sich selbst heraus nicht ein technisches Ding. [Rheinberger 2000:55]

Einer der Zustände, in welchem ein technisches Ding Fragen aufwirft, ist, wenn es *defekt* ist. Ein defekter Computer, sei er nun vollständig ausgefallen oder in bestimmten Funktionen gestört, wirft Fragen nach der Ursache des Defekts und seiner spezifischen Art, womit die technischen Details der Maschine in den Vorder-, der Nutzungscharakter jedoch in den Hintergrund geraten. Das epistemische Objekt *defekter Computer* stellt die Frage nach seiner Verfassung ins Zentrum.

Mit diesen Darstellungen steht nun eine Theorie zur Verfügung, die als *Epistemologie des Reparierens* genügen kann, wenn *die Tätigkeit des Reparierens selbst* noch in die Betrachtung einbezogen wird. Hier schlägt Rheinberger den Begriff des *Bastelns* vor, weil der Bastler experimentierend und nicht nach Schema vorgeht und weil Experimentatoren „ihre Geschichte nicht im Voraus erzählen [können]“, das heißt, stetig Neues und Unbekanntes hervorbringen:

> Ein Wissenschaftler ist vor allem ein „Bastler“, ein „bricoleur“, und nicht ein Ingenieur.[172] In ihrem nicht-technischen Charakter transzendiert das experimentelle Ensemble die technischen Objekte, aus denen es zusammengesetzt ist. [Rheinberger 2000:55f.]

Der Blick des Bastlers/Wissenschaftlers auf das Experimentalsystem fasst also stets den Doppelcharakter von technischen Objekten ins Auge. Die Verwendung des Werkzeugs geschieht für ihn deshalb auch stets reflektiert. Die Klassifikation des Experimentierens als „Basteln“ und damit des Wissenschaftlers als Bastler besitzt für Rheinberger aber zusätzlich noch eine wissenschaftshistorische Relevanz, führt sie doch die *Kontingenz von Entdeckungen* zurück in den Prozess der Wissenschaft:

> Dieses wilde Denken innerhalb der Wissenschaft, ohne das Rationalität, Logik und Präzision stumm und unfruchtbar blieben, diese „nächtliche Wissenschaft“, diese „Werkstatt des Möglichen“, dieses Labyrinth der Irrungen und Wirrungen ist gegen eine Geschichtsschreibung zu verteidigen, die versucht, die Entwicklung der modernen Wissenschaften in Standardisierung, Normierung und Regulierung aufgehen zu lassen. Auch die Präzisionsmessung gehört in diesen Zusammenhang, sofern sie nicht selbst als Forschungsprozeß betrieben wird. Dagegen setze ich den Begriff des Bastelns und bestehe auf den Werten des Unpräzisen, des Vor-Normativen, des nicht definierbaren und nicht standardisierbaren Überschusses als unverzichtbare Elemente des Forschungsprozesses […]. [Rheinberger 2000:56]

In der Folge ließe sich der oben dargestellte Reparaturprozess, vorgenommen von einem Bastler an einem ihm zuvor unbekannten Computer in diese Beschreibung integrieren. Wenn

172 Es sei ergänzt, dass der hier gebrauchte Begriff des Ingenieurs als bloßer ‚akademischer Techniker‘ dem Selbstverständnis des Ingenieurwesens keineswegs entspricht. Hierzu gehört seit Gründung des *VDI*, dass „die technischen Wissenschaften durch die Formulierung und die gemeinsame Lösung von Problemen gefördert werden“ [Scholl 1981:17]. Hierin offenbart sich bereits die Gradualität im Übergang von dem, was Rheinberger „Wissenschaftler“ und was er „Ingenieur“ nennt.

im Defekten aber Vorhandenes und Zuhandenes einander überlappen, technische zu epistemischen Dingen werden und die Reparaturmethoden sich zugleich an heuristischen Verfahren (*trial and error*) und autodidaktisch erworbenen, standardisierten Methoden orientieren, dann restituiert die Reparatur nicht bloß das vormalige Werkzeug/technische Objekt, sondern stellt zugleich eine Form von *Wissensarbeit* dar. Diese wird, angewendet auf ein historisches Objekt, welches aus dem dysfunktionalen Zustand von „Hardware" in den operativen Zustand von „Computer" versetzt wird, im Protokollieren in einem technischen Report zugleich ein Akt *operativer Historiografie*. Die Demonstration des funktionierenden Computers tritt neben den technischen Werkstattbericht und dessen theoretischer Flankierung als eine Einheit – als *Geschichtsarbeit*.

Der funktionierende Sol-20 erhält seinen Werkzeug-Charakter zurück, wird aber zugleich, weil er technisch obsolet ist, wohl nie mehr als reines Werkzeug genutzt werden. Er bleibt selbst dann, wenn er hinter seiner Funktionalität verschwindet (etwa, indem man ein Computerspiel auf ihm spielt) stets ein Beispiel sowohl für operative Geschichtsarbeit als auch für den Blick in die Black Box, die (funktionierende) Computer als Werkzeuge notwendig sein müssen. Für John von Neumann (und alle ihm nachfolgenden Computerarchitekten) war/ist der Computer noch ein epistemisches Ding, das es im Zuge des Blackboxing zu einem technischen Ding zu machen gilt. Für den Bastler stellt die „Black Box Computer" [Vgl. Becker 2012] wieder die reine Herausforderung eines epistemischen Dings dar.

4.4.6 Der Computer als Diagramm

Schließlich stellt sich die Frage, bis zu welchem Beschädigungsgrad ein Computer überhaupt noch repariert werden kann – oder mit anderen Worten: ab wann er seinen Status als epistemisches Ding *wieder verliert*. Im Unterschied zu anderen technischen Artefakten sind Computer als Werkzeuge Apparate, deren Werkzeug-Charakter darin besteht, gerade nicht das zu sein, was sie sind, weil ihre Aufgabe in der Simulation anderer Maschinen/Medien besteht. Als Computer selbst werden sie von Informatikern, Programmierern und Hobbyisten genutzt – mit der zuvor geschilderten Konsequenz, dass sie dabei einen Doppelcharakter zwischen Werkzeug und Werkstück erhalten. Auf der höchsten Abstraktionsstufe verlieren sie ihren materiellen Charakter (*die* Computer) und erscheinen als Architektur (*der* Computer), die sich mit Funktionsdiagrammen (z. B. Turingmaschinen) beschreiben lassen. Der eingangs in Erinnerung gerufene „First Draft" John von Neumanns leistet in seinem Blackboxing genau dies: aus der konkreten materiellen Maschine eine diagrammatische zu machen, die *rein theoretisch* über ihre prinzipiellen Funktionen beschrieben werden kann. (Nach den Ausführungen Heideggers zeigt sich aber, dass die theoretische Beschreibung und die konkrete Verwendung von Computern immer schon einander überlappen.)

Abb. 4.4.16: Der wieder lauffähige Sol-20 am Monitor und mit einem MP3-Recorder verbunden zeigt einen Speicherinhalt mit einem geladenen Programm.

Diese Beschreibbarkeit hat Konsequenzen für die Frage nach dem maximalen Beschädigungsgrad: Selbst dann, wenn der zu ‚reparierende' Computer als manifestes Objekt gar nicht mehr existiert[173] oder sogar nie existiert hat[174], lässt er sich als ein materielles, operatives Objekt restituieren, um im Anschluss (wieder) voll funktionsfähig zu sein. Diese restituierte Funktionalität kann jedoch nicht mehr in seiner Materialität liegen, sondern darin, dass der Charakter ‚des Computers' darin besteht eine andere Maschinen zu sein. Sobald er dieses Vermögen (wieder) besitzt, wird man ihn als repariert ansehen dürfen. Dass man aber auch hierfür die Ebene der abstrakten Diagramme verlassen muss, um sich der konkreten Technik zuzuwenden, schließt den in diesem Kapitel gezeichneten diskursiven Kreis.

173 Ein Beispiel hierfür ist das Restaurationsprojekt von Horst Zuse, der für das *Deutsche Technikmuseum* in Berlin den Z3-Computer seines Vaters Konrad Zuse nachgebaut hat, von dem kein vollständiges, geschweige denn funktionierendes Exemplar mehr existiert – wohl aber die technischen Beschreibungen. [vgl. Zuse 2011]

174 Zwei Beispiele hierfür: 1. die ebenfalls im Berliner *Technikmuseum* erstellte Machina Arithmeticae Dyadicae von G. W. Leibniz, die dieser 1679 auf dem Papier entworfen aber nie gebaut hatte. Die Konstruktion fand 1968/1972 in München statt [vgl Leibniz 1679; Stein 2016]. 2. die von Charles Babbage 1849 designte aber nie gebaute Difference Engine No. 2, die im Science Museum in London zwischen 1985 und 1991 nach den Originalentwürfen gebaut wurde und seitdem dort ausgestellt ist [vgl. Swade 1999:141f.].

4.4.7 Zusammenfassung

Am Beispiel des hier vorgestellten Reparaturprojektes ließ sich zeigen, auf welche Weise ein defekter Sol-20 mittels nicht-professioneller Praktiken und Tools repariert wurde und dabei seinen Status von der bloß historischen Hardware zum operativen Computer geändert wurde. Neben diese epistemologische Deutung des Reparaturprozesses haben sich im Reparaturbericht und dem Interview (Anhang B) bereits Einblicke in den Selbstlernprozess sowie spezifische didaktische Elemente (Gamification, E-Learning) gezeigt, die typisch für die Retrocomputing-Szenen sind. Die Vorgehensweisen zeigen einen deutlichen Unterschied zwischen einer professionellen, museologisch-konservatorischen *Erhaltung des materiellen Status eines Artefakts* (bei dem unter Umständen auch Dysfunktionalität in Kauf genommen wird, um dessen historische Integrität nicht zu [zer]stören) und der *nicht-professionellen Erhaltung oder Wiedergewinnung der technischen Funktionalität eines Artefakts* (selbst, wenn es dabei zu Anachronismen durch die Konfrontation historischer mit aktuellen Materialien kommt).

Aus diesen Erkenntnissen und den der zuvor geschilderten Retrocomputing-Projekte soll im folgenden Kapitel nun eine spezifische Didaktik als *Wille zum* und *Arbeit am Wissen* abgeleitet werden. Dabei wird eine Einordnung der Vorgehensweisen im Sinne informatischer Selbstausbildung/Autodidaktik vorgenommen und der Versuch unternommen, Retrocomputing als Praxis und Computerarchäologie als Theorie methodisch zu bündeln.

5. Retrocomputing als Wissenspraxis

Im folgenden Kapitel wird als Konsequenz aus den beschriebenen Projekten die Theorie der Computerarchäologie an die Wissenspraktiken des *Retrocomputings* gekoppelt. Hierzu werden zunächst noch einmal Fragen der Archäologie des Wissens bezüglich der *Epistemologie der Mikrocomputer* aufgeworfen und in eine Theorie des Selbstlernens überführt, die auf Ansätze der *Autodidaktik, Selbstlernkompetenz* und des *self regulated learning* zurückgreift. Im Begriff des *Hacking* werden sodann diese Selbstlernpraktiken und ihre epistemologischen Motivationen zunächst historisch an konkreten Beispielen zum frühen Mikrocomputer aufgezeigt, um schließlich zu einem Begriff von *Homecomputing* und dann *Retrocomputing* zu führen, der die Autodidaktiken der Vergangenheit im archäologischen Sinne aktualisiert und auf Praktiken des Selbstlernens der Gegenwart überträgt. An kontemporären Beispielen werden diese schließlich exemplifiziert.

5.1 Die Archäologie des Wissen(wollen)s

Wie in Kapitel 3 beschrieben, stellt Computerarchäologie nicht nur eine Form der Geschichtskritik dar, sondern schlägt auch einen Methoden-Kanon vor, der konkrete Projekte als ‚Arbeit an der Historie' ermöglicht. Diese Arbeit ist im Spannungsfeld von *Wissen* und *Macht* angesiedelt, die bei Foucault eng miteinander verbunden sind:

> Das Wort Wissen wird [..] gebraucht, um alle Erkenntnisverfahren und -wirkungen zu bezeichnen, die in einem bestimmten Moment und in einem bestimmten Gebiet akzeptabel sind. Und zweitens wird der Begriff Macht gebraucht, der viele einzelne, definierbare und definierte Mechanismen abdeckt, die in der Lage scheinen, Verhalten oder Diskurse zu induzieren. [Foucault 1992:32]

Diese Macht offenbart sich auf vielfältige Weise – ebenso wie die Praktiken ihr zu widersprechen. Wo die Foucault'sche Archäologie bereits einen *diskursiven Widerspruch* zum durch Macht organisierten *historischen Wissen* darstellt, stellt die Medienarchäologie mit ihrer techno-mathematischen Methodologie eine *operativen Widerspruch* zur durch ‚der Macht der Medien' etablierten und tradierten Wissensformen dar. Medienarchäologie geht damit immer auch mit einer *Medienepistemologie des Non-Diskursiven* einher, die die Bedingungen der *Möglichkeit und Unmöglichkeit* medial gespeicherten, übertragenen und prozessierten Wissens darstellt. An einzelnen Medien und ihrer Geschichte lässt sich zeigen, wie Mediennutzer immer schon versucht haben, sich einer so verstandenen Medien-Macht zu entziehen, indem sie Medientechnologien umzunutzen und sich ihre technische Funktionalität bewusst und zu eigen gemacht haben. (Zum Radio vgl. Kittler 1986:149f.; zur Fotografie vgl. Renner 2009; zum Telefon vgl. Wölbert 2014 usw.)

Computer bilden hiervon nicht nur keine Ausnahme, sondern werden von Pias [2002] sogar als diejenigen Medien, die die Aufforderung zu ihrem „Missbrauch" [vgl. ebd.:259f.

sowie Pias 2015:33] bereits in sich tragen, definiert. Ihre mehrfache Opazität bildet hierfür den Auslöser: Als komplexe operative Gefüge von Hardware und Software verbergen sie *technisches Wissen* in jeder ihrer Abstraktionsschichten [vgl. Tanenbaum 2006:22-24], welche die operative Geschlossenheit auf ihrer (Nutzer-)Oberfläche garantiert; als ökonomische Objekte verbergen sie *Betriebsgeheimnisse*, die ihren Marktwert garantieren und – vor dem Hintergrund der Fragestellung dieser Arbeit – als historische Gegenstände verbergen sie *Historeme, Diskurse und sind sichtbare Repräsentation einer unsichtbaren Macht der Archive* [vgl. Ernst 2013:87-138]. Die Vielfalt an Wissensformen, die in Computern kondensiert, hat daher immer schon das *Wissenwollen* ihrer Nutzer herausgefordert – insbesondere ab dem Moment, wo Computer in Privathände gelangt sind.

Ein Beispiel: Friedrich Kittler unternimmt in seinem Aufsatz *Protected Mode* [1993a] den Versuch Techniken der ISA-Ebene, die sich in CPU-internen Speicherschutz-Verfahren bei Intel-CPUs (ab Intel 80286) zeigen, als Beispiel für ein zu durchdringendes Machtwissen zu analysieren. Ihm zufolge war es zur Entstehungszeit seines Aufsatzes sogar der Informatik nicht möglich derlei Betriebsgeheimnisse zu lüften:

> Die Informatik [...] scheint mit internen Informationssperren konfrontiert. Im Raum der Codes, auf die sie faktisch zurückgreifen muß, auch wenn die Theorie ganz andere Modelle erzeugen könnte (und sollte), sind Entzifferungen wider Willen und Wissen der Codeentwickler ebenso möglich wie rar. [221]

Um den Real Mode der CPU zu aktivieren, muss der Nutzer auf der Ebene des Betriebssystem-Layers mittels Assembler-Programmierung in die Prozesse eingreifen – eine Programmiersprache, die zur Zeit des Kittler'schen Beitrags bereits „aufgrund ihrer Maschinenabhängigkeit, ihres niedrigen Abstraktionsniveaus und der Schwierigkeiten beim Schreiben und Verstehen [... nicht das ist], was man normalerweise unter einer Programmiersprache versteht." [Louden 1994:2], weshalb sie aus dem Kanon der Praktischen Informatik ausgeschlossen [vgl. ebd.] und in der Domäne der Technischen Informatik angesiedelt wurde. Als Programmiersprache genutzt wurden Assemblersprachen allerdings zu dieser Zeit dennoch – und zwar außerhalb universitärer und schulischer Informatik-Ausbildung: beim Programmieren privater Homecomputer, das aus dem *Willen zum Wissen* praktiziert wurde und wird.

Diesen *Willen zum Wissen*, der in der Neuzeit als *curiositas* (Neugier) den wissenschaftlichen Blick gelenkt hat [vgl. Ernst 2005/6], beschreibt Foucault als maßgebliche Triebfeder aufklärerischer Wissensproduktion und -distribution, die zur Zeit des Barock entsteht:

> [... A]n der Wende vom 16. zum 17. Jahrhundert ist (vor allem in England) ein Wille zum Wissen aufgetreten, der im Vorgriff auf seine wirklichen Inhalte Ebenen von möglichen beobachtbaren, meßbaren, klassifizierbaren Gegenständen entwarf; ein Wille zum Wissen, der dem erkennenden Subjekt (gewissermaßen vor aller Erfah-

> rung) eine bestimmte Position, einen bestimmten Blick und eine bestimmte Funktion (zu sehen anstatt zu lesen, zu verifizieren anstatt zu kommentieren) zuwies; ein Wille zum Wissen, der (in einem allgemeineren Sinn als irgendein technisches Instrument) das technische Niveau vorschrieb, auf dem allein die Erkenntnisse verifizierbar und nützlich sein konnten. [Foucault 1999:58]

Das Wissenwollen sieht Foucault in einer dialogischen Beziehung zur Macht, die dieses Wissen verwaltet und selbst von diesem verändert wird. Foucaults *Epistemologie* versucht „das Interface zwischen Wissen und Macht, zwischen Wahrheit und Macht sichtbar zu machen." [Foucault 2003a:521]:

> In Wirklichkeit handelt es sich bei den Machtbeziehungen um Kräfteverhältnisse und Konfrontationen; sie sind also stets umkehrbar. [...] Machtmechanismen lösen ständig Widerstand aus, sie provozieren und ermöglichen Widerstand. Gerade weil Widerstand möglich ist und auch wirklich geübt wird, versucht der jeweils Mächtige seine Macht umso heftiger und listiger zu verteidigen, je stärker der Widerstand ausfällt. [Foucault 2003a:524f.]

Der Begriff Macht ist bei Foucault als „Netz" zwischen *sozialen Entitäten* definiert (als Diskurse, Institutionen, administrative Maßnahmen, wissenschaftliche Aussagen usw. [vgl. Foucault 2003b:392]) Andere Dispositive der Macht, wie sie etwa in Medientechnologien (wie der Integration eines Protected Mode in Mikroprozessoren) wirken, finden in seiner Theorie keine Berücksichtigung. Wolfgang Ernst sieht diese Auslassung als Auftrag:

> Foucaults Machtanalyse wäre als Medienanatomie fortzuschreiben. Fragen wir ganz analog: Was ist das Nicht-Wahrnehmbare, die *An-aisthesis* am Medium? [...] Hier Aufklärung zu schaffen, ist die Aufgabe einer Medienarchäologie, die Schaltpläne aufdeckt, d. h. zur Entzifferung gibt. Hinter der medialen Oberfläche stehen keine Geheimnisse, sondern schlichte Algorithmen und Maschinenbauteile – man muß sie nur zu lesen wissen. [Ernst 2004:241]

Eine solche Lesekompetenz gilt es für denjenigen zu erwerben, der tiefgestaffeltes Wissen über Medien und ihre Macht erlangen möchte. Daher schlägt Friedrich Kittler im oben zitierten Text vor, mithilfe von Assemblerprogrammierung die Wissens-Barriere, die der Protected Mode darstellt, zu überwinden:

> Hundert Zeilen Assembler, aber auch nur Assembler, lösen also das Problem einer postmodernen Metaphysik: Auf die Gefahr hin, wahnsinnig zu werden, führen sie unter MSDOS jenseits von MSDOS. Mit der berühmt-berüchtigten Schallmauer, daß der Arbeitsspeicher unter DOS auf ein lächerliches MegaByte beschränkt bleibt, zergehen auch alle WINDOWS nachgerühmten Vorzüge zu nichts. In drastischer Paradoxie ermöglicht gerade das rückständigste aller Betriebssysteme den Ausstieg aus ihm. [Kittler 1993a:220]

Auch wenn die Motivation dafür Assembler zu programmieren nicht immer politisch wie in Kittlers Ansatz sein muss, spricht er doch aus, was ab Ende der 1970er-Jahre das Credo privater Programmierpraxis gewesen ist: Die ‚Geheimnisse' des Computers (seine Hardware und Programmierbarkeit) können nur autonom, unabhängig von extrinsischen Erkenntnisinteressen und -zielen und mittels selbst gewählter legaler und illegaler (z. B. Nutzung illegaler Opcodes[175] zur Assembler-Programmierung) Werkzeuge gelüftet werden. Denn neben urheberrechtlichen Hürden der Hersteller hatten sich die Informatik-Didaktiken von Schule und Universität aufgrund fortschreitender und sich stetig beschleunigender Hardware-Entwicklung [vgl. Knauer 1980:11] sowie zugunsten elaborierterer Fragestellungen [vgl. Forneck 1990] sukzessive von der konkreten Hardware des einzelnen Computers entfernt [vgl. Höltgen 2016b]. Die Didaktik, nach der der Wille zum Wissen über Computer im allgemeinen und das private Erlernen von Programmierung im besonderen vollzogen wurde und wird, soll im Folgenden zunächst theoretisch umrissen werden, bevor sie methodisch an historischen und kontemporären Beispielen exemplifiziert wird.

5.2 Computer-Autodidaktik

Neben der schulischen und universitären Informatik-Lehre existiert bereits seit Ende der 1950er-Jahre (vgl. 3.3) eine private (hobbymäßige) Informatik-Selbsausbildung, die sich in dreierlei Hinsicht von ersteren Formen unterscheidet: in ihrer Perspektive auf den *Gegenstand,* ihren *Methoden* und ihrer *Didaktik*. Hobbyistische Informatik rückt Computer als konkrete Apparate (Hardware) in den Fokus ihres Interesses und fragt nach ihrem Aufbau, ihrer Erweiterbarkeit und den Möglichkeiten ihrer Reparatur – erfordert also Wissen über Architekturen, Peripherie und Elektronik. Überdies beschäftigt sie sich mit der Programmierung von Computerhardware in unterschiedlichen, vor allem maschinennahen Sprachen und nach unterschiedlichen Programmierparadigmen. Die Theorie von Computern, ihre Geschichte sowie die Geschichte der Softwaregattungen, Firmen, Programmierer und Ingenieure sind ebenfalls ihre Themen, wenngleich ein Lernen und Wissen dieser Sachverhalte keine spezifisch informatischen Methoden jenseits von Quellenbeschaffung und Hermeneutik verlangt. Zur Ausbildung solcher Kenntnisse ist konkret nötig: ein Computer als Forschungsgegenstand (und Lernwerkzeug), Informationen über den Computer und seine Programmierung sowie eine Methode der Selbstausbildung. Letztere soll im Folgenden definiert und methodisch eingegrenzt werden.

5.2.1 Autodidaktik

Der Begriff Autodidakt wird häufig in Zusammenhang mit (historischen) Persönlichkeiten (Jean-Jacques Rousseau, Abraham Lincoln, Jakob und Wilhelm Grimm, ...) gebraucht,

175 Hierbei handelt es sich um Funktionen, die nicht intendiert in den Befehlssatz von Mikroprozessoren implementiert wurden aber dennoch abgerufen werden können [vgl. Höltgen 2013a:93].

die durch Selbstausbildung besondere Kenntnisse und Fähigkeiten erworben und darauf basierend „beachtliche bis herausragende Leistungen“ [Wikipedia: Autodidakt] auf kulturellem oder wissenschaftlichem Gebiet erbracht haben. Die Methodik dieser Selbstausbildung, verstanden als *Autodidaktik*, bleibt dabei zumeist undefiniert (oder tautologisch: „people who prefer to teach themselves“ [Solomon 2013:3]). Michael [2008] unternimmt den Versuch einer Begriffsbestimmung:

> Unlike the typical supervised learning frameworks, however, we examine the case where learning proceeds completely autonomously, without the presence of a teacher that designates when rules are sound. The agent attempts to learn to the extent that its environment provides the agent with sufficient information to do so. Induced rules are also applied in a completely autonomous manner, without the presence of a teacher that designates the order in which rules should be applied. We term this approach autodidactic, or self-taught, emphasizing both the supervised learning component and the autonomy component. [ebd.:7]

Die Abgrenzung von „completely autonomous“ zum „supervized learning“ ist dabei ausschlaggebend. Autodidaktische Lernprozesse finden zumeist *solitär* und auf Basis *selbst gewählter Methoden* („rules“), *Materialien* („information“) und *Lernumgebungen* („environment“) statt. Allerdings ist nicht jedes Lernen ohne Lehrer bereits Autodidaktik, wie Solomon [2003:13f.] konstatiert: Hinzu müssen beim Autodidakten drei wesentliche Prinzipien kommen:

1. Der Lernprogress verfährt (möglichst) kleinschrittig.
2. Die Untersuchung des Lerngegenstandes erfolgt autonom (ohne Prüfungs- oder Berichterstattungszwänge).
3. Die (Lern)Handlung wird aufgrund eines eigenen Impulses initiiert.

Der Autodidakt zeichnet sich darüber hinaus (4.) durch seine *Mitteilsamkeit* aus: „Any autodidacts worth their salt would stick to their own notions as long as possible. [...] Explaining your own ideas to others might be satisfying“ [Solomon 2003:12]. Wie Turkle [1984] beobachtet hat, etablieren sich in autonomen Programmier-Lerngruppen bei Kindern häufig derartige alternative Lehrer-Schüler-Verhältnisse, bei denen der Lehrer dem Schüler beratend zur Seite steht und nicht selten sogar von ihm lernt. Knauer [1980:18] „sind Fälle bekannt, wo Schüler sich selbst soweit ausgebildet haben, daß sie ‚alten Hasen‘ einiges vorraus [sic] hatten.“

In der interaktiven Auseinandersetzung mit Computern beobachtet Turkle allerdings noch einen weiteren Aspekt, den sie als „Evokation“ [vgl. Turkle 1984:10f.,130] bezeichnet und der die interpersonelle Lehr-Lern-Situation zu einer affektiven Beziehung zwischen Mensch und Technik werden lässt: Der User beginnt seinem Computer Persönlichkeitsmerkmale zuzuschreiben (diese in ihn zu evozieren). Dies ermöglicht es ihm, in

einen scheinbaren, nicht selten auch emotionalen [vgl. Turkle 1984:11] *Dialog mit dem Computer* zu treten.

Dieses Changieren zwischen Lehr- und Lernposition beginnt bereits bei der solitären Programmierlehre des Lernenden vor dem Computer: während der Mensch programmieren lernt, indem er „dem Computer etwas beibringt" [Turkle 1984:118]. Der Computer gibt dem Programmierer dabei ein *Feedback* durch sein erwartetes (Funktionieren) oder unvorhergesehenes Verhalten (Fehler), das vom Lerner als Erfolg oder Misserfolg gewertet und dementsprechend positiv oder negativ affektiv besetzt wird [vgl. Butler/Winne 1995:250]. Das ‚Verhalten des Computers', zumal wenn dieser evokatorisch aufgeladen wird, ersetzt damit das Feedback des Lehrenden.

5.2.2 Selbstlernkompetenz

Die Pädagogik untersucht autodidaktische Lernverfahren unter den Begriffen: *selbstständiges Lernen* [vgl. Moegling 2004], *selbstreguliertes Lernen* [Kaiser 2003:14 FN] bzw. *Self-Regulated Learning (SRL)* [Butler/Winne 1995] *oder Selbstlernkompetenz* [Kaiser 2003]. Darunter wird ein Set von Fähigkeiten gebündelt, die es dem Lernenden ermöglichen,

> ihn interessierende Fragestellungen oder von ihm zu lösende Probleme selbstständig zu konturieren, den zu ihrer Bearbeitung erforderlichen Informationsraum zu definieren und aus der Fülle von Informationen diejenigen auszuwählen, die zur Beantwortung der Frage beziehungsweise Lösung des Problems beitragen könnten. Er muss Antwort- oder Lösungsmöglichkeiten selbstständig entwerfen und auf ihre Tragfähigkeit überprüfen sowie eigenständig über deren Annahme oder Verwurf entscheiden können. Dabei anfallende Lern- und Arbeitsprozesse hat er ebenfalls selbstorganisiert und -regulierend durchzuführen. [Kaiser 2003:13]

Notwendig hinzu kommt in diesen Konzepten der „gezielte Einsatz von Metakognition" [ebd.:14], die den Lernenden befähigen soll, expliziten „Zugang zu denjenigen [impliziten, S.H.] Kompetenzen zu finden, die konstitutiv für die Fähigkeit zur Selbstlernaktivität sind." [ebd.:17] Dabei reflektiert er über seinen eigenen Lernprozess, um Geleistetes auf Stärken und Schwächen hin zu bewerten und sich auf anstehende Probleme vorzubereiten. Die Ausbildung der Metakognition ist jedoch an interpersonelle Lehr-/Lernprozesse gekoppelt, weil der Lernende zunächst eine methodisch fundierte Reflexion über seine Selbstlernpraktiken von Außen benötigt. Dem Autodidakten steht diese Instanz aber nicht zur Verfügung. Sein Lernen vollzieht sich in der Tat „[l]aisser-faire" [Moegling 2004], unabhängig vom direkten Einfluss eines Lehrers; meta-kognitive Reflexion findet, wenn überhaupt, unsystematisch statt. Nicht nur damit steht die Autodidaktik im Widerspruch zu „methodisierenden Didaktiken" [Moegling 2004:41] des Schulunterrichts. Von der vollständig freien Zeiteinteilung über die Frage der Lernlust und -unlust bis hin zur

freien Wahl der Lerninhalte zeigt sich Autodidaktik als unvereinbar mit organisiertem Schulunterricht.

Autodidaktik markiert damit auch ein erziehungswissenschaftliches und -politisches Moment der Ausbildung und setzt ein spezifisches Menschenbild voraus, bei dem der Wille zum Wissen, und die Selbstermächtigung des Lernprozesses im Zentrum stehen. Klaus Moegling [2004] sieht darin eine Bedingung für die Entwicklung von Selbstständigkeit überhaupt. Das Menschenbild, das dem zugrunde liegt, orientiert sich an der *konstruktivistischen Lerntheorie*:

> Lernen ist nur über die aktive Beteiligung des Lernenden möglich. Dazu gehört, dass der Lernende zum Lernen motiviert ist und dass er an dem, was er tut und wie er es tut, Interesse hat oder entwickelt. Bei jedem Lernen übernimmt der Lernende Steuerungs- und Kontrollprozesse. Wenn auch das Ausmaß eigener Steuerung und Kontrolle je nach Lernsituation variiert, so ist doch kein Lernen ohne jegliche Selbststeuerung denkbar. Lernen ist in jedem Fall konstruktiv: Ohne den individuellen Erfahrungs- und Wissenshintergrund und eigene Interpretationen finden im Prinzip keine kognitiven Prozesse statt. Lernen erfolgt stets in spezifischen Kontexten, sodass jeder Lernprozess auch als situativ gelten kann. Lernen ist schließlich immer auch ein sozialer Prozess: Zum einen sind der Lernende und all seine Aktivitäten stets soziokulturellen Einflüssen ausgesetzt, zum anderen ist jedes Lernen ein interaktives Geschehen. [Hubwieser 2007:10]

Konstruktivistische Ansätze für die Informatiklehre zeigen sich bereits bei Knauer [1980:9f.,12], wenn er kritisiert, dass Informatikunterricht in der Oberstufe erst einsetzt, wenn die Schüler „fast völlig ‚verdorben[..]'" [Knauer 1980:14] sind, weil ihnen bis dahin bereits jegliche Lernautonomie abhanden gekommen sei. Wie gestaltet sich hingegen nicht-schulische, autodidaktische, hobbyistische Informatikausbildung, die der konstruktivistischen Lerntheorie folgt, konkret?

5.2.3 Methoden des Selbstlernens

Informatikunterricht müsse Knauer zufolge als „Leitmotiv" den „Spieltrieb" [Knauer 1980:40] ansprechen. Dies würde am ehesten erreicht, „wenn der Schüler vor dem Rechner sitzt, aber noch nichts damit anfangen kann." [ebd.] In dieser Situation ist er zum Experimentieren gezwungen, setzt *Trial-and-Error-Verfahren* ein, berät sich mit anderen Schülern, sucht eigenständig in unterschiedlichen Quellen nach Vorgehensweisen und Problemlösungen, überträgt außerschulisches Wissen auf die Situation, nutzt den Computer dabei weniger als *Werkzeug* zur Wissenserlangung als als *Zeug* [Heidegger 2006:66ff.] des Wissens – macht ihn also zum epistemischen Objekt. Seine Selbstlernmethode basiert dabei stärker auf „Learning by doing" [Anzai/Simon 1979] als auf der Aneignung theoretischen, ‚passiven' Wissens. Als interaktiv definiert Cermak-Sassenrath Medien, „die

einen entscheidenden, inhaltlichen Eingriff des Teilnehmers in ihren Verlauf und ihr Ergebnis erlauben und sogar voraussetzen“ [2010:26]. Didaktisch wird Interaktivität in der hobbyistischen Informatiklehre dabei auf eine Weise genutzt, die den Methoden von *Trial and Error*, *Gamification* und *E-Learning* ähnelt.

5.2.3.1 Trial and Error

Insbesondere dann, wenn Informationen zum Lerngegenstand fehlen (oder nicht konsultiert werden), der Lerngegenstand selbst jedoch bereits vorliegt und damit zum Experimentieren einlädt, bieten sich heuristischen Vorgehensweisen des Wissenserwerbs an, zu denen auch das Lernen durch *Trial and Error* (Versuch und Irrtum) gehört, das der Kybernetiker W. Ross Ashby passenderweise als Hunt and Stick redefiniert:

> “Trial” is in the singular, whereas the essence of the method is that the attempts go on and on. “Error” is also ill-chosen, for the important element is the success at the end. “Hunt and stick” seems to describe the process both more vividly and more accurately. [Ashby 1957:230]

In der hobbyistisch betriebenen Informatik mit dem Computer als Lerngegenstand ist diese Vorgehensweise auf das Feld der Technischen und Praktischen Informatik beschränkt, das hier – ohne zusätzliche Kenntnisse zum Beispiel der Algorithmen-Theorie, Rechnerarchitektur, Programmiersprachen-Typologie usw. – in eingeschränktem Maß autodidaktisch betreten werden kann.

Derlei freies, unsystematisches Experimentieren mit Hardware und Software wird auch als *Hacking* bezeichnet und ist in Hinblick auf die oben diskutierte Epistemologie von besonderem Interesse. Denn insbesondere dort, wo auf den Versuch nicht der Erfolg, sondern der Irrtum (error) folgt, „manifestiert sich bekanntlich das Medium, das ansonsten hinter seinen Inhalten verborgen bleibt.“ [Ernst 2012b:323]. Hardwarefehler bringen die elektronische Verfasstheit des Computers ebenso zu Bewusstsein und in Erinnerung, wie Programmierfehler die Diskrepanz zwischen der geplanten/gedachten Funktion und ihrer formalsprachlichen Kodierung. Die Fehlersuche (das Debugging) vollzieht sich unter dieser Lernmethode dann ebenfalls auf „Versuch und Fehlerbeseitigung [...] Der Programmierer lernt aus den Fehlern, indem er sich Regeln und Vorschriften gibt, die der Vermeidung dieser Fehler dienen. In diesem Sinne ist das Programmieren eine Erfahrungswissenschaft, so wie die Ingenieurwissenschaften Erfahrungswissenschaften sind.“ [Grams 1990:5]

Mit der Anwendung des Trial-and-Error-Verfahrens vollzieht der Lerner allerdings bereits eine probate wissenschaftliche Methode: Ob in der Mathematik die heuristische Suche nach Nullstellen mittels Polynom-Darstellung von ganzrationalen Funktionen durchgeführt wird oder in der Informatik das automatische Beweisen mittels *generate and test*: Überall dort, wo (dem Anwender oder grundsätzlich) für Problemlösungen keine

effizienten Algorithmen bekannt sind, werden mit dem Brute-Force-Verfahren alle möglichen Lösungen durchprobiert.

Nicht nur aus eigenen, auch aus fremden Fehlern kann der Autodidakt lernen, wenn er durch Dysfunktionalitäten auf sie aufmerksam gemacht wird und der Computer damit in sein Bewusstsein rückt. Das Debugging des Spiels E.T- The Extraterrestrial (siehe 3.3.4) hat beispielhaft vorgeführt, welche Erkenntnisse nicht nur informatischer, sondern auch historischer Provenienz sich aus der Konzentration auf (fremde) Programmierfehler ableiten lassen. Am Beispiel des plötzlichen Gewahrwerdens von Fehlern in Computerspielen wurde dargestellt, wie Spieler auf diese Weise eine alternative Art des Spielens entdecken [vgl. Höltgen 2014c:295] und zu einem Spiel mit dem Code [vgl. ebd. 306f.] übergehen können.

5.2.3.2 Gamification

Ein solcher spielerisch-experimenteller („playful“ [Takhteyev/Dupont 2013:427]) Zugang zum Computer unterstreicht nicht nur einmal mehr die Lernautonomie, sondern ermöglicht es zudem, autodidaktische Programmierlehre an den Begriff des *Computerspiels* und damit der *Gamification* zu koppeln. Hierbei werden spieltypische Elemente und Prozesse auf Lernsituationen übertragen, um die Motivation des Lernenden zu steigern. [Vgl. Deterding et al. 2011:9] Die vielfach genutzte Definition „the use of game design elements in non-game contexts“ [Groh 2012:39 und Deterding et al. 2011:9] ist stark umstritten [vgl. ebd.:9; Bogost 2014], zumal die vorausgesetzten Begriffe des „game“ und des „non-game context“ nur bei extremer Zuspitzung auf dezidierte (Computer)Spiele eine scharfe Eingrenzung zulassen. Ansätze zu einer *Gamification der Computernutzung* existieren bereits seit der Homecomputer-Zeit (beispielsweise in [reh 1985] und überblicksartig bei [Deterding et al. 2011:10]).

Während die Didaktik der Informatik und die HCI-Forschung den bislang noch unscharf [vgl. Deterding et al. 2010:10] definierten Begriff Gamification vor allem an der Integration (computer)spielerischer Elemente in Lernumgebungen und den Lerneffekten von Computerspielen (Serious Games, Pervasive Games) untersuchen, widmet sich Deterding et al. zufolge die Medienwissenschaft den „playful media practices“ [2011:10], die im ausgehenden 20. Jahrhundert durch den breiten kulturellen Erfolg von Computerspielen sukzessive in verschiedene Sphären der Kultur diffundierten. Hier wird vorgeschlagen, den Begriff *Gamification* aus seinem bisherigen, vor allem durch das Design und die Deskription desselben geprägten Kontext zu lösen und ihn anstelle dessen als strukturanalytisches Moment verschiedener Technologien (Lerntechnologien, Simulationen, Planspielen, Operations Research usw. [vgl. Nohr 2019:309,312,324]) zu werten, die *spielerische Konzepte integrieren oder erkennen lassen*. Hier soll der Terminus Gamification aus diesen Perspektiven auf Selbstlernprozesse übertragen werden: In der „gameful interaction“ [Deterding et al. 2011:10] zwischen Lernendem und Computer zeigen sich Elemente von

Gamification, die nicht explizit (methodisch geplant) integriert worden sein müssen. Die daraus resultierende weitere Verunschärfung des Begriffs Gamification durch (Re-)Integration von „play“ (paidia) im Kontrast zu „game“ (ludus) [vgl. Groh 2012:39f.]) kann zugunsten der Möglichkeit in Kauf genommen werden, Computer als „epistemisches Spielzeug“ [Höltgen 2015a:54ff. sowie allgemeiner: von Samsonow 2012], untersuchen zu können, welches im Heidegger‘schen Sinn dabei gleichzeitig *Zeug* und *Spielzeug*, also Gegenstand (Thema) des Spiels und Spielgegenstand (Spielobjekt) ist. (Vgl. 4.4)

Neben dem bereits am Beispiel von E.T. - The Extraterrestrial vorgestellten interaktiven, zeitkritischen Eingriff in die laufenden Computerprozesse zum Zweck der Codemanipulation zeigen sich noch weitere Analogien zwischen Programmieren und Spielen. Michael Koch [2012] stellt eine Liste mit „spieletypischen Mechanismen auf, die als motivationsfördernd angesehen werden“ [ebd.], welche jeweils auf die autodidaktische Programmierlehre übertragen werden:

- *Sichtbarer Status:* die sichtbare Präsentation des bereits Erreichten, um sich mit anderen vergleichen zu können. – Dies zeigt sich in Praktiken des Code-Sharing, das in frühen hobbyistischen Programmierkulturen [vgl. Levy 1984:23,27f. und Turkle 1984:121] ebenso betrieben wird, wie heute.

- *Einsehbare Rangliste:* die aus diesem Vergleich resultierende Hierarchie. – Aufgrund der solitären Situation des Autodidakten zeigt sich auch dieser Aspekt nicht in allen Selbstlernprojekten. Dort allerdings, wo in Gruppen selbstreguliertes Lernen stattfindet, ergeben sich nach Turkle [vgl. ebd.:162f.] häufig Hierarchien, die zum Zweck der Kompetition transparent gemacht werden.

- *Quests:* Teilaufgaben, die gelöst werden müssen. – Hierunter fällt beispielsweise der Entwurf einzelner Algorithmen, Prozeduren oder Programm-Subroutinen.

- *Resultattransparenz:* die vorherige Kenntnis des Ergebnisses eines Spiels, das als Ziel angestrebt wird. – Dies wird durch das fertige Programm, das bestimmte Bedingungen erfüllen soll (Funktionalität, strukturelle Eleganz, Codelänge, Ausführungsgeschwindigkeit usw.), realisiert und zeigt sich etwa im „program bumming“ [Levy 1984:31 und Pias 2002b:257], also der kompetitiven Kürzung von Source Code unter Wahrung von dessen Funktionalität.

- *Rückmeldung:* das Feedback, welches das Spiel dem Spieler über den Erfolg oder Misserfolg seiner Spielhandlung gibt. – *Feedback* beim Programmieren erfolgt durch den Computer (Compiler oder Interpreter), der den Programmierer über Erfolg und Fehler informiert und Teil des dialogischen „Trial and Error“-Prozesses ist.

- *Epic Meaning:* das Bewusstsein des Spielers, an einem größeren Ziel zu arbeiten. – Hierin wäre das übergeordnete Erkenntnisinteresse an Computerprozessen zu se-

hen, dem die Etappenarbeit der Entwicklung eines spezifischen Programms zuarbeitet.

- *Fortschrittsanzeigen:* die Visualisierung des erreichten Fortschritts – Diese zeigt sich dem Programmierer quantitativ in der Länge seines bereits entwickelten Codes; qualitativ erfährt er über seine Fortschritte durch die sich sukzessive verringernden Fehlerausgaben des Compilers oder Interpreters bzw. die semantische Korrektheit seines Programms.

- *Community Collaboration:* Elemente, die verschiedene Spieler an einem gemeinsamen, größeren Projekt teilhaben lassen. – Insbesondere Plattformen für kollaborative Projektverwaltung wie *GitHub* bieten Programmierern diese Möglichkeit. [Vgl. Maibaum 2015:34]

- *Cascading Information:* Spieler erhalten nur die für den gegenwärtigen Spielstatus bzw. die aktuelle Aufgabe notwendigen Informationen, um eine Überforderung auszuschließen. – Dieser Aspekt zeigt sich analog in der ungeplanten individuellen Programmentwicklung (Hacking), in der jedes Problem separat gelöst und die Algorithmen modular entwickelt werden. Zudem beenden bereits frühe BASIC-Interpreter „die Ausgabe von Fehlern bei Studenten nach fünf Fehlern“ [Kurtz 2009:84], um das Debugging für die Programmieranfänger überschaubar zu halten und diese nicht zu demotivieren.

Der Gamification-Ansatz wird – auch jenseits meiner hier vorgenommenen Umdeutung – in der praktischen Programmierausbildung betrieben (eine Übersicht bietet Firebear [2015]). Entscheidend ist, dass das Programmierenlernen durch seinen experimentellen Charakter quasi als Spiel betrieben (wenn auch nicht unbedingt als solches vom Lernenden gesehen) wird, um sich von anderen Praktiken zu unterscheiden, die ebenfalls als Spiel deutbar wären (etwa das Lesen eines Buches, bei dem der Leseprozess ebenfalls Momente des sichtbaren Status, Quests, Resultattransparenz, epic meaning usw. gesehen werden könnte.) Die zeitkritische Interaktivität zwischen Programmierer und Computer ist es zusätzlich (vgl. Kap. 3.3.4), die die Zuschreibung des Lernprozesses als Gamification probat erscheinen lässt.

In Lernumgebungen, die der Lernende zumeist solitär nutzt, finden sich zahlreiche der oben aufgelisteten Spiel-Elemente wieder – etwa auf Online-Lernplattformen [vgl. Rudelt 2014]. Eine weitere Eskalation der hier vorgenommenen Umdeutung spielerischer Programmierdidaktik stellen Computerspiele dar, die selbst bereits Programmiermöglichkeiten inkorporieren, wie Wario Ware D.I.Y. [Vgl. Jones/Thiruvathukal 2012:216ff.], Minecraft [Mojang 2014] oder solche, bei denen mittels Spielhandlungen Änderungen am Code (Hacks) durchgeführt werden können, wie bei Super Mario World [vgl. Osgood 2015].[176] Eine

176 Siehe auch: *Super Mario World „Arbitrary Code Execution“*: https://www.youtube.com/watch?v=OPcV9uIY5i4 [letzter Abruf: 12.04.2016].

Eskalationsstufe dieser Beziehung stellen Programmierlehrbücher dar, bei denen man durch das Hacking von Computerspielen eine Programmiersprache systematisch erlernt [vgl. Richardson 2016].

5.2.3.3 E-Learning

Dort, wo elektronische Medien, insbesondere Computer, *explizit* für Lernzwecke eingesetzt werden, spricht man von E-Learning. E-Learning-Verfahren sind in einzelnen Fällen bereits sehr früh entstanden [vgl. Simons 2013a; Simons 2013b] und wurden spätestens mit der Verfügbarkeit von Mikrocomputern ab Mitte der 1970er-Jahre unter Begriffen wie *Intelligent Computer Assisted Instruction* [Collins/Crignetti 1975] und *Computer Based Training* [Dean/Whitlock 1989] systematisch didaktisch entwickelt. Unter E-Learning wird „das Lehren und Lernen mittels verschiedener elektronischer Medien verstanden" [Rey 2009:15], bei denen der Lernende in zumeist „selbst regulierten Lernprozessen" [Pachner 2009:51] unter Zuhilfenahme (medien-)technischer Mittel [vgl. ebd.] Lerninhalte erarbeitet. Diese werden in spezifischen *E-Learning-Szenarien und -Umgebungen* [vgl. ebd.:52] entwickelt und angeboten. Die *Szenarien* stellen dabei die „nach inhaltlichen und didaktischen Gesichtspunkten für bestimmte Zielgruppen" [51] ausgewählten Lehr-Lernmethoden dar; die *Umgebungen* umfassen „zusätzlich die technischen Voraussetzungen" [51] hierfür. E-Learning findet in vielfältigen Zusammenhängen, mit unterschiedlichsten Medien, Gegenständen, Lernzielen und Lerngruppen und -personen statt. [Vgl. beispielsweise Riekhof/Schüle 2002, Isaías et al. 2015 oder Rey 2009]

Unter den verschiedenen Formen des E-Learning spielt für den vorliegenden Zusammenhang die *virtuelle Lehre* [Effert 2001; Mader/Stöckl 1999] eine maßgebliche Rolle. Hierbei lernt der Lernende autonom und solitär. Lernumgebungen hierfür sind heute vor allem Internetplattformen und netzbasierte Kommunikations- und Kooperationssysteme (Chats, E-Mail, Repositories, Filehosting, Foren usw.) Daneben werden ‚traditionelle' Lehrmaterialien, wie Texte und Videos in der virtuellen Lehre eingesetzt. Die virtuelle Lehre ähnelt der Vorgehensweise bei der autodidaktischen Programmierlehre in einigen Aspekten. Um dies zu zeigen, wird im Folgenden – wie bereits beim Begriff der Gamification geschehen – auch den Begriff des E-Learning einer Erweiterung unterzogen. Denn dort, wo Computer zugleich das Objekt und das Werkzeug des (E-)Lernens sind, also „Unterricht an Computern und über Computer" [Harenberg 1984:103] zugleich stattfindet, ergeben sich epistemologisch interessante Wendungen im Prozess des Wissenserwerbs.

Dies zeigt sich deutlich an Homecomputern: Frühe Mikrocomputer stellten in Hinblick auf ihre Nutzbarkeit im Homecomputer-Zeitalter für ihre Besitzer eine ‚Provokation zum Lernen' dar. Dort, wo solche Homecomputer nicht bloß mit einem rudimentären BIOS und einem Monitorprogramm ausgestattet sind (die das Laden von Programmen und höheren Programmiersprachen vereinfachen), ist zumeist die Programmiersprache BASIC in ihrem ROM verbaut, die nach dem Einschalten sofort nutzbar ist. (Einzelne Systeme,

wie der Jupiter Ace, halten Programmiersprachen wie Forth bereit oder bieten, wie der für die schulische Informatik-Ausbildung konzipierte BBC Micro, an, mehrere ROM-Bausteine für unterschiedliche Programmiersprachen zu integrieren.) Nach dem Aufbauen und Einschalten erwarteten sie ihre Nutzer zumeist mit der Meldung „READY“ und einem Prompt („Eingabeaufforderungen“) in Form eines (bei manchen Systemen blinkenden) Cursors. Der Nutzer befand sich also sofort in einer Dialogsituation und erhielt für jede dann folgende Eingabe ein Feedback. Diese Situation konfrontierte ihn mit der Notwendigkeit zumindest rudimentäre Kenntnisse über die eingebaute Programmiersprache zu erlangen (falls er den Computer nicht sowieso für die Programmierung nutzen wollte), um andere Software in den Speicher laden zu können. Hilfestellung boten ihm dabei das Manual und die Sekundärliteratur.

Werden Homecomputer zum autodidaktischen Programmierenlernen eingesetzt, bieten ihre eingebauten BASIC-Interpreter besondere Möglichkeiten der Interaktivität und des Feedbacks: Anders als bei Compilersprachen lassen Interpreter-Sprachen die Entwicklung und Bearbeitung von Programmen zur Laufzeit zu. Daraus resultiert ein zeitnahes Feedback des Systems an seinen Nutzer. Auch einige kontemporäre BASIC-Dialekte basieren auf interaktiven Interpretern (etwa Chipmunk-BASIC[177], BBC-BASIC[178] und andere); diese finden sich aber auch bei moderneren Interpreter-Sprachen (PERL, Python, LUA, Ruby, APL, PHP und andere). In Emulator-Projekten werden die historischen BASIC-Dialekte ebenfalls als Interpreter integriert. All diesen Sprachen ist gemein, dass sie die Aufmerksamkeit des Nutzers während des gesamten Entwicklungsprozesses binden (weil keine Kompilierzeit anfällt) und er auf sein Programmieren konzentriert bleibt (weil er nicht von Meta-Prozessen wie dem Compiler abgelenkt wird). Der Lernprozess wird dabei zunächst auf den Nutzer und den Computer beschränkt; Hilfestellungen sind durch die oben genannten Medien erhältlich. (Online verfügbare Hilfen werden zumeist mit modernen Computern besorgt; selbst wenn der Homecomputer für die Anbindung ans Internet aufbereitet wurde, lässt seine technische Ausstattung es nur in Ausnahmefällen zu, mit ihn gleichzeitig das Programmieren zu lernen und Hilfsdokumente auf ihm anzuzeigen.)

Moderne Programmierlernumgebungen können in dieser Hinsicht als Nachfahren eines solchen interaktiven E-Learning-Feedback-Systems angesehen werden. Intelligent Tutoring Systems (ITS) unterstützen Programmierlerner durch flexible Feedback-Optionen, die dialogisch Hilfen bei der Behebung von Syntax-Fehlern oder dem Erreichen des Programmierziels geben [Gross/Pinkwart 2015; Gross/Mokbel/Hammer/Pinkwart 2015]. In Integrierte Development Environments (IDE) implementierte Möglichkeiten der Stack-Trace-Bearbeitung erinnern zeitweise gar an die Ursprünge der BASIC-Didaktik („One Error a time“), wenn, wie bei Processing und HelpMeOut, jeweils stets nur ein Fehler

177 http://www.nicholson.com/rhn/basic/ [letzter Abruf: 07.06.2019].

178 http://www.bbcbasic.co.uk/bbcbasic.html [letzter Abruf: 07.06.2019]

ausgegeben und dazu Korrekturhilfe angeboten wird [Coenen/Gross/Pinkwart 2018]. Die Programmier(lern)umgebungen übernehmen hierbei die Rolle früherer Interpreter-Systeme, wenngleich die didaktische Erfahrung mehrerer Jahrzehnte zu ausgefeilteren Feedback-Systemen mittels künstlich-intelligenter Tutoren [Le/Strickroth/Gross/Pinkwart 2013] geführt hat, so dass selfregulated Learning damit auch für Compiler-Sprachen und avanciertere Programmierparadigmen ermöglicht wird. Dennoch bleibt der zu programmierende Computer hierbei immer noch zugleich das ‚Werkstück' und das Werkzeug des Lernens.

Zusammenfassend kann gesagt werden, dass autodidaktische Methoden der hobbyistischen Informatik im allgemeinen und der Programmierlehre im besonderen Möglichkeiten darstellen, Wissen über Computern auf individuelle Weise (Inhalte, Geschwindigkeit, Methoden) zu erwerben. Der Erwerb von Selbstlernkompetenz findet zwar ohne explizierte Metakognition statt; Feedback erhält der Autodidakt jedoch vom System selbst. Durch die Erweiterung der didaktischen Konzepte der Gamification und des E-Learnings ist es möglich, den interaktiven Lernprozess, bei dem Computer zugleich Objekt und Werkzeug des Lernens sind, sowohl auf die historische Situation des Programmierlernenden als auch auf heutige Lernsituationen anzuwenden.

Abschließend sollen jeweils vier unterschiedliche Beispiele aus der Homecomputer-Zeit als auch aus dem Retrocomputing veranschaulichen, auf welche Weise das hier umrissene autodidaktischen Computerwissen umgesetzt wurde.

5.3 Homecomputing

Der Prototyp des Homecomputer-Nutzers ist der Hacker, der als Figur Ende der 1950er-Jahre am MIT auftaucht. Levy [1984:26f.] charakterisiert den Hacker über seine spezifischen Ethik, an deren Beginn der Wille zum Wissen über Computer steht: „Access to computers – and anything which might teach you something about the way the world works – should be unlimited and total. Always yield to the Hands-On Imperative!" [27] Dieser „access" ist durchaus in einem doppelten Sinne zu verstehen: Zum einen verschafften sich die MIT-Hacker auf unterschiedlichen Wegen *räumlichen Zugang* zu den dortigen Computer-Systemen [91,95]; zum anderen bestand das Bestreben auch immer auf *Zugänge zum Wissen* über Computer, bei der ‚Informationshürden' (wie sie etwa System Operators [vgl. ebd.:13f.] oder Betriebssystem-Restriktionen [vgl. ebd.:112-118] darstellten) überwunden werden mussten.

Sherry Turkle stellt 1984 fest, dass sich der modus operandi der nunmehr kindlichen und jugendlichen Hacker mehr als zwei Jahrzehnte später kaum gewandelt hat: An einer Schule, an der die Kinder freien Zugang zu den Computern hatten [vgl. Turkle 1984:117ff.], auf denen die Lehrprogrammiersprache LOGO installiert war, lernten sie „relativ frei [...] von schulischen Erwartungen" [117] selbstreguliert Programmieren. Ge-

leitet werden sie dabei von ihrer Neugier. Bestimmte Inhalte wurden von den Lehrern jedoch für unterschiedliche Jahrgangsstufen zugeordnet. Turkle rekapituliert folgende Anekdote:

> Als das System eingeführt wurde, meinten die Lehrer, die Richtungsbestimmung [von sich über den Bildschirm bewegenden Grafiken, SH] könnte für Zweitkläßler zu kompliziert sein, weil man dabei in Winkeln denken müsse. Man erklärte daraufhin diesen Kindern die Befehle, mit denen sie Sprites erscheinen lassen, ihnen Formen und Farben geben und ihnen einen Ort auf dem Bildschirm zuweisen konnten, aber nicht die Befehle, die sie brauchten, um sie in Bewegung zu versetzen. Die Bewegung blieb den Schülern der höheren Klassen vorbehalten. Diese Einschränkung konnte genau zwei Wochen aufrechterhalten werden. Das heißt, sie funktionierte solange, bis der Zweitkläßler Gary entdeckte, daß auf den Bildschirmen der älteren Kinder etwas Aufregendes passierte – er war sachverständig genug, um sich den Trick von einem stolzen und redseligen Drittkläßler erklären lassen zu können. In einem Punkt hatten die Lehrer recht gehabt: Gary verstand nicht, daß das, womit er umging, „Winkel" waren. Aber das war auch gar nicht notwendig. Er wollte den Computer veranlassen, etwas bestimmtes zu machen, und er fand eine Möglichkeit, die ihm fremde Vorstellungen des Winkels mit etwas gleichzusetzen, das ihm bereits vertraut war – mit Geheimcodes [...] [Turkle 1984:120f.]

Das Erkenntnisinteresse und die Vorgehensweise des hier vorgestellten Gary unterscheidet sich nicht von dem der MIT-Hacker. Wie Turkle weiter (und über andere Kinder) schreibt, entwickeln sich bei den jungen LOGO-Programmierern ohne äußere Einflussnahme unterschiedliche Programmierstile, je nachdem, welches Interesse diese dem Computer entgegenbringen (ob sie ihn beispielsweise als Kunstwerkzeug oder als zu erforschendes Ding sehen). In ihren Ausführungen zur Homecomputer-Zeit [203-240] findet sie diese Typen auch unter den jugendlichen Interviewpartnern, die sich nach dem Erwerb ihres Computers zumeist „selbst das Programmieren bei[brachten]" [214], um von da aus den Computer programmierend erforschen zu können:

> Ich kann das Ganze nicht von vorne bis hinten durchschauen ... In meinem Denken klafft eine große Lücke zwischen der Vorstellung, daß ein Stromkreis offen oder geschlossen sein kann, und dem binären Zahlensystem, und dann noch einmal von da aus zur Programmiersprache BASIC. Ich muß das alles irgendwie begreifen. [214f.]

Einzelne Nutzer, von denen Turkle berichtet, überwinden diese Lücke durch das Erlernen von Assemblersprachen, „um in unmittelbaren Kontakt mit dem zu sein, was innerhalb des Computers ‚wirklich passiert'." [Turkle 1984:220] Die Wahl der Programmiersprache hat kulturelle Implikationen: „Programmiersprachen und Programmierstile werden zu

Bausteinen bei der Schaffung von Computerkulturen, in diesem Fall der Kultur der ersten Hobbyanwender und der Hacker.“ [221] Die Betätigungsfelder sind dieselben, die sich heute auch im Retrocomputing zeigen: Programmierung hardwarenaher Sprachen [vgl. 222], Hardware-Hacking [222] sowie Fehlersuche und Reparatur der Homecomputer [223]. Als Teilnehmer der damals neu entstandenen privaten Computerkultur tritt der Hobbyist in Austausch mit Gleichgesinnten (zum Beispiel in Computerclubs [vgl. ebd.:211]), um gemeinsames Wissen über Computer systematisch zu sammeln.

Dieser Austauschprozess ist typisch für das Homecomputer-Zeitalter. Die Fähigkeiten des Autodidakten weiten sich ins Soziale aus, wo sein Lernen in soziale Lernhandlungen kulminiert: „Zwar gibt es Jugendliche, die sich an ihrem Heimcomputer isolieren“, schreibt Harenberg ein weitläufiges Gerücht über jugendliche Computernutzer widerlegend, „[...] Aber sogar unter den Hackern und Freaks, deren Welt nur noch aus Bits und Bytes, aus Load und Run zu bestehen scheint, sind solche Einzelgänger eine Minderheit. Die meisten haben alte Freunde verloren und neue gewonnen, mit denen sie fachsimpeln [...].“ [Harenberg 1984:119]

Claus Pias vergleicht die Figur des (historischen) Hackers diskursarchäologisch vor dem Hintergrund seines Umgangs mit Hardware, Wissen und Macht [Pias 2002b:256] mit einen *Spieler*. Ihm zufolge ist der Hacker „jemand, der sich sein Wissen selbst zusammensucht. Er ist respektlos gegenüber den willkürlichen Vorschriften von Programmen, Systemverwaltern oder Nutzungskontexten. [...] Er ist in seinem innersten Impuls ein Spieler, und seine historische Möglichkeitsbedingung ist der Digitalrechner als universale Spielmaschine.“ [ebd.] Er ist „ein Spieler mit digitalem a priori. Seine Existenz und Tätigkeit sind besonderen Spielmitteln und ihrer Kombinatorik geschuldet.“ [6] Der Hack „entspringt dem Spiel aller Computerspiele, dem Programmieren selbst.“ [ebd.][179]

Wie sich dieses spielerische Hacking, verstanden als Austragung des von Foucault skizzierten Wissens-Macht-Konfliktes, im Homecomputing als Autodidaktik und Wissenspraxis methodisch ausdifferenziert, zeigt sich an den folgenden Beispielen. Die vier historischen Publikationen sollen ebenso die Überschreitung zu einer (auto)didaktisch systematischen Wissens-Selbstaneignungspraxis nachvollziehbar machen. Deren Methoden werden von ihren Autoren zwar nicht explizit genannt, lassen sich aber implizit aus dem Aufbau, Stil und den verwendeten Materialien ihrer Bücher ableiten.

179 Dieser im positiven Sinne verstandene Hacker wechselt im Homecomputer-Zeitalter sein Image. In dem Maße, wie er sein Computerwissen ausnutzt, um nicht aus Wille zum Wissen, sondern mit kriminellen Intentionen in Systeme einzudringen, Daten zu stehlen oder zu manipulieren und Hardware und Software zu sabotieren, wird er zu einem Partisanen [vgl. Höltgen 2016c].

5.3.1 Das Handbuch des Schneider CPC6128

Die Programmierlehre findet zur Zeit der frühen Mikrocomputer zumeist im Privaten statt. Die Tatsache, dass hierbei lehrerlos gelernt wird, bedeutet jedoch nicht, dass die jungen Programmierer dieser Zeit ohne Hilfsmittel lernen mussten [vgl. Solomon 2003:4]. Thomas E. Kurtz, einer der Entwickler von BASIC, präferiert das am Handbuch orientierte Selbststudium: „Attending a class is pretty much a waste of time. [...] The way to learn new languages is to read the manual.“ [zit. n. Biancuzzi/Warden 2009:91] Und in der Tat geben die Manuals der Homecomputer sowie die erhältliche Literatur vielfältigste Hilfestellungen für das Selbstlernen.

Als Beispiel sei an dieser Stelle das deutschsprachige Handbuch des Schneider CPC6128 aufgeführt. Mit 520 Seiten gehört es zu den umfangreichsten Manuals, die Homecomputern von ihren Herstellern beigegeben wurden. Neben Aufbau- und Inbetriebnahmeanleitungen für den Computer bietet es Kurzeinführungen in die Bedienung der Peripherie, die Programmiersprache Locomotive-BASIC 1.1 (inklusive Grafik- und Soundprogrammierung) und die Betriebssysteme AMSDOS und CP/M (vgl. Abb. 5.1). Darauf folgen umfangreiche Kapitel, in denen diese Themen detaillierter ausgebreitet werden und zusätzlich eine systematische Einführung in die Programmierung mit LOGO (das dem Computer ebenfalls beigegeben ist). In Schlusskapitel mit dem Titel „Wenn Sie mal Zeit haben“ wird der Nutzer mit Informationen zum „Computer im allgemeinen“ sowie dem „CPC6128 im besonderen“ [Spital et al. 1985:4] versorgt. Hier (vgl. Abb. 5.2) finden sich sowohl Themen der Computergeschichte, der Theoretischen und Praktischen Informatik sowie plattformspezifische Informationen (bis hin zur hardwarenahen Programmierung z. B. von Interrupts [vgl. ebd.:9/30].

GOSUB

GOSUB ‹Zeilennummer›

```
GOSUB 210
```

KOMMANDO: Springt zu einem BASIC-Unterprogramm (engl. SUB-routine) in der angegebenen ‹Zeilennummer›. Nach dem Kommando RETURN, mit dem jedes Unterprogramm endet, fährt das Programm mit der Anweisung fort, die dem Aufruf des Unterprogramms folgt.

Verwandte Befehle: RETURN

GOTO

GOTO ‹Zeilennummer›

```
GOTO 90
```

KOMMANDO: Springt zur angegebenen ‹Zeilennummer›.

Verwandte Befehle: **keine**

GRAPHICS PAPER

GRAPHICS PAPER ‹Farbstift›

```
10 MODE 0
20 MASK 15
30 GRAPHICS PAPER 3
40 DRAW 640,0
run
```

KOMMANDO: Legt fest, auf welchem Farbhintergrund (‹Farbstift›-Skala 0 bis 15) die Zeichnungen auf dem Graphik-Bildschirm erscheinen sollen. Beim Zeichnen durchgehender Linien ist die Hintergrundfarbe unsichtbar. Im obigen Beispiel wird mittels des MASK-Befehls eine durchbrochene Linie gezeichnet, so daß die Hintergrundfarbe sichtbar wird.

Die GRAPHICS PAPER-Farbe bildet den Hintergrund für Zeichen, die unter Anwendung des TAG-Befehls auf den Graphik-Bildschirm geschrieben werden. Sie stellt außerdem die Standardfarbe dar, wenn der Graphik-Bildschirmbereich mit dem Befehl CLG gelöscht wird.

Verwandte Befehle: CLG, GRAPHICS PEN, INK, MASK, TAG, TAGOFF

GRAPHICS PEN

GRAPHICS PEN [‹Farbstift›][,‹Hintergrundmodus›]

```
10 MODE 0
20 GRAPHICS PEN 15
30 MOVE 200,0
40 DRAW 200,400
50 MOVE 639,0
60 FILL 15
run
```

KOMMANDO: Bestimmt den ‹Farbstift› (Bereich 0 bis 15) zum Zeichnen von Linien und Punkten. Der ‹Hintergrundmodus› ist entweder

0: Nicht transparenter Hintergrund

oder

1: Transparenter Hintergrund

(Beim Transparent-Modus erscheint der Hintergrund der unter Anwendung des TAG-Befehls auf den Bildschirm geschriebenen Zeichen sowie die Zwischenräume in gestrichelten Linien in der GRAPHICS PAPER-Farbe.)

Einer der beiden Parameter, jedoch nicht beide, können ausgelassen werden. Wird ein Parameter ausgelassen, ändert sich die jeweilige Einstellung nicht.

Verwandte Befehle: GRAPHICS PAPER, INK, MASK, TAG, TAGOFF

HEX$

HEX$ (‹vorzeichenloser ganzzahliger Ausdruck›[,‹Feldbreite›])

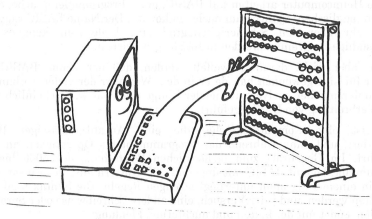
```
PRINT HEX$(255,4)
00FF
```

FUNKTION: Gibt den Wert des ‹vorzeichenlosen ganzzahligen Ausdrucks› in HEXadezimaler Form wieder. Die ‹Feldbreite› gibt an, wieviele Hexadezimalziffern (zwischen 0 und 16) verwendet werden sollen. Ist die Anzahl der tatsächlich benötigten Ziffern kleiner als vorgeschrieben, so werden der Hexadezimalzahl Nullen vorangestellt. Ist sie größer als vorgeschrieben, so wird die Hexadezimalzahl NICHT gekürzt, sondern in sovielen Ziffern ausgegeben, wie benötigt werden.

Der Wert des ‹vorzeichenlosen ganzzahligen Ausdrucks›, der in Hexadezimalform umgewandelt werden soll, muß zwischen –32768 und +65535 liegen.

Verwandte Befehle: BIN$, DEC$, STR$, UNT

Abb. 5.1: Doppelseiten aus dem deutschen Handbuch des Schneider CPC6128



Wenn Ihnen dieser Prozeß umständlich vorkommt, haben Sie recht, und die erste und wichtigste Wahrheit der Datenverarbeitung erkannt. Ein Computer ist ein Werkzeug zur Ausführung allereinfachster, immer wiederkehrender Aufgaben -sehr schnell und absolut präzise. BASIC interpretiert also die in einem Programm enthaltenen Anweisungen und wandelt sie in eine Sprache (Maschinencode) um, die von der Zentraleinheit verstanden wird. Der Computer versteht nur zwei Zustände: 'Ja' und 'Nein', die in binärer Schreibweise mit 1 und 0 dargestellt werden. In der Boole'schen Logik gibt es nur 'wahr' und 'falsch', kein 'jein' oder 'vielleicht'!

Das Umschalten zwischen diesen beiden bestimmten Zuständen bildet den Kern des Begriffs 'digital'. In der Natur verlaufen die meisten Prozesse allmählich und geradlinig von einem stabilen Zustand zu einem anderen; der Übergang verläuft langsam über eine Verbindungslinie zwischen den beiden Zuständen. In einer idealen digitalen Umgebung dagegen wird buchstäblich in Null-Komma-Nichts von einem Zustand auf einen anderen umgeschaltet. In Wirklichkeit verursachen die Grenzen der Halbleiter-Physik jedoch eine minimale Verzögerung, die sogenannte Laufzeitverzögerung. Die Summe vieler Laufzeitverzögerungen sind der Grund, weshalb es etwas Zeit braucht, bis die Information verarbeitet ist und die Antwort ausgegeben wird.

In jedem Fall muß der Computer eine bestimmte Zeit warten, bis eine Aufgabe fertig ist und er deren Ergebnis für die nächste Operation verwenden kann. In einer idealen Umgebung müßte also eine künstliche Verzögerung eingebaut werden. Der digitale Prozeß kennt nur schwarz und weiß; die dazwischenliegenden Grautöne sind bedeutungslos. Der lineare oder 'analoge' Übergang hingegen führt über alle Grautöne.

Da die letztendliche Antwort nur 0 oder 1 lauten kann, gibt es kein 'fast richtig'. Daß der Computer manchmal scheinbar Fehler bei Operationen mit numerischen Daten macht, hängt damit zusammen, daß er nur Zahlen bis zu einer bestimmten Größe verarbeiten kann. Numerische Daten, die diese Grenze überschreiten, werden abgeschnitten oder gerundet, damit sie in den verfügbaren Platz passen. Beispielsweise wird 999,999,999 zu 1,000,000,000.

Wie kann man nun in einer Welt, die nur 0 und 1 kennt, weiter als bis 1 zählen?

Bits und Bytes

Wir haben uns zufällig daran gewöhnt, mit Zahlen des Dezimalsystems umzugehen, in dem der Bezugspunkt die Zahl 10 ist, d.h. zur Darstellung von Mengen zwischen 0 und 9 (0 bis 9 ist der Schreibweise 1 bis 10 vorzuziehen) stehen uns zehn Ziffern zur Verfügung. Das Zahlensystem mit zwei Ziffern (0 und 1) heißt Dualsystem oder Binärsystem, und die Einheiten, mit denen das System arbeitet, heißen Bits (Abk. für Binary digIT).

Der Zusammenhang zwischen Bits und der dezimalen Schreibweise ist leicht zu verstehen:

Normalerweise wird die maximale Anzahl der verwendeten Binärziffern in einer Zahl angegeben, indem soviele Nullen vorangestellt werden, daß die Bit-Anzahl vollständig ist.

Aus der Dezimalzahl 7 wird:

00111

...in 5-Bit-Schreibweise.

Die Ziffern im Binärsystem können einfach als Indikatoren angesehen werden, die für ihre Spalte angeben, ob eine bestimmte Potenz von 2 gegeben ist. 1 = ja; 0 = nein

$2^0=1$
$2^1=2=2=2(2^0)$
$2^2=4=2x2=2(2^1)$
$2^3=8=2x2x2=2(2^2)$
$2^4=16=2x2x2x2=2(2^3)$

Abb. 5.2: Doppelseite aus dem CPC6128-Handbuch, Kapitel „Wenn Sie mal Zeit haben“

https://doi.org/10.18452/21165

Die Dokumentation der Programmiersprachen erfolgt grundsätzlich über Programmierbeispiele, in denen der Nutzer kleine Programme oder Befehle im Direktmodus einzugeben aufgefordert wird, um sich die Funktionsweise des Locomotive-BASIC zu vergegenwärtigen. Querverweise zu „verwandten Befehlen“ unterhalb jeder Befehlsbeschreibung machen das Handbuch wie einen Hypertext nutzbar und stellen damit eine Alternative zur hierarchisch-systematischen Lektüre dar. Der Programmierlehrling kann das Handbuch ganz nach seinen Wünschen und Erfordernissen nutzen und je nach Kenntnisstand die Kurzreferenz der Befehle oder ihre ausführliche Darstellung mit Programmbeispielen und Querverweisen verwenden. Das Fachwortglossar im Anhang [ebd.:Anhang 2/1-2/33] und die ebenfalls dort befindlichen „6 Programme für Computerspiele“ [ebd.:Anhang 3/1-3/30] zum Abtippen und Modifizieren machen das Handbuch zu einer Quelle, die den autodidaktischen Lernprozess vom Programmieranfänger bis zum Fortgeschrittenen ohne weitere Sekundärliteratur begleiten kann.

Jedem Homecomputer wurden mehr oder weniger umfangreiche Handbücher beigegeben, in denen auch die im ROM integrierte Programmiersprache dokumentiert und gelehrt wurde. Das Bundle Computer plus Handbuch reichte dabei für die grundlegende Selbstausbildung aus. In dem Moment, wo der Programmierlerner weiterreichende Informationen benötigte, konnte er sich an den Buchmarkt wenden. Damit war die rein autarke Lernsituation zwar aufgehoben; um aber zu wissen, über welches weitergehende Thema sich der Lernende detaillierter informieren will, musste er bereits ein Wissen über sein Nichtwissen besitzen. Das Erfragte muss also bereits im Gefragten enthalten sein. (Vgl. Heidegger 2006:5) Hierin lassen sich durchaus Ansätze zu einer Meta-Kognition des Selbstlernprozesses sehen.

5.3.2 C16-BASIC-Kurs

1986, zwei Jahre nachdem die Firma *Commodore* die Herstellung der drei 264er-Modelle C116/16/Plus 4, mit der sie ihre Produktpalette im unteren Preissegment erweitern wollte, ohne großen Erfolg eingestellt hatte, wurden die Lager des in großen Stückzahlen hergestellten Commodore 16 durch das Marketing als *Computer-Lernkurs* auf dem Karton umgelabelt (vgl. Abb. 5.3) und in ALDI-Discountern zu niedrigen Preisen angeboten.[180] Das Set enthielt einen Commodore-16-Homecomputer, einen Datenrecorder, zugehörige Kabel, eine Kassette mit Programmier-Lernsoftware (BASIC-Kurs) sowie die Bedienungshandbücher für den Computer, die Peripherie und das Lehrbuch *Lerne BASIC mit den Commodore 116/16/Plus 4* als „Eine Anleitung zum Selbststudium in 10 Lektionen“ [Scharnbacher 1985].

180 https://www.c64-wiki.de/index.php/Commodore-264-Serie [letzter Abruf: 12.04.2016].

Abb. 5.3: Oben: ursprünglicher Karton des Commodore C16 und der Datassette 1631; unten: Der Commodore 16 als „Computer-Lernkurs BASIC-Programmiersprache“-Paket

Der Homecomputer tritt in diesem Set weniger als Gegenstand der autodidaktischen Programmierlehre auf, als als deren notwendiges Werkzeug zum Erlernen von BASIC. Scharnbacher führt demzufolge streng systematisch und mit didaktischem Anspruch in die Programmierung ein: Er benutzt Struktogramme, nennt am Anfang jedes Kapitels das Lernziel und entwickelt die Lernfortschritte anhand eines sich als Leitmotiv durch das Werk ziehenden Programmierproblems: der Berechnung einfacher Bankzinsen. (Scharnbacher ist hauptberuflich Professor für Wirtschaftswissenschaft und befasst sich als solcher unter anderem mit der Gestaltung von Planspielen.) Die einzelnen Kapitel des Buches schließen mit Übungsaufgaben, deren Lösungen am Ende des Bandes abgedruckt sind.

Das Selbststudium muss hier, im Gegensatz zur Programmierlehre des CPC6128-Handbuches, linear in strenger Kapitelreihenfolge vollzogen werden. Mit seiner induktiven Lehrmethode, die „von der bloßen Kenntnis der Sprachelemente [bis] zur Programmierlogik“ [9] führt, steht der *Computer-Lernkurs* an der entgegen gesetzten Seite des Spektrums autodidaktischer Programmierlehre – dem CPC6128-Handbuch gegenüber. Beide Bücher richten sich an jugendliche oder erwachsene Nutzer – Scharnbacher aufgrund seines leitmotivischen Beispiels und der didaktischen Methodik, Spital et al. auf-

grund ihres sprachlichen Stils (konsequentes Siezen des Lesers) und der vergleichsweisen ‚Trockenheit' der Gestaltung.

5.3.3 Spielend programmieren lernen

Sekundärliterarische Lehrwerke, die sich dem Thema Programmieren von der spielerischen Seite her zuwenden, finden sich vergleichsweise häufig zur Zeit der Homecomputer (vgl. Abb. 5.4). Sie betonen den spielerischen (verstanden als „spaßigen") Aspekt der Programmierlehre nicht selten bereits in ihren Titeln [vgl. Schumacher 1982, Baumann 1983, Koch 1985, Gutzer 1987 oder Seibert 1989]. Dabei rufen die Begriffe „Spiel" und „Spaß" sowohl die anvisierte Methodik des ‚spielerischen Lernens' (playfulness) und damit Elemente von Gamification auf als auch das vordringliche Lernziel: die Programmierung (eigener) Computerspiele. In ihrer grafischen Aufmachung erwecken einige der Bücher [vor allem Schumacher 1982 und Baumann 1983] den Eindruck von Kinder(lehr)büchern. Dennoch richten sich alle hier genannten Werke an heranwachsende oder Erwachsene: Koch [1985:7], Schumacher [1982:3] und Seibert [1989:11] siezen ihren Leser. Baumann referenziert ihn neutraler als „den Spieler" [1983:1], lässt aber aufgrund seines Sprachstils, des mathematischen Niveaus und der in den Übungsaufgaben behandelten Themen unzweideutig erkennen, dass er höheres schulisches Wissen voraussetzt.

Mit Ausnahme von Baumann verstehen sich alle hier genannten Bücher als Einführungen in die Programmierung und stellen den BASIC-Befehlssatz (plattformübergreifend [Schumacher 1982, Koch 1985] oder plattformspezifisch für Commodore-64-Kompatible [Seibert 1989]) dementsprechend detailliert vor. Dabei führen sie induktiv in die Befehle, Funktionen, Operatoren und audiovisuellen Programmiermöglichkeiten von BASIC ein [Schumacher 1982:2], ermöglichen eine eher selbstbestimmte Vorgehensweise (wie Koch, dessen Programmier-Einführung einem Nachschlagewerk ähnelt [vgl. Koch 1985:12-38]) oder gehen in medias res in die Spielprogrammierung, indem Sie komplette BASIC-Listings mit Einführung und Programmbeschreibung anbieten – also in Grundzügen eine deduktive Lehrmethode zur Anwendung bringen.

Alle vier Werke eignen sich zum autonomen Selbststudium, allerdings auf unterschiedlichen Ausgangsniveaus – sowohl bezüglich der Programmierkenntnisse als auch der erforderlichen allgemeinen Vorbildung. Die Bücher stellen einen allerdings lediglich kleinen Ausschnitt an thematisch spezialisierter Programmierliteratur des Homecomputer-Zeitalters dar. BASIC und andere Programmiersprachen ließen sich vor dem Hintergrund aller möglichen Themen und Softwaregattungen autodidaktisch erlernen. Dort, wo Computerspiele diese thematische Basis bildeten, ließ sich – wie in einigen der hier genannten Beispiele – nicht selten die zugrunde liegende, spielerische Didaktik für den Leser erkennen.

Abb. 5.4: Frontcover der Bücher *Spielend programmieren lernen* und *Programmieren ganz einfach*

5.3.3 Mein Heimcomputer selbstgebaut

Dass sich Computerliteratur nicht allein an Programmierschüler richtet, sondern auch die Hardware zum Gegenstand hat, soll beispielhaft das letzte hier vorgestellte Buch zeigen: „Mein Heimcomputer selbstgebaut zum Lernen, Spielen, Messen, Steuern, Regeln …“ führt schrittweise in den Selbstbau eines Lerncomputers auf Basis des Mikroprozessors 2650 von *Signetics* ein. Die Autoren verfolgen mit dem Bauprojekt das Ziel, dem Leser/Lernenden „die Arbeitsweise eines Computers [zu] vermitteln“ [Glagla/Feiler 1984:9]. Sie begründen ihr Vorhaben damit, dass „über das materielle Innenleben, die Hardware, also alles, was man anfassen kann, [..] in den Bedienungsanleitungen [der Kleincomputer, S.H.] nicht viel zu erfahren“ [ebd.] ist. Demzufolge liefern sie dem Leser nach einer Einführung in die Digitalelektronik eine Anleitung zum Aufbau eines modularen Systems – von der Herstellung der Platinen bis zur Eingabe des BIOS (vgl. Abb. 5.5).

Der Komplexität der Thematik entsprechend ist die Vorgehensweise kleinschrittig, technisch detailliert und notwendigerweise induktiv angelegt. Für jeden Aufbauschritt werden Fotos, Zeichnungen, Platinenlayouts, Diagramme und Tabellen mit Messwerten angeboten. Nachdem der Aufbau der Hardware abgeschlossen ist, bieten die Autoren noch verschiedene Programme an, die in hexadezimalen Opcodes eingegeben werden müssen. (Der Selbstbaucomputer verfügt lediglich über binäre und hexadezimale Input- und Output-Schnittstellen.) Von den von den Autoren im Buch-Untertitel aufgeführten Zwecken „Lernen, Spielen, Messen, Steuern, Regeln, …“ sind lediglich die letzten drei als konkrete Anwendungsfälle zu verstehen. „Lernen“ bezieht sich auf den Konstruktionsprozess insgesamt; „Spielen“ kann (angesichts des einzigen einfachen Spielprogramms

„Würfelspiel“ [ebd.:265], bei dem drei Zufallszahlen erzeugt und als Dezimalziffern auf dem Siebensegment-Display angezeigt werden) ebenfalls nur metaphorisch gemeint sein – vielleicht im hier vermuteten Sinn einer spielerischen Annäherung an die Computerhardware über die Methode des Bastelns.

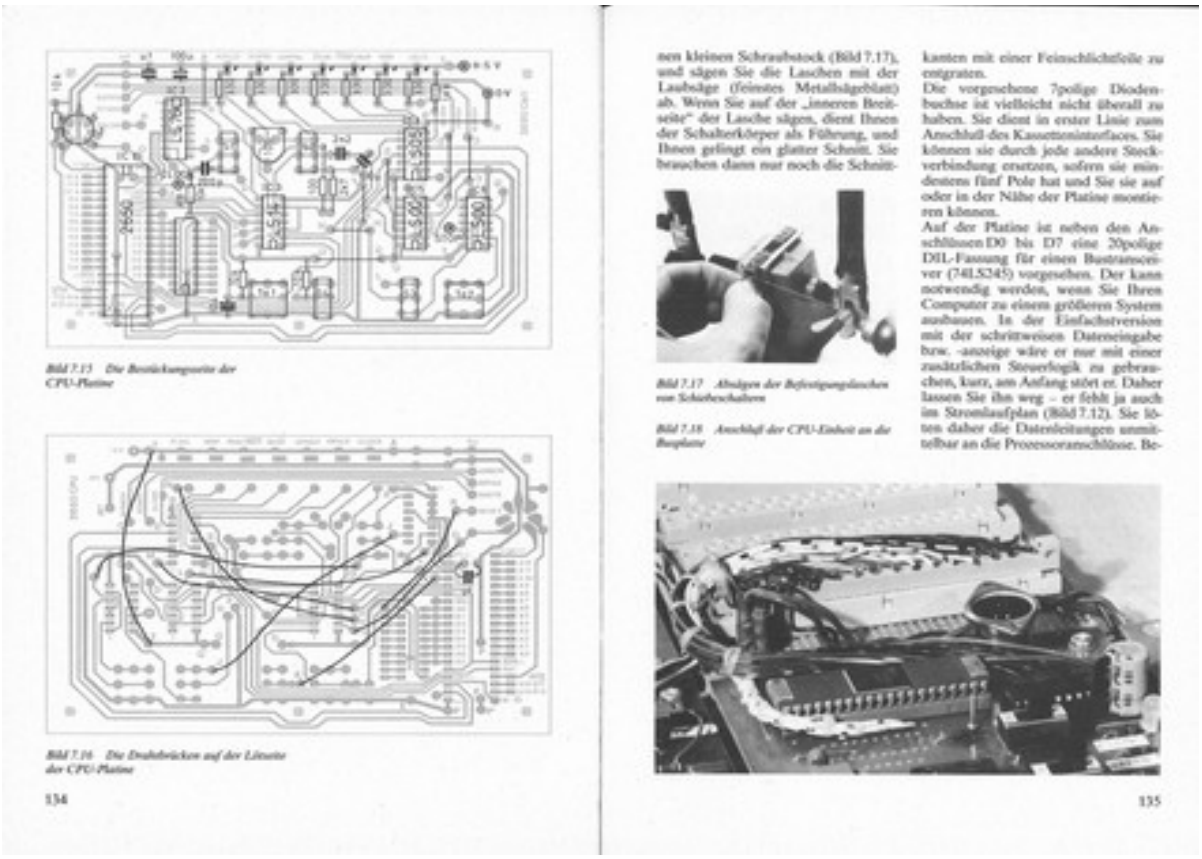


Bild 7.15 Die Bestückungsseite der CPU-Platine

Bild 7.16 Die Drahtbrücken auf der Lötseite der CPU-Platine

134

nen kleinen Schraubstock (Bild 7.17), und sägen Sie die Laschen mit der Laubsäge (feinstes Metallsägeblatt) ab. Wenn Sie auf der „inneren Breitseite“ der Lasche sägen, dient Ihnen der Schalterkörper als Führung, und Ihnen gelingt ein glatter Schnitt. Sie brauchen dann nur noch die Schnittkanten mit einer Feinschlichtfeile zu entgraten.

Die vorgesehene 7polige Diodenbuchse ist vielleicht nicht überall zu haben. Sie dient in erster Linie zum Anschluß des Kassetteninterfaces. Sie können sie durch jede andere Steckverbindung ersetzen, sofern sie mindestens fünf Pole hat und Sie sie auf oder in der Nähe der Platine montieren können.

Auf der Platine ist neben den Anschlüssen D0 bis D7 eine 20polige DIL-Fassung für einen Bustransceiver (74LS245) vorgesehen. Der kann notwendig werden, wenn Sie Ihren Computer zu einem größeren System ausbauen. In der Einfachstversion mit der schrittweisen Dateneingabe bzw. -anzeige wäre er nur mit einer zusätzlichen Steuerlogik zu gebrauchen, kurz, am Anfang stört er. Daher lassen Sie ihn weg – er fehlt ja auch im Stromlaufplan (Bild 7.12). Sie löten daher die Datenleitungen unmittelbar an die Prozessoranschlüsse. Be-

Bild 7.17 Absägen der Befestigungslaschen von Schalterschalter

Bild 7.18 Anschluß der CPU-Einheit an die Busplatte

135

Abb. 5.5: Doppelseite aus dem Buch [Glagla/Feiler 1984]

Bücher, Zeitschriftenreihen und Fernsehkurse, in denen der Aufbau einfacher 4-, 8- und 16-Bit-Computer Schritt für Schritt vollzogen und dem Rezipienten zur Nachahmung empfohlen wird, erschienen insbesondere ab der zweiten Hälfte der 1970er-Jahre. Zu dieser Zeit hatten sich fertige Tastaturcomputer mit Bildschirmanschluss auf dem Markt für Privatanwender noch nicht durchgesetzt, so dass ein Projekt wie das von Glagla/Feiler dem Leser durchaus ernst gemeint „nebenbei auch noch zu einem eigenen Homecomputer verhelfen“ [ebd.:9] konnte. Das 1984 erschienene Buch wurde aber auch noch weit über das Homecomputer-Zeitalter hinaus für den schulischen Informatik-Unterricht genutzt. In [Höltgen 2013c] berichtet der Lehrer einer Integrierten Gesamtschule, wie er die Plattform bis zum Jahr 2011 im Wahlpflichtfach Informationstechnische Grundausbildung der Jahrgangsstufen 9 und 10 eingesetzt hat. Die Schüler haben die Systeme dabei von Grund auf in Gruppenarbeit aufgebaut und programmieren gelernt. Als ein Ergebnis benennt Brandt das vertiefte Verständnis von Computerprozessen: „Die Schüler haben bei einer von ihnen selbst programmierten und zum Teil selbst gebauten

Maschine das Gefühl, dass sie die übersichtliche Hard- und die Software beherrschen.“ [Brandt zit. n. Höltgen 2013c:49] (Vgl. Abb. 5.6)

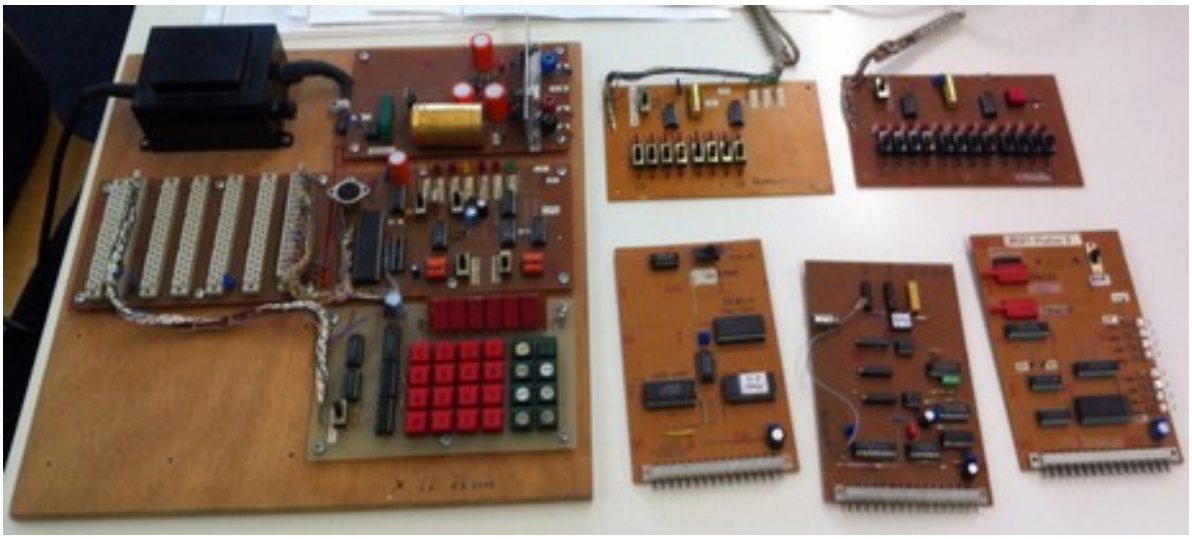


Abb. 5.6: Das von einem Schüler fertig aufgebaute Signetics-2650-System

5.4 Retrocomputing

Das zuletzt vorgestellte Schulprojekt gehört bereits in die Sphäre des *Retrocomputing*, bei dem historische Plattformen (oder wie oben: Architekturen) in heutiger Zeit genutzt und mit kontemporären Methoden und Technologien konfrontiert werden. Im folgenden sollen – ebenfalls kursorisch und auf ihre (auto)didaktischen Methoden hin befragt – drei solcher Retrocomputing-Projekte skizziert werden.

5.4.1 Das Computerspiel Rock (Berthold Fritz)

Fritz [2014] dokumentiert den Entstehungsprozess seines vollständig selbst entwickelten Computerspiels Rock für Atari-8-Bit-Computer. Seine Grundmotivation dafür benennt er: „Ich wollte endlich verstehen, wie die Dinge zusammenhängen und das Medium [Computer, S.H.] als Ausdruck meiner eigenen Kreativität nutzen.“ [190] Zu diesem Zweck nahm er sich vor 6502-Assembler zu lernen, um ein Spiel zu programmieren. Bei der Vorbereitung des Projektes musste er verschiedene Entscheidungen darüber treffen, unter welcher Umgebung und mit welcher Software er das Spiel programmieren wollte. Den Prozess der Spielerstellung selbst beschreibt er als modular (im Sinne des Software Engineering). Seine Fortschritte unterbricht er immer wieder mit Programmier-Experimenten, in denen er sich beispielsweise die Funktionen der Grafik, des Sounds oder der Speicherverwaltung der CPU vergegenwärtigen muss. Das Timing des Codes stellt aufgrund der Rasterstrahl-orientierten Grafikausgabe eine besondere Herausforderung an die Codeoptimierung dar.

Das Spiel, das im Rahmen einer Vortragsreihe und des nachfolgenden Buches öffentlich vorgestellt wurde, war zu diesem Zeitpunkt noch unvollendet, jedoch bereits spielbar. Den Entwicklungsprozess hat Fritz in seinem Weblog[181] sukzessive und detailliert dokumentiert und mit den dortigen Lesern diskutiert. Die Vorgehensweise von Fritz entspricht der autodidaktischen Methodik, mit der Programmierschüler bereits zur Zeit der Homecomputer Assembler gelernt haben: Eine kurze Evaluation der möglichen Entwicklungsumgebungen, propädeutische Programmierexperimente und die etappenweise Entwicklung von Spielalgorithmen. Die Werkzeuge Fritz' unterscheiden sich allerdings maßgeblich von denjenigen der 1980er-Jahre: Rock entstand in einem hybriden Cross-Platform-Development-Verfahren: auf einem modernen PC mit Atari-Emulation, in welcher er einen historischen Assemblierer (ATMAS II [vgl. ebd.:195]) laufen ließ. Dieses Setting garantierte zum einen eine möglichst sichere Entwicklungsumgebung, weil der aktuelle Status des Projektes jederzeit problemlos gespeichert werden konnte; zum anderen konnten die Programmbestandteile sofort im Emulator getestet werden, welcher wiederum die Hardware-Funktionen der Zielplattform vergleichsweise adäquat abbildete und so ein authentisches Feedback zu den in ihm ablaufenden Rock-Bestandteile lieferte. Über historische Lehrbücher, Dokumentationen [vgl. ebd.:209] und internetbasiertes Feedback konnte Fritz Wissen gewinnen und ergänzen.

5.4.2 Das Betriebssystem SymbOS (Jörn Mika)

Im selben Rahmen wie Rock hat auch Jörn Mika [2014] sein Retrocomputing-Projekt SymbOS (SYmbiosis Multitasking Based Operating System) vorgestellt. Dabei handelt es sich um ein präemptives Multitasking-Betriebssystem mit grafischem User-Interface und Maussteuerung, wie es von modernen PCs bekannt ist. SymbOS wurde jedoch für verschiedene 8-Bit-Computerplattformen auf Z80-Basis entwickelt – allen voran für Schneider-CPCs. Zur notwendigen Hardware-Erweiterung, wozu neben zusätzlichem RAM-Speicher, einer batteriegepufferten Uhr und der PS2-Mausschnittstelle auch ein IDE-Interface für den Anschluss von Massenspeichern (Festplatte, CF-Karte usw.) gehört, hat Mika zusammen mit Dr.Zed (Hacker-Name) das SymbiFace (mittlerweile in Revision 2[182]) entwickelt: eine Platine, die diese Hardware-Ergänzungen anschlussfertig für den CPC enthält. SymbOS wird mit Unterbrechungen seit dem Jahr 2000 von Mika kontinuierlich weiterentwickelt und liegt inzwischen (Stand Juni 2019) in der Version 3.0 vor, bei der nun auch eine LAN/WAN-Vernetzung des Computers möglich ist. Neben dem Betriebssystem selbst entwickelt Mika auch Applikationen dafür – darunter Browser, Videoplayer, Spiele, Programmiersprachen (Interpreter und Assemblierer) und anderes.

181 https://retrozock.com/category/projekt-rock/ [letzter Abruf: 12.04.2016].

182 http://www.symbos.de/blog.htm sowie http://www.symbos.de/changelg.htm [letzter Abruf: 07.06.2019].

Den frühen Entwicklungsprozess beschreibt Mika als stetigen Fort- und Rückschritt. Nachdem er die Idee zur Entwicklung des Betriebssystems begonnen hatte in die Tat umzusetzen, stockte der Programmierfortschritt aufgrund von Wissensmangel [vgl. Mika 2014:113]. Die lange Entwicklungszeit und die Tatsache, dass Mika den Assemblercode vollständig solitär entwickelt hat, lassen eine stetigen Kenntnisprogression vermuten. Den Aufbau moderner Betriebssysteme hat er mithilfe der einschlägigen Informatik-Literatur (wie etwa [Tanenbaum 2009]) erlernt. Dies lässt sich aus der von ihm verwendeten Fachterminologie und den Diagrammen in seinem Text entnehmen [vgl. Mika 2014:105-108]. Mika ist allerdings ausschließlich als Hobbyist vorgegangen – mit der Motivation zu demonstrieren, dass ein Multitasking-GUI-Betriebssystem „schon relativ effektiv und kompromisslos auf 8-Bit-Rechnern“ [97] funktioniert.

5.4.3 Der Emulator JavaCPC (Markus Hohmann)

Software-Emulatoren historischer Computerhardware gehören zu den bekanntesten Retrocomputing-Projekten, zumal sie nicht nur privat, sondern auch in professionellen Kontexten (etwa der Software Preservation) eingesetzt werden. Sie entstehen allerdings zumeist in hobbyistischen Kontexten. [Vgl. Lange 2016:318] Im Zuge steigender Leistungsfähigkeit von PCs lassen sich mittlerweile nicht nur Plattformen mit 32-Bit-Architektur emulieren; Emulatoren können inzwischen auch als plattformunabhängige Applets in HTML5, JavaScript oder Java in Webbrowsern genutzt werden.

Ein solitär in Java programmierter Emulator für Amstrad/Schneider-CPCs stellt der von Markus Hohmann seit 2006 entwickelte JavaCPC dar. [Vgl. Höltgen 2013c:66] Die Besonderheit bei diesem Projekt ist, dass der Entwickler zugleich eine moderne Programmiersprache (Java) lernen und die Hardware-Funktionen der Zielplattformen (sämtliche Amstrad/Schneider-CPCs nebst deren internationalen Versionen und nachgebauten Clones) in dieser Sprache reverse engineeren musste. Hohmann ging dabei nach einer ‚traditionellen Hackermethode‘ vor: Er analysierte den Java-Sourcecode eines existierenden CPC-Emulators und lernte zunächst durch Modifikation die Funktionsweise von Java und des Emulators. Die Originalplattform selbst stellte der Entwicklung zusätzliche Hürden in den Weg, sind bestimmte elektronische Bestandteile (wie etwa das Gate-Array) nicht dokumentiert und müssen via Black-Box-Analysen in ihrem Verhalten nachempfunden werden. Hohmann notiert, dass er allerdings ohne Elektronik-Kenntnisse [ebd.:67] vorgegangen ist und die Emulation bestimmter Komponenten aus anderen Emulatoren entnommen hat. Sein Bestreben war und ist es, seinen Emulator mit immer weiteren Features auszustatten – sei es der Skeumorphismus peripherer Elemente wie Peripherie-Geräusche, Bildschirm-Spiegelungen oder die Erweiterung des Original-Systems (bzw. seiner Emulation) um Komponenten, die nur virtuell möglich sind, wie Megabyte-große RAM-Speicher, simulierte Festplatten, eingebaute Assemblierer, Grafik-Editoren, eine CPC-Desktop-Umgebung und anderem.

Hohmanns Emulator-Projekt ist ein markantes Beispiel für vollständig autonome Programmierlehre. Er entwickelt JavaCPC unstetig weiter – zumeist, wenn er Feature-Anfragen bekommt oder Ideen für neue Erweiterungen entwickelt. Hohmann betreibt zudem ein Informationsportal zum CPC, zu dem auch ein Diskussionsforum und eine Software-Datenbank gehören. Ausschlaggebend für sein Tun ist zunächst *Nostalgie*, ein wichtiger Trigger für die Neubeschäftigung mit frühen Mikrocomputern [vgl. Felzmann 2014]: „Der Anreiz für das Ganze ist ganz klar die Tatsache, dass der CPC damals mein allererster Computer war." [Hohmann zit. n. Höltgen 2013c:66] Die Tatsache, dass er es aber nicht bei der möglichst historischen-authentischen Emulation seines „allerersten Computers" belässt, sondern die Software stetig weiterentwickelt, deutet darauf hin, dass ihn ein zusätzliches Interesse motiviert. Dies dürfte neben der Gewinnung von Erkenntnissen über die Plattform vor allem in der Zusammenarbeit mit der Community [vgl. Höltgen 2013:68] zu finden sein.

5.5. Zusammenfassung

Das vorangegangene Kapitel hat, als Ergänzung zu den Projekten unter Kapitel 4, die *Didaktik des Retrocomputing* dargestellt. Ausgehend von Foucaults Analysen zu den diskursiven Beziehungen zwischen Wissen und Macht wurden medientechnische Dispositive als Machtgefüge definiert, die deren Nutzer verstehen wollen. Weil sich die Dispositive der Macht, die diesem Willen zum Wissen entgegenstehen, nicht nur in der Technik, sondern auch in der Pädagogik äußern, greift der Lernwillige auf autodidaktische Formen des Wissenserwerbs zurück. Diese wurden mit Blick auf konstruktivistische Lerntheorien definiert und in ihren spezifischen Methoden Trial and Error, Gamification und E-Learning für die hobbyistische Informatikselbstlehre vorgestellt. Dabei haben alle drei Begriffe eine Erweiterung gegenüber ihren akademischen Definitionen erfahren, was sich deshalb als sinnvoll und fruchtbar erwies, weil bei der autodidaktischen Informatiklehre Computer sowohl Werkzeuge als auch Gegenstände des Lernens sind. An vier Diskursen (Manuals und Lehrbüchern) der Homecomputerzeit wurde gezeigt, wie sich diese Didaktiken historisch (als Hacking) dargestellt und in vier aktuellen Retrocomputing-Projekten unter Berücksichtigung moderner Lernwerkzeuge tradiert haben. In diesen Projekten zeigt sich sowohl eine Aktualisierung historischer Lehr-/Lernprozesse als auch eine Aktualisierung und Operativierung historischer Computerplattformen. In der Present-Progressive-Form der englischen Verben home-comput*ing* und retro-comput*ing* offenbaren sich also unterschiedliche Praktiken und Prozesse des Vergegenwärtigung des Historischen.

6. Schluss

6.1 Zusammenfassung

Das Ziel der vorangegangenen Arbeit war es, mit Hilfe der Computerarchäologie als Theorie und Methode eine neue Perspektive auf Computergeschichte und -geschichtsschreibung zu werfen. Diese sollte nicht allein in einer Geschichtskritik bestehen (die historiografische Ideen, Methoden oder Schulen kritisiert), sondern in einer Neuperspektivierung von Computern als grundsätzlich *ahistorische Objekte*. Nach einer Einführung in dieses Problemfeld im *ersten Kapitel* und der Diskussion bisheriger wissenschaftlicher Arbeiten im *zweiten Kapitel* wurden hierzu im *dritten Kapitel* zunächst systematisch jene Probleme erfasst, die bisherige Computergeschichtsschreibung aufwarf und aufwirft. In Historiografien, die zeit-, sozial- oder mentalitätsgeschichtlich argumentieren, werden Computer zu Historemen einer Kulturgeschichte der Computernutzung, -wirkung, -ästhetik o.ä.; innerhalb von Ingenieurs- und Wirtschaftsgeschichten, sind sie Gradmesser einer technischen und/oder ökonomischen Fortschrittsgeschichte. Die Ergebnisse solcher Historiografien stehen oftmals im unauflöslichen Widerspruch zueinander, weil ihre Argumente das Ergebnis subjektiver Interpretationen darstellen. Diese grundsätzlich konstruktivistische Geschichtsperspektive hinter diesen Zugängen wurde zunächst durch die Theorien Hayden Whites (Geschichte als poetisches Konstrukt) und Robin George Collingwoods (Geschichte als Re-Enactment im Denken des Historikers) hinterfragt, bevor sie diskursanalytisch (machtvolle Dispositive als Apriori historischer Diskurse) mit Michel Foucault und medienarchäologisch (Technik als non-diskursives Apriori historischer Diskurse) mit Wolfgang Ernst dekonstruiert wurde.

Eine Computerarchäologie als *non-historiografische Methode* akzentuiert gegenüber Computergeschichte(n) die Apparate selbst. Dieser Perspektivwechsel erfordert sowohl einen veränderten Fokus auf den Gegenstand (die Computer) als auch ein an diesen Gegenstand angepasstes Beschreibungsinstrumentarium: Das Wesen der Computer liegt im Comput*ing* – also im gegenwärtigen Prozessieren, Speichern und Übertragen von Informationen. Hierzu müssen Hardware, Software und Energie zusammenkommen, was bedeutet, dass Computing nur im operativen Zustand möglich ist und ein Computer daher nur als operatives Medium verstanden werden kann. Erst eine solche Einengung des Computerbegriffs eröffnet die Möglichkeit non-diskursive Methoden zur Beschreibung ‚historischer' Computer einzusetzen: informatische Analysen von Hardware-Architekturen, Algorithmen, Programmiersprachen und Programmcodes, ingenieurs- und naturwissenschaftliche Methoden (Messtechnik, Elektronik, Physik) und formalwissenschaftliche Darstellungen (Mathematik, Diagrammatik, Logik) sowie nicht zuletzt die Demonstration als Selbstausdruck des operativen Computings. Diese Annäherung an die Technologie definiert Computerarchäologie notwendigerweise als *Theorie mittlerer*

Reichweite, bei der wissenschaftliche Aussagen nicht mehr über *den Computer* im Kollektivsingular, sondern *den/einen Computer* als einzelnes Gerät verstanden werden müssen. Dies führt bereits zu einer ersten Öffnung der diskursiven ‚Black Box' Computer als Gegenstand des Wissens.

Der Einsatz computerarchäologischer Verfahren sollte in eine *Archäographie* von Computern münden, die im zentralen *vierten Kapitel* an vier Beispielen durchgeführt wurde. Hierzu wurden vier ‚Arbeitsfelder' des Retro-Computing, also der großteils hobbyistischen Beschäftigung mit ‚alten' (‚nicht mehr lieferbaren') Computern vorgestellt.

An einem Beispiel des *Demo Coding (4.1)* wurde die Implementierung eines physikalischen Prozesses (die Simulation eines springenden Balls) über mehrere Computerplattformen und Programmiersprachen untersucht. Die Simulationstiefe der Implementierungen unterschieden sich dabei ebenso voneinander wie das jeweils eingesetzte Wissen (was sich aus Beschreibungen, Interviews und den Codes selbst ablesen ließ). Um dennoch eine vergleichende Analyse der Implementierungen zu ermöglichen, wurde eine *computerphilologische Methode* vorgeschlagen, bei der Methoden der Sprach- und Literaturwissenschaft zum Einsatz kamen, die geeignet sind, sowohl die synchronen als auch die diachronen Aspekte diverser Implementierungen zueinander ins Verhältnis zu setzen. Die Frage der Lesbarkeit von Code (sowohl für Maschinen als auch für Menschen) bildete den Kulminationspunkt für eine interdisziplinäre Betrachtung von Softwaregeschichte.

Die diskrete Simulation von Prozessen mit Hilfe zellulärer Automaten war Gegenstand einer Archäographie zum *Computerspiel (4.2)*. Auf mehreren Computerplattformen und in unterschiedlichen Programmiersprachen und -verfahren wurden dazu Varianten des Game of Life implementiert. Zelluläre Automaten wurden so zu einem Werkzeug, um historische Computerdiskurse zu vergegenwärtigen (re-enactment), boten sie (in Form des Game of Life) doch die Möglichkeit unterschiedliche Grade maschineller Komplexität bis hin zur Turingmaschine zu simulieren. Hierbei zeigte sich ein erster Unterschied zwischen einer realen Computerplattform (Signetics Instructor 50) und seiner Emulation (WinArcadia), bei der letztere Möglichkeiten zur Verfügung stellte, die dazu genutzt werden konnten, mit dem System (bzw. seiner Emulation) etwas über es selbst zu lernen – was hier bei der Verwendung des im Emulator integrierten Speichermonitors zur Simulation einer Williams-Kilburn-Tube gezeigt wurde. Das Spielen mit zellulären Automaten und mit unterschiedlichen historischen und aktuellen Computer-Lernspielzeugen wurde schließlich als didaktische Methode (*Toy Computing*) in das Forschungsfeld des Unconventional Computing integriert.

Emulation zum Einsatz der *Software Preservation* wurde im dritten Anwendungskapitel (4.3) einer kritischen Reflexion unterzogen. Emulatoren wurden in ihrer Adäquanz als ‚Ersatz' für Computerhardware diskutiert und ihr prinzipielles Defizit (dass sie als symbolische Maschinen die materiellen Aspekte der Computertechnik nicht emulieren können) an Beispielen gezeigt. Die Entwicklung von neuen Spielen für ‚alte' Computer-

plattformen, bei denen Hardwareidiosynkrasien ausgenutzt werden, hatte dieses Problem verdeutlicht. *Retrofitting* als nachträgliche Anpassung ‚veralteter' Hardware an gegenwärtige Anforderungen sollte hierfür einen Ausweg zeigen und betonen, wie wichtig detaillierte technische Kenntnisse von (eben doch nicht) obsoleten Hardwares und Programmiersprachen heute sind. Für die Computerarchäologie liegt die Bedeutung von Emulatoren aber nicht im möglichst authentischen Ersatz von Hardware, sondern in ihren *errors of addition*, also jenen Aspekten, die sie besonders inauthentisch erscheinen lassen: Die Möglichkeit dem emulierten System mehr RAM, modernere Massenspeicher oder höhere Ausführungsgeschwindigkeiten zuzuteilen, lassen Emulatoren zu unzeitgemäßen Tools bei der Erforschung historischer Computermodelle werden.

Das letzte Praxisbeispiel stellte die Hardware Preservation (4.4) am Beispiel der Reparatur eines Sol-20-Homecomputers vor. Dieser wurde im Rahmen eines Retrocomputing-Projektes mit Hacking-Methoden, also Verfahren, die sich von professionellen, museologischen Museumspraktiken unterschieden, durchgeführt. Ziel hierbei war es, aus diesen Praktiken die Methoden der Wissensgewinnung abzuleiten, die sowohl den Reparaturprozess betreffen als auch die Kenntnisse, die über das zu reparierende System erworben werden. Hierfür wurde die Vorgehensweise des Reparateurs im Sinne einer Epistemologie (der Zeug-Analyse Martin Heideggers und der ‚Bastelei-Theorie' Hans-Jörg Rheinbergers) diskutiert. Zwei didaktische Methoden zeigten sich dabei: das experimentelle Trial-and-Error-Verfahren offenbarte sich bei der Reparatur eines Speicher-Fehlers als Form von *E-Learning*, bei dem sich Computer und Reparateur in einem wechselseitigen ‚Wissensaustausch' befinden; das Vorgehen des nicht-professionellen Reparateurs zeigte Aspekte einer spezifischen *Autodidaktik*, die auf ein technisches wie historisches Wissen-Wollen abzielt.

Dieses letzte Phänomen sowie die didaktischen Aspekte der drei anderen Experimente wurden im *fünften Kapitel* eingehender analysiert. Hobbyistisches Retrocomputing lässt sich stets als Wissenspraxis verstehen, bei der Computergeschichte auf Basis mannigfaltiger Quellen kritisch reflektiert und vermittels Hardware- und Softwareprojekten vergegenwärtigt (re-enacted) wird. Dies geschah und geschieht zumeist *autodidaktisch*. Nachdem zunächst die diskurs- und medienepistemologischen Aspekte des Verhältnisses von Wissen und Macht vorgestellt wurden, konnten Methoden der Computer-Autodidaktik beschrieben werden, die an diesem Spannungsfeld ansetzen: *Learning by doing, Hands-on-Imperative* durch *Trial and Error* (nichtprofessionelle Experimentalsituation), *Gamification* (kompetitive Verfahren der Wissensgewinnung) und *E-Learning* (dialogisches Lernen zwischen Mensch und Computer) wurden als zentrale Verfahren solchen Selbstlernens gekennzeichnet. Diese Vorgehensweisen zeigten sich bereits am Beginn privater Computernutzung, Mitte der 1970er-Jahre. Beispiele des Homecomputing (durch Analysen historischer Lern/Lehrbücher) konnten dies zeigen. Retrocomputing als Wissenspraxis hat dem gegenüber lediglich eine quantitative Erweiterung (zusätzliche, neue Quellen, Methoden

und Werkzeuge) nicht jedoch eine qualitative Veränderung erfahren. Die im vierten Kapitel vorgestellten Archäographien haben diese Selbstlerndidaktik zum Einsatz gebracht, indem sie im Rahmen von Universitätskursen im Fach Medienwissenschaft auf Basis konstruktivistischer Lerntheorie Methoden des Blended Learning einsetzten.

6.2 Möglichkeiten und Grenzen

Ein Anliegen der Arbeit war es, Computerarchäologie als *interdisziplinäre Theorie* mit einem an den Untersuchungsgegenstand angepassten Methodenset vorzustellen. Zentral war es hierbei eine Doppelperspektive einzunehmen, die Computer aus informatischer und medienwissenschaftlicher Sicht analysiert. Hieraus sollten beide Disziplinen Nutzen ziehen, indem sie die Fragestellungen und Antwortmöglichkeiten des jeweils anderen Faches in ihren eigenen Forschungs- und Lehransätzen berücksichtigen. Dies soll an dieser Stelle noch einmal in konziser Weise formuliert und dabei die möglichen Schwierigkeiten genannt werden, die sich sowohl aus fachlicher Sicht ergeben als auch grundsätzlich aus der Computerarchäologie hervorgehen.

6.2.1 Interdisziplinarität

Informatik aspektiert die Geschichte der Computer und ihrer eigenen Disziplin bereits im Rahmen von Lehrveranstaltungen des Fachgebiets *Informatik und Gesellschaft*. Das Wissen um historische Aspekte ist darüber hinaus integraler Bestandteil der Fachgebiete Technische Informatik, Praktische Informatik, Theoretische Informatik und anderer, denn das kontemporäre Wissen dieser Bereiche fußt notwendigerweise auf der historischen Entwicklung der Technologien: Die Von-Neumann-Architektur, die Boole'sche Logik, der Dijkstra-Algorithmus, die Turing-Maschine usw. tragen neben ihrer ‚zeitlosen' Gültigkeit als Grundlagenkonzepte bereits in ihren Namen historische Bezüge. Die Personennamen (von Neumann, Boole, Turing etc.) sind allerdings weniger als Biographeme denn als Statthalter von Ideen und Epistemen zu verstehen, für die sich eine epistemologische Medienwissenschaft interessiert. Hier zeigen sich bereits Überschneidungspunkte für eine interdisziplinäre Diskussion.

Diese Diskussion bedarf allerdings einer Vorbereitung von beiden Seiten aus: Eine an der Technik der Medien interessierte Medienwissenschaft benötigt medientechnisches Wissen als Grundlage – nicht nur, um die Fachbeiträge aus der Informatik, Elektrotechnik, Physik usw. überhaupt rezipieren zu können, sondern auch, um sich Medien aus einer nicht-hermeneutischen Perspektive annähern zu können. Dies müsste innerhalb der Curricula berücksichtigt werden, wobei gleichzeitig bedacht werden sollte, dass ein Fachstudium der betreffenden technischen Disziplinen wünschenswert wäre (etwa als Zweit- oder Nebenfach neben der Medienwissenschaft), was aber weder immer möglich

noch grundsätzlich notwendig ist. Im Vorangegangenen wurde implizit eine Form der didaktische Reduktion vorgestellt und zur Anwendung gebracht, die hierfür Abhilfe schaffen könnte: *Retro-Didaktik.*

6.2.2 Retro-Didaktik

Auf der Basis einfacherer Systeme (Hardware-Architekturen, Betriebssysteme, Programmiersprachen, Schaltungskomplexität usw.) lassen sich *im Prinzip* viele notwendige Grundlagen der Fachdisziplinen vermitteln. Die Tatsache, dass moderne wie obsolet gewordene Computerplattformen nach dem selbem Prinzip der Von-Neumann-Maschine arbeiten, ermöglicht es zudem auch aktuelle Probleme und Fragestellungen auf Computern geringerer Komplexität (wie beispielsweise 8-Bit-Homecomputern) zu implementieren.[183] Von diesem Prinzip haben die Projekte des vierten Kapitels in dieser Arbeit Gebrauch gemacht. Eine Retro-Didaktik wäre darüber hinaus aber für unterschiedliche Disziplinen und ihre fachspezifischen Fragestellungen (vor dem Hintergrund der Digital Humanities) auszuformulieren.[184]

Im Gegenzug könnte die Einbringung medienarchäologischer Themen in die Fachgebiete der Informatik zu einem größeres ‚historisches Selbstbewusstsein' der eigenen Disziplin beitragen. Dies erscheint vor allem vor dem Hintergrund eines immer schneller voranschreitenden Wechsels von Hardware, Software, Datenformaten, Standards, Verfahrensweisen usw. sinnvoll zu sein, um jüngst oder schon länger obsolet gewordene Technologien zum Zwecke der Sicherheit, Kompatibilität und/oder Nachhaltigkeit zu aktualisieren. Im Zusammenhang mit der Diskussion des *Retrofitting* (vgl. Kapitel 4.3.5) wurde die Notwendigkeit dessen bereits angesprochen. Sie zeigt sich aber auch im Rahmen der Informatik-Didaktik für alle Bereiche, wenn es um die bereits erwähnte didaktische Reduktion und historische Fundierung von Wissen geht.

Die Grenzen einer solchen Didaktik zeigen sich selbstverständlich an technologischen Umbrüchen, die gänzlich neuartige Apparate und Verfahren einführen (wie aktuell mit dem Quantencomputing zu erwarten). Hier wäre eher zu prüfen inwiefern ein epistemologischer ‚Rückschritt' hinter die Computertechnologie selbst aktuelles Grundlagenwissen zu vermitteln erlaubt (wenn etwa in Quantencomputern immer noch logische Gatter auf Basis Boole'scher Algebra implementiert werden). Hier zeigen diverse Projekte des *Unconventional Computing* bereits mögliche Reduktionen – zum Beispiel die Implementierung massiv-parallel rechnender Systeme mit Hilfe zellulärer Automaten, logischer Schaltnetze mit Billard-Ball-Computern und ähnliches.

183 So existieren beispielsweise Implementierungen für lernfähige neuronale Netze für den Commodore 64 in BASIC: https://gururise.github.io/neural-network-on-a-c64/ bzw. https://www.fourmilab.ch/documents/commodore/BrainSim/ [beide letzter Abruf: 15.07.2019].

184 Für die Medienwissenschaft geschieht dies in der Lehrbuch-Reihe „Medientechnisches Wissen" [Höltgen 2017a; 2018b].

6.2.3 Reichweite

Die (mittlere) Reichweite der Computerarchäologie als Theorie hilft bei der Akzentuierung konkreter Systeme als Gegenstände des Forschens und Wissens – auch in Hinblick auf die Formulierung falsifizierbarer Aussagen über diese. Dass sich hieraus keine grundsätzlich auf alle Computer übertragbaren und für alle Zeiten verallgemeinerbaren Erkenntnisse produzieren lassen, wurde im Vorangegangenen als eine Möglichkeit dargestellt, dem ‚Black Boxing' der Computertechnologie entgegenzutreten: Anstelle einer Technik-Metaphorik sollte die konkrete Analyse von Apparaten und Prozessen treten. In letzter Konsequenz ließe dies lediglich wissenschaftliche Aussagen zu einem Gerät/Prozess zu einem spezifischen Zeitpunkt zu, was einem naturwissenschaftlichen Experimentalsetting entspräche, eine kulturwissenschaftliche Theorien des (sogenannten) Computers jedoch erschweren oder gar verunmöglichen würde. Allerdings wäre der Ephemerität von Medienprozessen damit Rechnung getragen; die Übertragbarkeit von Forschungsergebnissen auf andere Geräte/Prozesse wäre über den ‚Umweg' des wiederholten Experiments möglich, bei dem die jeweils aktuellen Voraussetzungen mit in die Auswertung einfließen müssten. Insbesondere die Demonstration als Methode und Selbstausdruck des Mediums bildet hierbei einen kritischen Gradmesser zur Validierung von Erkenntnissen.

6.2.4 Beschreibungs(in)kompetenz

Neben natur-, formal- und ingenieurwissenschaftlichen Methoden wurden in der Arbeit auch neue Methoden der Beschreibung von Computerprozessen vorgeschlagen. Diese könnten dabei helfen zeitbasierte Prozesse, die sich allein operativ *erfahren* ließen, *beschreibbar* zu machen. Neben der operativen Diagrammatik, die bereits von C. S. Peirce als ein solches Verfahren etabliert wurde (und die sich längst in Entwurfs- und Darstellungswerkzeugen der Informatik findet), wurde in Kapitel 4.1 versucht sprach- und literaturwissenschaftliche Methoden zur synchronen und diachronen Analyse von Codes einzusetzen. Dieses Verfahren ist bereits auf Grenzen gestoßen, als damit versucht wurde das Prinzip-Schaltdiagramm eines Analogcomputerspiels mit symbolischen Sourcecodes zu vergleichen (vgl. Kapitel 4.1.4.1).

Diese Übersetzungslücke zwischen ikonischem und symbolischen Beschreibungen nahm das im Kapitel 4.3 diskutierte Dilemma der Software-Emulation von Hardware bereits vorweg und führte die Konsequenz einer mittel-weitreichenden Computerarchäologie noch einmal vor Augen: Am Beispiel der entropisch erzeugten Zufallszahlen in SRAM-Bausteinen im Vergleich zu den algorithmisch erzeugten LFBS-Zufallszahlen im Emulator zeigt sich die Unwiederholbarkeit eines physikalischen Computerprozesses deutlich.

Diese Unwiederholbarkeit offenbart sich nicht nur auf den Unterflächen, sondern auch an den Schnittstellen der Systeme – etwa in Hinblick auf die Interaktivität zwischen Computer und Mensch, ermöglicht durch die Interrupt-Technologie: „the interrupt

mechanism turned the computer into a nondeterministic machine with a nonreproducible behavior“ [Dijkstra 2000]. Eine allein auf die apparativen Prozesse konzentrierte Betrachtung von Computern, die den „User als Gestalt der Anschlüsse“ [Pias 2005a] nicht berücksichtigt, bliebe ohnehin unvollständig. Hier können informatische Methoden der Human-Computer-Interaction [vgl. Kammersgaard 1989] angesetzt werden, um mit dem menschlichen Faktor (auch im Hinblick auf das kybernetische Mensch-Maschine-Feedback-System) ‚rechnen zu können‘.

Versuche, den prinzipiellen Indeterminismus, der die theoretischen Turing-Maschinen und Von-Neumann-Architekturen von operativen Computern unterscheidet, schriftlich zu fixieren, erscheinen ebenso defizitär. Dort, wo es jedoch lediglich um die bereits kodifizierten Algorithmen geht und das symbolische Zeichensystem nicht verlassen wird, können computerphilologische Analysen genutzt werden, um in eine synchrone wie diachrone Code-Text-Analyse einzumünden und die Lesbarkeit und Vergleichbarkeit von Kodierungsverfahren zu unterstützen.

6.2.5 Software/Hardware Preservation

Der Einsatz von Original-Hardware zur Ausführung von Original-Software und zur Nutzung von Original-Daten stellt dasjenige Problem dar, an dem die größten Berührungsflächen zwischen den hobbyistischen Retrocomputing-Szenen und professionellen Anforderungen (etwa zur Musealisierung von Informationstechnologie) bestehen. Zwei aktuelle Tendenzen zeigen sich hier: Die Nutzung von Software-Emulatoren zur Kompensation der Originalhardware (die entweder defekt ist, nicht mehr vorhanden ist oder vor weiterem Verschleiß bewahrt werden soll) und der Einsatz von Originalhardware, die gegebenenfalls für diese Zwecke ‚fit‘ gemacht werden muss, indem defekte oder empfindliche Systembestandteile durch neue/robustere ausgetauscht werden.

Wo das erste Verfahren an die erwähnten prinzipbedingten Schranken der Emulation stößt und zudem eine historiografische oder museologische Agenda authentischer Geschichtsdarstellung stört, da sieht sich die zweite Agenda mit dem Theseus-Paradoxon konfrontiert: Wie viele Teile eines Computers kann man reparieren/austauschen, bis es sich nicht mehr um denselben Computer handelt? Angesichts des unaufhaltsamen Verfalls der materiellen Bestandteile realer Systeme scheint die Hardware Preservation allerdings auf verlorenem Posten zu stehen. Ihr Insistieren auf eine dauerhafte Erhaltung des Realen (der Computer, Peripherien, Datenträger) erscheint vor diesem Hintergrund fragwürdig. An dieser Stelle müsste die Forderung nach ‚neuer alter Hardware‘ gestellt werden: Die Retrocomputing-Szenen sammeln seit einiger Zeit nicht mehr nur *historische* technische Dokumente, sondern erzeugen auch *neue* – etwa in Form reverse engineerter historischer ICs. So existieren bereits Javascript-Versionen von 8-Bit-CPUs und anderer historischer Bausteine[185]: „[...] we don‘t need actual 6502 chips to drive old hardware or

185 Vgl. http://www.visual6502.org/ [letzter Abruf: 15.07.2019].

to study how old hardware works [...] We‘re not crippled by the fact that the original 6502 is no longer made.“ [Greg James zit. n. Swaminathan 2011]

6.3 Ausblick

Die vorliegende Arbeit versteht sich als Eingrenzung eines interdisziplinären Forschungsfeldes zwischen den Disziplinen Informatik, Medienwissenschaft, Linguistik, Geschichtswissenschaft, Museumskunde und -pädagogik sowie angrenzender Arbeitsbereiche. Frühe Mikrocomputer der 1970er und 1980er Jahre bilden ‚die Medien‘ zwischen diesen Arbeitsfeldern, denn an ihnen konnte gezeigt werden, worin gemeinsame wissenschaftliche Interessen dieser Disziplinen bestehen könnten.

Der reichhaltigen Kultur des Retrocomputing konnte diese Arbeit allerdings kaum vollständig gerecht werden. Es existieren zahlreiche weitere Betätigungsfelder, auf denen Hobbyisten im autodidaktischen Verfahren seit Jahrzehnten Computergeschichte(n) lernen, operativieren und aktualisieren. Zudem hat die Diskussion des Forschungsstandes (der fortlaufend um neue Beiträge bereichert wird) gezeigt, dass etliche Disziplinen ein vielfältiges Interesse an diesen Themen besitzen. Hiervon konnte nur weniges in der Arbeit berücksichtigt werden. Insbesondere zwei Themengebiete scheinen für eine computerarchäologische Forschung besonders ergiebige Quellen zu liefern: die Programmiersprachen und die Computerspiele. Beide Themen sind im Verlauf der Arbeit auf unterschiedliche Weise diskutiert worden und haben gezeigt, dass sie hoch-komplexe Theorie-, Technik- und Kulturgeschichten besitzen, die es computerarchäologisch zu dekonstruieren gälte.

Die computerarchäologische Erforschung der Programmiersprachen, insbesondere jener, die eine Rolle bei und während der Popularisierung der Computertechnik gespielt haben (8-Bit-Assembler, BASIC, Pascal, C, Forth u.a.), könnte einen tiefen (unterflächlichen) Einblick in vergangene und gegenwärtige Computerkulturen liefern. Die enorme Menge an Sekundärliteratur und grauer Literatur (die von Szenezeitschriften bis hin zu handgeschriebenen Programmen reicht) stellt eine noch unerforschte weil bislang nicht systematisch erfasste Quelle dar, die ebenso wie die Hardware und Software vom Verfall bedroht ist und mit adäquaten Bewahrungsstrategien gesichert werden müsste.

Computerspiele als im aspektierten Zeitraum für Homecomputer wichtigste Software-Gattung liefern eine ebenso reichhaltige und ebenso bedrohte Quellengattung. Ihre systematische und historische Erforschung in den Game Studies [Beil/Hensel/Rauscher 2018 sowie dazu: Höltgen 2019b] könnte durch eine techniknahe ‚Game Science‘ ergänzt werden, die im Sinne der Computerarchäologie die Unterflächen mit den Oberflächen der Spiele als zusammenhängend analysiert, und dabei Schaltungen, Codes und Apparate mit einbezieht, indem hierfür formale Beschreibungssysteme (Spieltheorie, Diagrammatik, Re-Enactments, ...) nutzt. So könnten auch Phänomene, die bislang kaum in den Game Studies erörtert wurden (BASIC-Abtipp-Spiele, komputierende Spielzeuge, Hacking und

Misusing von Computerspielen etc.), als Gamification in dem hier verstandenen Sinn eines spielerischen Wissenserwerbs Berücksichtigung finden.

Apparat

Bibliografie

- Ahl, David H. (1982): BASIC Computer Spiele. Band 1. Microcomputer Edition. Düsseldorf: Sybex.
- Aaronson, Scott (o.J.): The Power of the Digi-Comp II/ My First Conscious Paperlet. In: https://www.scottaaronson.com/blog/?p=1902 [letzter Abruf: 22.01.2018].
- Adamatzky, A./Teuscher, Chr. (2006): Editorial. In: Dies. (Hgg.): From Utopian to Genuine Unconventional Computers. o. O.: Luniver, S. V-VII.
- Adamatzky, Andrew (2002a)(Hg): Collision-Based Computing. London u.a.: Springer.
- Adamatzky, Andrew (2002b): Preface. In: [Adamatzky 2002a], S. III-XIV.
- Adamatzky, Andrew (2002c): New Media for Collision-Based Computing. In: [Adamatzky 2002a], S. 411-442.
- Adamson, I./Kennedy, R. (1984): The Quantum Leap – to where? In: Personal Computer World, Juni 1984. Online: http://rk.nvg.ntnu.no/sinclair/computers/ql/ql_sst.htm [letzter Abruf: 07.02.2018].
- AEG/Telefunken (o.J.a): Demonstrationsbeispiel Nr. 5: Ball im Kasten. In: AEG/Telefunken (Hg.): Analog- und Hybridrechner Lehrgangshandbuch. Kapitel 11: Programmierhilfen, Anwendungsbeispiele und Demonstrationsbeispiele, S. 1-8.
- AEG/Telefunken (o.J.b): Datenverarbeitung Informationsblatt Transistorisierter Tischanalogrechner RE 742. In: http://fafner.dyndns.org/~vaxman/my_machines/telefunken/ra742/telefunken_informationsblatt_ra742.pdf [letzter Abruf: 07.07.2016].
- Agar, Jon (1998): Introduction: History of Computing: Approaches, New Directions and the Possibility of Informatic History. In: History and Technology, Vol. 15, No. 1-2, S. 1-5.
- Agrifoglio, Rocco (2015): Knowledge Preservation Through Community of Practice Theoretical Issues and Empirical Evidence. (SpringerBriefs in Information Systems) Cham u.a.: Springer.
- Alberts, Gerard/Oldenziel, Ruth (Hgg.) (2014): Hacking Europe. From Computer Cultures to Demo Scene. London u. a.: Springer.
- Alcock, Donald (1977): Illustrating BASIC (A Simple Programming Language). Cambridge u.a.: Cambridge University Press.
- Antoniol, G./Ayari, K./Dipenta, M.(2008): Is it a Bug or an Enhancement? A Text-based Approach to Classify Change Requests. In: CASCON '08 Proceedings of the 2008 conference of the center for advanced studies on collaborative research: meeting of

minds, Article No. 23, https://dl.acm.org/citation.cfm?id=1463819 [letzter Abruf: 21.03.2018].

- Anzai, Yuichiro/Simon, Herbert A. (1979): The Theory of Learning by Doing. In: Psychological Review, Vo. 87, No. 2, S. 124-140.
- Arsenault, Dominic (2016): Playing with Super Power: The Super NES. Cambridge/London: MIT Press.
- Ashby, W. Ross (1957): An Introduction to Cybernetics. London: Chapman & Hall Ltd.
- Atice, Nathan (2015): I am Error. The Nintendo Family Computer / Entertainment System Platform. Cambridge/London: MIT Press.
- Atkinson, Paul (1998): Computer Memories: The History of Computer Form. In: History and Technology, 1998, Vol. 15, pp. 89-120.
- Aycock, John (2006): Retrogame Archology. Exploring Old Computer Games. o. O.: Springer.
- Backus, J. W: (1959): The Syntax ans Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference. New York: IBM.
- Bagnall, Brian (2010): Volkscomputer. Aufstieg und Fall des Computer-Pioniers Commodore und die Geburt der PC-Industrie. Utting: Gameplan.
- Balke, F./Garderer, R. (2017): Einleitung. In: Dies. (Hgg.): Medienphilologie. Konturen eines Paradigmas. Göttingen: Wallstein, S. 7-24.
- Banks, C. M. (2009): What Is Modeling and Simulation? In: Dies./Sokolowski, J. A. (Hgg.): Principles of Modeling and Simulation. A Multidisciplinary Approach. Hoboken/New Jersey: Wiley & Son, S. 3-24.
- Baranovska, M. (2018): Maschine spricht Spannung. In: Höltgen, S./Dies. (Hgg.): Hello, I'm ELIZA. 50 Jahre Chatbots. (Computerarchäologie, Band 5). Bochum: projekt, S. 125-142.
- Battle, Jim (2006): Sol-20.org. http://www.sol20.org/ [letzter Abruf: 25.11.2016].
- Bauer, F. L. (2009): Historische Notizen zur Informatik. Berlin/Heidelberg: Springer.
- Baumann, Konrad/Lanz, Herwig (1998): Mensch-Maschine-Schnittstellen elektronischer Geräte. Leitfaden für Design und Schaltungstechnik. Berlin/Heidelberg: Springer.
- Baumann, Rüdiger (1983): Strukturiertes Programmieren mit BASIC. Stuttgart: Klett.
- Becker, Rainer C. (2012): Black Box Computer. Zur Wissensgeschichte einer universellen kybernetischen Maschine. Bielefeld: Transcript.
- Beil, Benjamin (2013): Die Sehnsucht nach dem Pixelklumpen. Retro-Gaming und das popkulturelle Gedächtnis des Computerspiels. In: Kleiner, M. S./Wilke, T. (Hgg.): Performativität und Medialität Populärer Kulturen. Wiesbaden, Springer, S. 319-335.

- Beil, Benjamin/Hensel, Thomas/Rauscher, Andreas (2018): Game Studies. Wiesbaden: Springer VS.
- Bellos, Ales (2014): The Grapes of Math. How Life reflects numbers and numbers reflect life. New York u.a.: Simon & Schuster.
- Berz, Peter (2009): Bitmapped Graphics. In: Vollmar, A. (Hg.): Zeitkritische Medien. Berlin: Kadmos, S. 127-154.
- Beyer, Kurt W. (2009): Grace Hopper and the Invention of the Information Age. Cambridge/London: MIT Press.
- Biancuzzi, Federico/Warden, Shane (2009): Masterminds of Programming. Conversations with the Creators of Major Programming Languages. Beijing u.a.: O'Reilly.
- Bischoff, Manon (2019): Festkörperphysik – Topologische Materialien. Rätselhafte neue Stoffe stellen heute eine Revolution der Halbleiterindustrie in Aussicht. In: Spektrum der Wissenschaft, 2, 2019, S. 50-59.
- Bogost, Ian (2012): Alien Phenomenology. Or what it‘s like to be a Thing. London/Minneapolis: University of Minnesota Press.
- Bogost, Ian (2014): Why Gamification is Bullshit. In: Walz, Steffen P./Deterding, Sebastian (Hgg.): The Gameful World. Approaches, Issues, Application. Boston: MIT, S. 65-80.
- Bogost, Ian/Montfort, Nick (2009a): Platform Studies: Frequently Questioned Answers. In: Proceedings of the Digital Arts and Culture Conference, 12.-15.12.2009. http://pdf.textfiles.com/academics/bogost_montfort_dac_2009.pdf [letzter Abruf: 11.01.2019].
- Bogost, Ian/Montfort, Nick (2009b): Racing the Beam. The Atari Video Computer System. Cambridge/London: MIT Press.
- Bolter, J. David (1990): Der digitale Faust. Philosophie des Computer-Zeitalters. München: Oktogon.
- Bolter, Jay David/Grusin, Richard (1999): Remediation. Understanding New Media. Cambridge/London: MIT Press.
- Bolz, Norbert (1994): Computer als Medium – Einleitung. In: Bolz/KittlerTholen (1994) a. a. O., S. 9-16.
- Bolz, Norbert/Kittler, Friedrich/Tholen, Christoph (Hgg.) (1994): Computer als Medium. München: Fink. [Vorwort: S. 7.]
- Bond, Paul (1986): Blitter. How to mimic a Commodore Amiga. In: Amstrad Action, Nr. 10, July 1986, S. 78.
- Boole, George (2001): Die mathematische Analyse der Logik. Halle: Hallescher Verlag.

- Borges, J. L. (1975): On Exactitude in Science. In: Ders.: A Universal History of Infamy (translated by Norman Thomas de Giovanni), London: Penguin Books, S. 131.
- Borghoff, Uwe M./Krebs, Nico/Röding, Peter (2014): Der Museumsansatz bei der digitalen Langzeitarchivierung in Theorie und Praxis. In: Hollemann, Michael/Schüller-Zwierlein, André (Hgg.): Diachrone Zugänglichkeit als Prozess. Kutlurelle Überlieferung in systematischer Sicht. Berlin/München/Boston: DeGruyter, S. 366-385.
- Borst, Arno (1991): Computus. Zeit und Zahl in der Geschichte Europas. Berlin: Wagenbach.
- Boswell, Paul/Boswell, Alyssa (2017): Turing Tumble Puzzle Book. Kickstarter Edition. O. O.: Turing Tumble LCC.
- Botz, D. (2011): Kunst, Code und Maschine. Die Ästhetik der Computer-Demoszene, Bielefeld: Transcript.
- Bouton, Charles L. (1901): Nim, A Game with a Complete Mathematical Theory. In: The Annals of Mathematics, 2nd Ser., Vol. 3, No. 1/4. (1901 - 1902), S. 35-39.
- Braguinski, Nikita (2014): Circuit Bending. Ein unheimlicher Spaß. In: Retro Nr. 34 (Sommer 2014), S. 38f.
- Braguinski, Nikita (2018): RANDOM. Die Archäologie elektronischer Spielzeuge. (Reihe Computerarchäologie Band 3). Bochum: Projekt.
- Bresenham, J. E: (1965): Algorithm for computer control of a digital plotter. In: IBM Systems Journal, Vol. 4, No. 1, S. 25-30.
- Brückmann/Englisch/Gerits/Stiegers (1985): CPC 664/6128 Intern. Düsseldorf: Data Becker.
- Brückner, Michael (2011): Steinzeit-PCs können hunderttausende Euro bringen. In: Welt Online, 12.11.2011, https://www.welt.de/wirtschaft/webwelt/article13712153/Steinzeit-PCs-koennen-hunderttausende-Euro-bringen.html [letzter Abruf: 24.10.2016].
- Busjahn, Teresa u.a. (2015): Eye movement in code reading: relaxing the linear order. In: Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension, S. 255-265.
- Busjahn, Teresa/Schulte, Carsten (2013): The use of code reading in teaching programming. In: Proceedings of the 13th Koli Calling International Conference on Computing Education Research, S. 3-11.
- Butler, Deborah L./Winne, Philip H. (1995): Feedback and Self-Regulated Learning: A Theoretical Synthesis. In: Review of Educational Research, Fall 1995, Vol 65, No. 3, S. 245-281.
- Caliskan, Aylin u.a. (2018): When Coding Style Survives Compilation: De-anonymizing Programmers from Executable Binaries. In: Network and Distributed

Systems Security (NDSS) Symposium 2018, http://dx.doi.org/10.14722/ndss.2018.23304 [letzter Abruf: 13.03.2018].

- Calude, Cristian S. (2015): Unconventional Computing: A Brief Subjective History. In: Adamatzky, A. (Hg.): Advances in Unconventional Computing. Vol. 1: Theory. Basel: Springer, S. 855-864.
- Camper, Brett (2009): Fake Bit: Imitation and Limitation. In: Proceedings of the Digital Arts and Culture Conference, 2009, University of California. https://escholarship.org/uc/item/3s67474h [letzter Abruf: 09.01.2019].
- Cermak-Sassenrath, Daniel (2010): Interaktivität als Spiel. Neue Perspektiven auf den Alltag mit dem Computern. Bielefeld: transcript.
- Ceruzzi, Paul E. (1998): A History of Modern Computing. Cambridge: MIT Press.
- Ceruzzi, Paul E. (2003a): Eine kleine Geschichte der EDV. (Aus dem Amerikanischen von Prof. Dr. Arne Willner) Bonn: mitp-Verlag.
- Ceruzzi, Paul E. (2003b): A history of modern Computing. Second Edition. Boston: MIT Press.
- Chomsky, Noam (1956): Three Models for the Description of Language. In: IRE Transactions of Information Theory, Vol. 2, No. 3, Sep. 1956, S. 113–124.
- Chomsky, Noam (1957): Syntactic Structures. The Hague/Paris: Mouton Publ.
- Coenen, Jarno/Gross, Sebastian/Pinkwart, Niels (2018): Comparison of Feedback Strategies for Supporting Programming Learning in Integrated Development Environments (IDEs). In: Le, Nguyen-Thinh u.a. (Hgg.): Advanced Computional Methods for Knowledge Engineering. Proceedings of the 5th International Converence on Computer Science, Applied Mathematics and Applications, ICCSAMA 2017. Cham u.a.: Springer, S, 72-83.
- Collingwood, R. G. (1955): Philosophie der Geschichte. Stuttgart: Kohlhammer.
- Collingwood, R. G. (1999a): Philosophy of History. In: Ders.: The Principles of History And Other Writings in Philosophy of History, hg. v. Dray, William H.; Dussen, W. J. van der, Oxford: Oxford Univ. Press, S. 219-234.
- Collingwood, R. G. (1999b): Notes on Historiography. In: Ders.: The Principles of History And Other Writings in Philosophy of History, hg. v. Dray, William H.; Dussen, W. J. van der, Oxford: Oxford Univ. Press, S. 235-249.
- Comicro AG (1978): TRS-80 bekommt „Flügel“. In: Hobbycomputer. Sonderheft der ELO Funkschau Elektronik. Oktober 1978, S. 94.
- Coners, Enno/Benda, Rainer/Zahn, Christian/Kretzinger, Boris (2012): Die Commodore Story. Winnenden: CSW.
- Conway, J. H. (1983): Über Zahlen und Spiele. Braunschweig/Wiesbaden: Vieweg & Sohn.

- Conway, J. H./Berlekamp, E. R./Guy, R. K. (1985a): Was heißt „Leben“? In: Dies.: Gewinnen. Strategien für mathematische Spiele. Band 4: Solitairspiele, S. 123-155.
- Conway, J. H./Berlekamp, E. R./Guy, R. K. (1985b): Grünes Hackenbusch, Nim und Nim-Zahlen In: Dies.: Gewinnen. Strategien für mathematische Spiele. Band 1: Von der Pike auf, S. 42-44.
- Cook, Matthew (2004): Universality in Elementary Cellular Automata. In: Complex Systems, 15/2004, S. 1–40.
- D'Ambros, M./Gall, H. C./Lanza, M./Pinzger, M. (2008): Analyzing software repositories to understand software evolution. In: Mens, T/Demeyer, S. (Hgg.): Software Evolution. Heidelberg, Germany, 37-67.
- Dale, Rodney (1985): The Sinclair Story. London: Duckworth.
- De Baugrande, R.-A./Dressler, W. U. (1981): Einführung in die Textlinguistik. Tübingen: Max Niemeyer Verlag.
- Dean, Christopher/Whitlock, Quentin (1989): A Handbook of Computer Based Learning. Second Edition. New York: Nichols Publishing.
- Deterding, Sebastian/Dixon, Dan/Khaled, Rilla/Nacke, Lennard (2011): From Game Design Elements to Gamefulness: Defining „Gamification“. In: http://dl.acm.org/citation.cfm?id=2181040 [letzter Abruf: 20.04.2016].
- Deutscher Wissenschaftsrat (2014): Bedeutung und Weiterentwicklung von Simulation in der Wissenschaft. Positionspapier. In: https://www.wissenschaftsrat.de/download/archiv/4032-14.pdf [letzter Abruf: 13.08.2018].
- Dijkstra, E. W. (1968): Dijkstra: Go To Statement Considered Harmful. In: Communications of the ACM. 11, 3, 1968, S. 147-148.
- Dijkstra, E. W. (1972): Notes on Structured Programming. In: Dahl, O.-J./Dijkstra, E. W./Hoare, C. A. R. (Hgg.): Structured Programming. A.P.I.C. Studies in Data Processing, No. 8. New York/London: Adacemic Press, S. 1-82.
- Dijkstra, E. W. (1978): On the Foolishness of „Natural Language Programming“. In: Proceedings: Program Construction, International Summer School, July, 26 – August, 06 1978, S. 51-53.
- Dijkstra, E. W. (2000): My recollections of operating system design. In: https://www.cs.utexas.edu/users/EWD/transcriptions/EWD13xx/EWD1303.html [letzter Abruf: 15.07.2019].
- Dolan, S. (2013): mov is Turing-complete. In: https://www.cl.cam.ac.uk/~sd601/papers/mov.pdf [letzter Abruf: 16.08.2018].
- Dotzler, Bernhard (2006): Diskurs und Medium. Zur Archäologie der Computerkultur. München: Fink.

- Dotzler, Bernhard J. (1996): Papiermaschinen. Versuch über Communication & Control in Literatur und Technik. Berlin: Akademie Verlag.
- Dotzler, Bernhard J. (2006): Diskurs und Medium. Zur Archäologie der Computerkultur. München: Fink.
- Douglas, Kimberly (2000): Digital Archiving in the Context of Cultural Change. In: Serials Review v. 26(3) October 2000, S. 55-59.
- Düllo, Thomas/Liebl, Franz (Hgg.) (2005): Cultural Hacking. Kunst des Strategischen Handelns. Wein/New York: Springer.
- Dunietz, Jesse (2015): The Computer Made of Nothing but Plastic and Marbles. The confounding logic of Dr. NIM. In: https://motherboard.vice.com/en_us/article/kbz5ay/the-computer-made-of-nothing-but-plastic-and-marbles [letzter Abruf: 28.08.2018].
- E.S.R. Inc. (1966): How to Play DR. NIM. In: http://www.one-leggedsandpiper.com/Christmas/Presents/Dr-Nim-Manual.pdf [letzter Abruf: 29.08.2018].
- E.S.R. Inc. (o.J.): Digi-Comp II. Mechanical Binary Digital Computer. Instruction Manual. In: http://cdn2.evilmadscience.com/KitInstrux/DCII-manual.pdf [letzter Abruf: 29.80.2018].
- Ebeling, Knut: Das technische Apriori. In: Engell, Lorenz/Siegert, Bernhard/Vogl, Joseph (Hgg.): Kulturgeschichte als Mediengeschichte (oder vice versa?). Weimar: Bauhaus Universität, S. 11-22.
- Efferth, Thomas (2001): Welchen Beitrag kann die Virtuelle Lehre für den Bildungsauftrag der Hochschule leisten? In: http://www.studgen.uni-mainz.de/manuskripte/efferth.pdf [letzter Abruf: 12.04.2016].
- Eilam, Eldad (2005): Reversing: Secrets of Reverse Engineering. Indianapolis: Wiley.
- Eilebrecht, K./Starke, G. (2010): Patterns kompakt. Entwurfsmuster für effektive Software-Entwicklung. Heidelberg: Spektrum Akademischer Verlag.
- Enge, J. (2018): Friedrich Kittler's Digital Legacy – PART I - Challenges, Insights and Problem-Solving Approaches in the Editing of Complex Digital Data Collections. In: In: Digital Humanities Quarterly, 2017 (11.2), http://digitalhumanities.org/dhq/vol/11/2/000307/000307.html [letzter Abruf: 15.03.2018].
- Ernst, Wolfang (2012a): Gleichursprünglichkeit: Zeitwesen und Zeitgegebenheit technischer Medien. Berlin: Kadmos.
- Ernst, Wolfang (2016): Technomathematische Philologie als Beitrag zur Archäologie digitaler Medienkultur. Kritische Lesarten der Medienwissenschaft. In: https://www.musikundmedien.hu-berlin.de/de/medienwissenschaft/medientheori

en/Schriften-zur-medienarchaeologie/pdfs/comp-system-reif.pdf [letzter Abruf: 21.5.2019].

- Ernst, Wolfgang (2001): Medien@rchäologie (Provokation der Mediengeschichte). In: Stanitzek, Georg/Voßkamp, Wilhelm (Hgg.): Schnittstelle: Medien und Kulturwissenschaften. Mediologie Band 1. Köln: DuMont, S. 250-267.
- Ernst, Wolfgang (2003): Im Namen von Geschichte. Sammeln – Speichern – Er/Zählen. Infrastrukturelle Konfigurationen des deutschen Gedächtnisses. München: Fink.
- Ernst, Wolfgang (2004): Das Gesetz des Sagbaren. Foucault und die Medien. In: Gente, Peter (Hg.): Foucault und die Künste. Frankfurt am Main: Suhrkamp, S. 238-259.
- Ernst, Wolfgang (2005/6): Kalter Sinn. Der medienarchäologische Blick, das medienarchäologische Ohr. Vorlesungsmanuskript. In: (https://www.medienwissenschaft.hu-berlin.de/de/medienwissenschaft/medientheorien/downloads/skripte/kalter-sinn.pdf [letzter Abruf: 11.01.2019].
- Ernst, Wolfgang (2012b): Chronopoetik: Zeitweisen und Zeitgaben technischer Medien. Berlin: Kadmos.
- Ernst, Wolfgang (2013): Signale aus der Vergangenheit. Eine kleine Geschichtskritik. München: Fink.
- Ernst, Wolfgang (2018): Humanities of the Digital: Media Philology. In: https://www.musikundmedien.hu-berlin.de/de/medienwissenschaft/medientheorien/ernst-in-english/pdfs/med-philology-budapest.pdf [letzter Abruf: 12.04.2018].
- Ewert, Thorsten (o. J.): Strukturen der Informatik. In: Die Welt der Strukturwissenschaften. http://www.strukturwissen.de/informatik.html [letzter Abruf: 19.11.2015].
- Felzmann, Sebastian (2014): Been there, done that. Mediennostalgie als kreative Praxis zur Schaffung neuer retroider Spiele. In: [Höltgen 2014e:25-40].
- Fernandez-Vara, Clara/Montfort, Nick (2013): Videogame Editions for Play and Study. TROPE-13-02, October 2013, https://nickm.com/trope_tank/TROPE-13-02.pdf [letzter Abruf: 15.03.2019].
- Firebear Team (2015): Play & Code – Lerne zu programmieren durch Spielen. In: https://firebearstudio.com/blog/play-code-lerne-zu-programmieren-durch-spielen.html [letzter Abruf: 12.04.2016].
- Fischer, Othmar (1982): Mikroprozessoren für Anfänger. Wien/München: Oldenbourg.
- Fish, Stanley (1979): Literatur im Leser. Affektive Stilistik. Warning, R. (Hg.): Rezeptionsästhetik. München: Fink (UTB), S. 196-227.

- Flowers, T. H. (1983): The Design of Colossus. In: Annals of the History of Computing, July 1983, S. 239-252. https://www.computer.org/csdl/mags/an/1983/03/man1983030239.pdf [letzter Abruf: 19.03.2018].
- Forneck, Hermann J. (1990): Entwicklungstendenzen und Probleme der Didaktik der Informatik. In: Beiträge zur Didaktik der Informatik, hg. v. Günther Cyranek. Frankfurt am Main: Diesterweg, S. 18-53.
- Foucault, Michel (1976): Mikrophysik der Macht. Berlin: Merve.
- Foucault, Michel (1981): Archäologie des Wissens. Frankfurt am Main: Suhrkamp.
- Foucault, Michel (1992): Was ist Kritik? Berlin: Merve.
- Foucault, Michel (1999): Die Ordnung des Diskurses. In: Ders.: Botschaften der Macht. Reader Diskurs und Medien. München: Deutsche Verlags Anstalt, S. 54-76.
- Foucault, Michel (2001a): Über die Archäologie der Wissenschaften. Antwort auf den Cercle d'épistémologie. In: Ders.: Schriften in vier Bänden. Dits et Ecrits. Band 1: 1954-1969, S. 887-931.
- Foucault, Michel (2001b): Was ist ein Autor? In: Ders.: Schriften in vier Bänden. Dits et Ecrits. Band 1: 1954-1969, S. 1003-1041.
- Foucault, Michel (2003a): Macht und Wissen. In: Ders.: Schriften in vier Bänden. Dits et Ecrits. Band 3: 1976-1979, S. 515-534.
- Foucault, Michel (2003b): Das Spiel des Michel Foucault. In: Ders.: Schriften in vier Bänden. Dits et Ecrits. Band 3: 1976-1979, S. 391-429.
- Foucault, Michel (2008): Dispositive der Macht. Über Sexualität, Wissen und Wahrheit. Berlin: Merve.
- Fredkin, Edward F./Toffoli, Tommaso (2002): Conservative Logic. In: [Adamatzky 2002a], S. 47-82.
- Frei, Phil/Su, Victor/Mikhak, Bakhtiar/Ishii, Hiroshi (2000): curlybot: Designing a New Class of Computational Toys. Ibn: CHI Letters, Volume 2, issue 1, S. 129-136.
- Freiberger, Paul/Swaine, Michael (2000): Fire in the Valley. The Making of the Personal Computer. Second Edition. New York u.a.: McGraw-Hill.
- Freundorfer, Stephan (2009): Als E.T. die Videospielindustrie killte. In: Spiegel Online, 10.03.2009. http://www.spiegel.de/einestages/game-crash-1984-a-948205.html [letzter Abruf: 11.01.2019].
- Fritz, Berthold (2014): Rock. Vom Wiedereinstieg in die 8-Bit-Spielprogrammierung. In: [Höltgen 2014e:211-228].
- Gardner, Martin (1970): The fantastic combinations of John Conway's new solitaire game ‚life'. In: Scientific American 223 (October 1970): S. 120-123.

- Gardner, Martin (1997): Nim and Tac-Tix. In: Ders.: Hexaflexagons and other mathematical Diversions. The First Scientific American Book of Puzzles & Games. Chicago/London: Univ. of Chicago Press, S. 151-162.
- Gazzard, Alison (2016): Now the Chips are Down: The BBC Micro. Cambridge/London: MIT Press.
- Genette, Gerard (1993): Palimpseste. Die Literatur auf zweiter Stufe. Frankfurt am Main: Suhrkamp.
- Genette, Gerard (2001): Paratexte. Das Buch vom Beiwerk des Buches. Frankfurt am Main: Suhrkamp.
- Gerhardt, M./Schuster, H. (1995): Das digitale Universum. Zelluläre Automaten als Modelle der Natur. Wiesbaden: Vieweg.
- Gfeller, Johannes (2013): Zur Ausbildung im Fach Medienkonservierung. In: Serexhe, Bernhard (Hg.): digital art conservation. Konservierung digitaler Kunst: Theorie und Praxis. Wien: Ambra, S. 601-618.
- Gillies, Constantin (2014): Extraleben. reverse engineert. In: [Höltgen 2014e:175-188].
- Glagla, Joseph/Feiler, Dieter (1984): Mein Heimcomputer selbstgebaut zum Lernen, Spielen, Messen, Steuern, Regeln, ... Ravensburg: Otto Meier Verlag.
- Godden, B. (1984): CPC464 Firmware. ROM Routines and Explanations. Brentwood: AMSOFT/Amstrad.
- Godfrey, J. T. (1968): Binary Digital Computer (Patent). In: https://patents.google.com/patent/US3390471 [letzter Abruf: 29.08.2018].
- Goldberg, Marty/Vendel, Curtis (2012): Atari Inc. Business Is Fun. Carmel: Syzygy Company Press.
- Göpfert, Rebekka (1996): Oral History. Über die Zusammensetzung individueller Erinnerung im Interview. In: Clemens Wischermann (Hg.): Die Legitimierung der Erinnerung und die Geschichtswissenschaft. Stuttgart: Franz-Steiner-Verlag, S. 101-111.
- Gosper, R. W. (1984): Exploiting Regularities In Large Cellular Spaces. In: Physica 10D (1984), S. 75-80.
- Grams, Timm (1990): Denkfallen und Programmierfehler. Berlin u.a.: Springer.
- Grechenig, Thomas/Bernhard, Mario/Breiteneder, Roland/Kappel, Karin (2010): Softwaretechnik. Mit Fallbeispielen aus realen Entwicklungsprojekten. München: Pearson.
- Griffith, A. (2002): GCC. The Complete Reference. O. O.: McGraw-Hill/Osborne.
- Groh, Fabian (2012): Gamification: State of the Art Definition and Utilization. In: Asaj, Naim u.a. (Hgg.): Proceedings of the 4th Seminar on Research Trends in Media

Informatics, Institute of Media Informatics Ulm University, 14th February 2012, S. 39-46. Zit. n. http://d-nb.info/1020022604/34/#page=39 [letzter Abruf: 20.04.2016].

- Gross, Sebastian/Pinkwart, Niels (2015): How Do Learners Behave in Help-Seeking Wehn Given a Choice? In: Conati, C./Heffernan, N./Mitrovic, A./Verdejo, M. F. (Hgg.): Artificial Intelligence in Education. 17Th International Conference, AIED 2015 Madrid, Spain, June 22-26, 2015 Proceedings. Cham u.a.: Springer, S. 600-603.
- Gross, Sebastian/Mobkel, Bassam/Hammer, Barbara/Pinkwart, Niels (): Learning Feedback in Intelligent Tutoring Systems. Report of the FIT Project, Conducted from December 2011 to March 2015. In: KI - Künstliche Intelligenz 29 (4): S. 413-418.
- Guffey, Elizabeth E. (2006): Retro. The Culture of Revival. London: Reaktion Books.
- Günthner, Willibald/Klenk, Eva/Tenerowicz-Wirth, Peter (2017): Adaptive Logistiksysteme als Wegbereiter der Industrie 4.0. In: Vogel-Heuser, Birgit/Bauernhansl, Thomas/ten Hompel, Michael (Hgg.): Handbuch Industrie 4.0, Bd. 4; Allgemeine Grundlagen, 2. Auflage. Heidelberg u.a.: Springer Vieweg, S. 97-124.
- Gupta, R. C. (1967): Bhaskara I‘s Approximation To Sine. In: Indian Journal of HIstory of Science, Vol 2, No. 2, S. 122-136.
- Guri, Mordechai/Zadov, Boris/Bykhovsky, Dima/Elovici, Yuval (2018): PowerHammer: Exfiltrating Data from Air-Gapped Computers through Power Lines. In: https://arxiv.org/pdf/1804.04014.pdf [letzter Abruf: 12.04.2019].
- Gutzer, Hannes (1987): Spiel + Spass mit dem Computern. BASIC-Programme lustig und lehrreich. Leipzig/Jena/Berlin: Urania.
- Hagen, Wolfgang (1997): Der Stil der Sourcen. Anmerkungen zur Theorie und Geschichte der Programmiersprachen. In: Warnke, Martin/Coy, Wolfgang/Tholen, Georg Christoph (Hgg.): HyperKult. Geschichte, Theorie und Kontext digitaler Medien. Basel: Stroemfeld, S: 33-68.
- Hagen, Wolfgang (2004): Die Camouflage der Kybernetik. In: Pias, Claus (2004): Cybernetics - Kybernetik. The Macy-Conferences 1946-1953. Band II: Essays & Dokumente.Zürich/Berlin: diaphanes, S. 191-208.
- Hammerman, R./Russell, A. R.: Ada's Legacy: Cultures of Computing from the Victorian to the Digital Age. New York: Morgan & Claypool Publishers-ACM.
- Harenberg, Werner (1984): Alarm in den Schulen: Die Computer kommen. In: Der Spiegel Nr 47, (19. November 1984), S. 97-129.
- Hartmann, Doreen (2017): Digital Art Natives. Praktiken, Artefakte und Strukturen der Computer-Demoscene. Berlin: Kadmos.
- Heath Company (1959): Operational manual for the Heath educational analog computer model EC-1 (1959). In:

http://www.computerhistory.org/collections/catalog/102649920 [letzter Abruf: 05.07.2016].

- Heidegger, Martin (2006): Sein und Zeit. Tübingen: Niemeier.
- Heilige, H. D. (2004): Sichtweisen der Informatikgeschichte: Eine Einführung. In: Ders. (Hg). a. a. O., S. 1-28.
- Heilige, H. D. (Hg.) (2004): Geschichten der Informatik. Visionen, Paradigmen, Leitmotive. Berlin/Heidelberg: Springer.
- Heilige, H. D. (o. J.): Die Fachgruppe InfoHist. Informatik- und Computergeschichte. In: http://www.informatik-geschichte.de/ [letzter Abruf: 11.01.2019]
- Heineman, David S. (2014): Public Memory and Gamer Identity: Retrogaming as Nostalgia. In: Journal of Games Criticism, Vol. 1, Issue 1, S. 1-24.
- Hering, Ekbert (1992) Software-Engineering. Braunschweig: Vieweg.
- Herrmann, Debra S. (1999): Software Safety and Reliability. Techniques, Approaches, and Standards of Key Industrial Sectors. Los Alamos: IEEE Computer Society Press.
- Hertzfeld, Andy (2004): Revolution in the Valley. New York u.a.: O'Reilly.
- Hilf, W. & Nausch, A. (1984): M6800-Familie. Teil 1: Grundlagen und Architektur. München: te-wi.
- Hobi, Viktor (1988): Kurze Einführung in die Grundlagen der Gedächtnispsychologie. In: von Ungern-Sternberg, Jürgen/Reinau, Hansjörg (Hgg.): Vergangenheit in mündlicher Überlieferung. Stuttgart: Teubner, S. 9-33.
- Hoelzer, Helmut (1994): 50 Jahre Analogcomputer. In: Bolz/KittlerTholen (1994) a. a. O., S. 69-90.
- Hoffmann, Dirk W. (2007): Grundlagen der Technischen Informatik. München: Hanser.
- Holcomb, Daniel E. (2009): Power-Up SRAM State as an Identifying Fingerprint and Source of True Random Numbers. In: IEEE Transactions on Computers, Vol. 58, No. 9, September 2009, S. 1198-1210.
- Holenstein, Elmar (1992): Einführung: Semiotica universalis. In: Jakobson, Roman (1992): Semiotik. Ausgewählte Texte 1919-1982. Hg. v. Elmar Holenstein. Frankfurt am Main: Suhrkamp, S. 9-40.
- Holl, Susanne (2017): Friedrich Kittler's Digital Legacy – PART II - Friedrich Kittler and the Digital Humanities: Forerunner, Godfather, Object of Research. An Indexer Model Research. In: Digital Humanities Quarterly, 2017 (11.2), http://digitalhumanities.org/dhq/vol/11/2/000308/000308.html [letzter Abruf: 15.03.2018].
- Holl, Ute (2015): Medientheorie (oder, und, trotz) Kulturtechnikforschung. In: Texte zur Kunst. Juni 2015, 25. Jg., Heft 98, S. 80-87.

- Hollywood, California, USA — June 14 - 16, 2006
- Höltgen, S./Coners, E./Eddiks, T./Lange, A./Paul, A. (2014): Hardwhere? Softwhere? Eine Podiumsdiskussion über die (Un)Möglichkeiten ihrer Musealisierung. In: [Höltgen 2014e:261-299].
- Höltgen, S./Maibaum, J./Rech, M. (2012): Tennis for Two. Ein Analogcomputerspiel aus der Medienwissenschaft. Projektbericht, Anleitungen, Schaltungen, Skizzen und Scribbles. In: https://www.musikundmedien.hu-berlin.de/de/medienwissenschaft/medientheorien/signallabor/praxisarbeiten/T43.pdf/view [letzter Abruf: 21.03.2018].
- Höltgen, Stefan (2003): Simulation, Simulationstheorie, Simulakrum. In: Wulff, H.-J./ Bender, T. (Hgg.): Lexikon der Filmbegriffe. Mainz: Bender. http://www.bender-verlag.de/lexikon/index.php [letzter Abruf: 14.08.2018].
- Höltgen, Stefan (2009): Schnittstellen. Die Konstruktion fon Authentizität im Serienmörderfilm. Inaugural-Dissertation. Bonn 2009, http://txt3.de/schnittstellen [letzter Abruf: 15.03.2019]
- Höltgen, Stefan (2010): Simulation/Simulationstheorie. In: Ders./Baum, P. (Hgg.): Lexikon der Postmoderne. Bochum: Projektverlag, S. 163f.
- Höltgen, Stefan (2012): Data, Dating, Datamining. Der Computer als Medium zwischen Mann und Frau – innerhalb und außerhalb von Fiktionen. In: Bukow, Gerhard Chr./Fromme, Johannes/Jörissen, Benjamin (Hgg.): Raum, Zeit, Medienbildung. Untersuchungen zu medialen Veränderungen unseres Verhältnisses zu Raum und Zeit. Wiesbaden: Springer VS, S. 265-294.
- Höltgen, Stefan (2013a): Game Circuits. Platform Studies und Medienarchäologie als Methoden zur Erforschung von Computerspielen. In: Benjamin Bigl & Sebastian Stoppe (Hgg.): Playing with Virtuality. Theories and Methods of Computer Game Studies. Frankfurt am Main: Peter Lang, S. 83-100.
- Höltgen, Stefan (2013c): Die Notwendigkeit der hardwarenahen Programmierung. Informatikunterricht mit dem 2650 – Ein Interview mit Reinhard Brandt. In: Retro Nr. 29 (Winter 2013/14), S. 48f.
- Höltgen, Stefan (2013d): die CPC-Hardware als Software-Experiment. Interview mit Markus Hohmann, dem Entwickler von JavaCPC. In: Retro Nr. 28 (Herbst 2013), S. 66-68.
- Höltgen, Stefan (2014a): Die NOPs und HALTs digitaler Medien. Programmierlehre maschinennaher Sprachen für Medienwissenschaftler. In: Grundlagenstudien aus Kybernetik und Geisteswissenschaft, Band 55, Heft 4, Dezember 2014, S. 139-153.
- Höltgen, Stefan (2014b): „All Watched Over by Machines of Loving Grace“. Öffentliche Erinnerung, demokratische Information und restriktive Technologien am Beispiel der „Community Memory“. In: Ramón Reichert (Hg.): Big Data. Analysen

zum digitalen Wandel von Wissen, Macht und Ökonomie. Bielefeld: transcript, S. 386-404.

- Höltgen, Stefan (2014c): Sprachregeln und Spielregeln. Von Computerspielen und ihren Programmierfehlern. In: Huberts, Christian/Standke, Sebastian (Hgg.): Zwischen|Welten. Atmosphären im Computerspiel. Glückstadt: vwh, S: 295-316.
- Höltgen, Stefan (2014d): „'t's more fun to compute!' - Theoretische und operative Begriffsbestimmung von 'Computerarchäologie'". Vortrag/Performance im Rahmen des Forschungskolloquiums „Medien, die wir meinen", gehalten am 09.07.2014. In: https://www.medienwissenschaft.hu-berlin.de/de/medienwissenschaft/medientheorien/kolloquium1/stefan-hoeltgen-2014-its-more-fun-to-compute [letzter Abruf: 11.01.2019].
- Höltgen, Stefan (2014e) (Hg.): SHIFT – RESTORE – ESCAPE. Retrocomputing und Computerarchäolgie. Winnenden: CSW.
- Höltgen, Stefan (2015a): It's more fun to compute! Retro-Games als Wissensobjekte. In: Letourneur, Ann-Marie/Mosel, Michael/Raupach, Tim (Hgg.): Retro-Games und Retro-Gaming. Nostalgie als Phänomen einer performativen Ästhetik von Computer- und Videospielen. Glückstadt: vwh 2015, S. 49-66.
- Höltgen, Stefan (2015b): JUMPs durch exotische Zonen. Portale, Hyperräume und Teleportation in Computern und Computerspielen. In: Hensel, Thomas/Neitzel, Britta/Nohr, Rolf F. (Hgg.) „The cake is a lie!" Polyperspektivische Betrachtungen des Computerspiels am Beispiel von 'Portal'. Münster: LIT, S. 107-134.
- Höltgen, Stefan (2016a): Time Invaders. Zeit(ge)schichten in Computer(spiele)n. In: [Höltgen/van Treeck 2016], S. 51-69.
- Höltgen, Stefan (2016b): GO BACK GOTO. Ein kurzer Überblick über die Entfernung der Schulinformatik von den Computern. In: Grundlagenstudien aus Kybernetik und Geisteswissenschaften, Band 57, Heft 3 (September 2016), S. 141-152.
- Höltgen, Stefan (2016c): Spiel, Raum und Krieg. Der Hacker als Partisan im Kalten Krieg. In: Nowak, Lars (Hg): Medien – Krieg – Raum. Paderborn: Fink, S. 393-416.
- Höltgen, Stefan (2016d): Tennis 2650. In: http://www.simulationsraum.de/blog/2016/05/24/tennis-2650/ [letzter Abruf: 04.07.2016].
- Höltgen, Stefan (2016e): Schiffe (ver)raten. In: http://www.simulationsraum.de/blog/2016/07/04/schiffe-verraten/ [letzter Abruf: 04.07.2016].
- Höltgen, Stefan (2017a): Das Spiel des Lebens. Ein Computer kann alles sein, was ein Computer sein kann. Sogar ein Computer! In: LOAD, Ausgabe 3/2017, S. 25-27. https://www.classic-computing.org/2017-load3/ [letzter Abruf: 16.08.2018].

- Höltgen, Stefan (2017b): Logik. In: Ders. (Hg.): Medientechnisches Wissen. Band 1: Logik, Informations- und Speichertheorie. Berlin u.a.: DeGruyter, S: 15-148.
- Höltgen, Stefan (2018a): Computer/sprachen: ELIZA und BASIC. Urszenen des Homecomputing (und) künstlicher Intelligenz. In: Ders./Baranovska, M. (Hgg.): Hello, I'm ELIZA. 50 Jahre Chatbots. (Computerarchäologie, Band 5). Bochum: Projektverlag, S. 97–122.
- Höltgen, Stefan (2018b): Assembler für Medienwissenschaftler. In: Ders. (Hg.): Medientechnisches Wissen, Band 2: Informatik, Programmieren, Kybernetik. Boston/Berlin: DeGruyter, S. 136-168.
- Höltgen, Stefan (2019a): DaimoGraphien. Für ein Argumentieren jenseits des Diskursiven. In: Ders./Hiller, Moritz (Hgg.): Archäographien. Aspekte einer radikalen Medienarchäologie. Festschrift für Wolfgang Ernst. Berlin/Basel: Schwabe, S. 301-314.
- Höltgen, Stefan (2019b): Unterwegs zum Computerspiel. Rezension zu: [Beil/Hensel/Rauscher 2018]. In: http://www.paidia.de/unterwegs-zum-computerspiel-rezension/ [letzter Abruf: 15.07.2019].
- Höltgen, Stefan (2020): Der sogenannte Computer. Zum Problem des Kollektivums der Digitalisierung. In: Biermann, Ralf/Holze, Jens/Verständig, Dan (Hgg.): Medienbildung zwischen Subjektivität und Kollektivität (Reihe: Medienbildung und Gesellschaft). SpringerVS. (In Vorbereitung)
- Höltgen, Stefan/Groth, Marius (2018c): Wissens-Appa/Repa/raturen. Ein epistemologisch-archäologischer Werkstattbericht von der Reparatur eines frühen Mikrocomputers. In: Krebs, Stefan/Schabacher, Gabriele/Weber, Heike (Hgg.): Kulturen des Reparierens. Dinge, Wisse, Praktiken. Bielefeld: transcript, S. 239–264.
- Höltgen, Stefan/Maibaum, Johannes/Rech, Matthias (2012): Tennisspielen mit Physik. In: Retro Nr. 24 (Sommer 2012), S. 32-37.
- Höltgen, Stefan/van Treeck, Jan Claas (2016) (Hgg): Time To Play. Zeit und Computerspiel. Glücksstadt: vwh.
- Holtius, S. (1993): lntertextualität. Aspekte einer rezeptionsorientierten Konzeption. Tübingen: Stauffenberg.
- Holzmann, G./Meyer, H./Schumpich, G. (2010): Technische Mechanik, Kinematik und Kinetik. Wiesbaden: Vieweg+Teubner.
- Hopper, Grace (1947): Log Book With Computer Bug. In: Grace Murray Hopper Collection, Archives Center, National Museum of American History, Smithsonian Institution. http://americanhistory.si.edu/collections/search/object/nmah_334663 [letzter Abruf: 20.02.2018].
- Horn, B. K. P. (1976): Circle Generators for Display Devices. In: Computer Graphics and Image Processing 5, S. 280-288.

- hs (1986a): Amigas Nachfolger? In: HC Mein Home-Computer, 7/1986, S. 115.
- hs (1986b): Liebe auf den zweiten Blick. Amiga – Die neue Freundin aus dem Hause Commodore. Ein Computer, der mit neuartiger Technik neue Leistungsmaßstäbe setzen will. In: HC Mein Home-Computer, 1/1986, S. 18f.
- Hubwieser, Peter (2007): Didaktik der Informatik. Grundlagen, Konzepte, Beispiele. Berlin/Heidelberg: Springer.
- Hugg, Steven (2016): Making Games for the Atari 2600. Wrodclaw: Amazon.
- Hung, Patrick C. K. (Hg.) (2015): Mobile Services for Toy Computing. Cham u.a.: Springer.
- Husserl, E. (1928): Vorlesungen zur Phänomenologie des inneren Zeitbewußtseins. Halle: Niemeyer.
- IBM (1962): Rererence Manual. Catalog for Programs for IBM Data Processing Systems. KWIC Index, April 1962, No. 1. In: ftp://bitsavers.informatik.uni-stuttgart.de/pdf/ibm/pgmCatalog/C20-8090_Catalog_of_Programs_for_IBM_Data_Processing_Systems_KWIC_Index_Apr62.pdf [Abrufdatum: 05.02.2019].
- Isaías, Pedro/Spector, J. Michael/Ifenthaler, Dirk/Sampson, Demetrios G. (Hgg.) (2015): E-Learning Systems, Environments and Approaches. Theory and Implementation. Cham: Springer.
- Jacobs, James (2007): Interton VC 4000 Coding Guide, Version vom 27.09.2010. In: https://web.archive.org/web/20121001021823/http://amigan.yatho.com/i-coding.txt [letzter Abruf: 18.03.2019)
- Jäger, Siegfried (2012): Kritische Diskursanalyse. Eine Einführung. (Edition: DISS) Münster: Unrast.
- Janich, N. (2008): Intertextualität und Text(sorten)vernetzung. In: Dies. (Hg.): Textlinguistik. 15 Einführungen. Tübingen: Narr, S. 177-198.
- Janneck, J. W./Mossakowski, T. (1985): ROM-Listing CPC 464/664/6128. München: Markt+Technik.
- Jauß, Hans Robert (1975a): Über die Partialität der rezeptionsästhetischen Methode. Nachwort zu Racines und Goethes Iphigenie. In: Warning, R. (Hg.): Rezeptionsästhetik. München: Fink (UTB), S. 353-400.
- Jauß, Hans Robert (1975b): Literaturgeschichte als Provokation der Literaturwissenschaft. In: Warning, R. (Hg.): Rezeptionsästhetik. München: Fink (UTB), S. 126-162.
- Jérôme, Durand-Lose (2002): Computing Inside the Billard Ball Model. In: [Adamatzky 2002a], S. 135-160.
- Jones, Steven E./Thiruvathukal, George K. (2012): Codename Revolution. The Nintendo Wii. (Reihe: Platform Studies, Band 3) Cambridge/London: The MIT Press.

- Jones, Steven. E. (2014): The Emergence of the Digital Humanities. New York/London: Routledge.
- Jordan, Stefan (2013): Theorien und Methoden der Geschichtswissenschaft. Wien/Köln/Weinmar: Böhlau. (UTB)
- Joyce, D. (2015): Aryabhata'strigtable. Math 105 History of Mathematics. In: https://mathcs.clarku.edu/~djoyce/ma105/aryatrig.pdf [letzter Abruf: 23.02.2018].
- Kaiser, Armin (2003): Selbstlernkompetenz, Metakognition und Weiterbildung. In: Ders. (Hg.): Selbstlernkompetenz. Metakognitive Grundlagen selbstregulierten Lernens und ihre praktische Umsetzung. München/Unterschleißheim: Kluwer 2003, S. 12-34.
- Kammersgaard, John (1989): Four Different Perspectives on Human-Computer Interaction. In: Preece, Jenny/Keller, Laurie (Hgg.): Human-Computer Interaction. Selected Readings. Englewood Cliffs: Prentice Hall, S. 42-63.
- Katz, V. J. (1987): The Calculus of Trigonometric Functions. In: HISTORIA MATHEMATICA 14 (1987), S. 311-324.
- Keefer, Erwin (2006): Zeitsprung in die Urgeschichte. Von wissenschaftlichen Versuch und lebendiger Vermittlung. In: Archäologie in Deutschland. Sonderheft 6: Lebendige Vergangenheit. Vom archäologischen Experiment zur Zeitreise. Stuttgart: Theiss, S. 8-36.
- Keichel, C. (2017): Stark verpixelt. Was vom Computerspiel(en) übrig blieb. (Reihe: Computerarchäologie Band 2). Bochum: projekt.
- Keller, A. (2017): Alter Code in neuen Schläuchen. Über die Wiederverwendbarkeit von Software aus den 1980er Jahren. In: Deutschlandfunk: Computer und Kommunikation. Sendung vom 25.11.2017, http://ondemand-mp3.dradio.de/file/dradio/2017/11/25/alter_code_in_neuen_schlaeuchen_dlf_20171125_1636_c006411d.mp3 [letzter Abruf: 05.4.2018].
- Kent, Steven L. (2001): The Ultimate History of Video Games. The Story behind the Craze that touched our lives and changed the World. New York: Three Rivers Press.
- Kernighan, B. W./Ritchie, D. M. (1990): Programmieren in C. Mit dem C-Reference Manual in deutscher Sprache. Zweite Ausgabe ANSI C. München: Hanser.
- Kittler, Friedrich (1986): Grammophon, Film, Typewriter. Berlin: Brinkmann & Bose.
- Kittler, Friedrich (1993): Draculas Vermächtnis. Technische Schriften. Leipzig: Reclam.
- Kittler, Friedrich (1993a): Protected Mode. In: [Kittler 1993:208-224].
- Kittler, Friedrich (1993b): Es gibt keine Software. In: [Kittler 1993:225-242].
- Kittler, Friedrich (1993c): Die Welt des Symbolischen – eine Welt der Maschine. In: [Kittler 1993:58-80].

- Kittler, Friedrich (1996): Computeranalphabetismus. In: Matejovski, D./Kittler, F. (Hgg.): Literatur im Informationszeitalter. Frankfurt am Main/New York: Campus, S. 237-251.
- Kittler, Friedrich (2007): Museen an der digitalen Grenze. In: Helas, Philine/Polte, Maren/Rückert, Claudia/Uppenkamp, Bettina (Hgg.): Bild/Geschichte. Frestschrift für Horst Bredekamp. Berlin: Akademie-Verlag, S. 109-118.
- Klöter, Patric (2008): Technische Daten der Amiga Computer. In: AmigaFuture, http://www.amigafuture.de/kb.php?mode=article&k=367 [letzter Abruf: 19.07.2016].
- Knauer, Willi (1980): Informatik als Schulfach. (Reihe: EDV in der Anwendung, hrsg. v. Prof. Dr. Martin A. Graef, Band 12). Tübingen: Neuer Verlag Bernhard Bruscha.
- Kneusel, Robert T. (2018): Random Numbers and Computers. Cham: Springer.
- Knuth, D. E./Pardo, L. T. (1976): The Early Development of Programming Languages. Stanford: Stanford University.
- Koch, Karl-Heinz (1985): Spielend programmieren lernen. Einführung in das kreative Programmieren mit dem Heimcomputer. Lernen Sie, Computerspiele selbst zu gestalten. (ht Ratgeber). München: Humboldt-Taschenbuchverlag.
- Koch, Michael (2012): Gamification – Steigerung der Nutzungsmotivation durch Spielkonzepte. In: http://www.soziotech.org/gamification-steigerung-der-nutzungsmotivation-durch-spielkonzepte/ [letzter Abruf: 12.04.2016].
- König, Wolfgang (2009): Technikgeschichte. Eine Einführung in ihre Konzepte und Forschungsergebnisse. Stuttgart: Franz Steiner Verlag.
- König, Wolfgang (2009a): Technikgeschichte. Eine Einführung in ihre Konzepte und Forschungsergebnisse. (Reihe: Grundzüge der modernen Wirtschaftsgeschichte – 7) Stuttgart: Franz-Steiner-Verlag.
- Kornwachs, K. (1999): Das Projekt: Wissen für die Zukunft. In: Kornwachs, K./Berndes, S. (Hgg.): Wissen für die Zukunft. II: Veröffentlichte Arbeiten (Textteil). Abschlußbericht an das Zentrum für Technik und Gesellschaft. Berichte der Fakultät für Mathematik. Naturwissenschaften und Informatik der Brandenburgischen Technischen Universität Cottbus, PT - 3/1999, Cottbus, S. 7-44.
- Krämer, Sibylle (2005): Operationsraum Schrift. Über einen Perspektivwechsel in der Betrachtung der Schrift. In: Grube, Gernot/Kogge, Werner/Krämer, Sybille (Hgg.): Schrift: Kulturtechnik zwischen Auge, Hand und Maschine. München: Fink, S. 23-60.
- Kretzinger, Boris (2005): Commodore: Aufstieg und Fall eines Computerriesen. Morschen: Skriptorium.
- Kuhn, Thomas S. (1996): Die Struktur wissenschaftlicher Revolutionen. Frankfurt am Main: Suhrkamp.

- Kunkel-Razum, K. (2016): Textkohäsion. In: Dies.: Die Grammatik. (DUDEN, Band 4). Berlin: Bibliographisches Institut, S. 1079-1135.
- Küppers, B.-O. (2000): Die Strukturwissenschaften als Bindeglied zwischen Geistes- und Naturwissenschaften. In: Ders. (Hg.): Die Einheit der Wirklichkeit. Zum Wissenschaftsverständnis der Gegenwart. München: Fink, S. 89-106.
- Kurtz, Thomas E. (2009): BASIC. In: Biancuzzi, F./Warden, S. (Hgg.): Visionäre der Programmierung. Die Sprachen und ihre Schöpfer. Beijing u.a.: O'Reilly, S. 81-102.
- Lange, Andreas (2016): Playing in Different Times. Herausforderungen und Möglichkeiten der Bewahrung von Computerspielen. In: [Höltgen/van Treeck 2016], S. 314-322.
- Lauer, G. (2009): Lektüre im Computerzeitalter. In: Frankfurter Allgemeine, 26.08.2009, http://www.faz.net/aktuell/feuilleton/geisteswissenschaften/literatur-rechnen-lektuere-im-computerzeitalter-1840973.html [letzter Abruf: 16.03.2018].
- Le, Nguyen-Thinh/Strickroth, Sven/Gross, Sebastian/Pinkwart, Niels (2013): A Review of AI-Supported Tutoring Approaches for Learning Programming. In: Nguyen, N. T. u.a. (Hgg.): Advanced Computational Methods for Knowledge Engineering. Heidelberg u.a.: Springer, S. 267-279.
- Lee, Younghwa (2011): Understanding anti-plagiarism software adoption: An extended protection motivation theory perspective. In: Decision Support Systems 50 (2011), S. 361–369
- Leeker, Martina (2001): Medientheater/Theatermedien. In: Dies. (Hg.): MASCHINEN, MEDIEN, PERFORMANCES. Theater an der Schnittstelle zu digitalen Welten. Berlin: Alexander, S. 374-403.
- Leibniz G. W: (1697): Machina arithmeticae dyadicae. In: http://dokumente.leibnizcentral.de/index.php?id=95 [letzter Abruf: 25.11.2016].
- Leibniz, G. W. (1697): Brief an den Herzog von Braunschweig-Wolfenbüttel Rudolph August, 2. Januar 1697. In: http://www.hs-augsburg.de/~harsch/germanica/Chronologie/17Jh/Leibniz/lei_bi-na.html [letzter Abruf: 11.01.2019].
- Lenz, K. (1924): Die Rechenmaschinen und das Maschinenrechnen. Wiesbaden: Springer.
- Levy, Stefan (1993): KI. Künstliches Leben aus dem Computer. München: Droemer/Knaur.
- Levy, Steven (1984): Hackers. Heroes of the Computer Revolution. Garden City/New York: Anchor Press/Doubleday.
- Levy, Steven (2010): Hackers. Heroes of the Computer Revolution. Beijing u.a.: O'Reilly.

- Li, Zhenmin/Zhou, Yuanyuan (2005): PR-Miner. Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code. In: ACM SIGSOFT Software Engineering Notes Homepage, Volume 30 Issue 5, September 2005, S. 306-315, https://dl.acm.org/citation.cfm?id=1081755 [letzter Abruf: 21.08.2018].
- Link, David (2016): There must be an Angel. On the Beginnings of the Arithmetics of Rays. In: Ders.: Archaeology of algorithmic artefacts. Minneapolis: Univocal Publishing, S. 55-78.
- Lippe, Wolfram-M. (2013): Die Geschichte der Rechenautomaten. Von der Himmelsscheibe von Nebra bis zu den ersten Rechenmaschinen. Berlin/Heidelberg: Springer.
- Loebel, Jens-Martin (2014): Lost in Translation. Leistungsfähigkeit, Einsatz und Grenzen von Emulatoren bei der Langzeitbewahrung digitaler multimedialer Objekte am Beispiel von Computerspielen. Glückstadt: vwh.
- Löthe, H./Quehl, W. (1982): Systematisches Arbeiten mit BASIC. Problemlösen – Programmieren. Stuttgart: Teubner.
- Louden, Kenneth C. (1994): Programmiersprachen. Graundlagen, Konzepte, Entwurf. Thompson Publ. 1994.
- Luhmann, Niklas (1977): Differentiation of Society. In: Canadian Journal of Sociology / Cahiers canadiens de sociologie, Vol. 2, No. 1 (Winter, 1977), S. 29-53. http://www.jstor.org/stable/3340510 [letzter Abruf: 25.11.2016].
- Lytoard, Jean-François (2012): Das postmoderne Wissen. Wien: Passagen.
- Mader, Günter/Stöckl, Walter (1999): Virtuelles Lernen: Begriffsbestimmung und aktuelle empirische Befunde. Innsbruck: Studien Verlag.
- Maher, Jimmy (2012): The Future Was Here. The Commodore Amiga. Cambridge/London: MIT Press.
- Mahoney, Michael S. (2000): (Review for) Paul E. Ceruzzi, A History of Modern Computing, MIT Press, Cambridge, Mass., 1998, $35.00, 416 pp., ISBN 0262032554. In: Annals of the History of the Computer, Issue No. 03 (July-September), Vol. 22, S. 93f.
- Maibaum, Johannes (2015): Retrocomputing als praktische Medienarchäologie. Über die Entwicklung eines Flash-Speicher-Moduls für die Videospielkonsole „Interton VC 4000“ (Projektarbeit). In: https://www.musikundmedien.hu-berlin.de/de/medienwissenschaft/medientheorien/signallabor/maibaum_projektbericht-multi-rom.pdf [letzter Abruf: 15.03.2019].
- Maibuam, Johannes/Höltgen, Stefan (2018): Programmieren für Medienwissenschaftler. In: Höltgen, Stefan (Hg.): Medientechnisches Wissen, Band 2. Berlin u.a.: DeGruyter, S. 133-273.

- Mainzer, Klaus/Chua, Leon (2012): The Universe as Automaton. From Simplicity and Symmetry to Complexity. Heidelberg u.a.: Springer.
- Malek, Mirislaw (2006): A Perspective on Parallel and Distributed Computing. In: Reisig, W./Freytag, J.-C. (Hgg.): Informatik. Aktuelle Themen im historischen Kontext. Berlin/Heidelberg: Springer, S. 197-220.
- Malone, Michael S. (1996): Der Mikroprozessor. Eine ungewöhnliche Biographie. Berlin/Heidelberg: Springer.
- Maurer, H./Kappe, F./Zaka, B. (2006): Plagiarism - A Survey. In: Journal of Universal Computer Science, vol. 12, no. 8, S. 1050-1084.
- McDaniel, Dean (1975): Kill the Bit game. In: http://altairclone.com/downloads/kill-bits.pdf [letzter Abruf: 11.07.2016].
- McLuhan, Marshall (1992): Die Magischen Kanäle. „Understanding Medie". Düsseldorf u.a.: Econ.
- MEGA (2012). T42 – Tennis for Two. In: http://www.m-e-g-a.org/de/research-education/research/t42-tennis-for-two/ [letzter Abruf: 11.07.2016].
- Mens, T. u.a. (2005): Challanges in Software Evolution. In: Proceedings of the 2005 Eighth International Workshop on Principles of Software Evolution (IWPSE'05), https://pdfs.semanticscholar.org/ba8d/e5481506a1b027fa96a4499fc6c51c74614c.pdf [letzter Abruf: 21.03.2018].
- Mertens, P. (1986): Simulation, diskrete / Simulation, kontinuierliche. In: Schneider, H.-J. (Hg.): Lexikon der Informatik und Datenverarbeitung. 2. Auflage. München/Wien: Oldenbourg, S. 534.
- Merton, R. K. (1968): Social Theory and Social Structure. London: Collier-Macmillan.
- Merton, R. K. (1973): Der Rollen-Set: Probleme der soziologischen Theorie. In: Hartmann, Heinz (Hg.): Moderne amerikanische Soziologie. Neuere Beiträge zur soziologischen Theorie. Stuttgart: Ferdinand Enke Verlag, S. 316-333.
- Mihocka, Darek (1987): Inside the ST Xformer. A Special Inclusion. In: ST-LOG. The Atari ST Monthly Magazine, Nr. 17, September 1987, S. 43-47.
- Mihocka, Darek (1988): Inside the ST Xformer. A Special Inclusion, Part 2: the hardware. In: ST-LOG. The Atari ST Monthly Magazine, Nr. 18, April 1988, S. 71-80
- Mika, Jörn (2014): SymbOS. Ein GUI-Mulstitasking-Betriebssystem für Z80-basierte Computer. In: [Höltgen 2014e:97-114].
- Mitnick, Kevin/Simon, William L. (2011): Ghost in the Wires. My Adventure as the World's Most Wanted Hacker, New York/Boston/London.
- Miyazaki, Shintaro (2013): Algorhythmisiert. Eine Medienarchäologie digitaler Signale und (un)erhörter Zeiteffekte. Berlin: Kadmos.

- Mohr, Volker (2007): Amiga. Die Geschichte einer Computerlegende. Morschen: Skriptorium.
- Mohr, Volker (2014): Aus Neu mach Alt. RISC OS auf BeagleBoard und Raspberry Pi. In: [Höltgen 2014e:157-174].
- Mojang (2014): Minecraft. Das Schaltkreis-Handbuch. Die hohe Redstone-Schule. Köln: Egmont.
- Moll, S. O. (2014): Bouncing the Beam. Demo Hacking des Atari VCS/2600. In: [Höltgen 2014e:65-80].
- Montford, Nick/Bogost, Ian (2009): Racing the Beam. The Atari Video Computer System. Platform Studies Bd. 1. Cambridge: MIT.
- Montfort, Nick (2013): Beyond the Journal and the Blog. The Technical Report for Communication in the Humanities. In: Amodern 1: The Future of the Scholarly Journal, http://amodern.net/article/beyond-the-journal-and-the-blog-the-technical-report-for-communication-in-the-humanities/ [letzter Abruf: 28.08.2018].
- Moore, E. A. (1910): A Generalization of the Game Called Nim. In: Annals of Mathematics 11,3, S. 93-94.Höltgen/Nückel (2015)
- Morris, D. (1971): Computer-controlled Apparatus for Playing Game of Nim. US-Patent 3,606,331, In: https://patents.google.com/patent/US3606331A/en [letzter Abruf: 28.08.2018].
- Mortensen, Hviid/Kapper, Lise (2015): Beyons simple nostalgia: Transforming visitors‘ experinece of retro-gaming and vintage computing in the museum. In: Aktualitet. Litteratur, Kultur og Medier, 9:2, 2015, https://tidsskrift.dk/aktualitet/issue/view/3688 [Abrufdatum: 15.03.2019].
- Muñoz-Cremers, Omar (1999): Now that our Youth has been Emulated ... In: Mediamatic, Vol. 9/4 + 10/1 (Herbst 1999), S. 107-112.
- Murlasits, Elke/Reisinger, Gunther (Hgg.): museum multimedial. Audiovisuelle Traditionen in aktuellen Kontexten. Wien/Berlin: LIT.
- N. N. (1983): Atari Parts Are Dumped. In: The New York Times, 28.09.1983. http://www.nytimes.com/1983/09/28/business/atari-parts-are-dumped.html [letzter Abruf: 11.01.2019].
- N. N. (2014): Minecraft. Das Schaltkreis-Handbuch. Die hohe Redstone-Schule. Köln: Egmont.
- N. N. (2015a): Die Digitalisierung des Rundfunks. „Berliner MEDIEN Diskurs“ über Chancen und Herausforderungen der DAB-Übertragung. In: http://www.kas.de/wf/de/33.43280/ [letzter Abruf: 19.11.2015].
- N. N. (2015b): Termine und Veranstaltungen. In: Return Ausgabe 22 (3/2015), S. 11.

- Nake, Frieder (2005): Das doppelte Bild. In: Margarete Pratschke, (Hg.): Bildwelten des Wissens. Kunsthistorisches Jahrbuch für Bildkritik. Band 3, Nummer 3: Digitale Form. Berlin: Akademie-Verlag 2005, S. 40-50.
- Nake, Frieder (2008): Zeigen, Zeichnen und Zeichen. Der verschwundene Lichtgriffel. In: Heilige, Hans Dieter (Hg.): Mensch-Computer-Interface. Zur Geschichte der Computerbedienung. Bielefeld: transcript, S. 121-156.
- National Semiconductor (1977): SC/MP-II Microprocessor Retrofit Kit Users Manual. Santa Clara: National Semiconductor Corporation.
- Nelson, Ted (1987): Computer Lib/Dream Machines. (2nd ed.) Redmont: Tempus Books/Microsoft Press.
- Neumann, Gerhard (1999): Roland Barthes. Literatur als Ethnografie. Zum Konzept einer Semiologie der Kultur. In: Jörg Glauser & Annegret Heitmann (Hgg.): Verhandlungen mit dem New Historicism. Das Text-Kontext-Problem in der Literaturwissenschaft. Würzburg: Königshausen & Neumann.
- Nevison, John M. (1982): The Little Book of BASIC Style. How to write a program you can read. Reading u.a.: Addison-Wesley Publ.
- Nixon, Helen (2011): ‚From bricks to clicks': Hybrid commercial spaces in the landscape of early literacy and learning. In: Journal of Early Childhood Literacy, 11(2), 2011, S. 114–140.
- Nohr, Rolf F. (2019): Unternehmensplanspiele 1955–1975. Die Herstellung unternehmerischer Rationalität im Spiel. Münster: Lit.
- Nückel, Thomas/Borbach, Christoph (2016): Game of Memories. Zeitschichten und einem zellulären Automat. In: [Höltgen/van Treeck 2016], S. 70-94.
- Oltean, Mihai (o.J.): Evolving winning Strategies for Nim-like Games. In: https://pdfs.semanticscholar.org/398c/211d13151caa680183499299827f0978bdc4.pdf [letzter Abruf: 29.08.2018].
- Osgood, Rick (2015): Reprogramming Super Mario World from Inside The Game. In: http://hackaday.com/2015/01/22/reprogramming-super-mario-world-from-inside-the-game/ [letzter Abruf: 12.04.2016].
- Othmer, Thorsten (2013): Bis an die Grenzen der Hardwareleistung. Interview mit dem VC-4000-Programmierer Hans-Heinz Bieling. In: Retro Nr. 29, Winter 2013/14, S. 52f.
- Pachner, Anita (2009): Entwicklung und Förderung von selbst gesteuertem Lernen in Blended-Learning-Umgebungen. Eine Interviewstudie zum Vergleich von Lernstrategietrainung und Lerntagebuch. Münster u.a.: Waxmann.
- Peirce, C. S. (1983): Charles Sanders Peirce. Phänomen und Logik der Zeichen. Frankfurt/Main: Suhrkamp.

- Peschel, O. (1869): Der Atlas des Andrea Bianco vom Jahre 1436 in zehn Tafeln. Venedig: Münster.
- Pfaltzgraff, David J. (1969): Analog Simulation of the Bouncing-Ball-Problem. In: American Journal of Physics, Vol 37, No. 10 (Oktober 1969), S. 1008-1013.
- Pias, Claus (2002a): Computer Spiel Welten. München: sequenzia.
- Pias, Claus (2002b): Der Hacker. In: Eva Horn/Stefan Kaufmann/Ulrich Bröckling (Hgg.), Grenzverletzer. Von Schmugglern, Spionen und anderen subversiven Gestalten. Berlin: Kadmos, S. 248-270.
- Pias, Claus (2005a): Die Pflichten des Spielers. Der User als Gestalt der Anschlüsse. In: Warnke, Martin/Coy, Wolfgang/Tholen, Georg Christoph (Hgg.): HyperKult II. Zur Ortsbestimmung analoger und digitaler Medien. Bielefeld: Traskript, S. 313-341.
- Pias, Claus (2005b): „Children of the revolution". Video-Spiel-Computer als Kreuzungen der Informationsgesellschaft. In: Ders. (Hg.): Zukünfte des Computers. Zürich/Berlin: diaphanes, S. 217-240.
- Pias, Claus (2015): Friedrich Kittler und der „Mißbrauch von Heeresgerät". Zur Situation eines Denkbildes 1964 – 1984 – 2014. In: Merkur. Deutsche Zeitschrift für Europäisches Denken. Nr. 791 (April 2015), S. 31-44.
- Pias, Claus (2017): Medienphilologie und ihre Grenzen. In: Balke, F./Garderer, R. (Hgg.): Medienphilologie. Konturen eines Paradigmas. Göttingen: Wallstein, S. 364-385.
- Pöhls, Marcus/Peitek, Norman (2016): Retrofit: Love Working with APIs on Android. You need to take delight building API clients on Android. O. O: Future Studio.
- Pöpsel, J./Claussen, U./Klein, R.-D./Plate, J. (1994): Computergrafik. Algorithmen und Implenentierung. Berlin u.a.: Springer.
- Pratt, T./Zelkowitz, M. (1998): Programmiersprachen. Design und Implementierung. München u.a.: Prentice Hall.
- Processor Technology (1978): 16KRA. Dynamic Read/Write Memory Module. User's Manual. Pleasanton: Processor Technology Corporation. (http://www.sol20.org/manuals/img/16kra-img.pdf [letzter Abruf: 28.06.2016].
- Rafferty, Laura/Hung, Patrick C. K. (2015): Introduction to Toy Computing. In: [Hung 2015:1-7]
- Raymond, Darrell R. (1991): Reading Source Code. In: Proceedings of the 1991 conference of the Centre for Advanced Studies on Collaborative research, S. 3 - 16.
- Rechenberg, P. (2000): Was ist Informatik? Eine allgemeinverständliche Einführung. München, Wien: Hanser.
- Redheffer, Raymond (1948): A Machine for Playing the Game Nim. In: The American Mathematical Monthly, Vol. 55, No. 6 (Jun. - Jul., 1948), S. 343-349.

- reh (1985): Vom Würfelspiel zu Maus und GEM. In: HC – Mein Home-Computer, 10/1985, S. 122-124.
- Reinefeld, A. (2006): Entwicklung der Spielbaum-Suchverfahren: Von Zuses Schachhirn zum modernen Schachcomputer. In: Reisig, W./ Freytag, J.-C. (Hgg.): Aktuelle Themen im historischen Kontext. Berlin/Heidelberg: Springer, S. 241-274.
- Reinhard, Andrew (2014): The Video Game Graveyard. In: Archaeology. A publication of the Archaeological Institute of America. Online, 09.07.2014. http://www.archaeology.org/issues/139-1407/trenches/2189-new-mexico-atari-dump-site-excavation [letzter Abruf: 19.01.2019].
- Reißmann, Ole (2014): Verschollene Atari-Spiele: Komm, wir jagen einen Außerirdischen. In: Spiegel Online, 15.03.2014. http://www.spiegel.de/netzwelt/games/atari-e-t-spiele-in-der-wueste-als-doku-a-958137.html [letzter Abruf: 11.01.2019].
- Rendell, P. (2016): Turing Machine Universality of the Game of Life. Heidelberg u.a.: Springer.
- Rennard, Jean-Phillipe (2002): Implementation of Logical Functions in the Game of Life. In: [Adamatzky 2002a], S. 491-512.
- Renner, Eric (2009): Pinhole Photography – Rediscovering a Historic Technique. Amsterdam u.a.: Elsevier.
- Resnick, M./Martin, F./Sargent, R./Silverman, B. (1996): Programmable Bricks: Toys to think with. In: IBM Systems Journal, Vol. 35, Nos. 3&4, S. 443-452.
- Resnick, Mitchel u.a. (1998): Digital Manipulatives: New Toys to Thik with. In: CHI 1998, April, S. 281-287.
- Reunanen, Markku/Silvast, Antti (2009): Demoscene Platforms: A Case Study on the Adoption of Home Computers. In: Impagliazo, J./Järvi, T./Paju, P. (Hgg.): HiNC 2, IFIP AICT 303, S. 289-301.
- Reunanen, Markku/Silvast, Antti (2014): Multiple Users, Diverse Users: Appropriation of Personal Computers by Demoscene Hackers. In: Alberts, G./Oldenziel, R. (Hgg): Hacking Europe. From Computer Cultures to Demoscenes. London u.a.: Springer, S. 151-166.
- Rey, Günter Daniel (2009): E-Learning. Theorien, Gestaltungsempehlungen und Forschung. Bern: Huber.
- Rheinberger, Hans-Jörg (2000): Experiment: Präzision und Bastelei. In: Meinel, Christoph (Hrsg.): Instrument / Experiment. Historische Studien. Berlin/Diepholz: Verlag für Geschichte der Naturwissenschaft und Technik, S. 52-60.
- Rheinberger, Hans-Jörg (2001): Experimentalsysteme und epistemische Dinge. Eine Geschichte der Proteinsynthese im Reagenzglas. Göttingen: Wallstein.
- Richardson, Craig (2016): Python programmieren lernen mit Minecraft. Heidelberg: dpunkt.

- Rieger, B. (1970): Computer und Literatur. Zur Diskussion um quantifizierende Verfahrensweisen in der Literaturwissenschaft. In: WDR-3-Radio, 19.06.1970, Transkript: https://ww-w.uni-trier.de/fileadmin/fb2/LDV/Rieger/Publikationen/Aufsaetze/70/comp_lit70.pdf [letzter Abruf: 25.03.2018].
- Riekhof, Hans-Christian/Schüle, Hubert (Hgg.) (2002): E-Learning in der Praxis. Strategien, Konzepte, Fallstudien. Wiesbaden: Gabler, Springer.
- Roberts, H. E./Yates, W. (1975): Altair 8800. The most powerful minicomputer project ever presented – can be built for under $400. In: Popular Electronics, Januar 1975, S. 33-38. http://www.swtpc.com/mholley/PopularElectronics/Jan1975/PE_Jan1975.htm [letzter Abruf: 25.11.2016].
- Roch, Axel (2009): Claude E. Shannon. Spielzeug, Leben und die geheime Geschichte seiner Theorie der Information. Berlin: Gegenstalt.
- Rojas, Raúl (1997): Konrad Zuse's Legacy: The Architecture of the Z1 and Z3. In: IEEE Annals of the History of Computing, Vol. 19, No. 2, 1997, S. 5-16.
- Rojas, Raúl et al. (2004): Konrad Zuses Plankalkül – Seine Genese und eine moderne Implementierung. In: Heilige (2004) a. a. O., S. 215-236.
- Rothenberg, Jeff (1999): Avoiding Technological Quicksand: Finding a Viable Technical Foundation for Digital Preservation. A Report to the Council on Library and Information Resources. https://dl.acm.org/citation.cfm?id=520971 [letzter Abruf: 13.06.2019].
- Rothstein, B./Kröger-Bidlo, H./Gräsel, C./Rupp, G. (2014): Überlegungen zur Messung des Kohäsionsgrads von Texten. In: Linguistische Berichte 237 (2014), S. 37-56.
- Rougetet, Lisa (2014): The Prehistory of Nim Game. In: G4G11 Exchange Book VOLUME 1, S. 106-140. http://www.gathering4gardner.org/g4g11-exchange-book/ [letzter Abruf: 28.08.2018].
- Rougetet, Lisa (2016): Machines designed to play Nim Games. Teaching supports for mathematics, algorithmics and computer science (1940-1970). In: History and Pedagogy of Mathematics, Jul 2016, Montpellier, France. https://hal.archives-ouvertes.fr/hal-01349260 [letzter Abruf: 28.08.2018].
- Rudelt, Tom (2014): Codecademy – Programmieren online lernen? In: http://blog.hwr-berlin.de/elerner/2014/10/30/codecademy-programmieren-online-lernen/ [letzter Abruf: 12.04.2016].
- Ruiz de la Infante, Francisco/Thomas, Jérôme u.a. (2013): Die Konservierung: ein Aspekt der Lehre in einer Kunsthochschule (oder nicht)? In: Serexhe, Bernhard

(Hg.): digital art conservation. Konservierung digitaler Kunst: Theorie und Praxis. Wien: Ambra, S. 619-631.

- Russwurm, Siegfried (2013): Software: Die Zukunft der Industrie. In: Sendler, Ulrich (Hg.): Industrie 4.0. Beherrschung der industriellen Komplexität mit SysLM. Heidelberg u.a.: Springer Vieweg, S. 21-36.
- Sander, Phillip (2019): Genealogie der Programmiersprachen. Analyse zur babylonischen Sprachverwirrung im Diskurs der Programmierung. Masterarbeit, Humboldt-Universität zu Berlin.
- SAS (1990): SAS/C Compiler for AmigaDOS User's Manual. Version 5.10. The C Development System for Amiga. Cary: SAS Institute Inc.
- Scherrer, Kai (2018): Der Boing-Ball auf der Hebebühne – Funktionsweise der originalen „Boing!"-Demo auf dem Amiga. Vortrag im Rahmen der Kurztagung „50 Jahre Computerdemos. Geschichte, Theorie, Programmierung" auf dem Vintage Computing Festival 2018 (14.10.2018), https://www.youtube.com/watch?v=w6HaRAA8UCo [letzter Abruf: 11.01.2019].
- Schildt, G. H./Kahn, D./Krügel, Chr./Moerz, Chr. (2005): Einführung in die technische Informatik. Wien/New York: Springer.
- Schmidt-Biggemann, Wilhelm (2003): Geschichte, Ereignis, Erzählung. Über Schwierigkeiten und Besonderheiten von Geschichtsphilosophie. In: Speer, Andreas (Hg.): Anachronismen. Tagung des Engeren Kreises der Allgemeinen Gesellschaft für Philosophie in Deutschland (AGPD) vom 3. bis 6. Oktober 2001 in der Würzburger Residenz. Würzburg: Königshausen & Neumann, S. 25-50.
- Schmitz, Anke (2016): Verständlichkeit von Sachtexten. Wirkung der globalen Textkohäsion auf das Textverständnis von Schülern. Wiesbaden: Springer VS.
- Schmitz, Jürgen (1991): Automatische Analyse von Fortran- und C-Programmen. Klassische Programmiersprachen gehören noch nicht zum alten Eisen. In: Computerwoche, 08.02.1991. https://www.computerwoche.de/a/automatische-analyse-von-fortran-und-c-programmen,1138427 [letzter Abruf: 11.01.2019].
- Schneider, Helmut (1976): Fläche durch Farbe in Bewegung. Frank Kupka war der erste abstrakte Maler. In: Die Zeit, 09/1976, http://www.zeit.de/1976/09/flaeche-durch-farbe-in-bewegung/komplettansicht [letzter Abruf: 19.03.2018].
- Schneider, Ulrich J. (2004): Philosophische Archäologie und Archäologie der Philosophie: Kant und Foucault. In: Ebeling, Knut/Altekamp, Stefan (Hg.): Die Aktualität des Archäologischen in Wissenschaft, Medien und Künsten. Frankfurt am Main: Fischer, S. 79-97.
- Schneider, W. (1982): BASIC für Fortgeschrittene. Wiesbaden: Springer.
- Schneider, W. (1985): Strukturiertes Programmieren in BASIC. Braunschweig/Wiesbaden: Vieweg & Sohn.

- Schöler, Thorsten (2018): Informatik. In: Höltgen, Stefan (2018) (Hg.): Medientechnisches Wissen, Band 2: Informatik, Programmieren, Kybernetik. Bosten/Berlin: DeGruyter, S. 7-130.
- Scholl, L. U. (1981): Der Ingenieur in Ausbildung, Beruf und Gesellschaft 1856 bis 1881. In: Ludwig, K.-H./König, W. (Hgg.): Technik, Ingenieure und Gesellschaft. Geschichte des Vereins Deutscher Ingenieure 1856-1981. Düsseldorf: VDI-Verlag, S. 1-66.
- Schönfisch, Birgitt (1993) - Zelluläre Automaten und Modelle für Epidemien. Eßlingen: Universität Tübingen. (Dissertation)
- Schwarz-Friesel, M./Consten, M./Knees, M. (2007): The function of complex anapors in texts: Evidence from corpus studies and ontological consideration. In: Dies. (Hgg.): Anaphors in Text. Cognitive, formal and applied approaches to anaphoric reference. Amsterdam/Philadelphia: John Benjamins Publishing Company, S. 81-102.
- Schweibenz, Werner (2012): Das Museumsobjekt im Zeitalter seiner digitalen Reproduzierbarkeit. In: Murlasits, Elke/Raisinger, Gunther (Hgg.): museum multimedial. Audiovisionäre Traditionen in aktuellen Kontexten. Wien/Münster: Lit, S. 47-70.
- Shannon, C. E. (1938): A Symbolic Analysis of Relay and Switching Circuits. In: Transactions Institute American Engineering, Band 57.
- Shannon, Claude (1950): Programming a Computer for Playing Chess. In: Philosophical Magazine, Ser. 7, Vol. 41, No. 314 - March 1950, https://vision.unipv.it/IA1/aa2009-2010/ProgrammingaComputerforPlayingChess.pdf [letzter Abruf: 12.07.2019].
- Shibata, Yuu/Hiraki, Kei (2006): Applying recent techniques for retro games: in the case of undo function. In: ACE '06 Proceedings of the 2006 ACM SIGCHI international conference on Advances in computer entertainment technology, Article No. 21. https://dl.acm.org/citation.cfm?id=1178848 [letzter Abruf: 090.70.2019].
- Shores, c. (2009): The Development of Digital Carry Technology. In: Pirates & Revolutionaries – Research Machinery. http://piratesandrevolutionaries.blogspot.com/2009/03/development-of-digital-carry-technology.html [letzter Abruf: 11.01.2019].
- Sieg, Peter (2013): Retro-Computing Simulation – emulation – Projekte: „Exotic Flavor“. Scotts Valley: CreateSpace Independent Publishing Platform.
- Siegert, Bernhard (1993): Relais. Geschicke der Literatur als Epoche der Post 1751-1913. Berlin: Brinkmann & Bose.
- Signetics (1978a): 2650 Microprocessor Course. An Applications-oriented audio-visual Learning Exprience in fundamental Microprocessor Design. Sunnyvale: Signentics Corporation.

- Signetics (1978b): Signetics Instructor 50 Desktop Computer User‘s Guide. Sunnyvale: Signetics Corporation.
- Signetics (1981): Signetics Programmable Video Interface (PVI) 2636. 7/81, In: https://amigan.yatho.com/2636PVI.pdf [letzter Abruf: 18.03.2019].
- Simons, Volkhard (2013a): Klingende KONTAKT-Tafeln. Wie der Computer in den Musikunterricht fand. In: Retro Nr. 28 (Sommer 2013), S. 34-37.
- Simons, Volkhard (2013b): Das KÖLN-Programm. Wie sich Computer und Schule offline vernetzten. In: Retro Nr. 29 (Winter 2013/14), S. 24-25.
- Smith, Alvy Ray (2016): The Dawn of Digital Light. In: IEEE Annals of the History of Computing, Vol. 38, Issue, No. 04 (Oct.-Dec. 2016). https://www.computer.org/csdl/mags/an/2016/04/man2016040074.html [letzter Abruf: 21.08.2018].
- Smith, Tony (2014): You’re NOT fired: The story of Amstrad’s amazing CPC 464. 30 years ago, a budget micro shocked a nation - with how good it was. In: The Register, 12.02.2014, https://www.theregister.co.uk/2014/02/12/archaeologic_amstrad_cpc_464/ [letzter Abruf: 16.03.2018].
- Solomon, Joan (2003): Theories of learning and the range of autodidactism. In: The Passion to Learn. An Inquiery into Autodidactism, ed. by Joan Solomon. London, New York: Routledge/Falmer, p. 3-23.
- Sommergut, W. (1994): Programmieren in C. Einführung auf der Grundlage des ANSI-C-Standards. München: dtv.
- Sommerville, Ian (2012): Software Engineering. 9., aktualisierte Auflage. Hallbergmoos: Pearson Deutschland GmbH.
- Spital, I./Perry, R./Poel, W./Lawson, C. (1985): Colour Personal Computer ‚CPC6128‘ Benutzerhandbuch. Türkheim: Schneider Computer Division.
- Squire, Kurt (2005): Game-Based Learning: State of the Field. In: https://s3.amazonaws.com/academia.edu.documents/8132658/game-based_learning.pdf?AWSAccessKeyId=AKIAIWOWYYGZ2Y53UL3A&Expires=1535630474&Signature=Lb8Xp39mx0J6dw6ZGKnIETEEmPI%3D&response-content-disposition=inline%3B%20filename%3DGame-based_learning_Pres [letzter Abruf: 29.08.2018].
- Steil, M. (2014): Rasterstrahl-Hacken. Der Grafikchip des Commodore 64. In: [Höltgen 2014e:115-138].
- Stein, E. (2016): Leibniz-Ausstellung der Leibniz-Universität Hannover. In: https://www.uni-hannover.de/de/universitaet/leibniz/leibnizausstellung/ [letzter Abruf: 25.11.2016].

- Stöckle, Frieder (1990): Zum praktischen Umgang mit der Oral History. In: Herwart Vorländer (Hg.): Oral History. Mündlich erfragte Geschichte. Göttingen: Vandenhoek & Ruprecht, S. 131-158.
- Stoll, Clifford (1989): Kuckucksei. Die Jagd auf die deutschen Hacker, die das Pentagon knackten. Frankfurt am Main: Krüger.
- Strübel, G. (1986): Simulation, diskrete / Simulation, kontinuierliche. In: Schneider, H.-J. (Hg.): Lexikon der Informatik und Datenverarbeitung. 2. Auflage. München/Wien: Oldenbourg, S. 534f.
- Swade, Doron (2000): Virtual Objects. Threat or Salvation? In: Lindqvist, Svante u.a. (Hgg.): Museums of Modern Science. Canton: Nobel Symposium 112. Science History Publications/USA, S. 139-147.
- Swalwell, Melanie (2013): Moving on from the Original Experience: Game history, preservation and presentation. In: DiGRA '13 – Proceedings of the 2013 DiGRA International Conference: DeFragging Game Studies, August, 2014, Volume: 7, http://www.digra.org/digital-library/publications/moving-on-from-the-original-experience-games-history-preservation-and-presentation/ [letzter Abruf: 21.03.2019].
- Swaminathan, Nikhil (2011): Digging in Technology's Past. „Digital archaeologists" excavate the microprocessor that ushered in the home computer revolution. In: Archaeology. A publication of the archaeological Institute of America. Vol. 64, No. 4, July/August 2011. Online: https://archive.archaeology.org/1107/features/mos_technology_6502_computer_chip_cpu.html [letzter Abruf: 12.12.2018].
- Sydow, A. (1974): Programmierungstechnik für elektronische Analogrechner. Berlin: VEB Verlag Technik.
- Takhteyev, Yuri/DuPont, Quinn (2013): Retrocomputing as Preservation and Remix. In: iConference 2013 Proceedings, S. 422-432. doi:10.9776/13230, https://www.ideals.illinois.edu/bitstream/handle/2142/38392/230.pdf?sequence=4 [letzter Abruf: 17.01.2017].
- Tanenbaum, Andrew S. (2006): Computerarchitektur. Strukturen – Konzepte – Grundlagen. München: Pearsons.
- Tanner, Jakob (2004): Historische Anthropologie zur Einführung. Hamburg: Junius.
- Texas Instruments (2004): SN74LVC125A. Quadruple Bus Buffer Gate With 3-State Outputs. Datenblatt. Dallas/Texas, Feb. 2004. In: http://pdf1.alldatasheet.com/datasheet-pdf/view/171748/TI/LVC125A.html [letzter Abruf: 19.03.2019].

- Thibadeau, Kenneth (1994): Digital Preservation Techniques: Evaluating the Options. In: ISAD (G): Descrizione Archivistica: Standard Internazionale, San Miniato 1994, 11(2), S. 101-109.
- Thorvalds, Linus (2001): Prologue. What Makes Hackers Tick? a.k.a. Linus's Law. In: Himanen, Pekka (Hg.): The Hacker Ethik and the Spirit of the Information Age. New York, S. xvi f.
- Tijms, Arjan (2000): Binary translation: Classification of emulators. In: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.75.807 [letzter Abruf: 15.03.2019].
- Tomczyk, M. S. (1984): The Home Computer Wars. An Insider's Account of Commodore and Jack Tramiel. In: http://www.stonan.com/dok/The.Home.Computer.Wars.pdf [letzter Abruf: 07.07.2019].
- Trionfo, Adam (2012): Changing Atari VCS Graphics - The Easy Way. In: OC&GS Newsletter, April 1999 (Re-Written and Updated, March 2012), http://www.orphanedgames.com/ocgs/issue12/vcsgraph.html [letzter Abruf: 06.03.2012].
- Turing, Alan M. (1947): Lecture to the London Mathematical Society on 20 February 1947. In: Ince, D. C. (Hg.) (1992): Collected Works of A. M. Turing. Mechanical Intelligence. Amsterdam u.a.: Elsevier, S. 106-124.
- Turing, Alan M. (1952): The Chemical Basis of Morphogenesis. In: Philosophical Transactions of the Royal Society of London. Series B, Biological Sciences, Vol. 237, No. 641. (Aug. 14, 1952), S. 37-72.
- Turing, Alan M. (1987): Intelligence Service. Schriften. Hg. v. Friedrich Kittler und Bernhard Dotzler. Berlin: Brinkmann & Bose.
- Turing, Alan M. (1987a): Intelligente Maschinen. In: [Turing 1987:81-113].
- Turing, Alan M. (1987b): Spielprogramme. In: [Turing 1987:115-145].
- Turkle, Sherry (1984): Die Wunschmaschine. Vom Entstehen der Computerkultur. Reinbek bei Hamburg: Rowohlt.
- Tyschtschenko, Petro Taras (2014): Meine Erinnerungen an Commodore und Amiga. Rödermark: Power Service.
- U.M. Borghoff . P. Rödig J. Scheffczyk . L. Schmitz (2003): Long-Term Preservation of Digital Documents. Principles and Practices. Heidelberg: Springer.
- Ulmann, Bernd (2010): Analogrechner. Wunderwerke der Technik - Grundlagen, Geschichte und Anwendung. München: Oldenbourg.
- Ulmann, Bernd (2014): AN/FSQ-7: The Computer that shaped the Cold War. München: Oldenbourg.

- Ulmann, Bernd (2017): High-speed Bouncing Ball. In: Analog Paradigm. Analog Computer Applications. Issue #5, 14-JAN-2017, http://analogparadigm.com/downloads/alpaca_5.pdf [letzter Abruf: 27.02.18].
- Van Aken, J. R. (1984): An Efficient Ellipse-Drawing Algorithm. In: IEEE CG&A, Sept. 1984, S. 24-35.
- Van Hulle, D. (2008): Editionswissenschaft. In: Nünning, A. (Hg.): Metzler Lexikon Literatur- und Kulturtheorie. 4. Auflage. Stuttgart: J. B. Metzler, S. 148-150.
- Volder, J. E. (1959): The CORDIC Trigonometric Computing Technique. In: IRE Transactions On Electronic Computers, September 1959, S. 330-334.
- Völz, Horst (2017): Informations- und Speichertheorie. In: Höltgen, Stefan (Hg.): Medientechnisches Wissen. Band 1: Logik, Informations- und Speichertheorie. Berlin u.a.: DeGruyter, S. 150-282.
- von Neumann, John (1966): Theory of Self-Reproducing Automata. Edited and completed by Arthur C. Burks. Urbana/London: Univ. of Illinois Press.
- von Neumann, John (1993): First Draft of a Report on the EDVAC. In: EEE Annals of the History of Computing, Vol. 15, No. 4,1993, S. 27-76.
- Von Samsonow, Elisabeth (2012): Zum Spielzeugstatus zeitgenössischer Apparate. In: Bertz, Peter u.a. (Hgg.): Spielregeln. 25 Aufstellungen. Eine Festschrift für Wolfgang Pircher. Zürich/Berlin: diaphanes, S. 159-171.
- Vorländer, Herward (1990): Mündliches Erfragen von Geschichte. In: Ders. (Hg.): Oral History. Mündlich erfragte Geschichte. Göttigen: Vandenhoek & Ruprecht, S. 7-28.
- Wagner, Christian (2014): Model-Driven Software Migration: A Methodology. Reengineering, Recovery and Modernization of Legacy Systems. Wiesbaden: Springer.
- Wainwright, Robert T. (1974): Life is universal! In: Proceedings of the 7th conference on Winter simulation (Washington, DC – January 14-16, 1974), Volume 2, S. S. 449-459.
- Wallace, James;/Erickson, Jim (1992): Hard Drive: Bill Gates and the Making of the Microsoft Empire. John Wiley & Sons.
- Warning, R. (1979): Rezeptionästhetik als literaturwissenschaftliche Pragmatik. In: Ders. (Hg.): Rezeptionsästhetik. Theorie und Praxis. München: Fink (UTB), S. 9-41.
- Warnke, Martin/Coy, Wolfgang/Tholen, Georg Christoph (Hgg.) (1997): HyperKult. Geschichte, Theorie und Kontext digitaler Medien. Basel: Stroemfeld/Nexus.
- Wegener, I. (1997): Grenzen der berechenbarkeit. In: Rechenberg, P./Pomberger, G. (Hgg.): Informatik-Handbuch. München: Hanser, S. 99-106.
- Weiser, Marc (1991): The Computer for the 21st Century. In: http://www.ubiq.com/hypertext/weiser/SciAmDraft3.html [letzter Abruf: 25.11.2016].

- Weizenbaum, Joseph (2018): ELIZA – Ein Computerprogramm zur Untersuchung der Kommunikation zwischen Mensch und Maschine mithilfe natürlicher Sprache. Übers. v. Jan-Claas van Treeck. In: Höltgen, Stefan/Baranovska, Marianna (2018): Hello, I'm ELIZA. Fünfzig Jahre Gespräche mit Computern. Bochum: Projektverlag, S. 23-52.
- Wertheimer, Max (1923): Untersuchungen zur Lehre von Gestalt II. Berlin: Springer.
- White, Hayden (1991): Auch Klio dichtet oder Die Fiktion des Faktischen. Studien zur Tropologie des historischen Diskurses. Stuttgart: Klett.
- White, Hayden (1991a): Der historische Text als literarisches Kunstwerk. In: Ders. (1991), a. a. O., S. 101-122.
- White, Hayden (1991b): Die Last der Geschichte. In: Ders. (1991), a. a. O., S. 36-63.
- White, Hayden (2015): Metahistory. Die historische Einbildungskraft im 19. Jahrhundert in Europa. Frankfurt am Main: Fischer.
- Wieland, H. R. (2011): Computergeschichte(n) – nicht nur für Geeks. Von Antikythera zur Cloud. Bonn: Galileo Press.
- Wilkes, M. V. (1951): The Best Way to Design an Automated Calculating Machine, Manchester University Computer Inaugural Conf., 1951, S. 16–18 (Nachdruck: Wilkes, M.V.: The Genesis of Microprogramming. In: IEEE Annals of the History of Computing, Band 8, 1986, S. 116-126.)
- Williams, Frederic C. (1960): Apparatus for Storing Trains of Pulses. Patent vom 30.08.1960. In: https://patents.google.com/patent/US2951176 [letzter Abruf: 17.08.2018].
- Williams, M. R.:(1985): Hackers. Heroes of the Computer Revolution. In: Computing Reviews, Review: CR108949. http://www.computingreviews.com/review/review_review.cfm?review_id=108949 [letzter Abruf: 11.01.2019].
- Wiltshire, Alex (2015): Britsoft. An Oral History. Based on the 2014 documentary feature film From Bedroom to Billions. O. O.: Read-Only Memory._Co1o%%US
- Woigk, G. (2012): Das Schneider CPC Systembuch. 4. Version basierend auf dem ursprünglichen Manuskript, mit Änderungen angelehnt an die im Sybex-Verlag veröffentlichte Fassung. https://k1.spdns.de/Vintage/Schneider%20CPC/Das%20Schneider%20CPC%20Systembuch/ [letzter Abruf: 12.02.2018].
- Wölbert, Christian (2014): Handy-Selbstbau-Projekt in Hamburg: bestücken, backen, telefonieren. In: heise online, 19.08.2014, http://www.heise.de/make/meldung/Handy-Selbstbau-Projekt-in-Hamburg-bestuecken-backen-telefonieren-2294509.html [letzter Abruf: 20.04.2016].
- Wolfram, Steven (2001): A New Kind of Science. Champaign: Wolfram Media.

- Wolmeringer, Gottfried (2008): Coding for Fun. IT-Geschichte zum Nachprogrammieren. Bonn: Galileo Press.
- Wozniak, Steve/Smith, Gina (2008): iWoz. Wie ich den Personal Computer erfand und Apple mitgründete. München: dtv.
- Wt (1985): Neues Spiel – neues Glück. In: HC – Mein Home-Computer, Nr. 3 (1985), S. 112-115.
- Wüster, Eugen (1974): Die allgemeine Terminologielehre – Ein Grenzgebiet zwischen Sprachwissenschaft, Logik, Ontologie, Informatik und den Sachwissenschaften. In: Linguistics, Vol. 119, No. 1 (1974), S. 61–106.
- Wüthrich, Hans Jürg (2007): Emulatoren. Wie Computersysteme und Spielkonsolen unsterblich werden. Aktualisierte Neuausgabe. Morschen: Skriptorium.
- Yang, Zijiang/Paradim Joseph C. (2004): DEA Evaluation of a Y2K Software Retrofit Program. In: IEEE TRANSACTIONS ON ENGINEERING MANAGEMENT, VOL. 51, NO. 3, AUGUST 2004, S. 279-387.
- Zacher, Hans J. (1973): Die Hauptschriften zur Dyadik von G. W. Leibniz. Ein Beitrag zur Geschichte des binären Zahlensystems. Frankfurt am Main: Vittorio Klostermann.
- Zaks, Rodnay (1981): Don't! Or How to Car for your Computer. Berkeley u.a.: Sybex.
- Zaks, Rodnay (1986): Programmierung des 6502. 9. Auflage. Düsseldorf: Sybex.
- Zaks, Rodnay (1987): Programmierung des Z80. Düsseldorf: Sybex.
- Zeller, Frauke (2005): Mensch-Roboter Interaktion: Eine sprachwissenschaftliche Perspektive (Diss.) Kassel: Kassel Univ. Press.
- Zhuang, Y./Li, W.-Y./Wang, H./Hong, S./Wang, H. (2017): A Bibliographic Review of Cellular Automaton Publications in Last 50 Years. In: Journal of Cellular Automata, Vol. 12, S. 475–492.
- Zimmermann, Jens (2010): Archäologie (des Wissens). In: Ders./Siegfried Jäger (Hgg.): Lexikon Kritische Diskursanalyse. Eine Werkzeugkiste. Münster: Unrast, S. 26f.2003a
- Zuse, Horst (2011): 70 Jahre Rechner Zuse Z3. Nachbau der Z3 durch Horst Zuse. In: http://www.horst-zuse.homepage.t-online.de/horst-zuse-z3-html/z3-broschuere-mit%20foto-allgemein-2011-1.pdf [letzter Abruf: 25.11.2016].
- Zuse, Konrad (1967): Rechnender Raum. In: Elektronische Datenverarbeitung. 8/1967, S. 336–344.
- Zuse, Konrad (1969): Rechnender Raum. Braunschweig: Vieweg.
- Zuse, Konrad (1975): Gesichtspunkte zur Beurteilung algorithmischer Sprachen. Berichte der Gesellschaft für Mathematik und Datenverarbeitung, Nr. 105. St. Augustin: Gesellschaft für Mathematik und Datenverarbeitung mbH.

Abbildungsverzeichnis

Anhang

A. Programmlisting „16KRA Long Memory Text Programm“ [Processor Technology 1978:A2-5–A2-9]

```
                        1000 *** TEST FOR PROCESSOR TECHNOLOGY 16KRA ***
                        1002 *
                        1004 * COPYRIGHT (C) 1977
                        1006 * SOFTWARE TECHNOLOGY CORPORATION
                        1008 *
                        1010 * THIS VERSION OF THE 16KRA TEST IS WRITTEN
                        1012 * TO BE LOADED INTO RAM AT ADDRESS 0000 .
                        1014 *
                        1016 * THE 16KRA TO BE TESTED MUST BE AT ADDRESS
                        1018 * 1000 HEX. AS A CONTIGUOUS BLOCK OF 16K.
                        1020 *
                        1022 *  THE TEST MAY BE TERMINATED AT ANY TIME BY
                        1024 *  HITTING THE ESCAPE KEY ON THE KEYBOARD.
                        1026 *
                        1028 *  THIS PROGRAM USES SOLOS OR CUTER FOR I/O
                        1030 *  IT MUST BE MODIFIED IF OTHER OPERATING
                        1032 *  SYSTEMS ARE USED.
                        1034 *
                        1036 *
                        1038 *
     0000               1040 INIT    EQU     $       **** INITIALIZATION ****
0000 22 8B 01           1042         SHLD    IOADR   SAVE CALLER'S I/O TABLE ADDRESS
0003 21 00 00           1044         LXI     H,0     CLEAR WORKING STORAGE
0006 22 8F 01           1046         SHLD    COUNT
0009 22 91 01           1048         SHLD    ROW1
000C 22 93 01           1050         SHLD    ROW2
                        1052 *
000F 26 10              1054         MVI     H,10H
0011 22 8D 01           1056         SHLD    BDADR
0014 97                 1058         SUB     A
0015 32 95 01           1060         STA     PAGE    PAGE NUMBER
0018 32 96 01           1062         STA     FILL    STATIC TEST FILLER
                        1064 *
     001B               1066 MAIN    EQU     $       **** MAIN ****
001B 3A 96 01           1068         LDA     FILL    FILL STATIC TEST PAGE
001E 07                 1069         RLC
001F CD 29 01           1070         CALL    WRITE
0022 97                 1072         SUB     A       START WITH 1 BIT PATTERN
0023 37                 1074         STC     .       CARRY HAS THE BIT
                        1076 *
     0024               1078 LOOP1   EQU     $       **** LOOP 1 ****
0024 F5                 1080         PUSH    PSW     SAVE MASTER PATTERN
0025 CD 04 01           1082         CALL    NXTPG   GO PAST STATIC TEST PAGE
0028 1E 03              1084         MVI     E,3     TEST NEXT 3 PAGES
                        1086 *
     002A               1088 TEST1   EQU     $       **** TEST 1 ****
002A CD 1F 01           1090         CALL    TEST    TEST PAGE
002D 1D                 1092         DCR     E       3 PAGES TESTED?
002E C2 2A 00           1094         JNZ     TEST1   NO, DO NEXT ONE
                        1096 *
0031 3A 96 01           1098         LDA     FILL    CHECK STATIC TEST PAGE
0034 07                 1099         RLC
0035 CD 42 01           1100         CALL    READ    FOR DROPPED BITS
                        1102 *
0038 F1                 1104         POP     PSW     RESTORE MASTER PATTERN
0039 1F                 1106         RAR     .       PERMUTE
003A D2 24 00           1108         JNC     LOOP1   REPEAT UNTIL CARRY COMES AROUND
                        1110 *
```

```
003D BF                 1112            CMP     A       CLEAR CARRY
003E 3E FF              1114            MVI     A,0FFH  7 BIT PATTERN
                        1116 *
     0040               1118 LOOP2      EQU     $       **** LOOP 2 ****
0040 F5                 1120            PUSH    PSW     SAVE MASTER PATTERN
0041 CD 04 01           1122            CALL    NXTPG   SKIP PAST STATIC TEST PAGE
0044 1E 03              1124            MVI     E,3     TEST REMAINING 3 PAGES
                        1126 *
     0046               1128 TEST2      EQU     $
0046 CD 1F 01           1130            CALL    TEST    TEST PAGE
0049 1D                 1132            DCR     E       3 PAGES TESTED?
004A C2 46 00           1134            JNZ     TEST2   NO, DO NEXT ONE
                        1136 *
004D 3A 96 01           1138            LDA     FILL    CHECK STATIC TEST PAGE
0050 07                 1139            RLC
0051 CD 42 01           1140            CALL    READ    FOR DROPPED BITS
                        1142 *
0054 F1                 1144            POP     PSW     RESTORE MASTER PATTERN
0055 1F                 1146            RAR     .       PERMUTE
0056 DA 40 00           1148            JC      LOOP2   REPEAT UNTIL CARRY COMES AROUND
                        1150 *
0059 CD 04 01           1152            CALL    NXTPG   REPEAT ENTIRE TEST
005C 3A 95 01           1154            LDA     PAGE    STARTING WITH
005F B7                 1156            ORA     A       NEXT PAGE IF WE HAVEN'T
0060 C2 1B 00           1158            JNZ     MAIN    BEEN AROUND 4 TIMES ALREADY
                        1160 *
0063 3A 96 01           1162            LDA     FILL    INVERT FILLER
0066 2F                 1164            CMA     .
0067 32 96 01           1166            STA     FILL    AND TEST AGAIN
006A B7                 1168            ORA     A       WITH COMPLIMENT
006B C2 1B 00           1170            JNZ     MAIN    UNLESS ALREADY DONE
                        1172 *
006E CD 7A 00           1174            CALL    MAP     OUTPUT CHIP MAP
0071 21 79 01           1176            LXI     H,CMPLT  'COMPLETED'
0074 CD F7 00           1178            CALL    STRNG   OUTPUT LINE
0077 C3 1B 00           1180            JMP     MAIN    AND CONTINUE TEST
                        1182 *
     007A               1184 MAP        EQU     $       **** MAP ****
007A CD BF 00           1186            CALL    CRLF
007D 2A 91 01           1188            LHLD    ROW1    DISPLAY CHIPS IN ROW 1
0080 CD 8A 00           1190            CALL    LINE    FORMAT THE LINE
0083 2A 93 01           1192            LHLD    ROW2    DISPLAY CHIPS IN ROW 2
0086 CD 8A 00           1194            CALL    LINE    FORMAT THE LINE
0089 C9                 1196            RET     .       RETURN
                        1198 *
     008A               1200 LINE       EQU     $       **** LINE ****
008A 16 04              1202            MVI     D,4     # OF BITS PER QUADRANT
008C 1E 02              1204            MVI     E,2     # OF ROWS
                        1206 *
     008E               1208 QUAD       EQU     $       **** QUAD ****
008E 7D                 1210            MOV     A,L     PAGE 0 OR 2
008F 1F                 1212            RAR     .       CARRY MEANS CHIP HAD ERRORS
0090 6F                 1214            MOV     L,A     REMAINING BITS GO BACK
0091 CD B3 00           1216            CALL    CHIP    DISPLAY CHIP STATUS
                        1218 *
0094 7C                 1220            MOV     A,H     PAGE 1 OR 3
0095 1F                 1222            RAR     .       TEST BIT,  CARRY IS N.G.
0096 67                 1224            MOV     H,A     RETURN THE REST
0097 CD B3 00           1226            CALL    CHIP    DISPLAY CHIP STATUS
```

```
                        1228 *
009A CD AE 00           1230          CALL    SPACE   FOR READABILITY
009D 15                 1232          DCR     D       QUADRANT DONE?
009E C2 8E 00           1234          JNZ     QUAD    NO
                        1236 *
00A1 16 04              1238          MVI     D,4     YES, RESTORE CHIP COUNT
00A3 CD AE 00           1240          CALL    SPACE   SEPARATE QUADRANTS
00A6 1D                 1242          DCR     E       IS LINE DONE?
00A7 C2 8E 00           1244          JNZ     QUAD    NO, FORMAT OTHER QUADRANT
00AA CD BF 00           1246          CALL    CRLF    LINE IS DONE
00AD C9                 1248          RET     .       RETURN
                        1250 *
     00AE               1252 SPACE     EQU     $       **** SPACE ****
00AE 3E 20              1254          MVI     A,' '   WRITE A SPACE
00B0 C3 BA 00           1256          JMP     MARK
                        1258 *
     00B3               1260 CHIP      EQU     $       **** CHIP ****
00B3 3E 47              1262          MVI     A,'G'   MARK CHIP 'G'
00B5 D2 BA 00           1264          JNC     MARK    IT'S OK, ELSE
00B8 3E 58              1266          MVI     A,'X'   MARK CHIP 'X'
                        1268 *
00BA CD CA 00           1270 MARK     CALL    PUT     OUTPUT MARK
00BD BF                 1272          CMP     A       CLEAR CARRY BIT
00BE C9                 1274          RET     .       RETURN
                        1276 *
     00BF               1278 CRLF      EQU     $       **** CRLF ****
00BF 3E 0D              1280          MVI     A,0DH   OUTPUT CARRIAGE RETURN
00C1 CD CA 00           1282          CALL    PUT
00C4 3E 0A              1284          MVI     A,0AH   FOLLOWED BY A LINE FEED
00C6 CD CA 00           1286          CALL    PUT
00C9 C9                 1288          RET     .       AND RETURN
                        1290 *
     00CA               1292 PUT       EQU     $       **** PUT ****
00CA E5                 1294          PUSH    H       SAVE
00CB 01 19 00           1296          LXI     B,SOUT   OUTPUT ROUTINE JUMP LOCATION
00CE 2A 8B 01           1298          LHLD    IOADR   ADDRESS OF 'CUTER'/'SOLOS'
00D1 09                 1300          DAD     B       FORM TRUE ADDRESS
00D2 47                 1302          MOV     B,A     CHARACTER TO O/P IN B
00D3 E3                 1304          XTHL    .       RESTORE H
00D4 C9                 1306          RET     .       DESTINATION IS ON TOP OF STACK
                        1308 *
     00D5               1310 GET       EQU     $       **** GET ****
00D5 01 E3 00           1312          LXI     B,CHECK  RETURN ADDRESS
00D8 C5                 1314          PUSH    B       PUT ON STACK
00D9 E5                 1316          PUSH    H       SAVE
00DA 01 1F 00           1318          LXI     B,SINP   INPUT ROUTINE JUMP LOCATION
00DD 2A 8B 01           1320          LHLD    IOADR   ADDRESS OF 'CUTER'/'SOLOS'
00E0 09                 1322          DAD     B       FORM TRUE ADDRESS
00E1 E3                 1324          XTHL    .       RESTORE H
00E2 C9                 1326          RET     .       DESTINATION IS TOS, RETURNS TO CHECK
                        1328 *
     00E3               1330 CHECK     EQU     $       **** CHECK ****
00E3 FE 1B              1332          CPI     1BH     ESCAPE KEY?
00E5 C0                 1334          RNZ     .       NO, CONTINUE TESTING
                        1336 *
     00E6               1338 ABORT     EQU     $       **** ABORT ****
00E6 CD 7A 00           1340          CALL    MAP     OUTPUT WHAT WE'VE GOT SO FAR
00E9 21 83 01           1342          LXI     H,TERM   'ABORTED'
00EC CD F7 00           1344          CALL    STRNG   OUTPUT LINE
```

https://doi.org/10.18452/21165

```
00EF 2A 8B 01          1346          LHLD    IOADR   ADDRESS OF 'SOLOS'/'CUTER'
00F2 23                1348          INX     H       BUMP TO RETURN TO COMMAND PROCESSO
00F3 23                1350          INX     H
00F4 23                1352          INX     H
00F5 23                1354          INX     H
00F6 E9                1356          PCHL    .       EXIT TO OUR CALLER
                       1358 *
     00F7              1360 STRNG    EQU     $       **** STRING ****
00F7 7E                1362          MOV     A,M     GET CHARACTER FROM STRING
00F8 23                1364          INX     H       BUMP STRING POINTER
00F9 FE 0D             1366          CPI     0DH     IS IT CR?
00FB CA BF 00          1368          JZ      CRLF    YES, END OF STRING
00FE CD CA 00          1370          CALL    PUT     NO, OUTPUT CHARACTER
0101 C3 F7 00          1372          JMP     STRNG   CONTINUE
                       1374 *
     0104              1376 NXTPG    EQU     $       **** NEXT PAGE ****
0104 F5                1378          PUSH    PSW     SAVE
0105 CD D5 00          1380          CALL    GET     LOOK FOR 'ESCAPE' KEY
0108 3A 95 01          1382          LDA     PAGE    GET CURRENT PAGE NUMBER
010B C6 10             1384          ADI     10H     ADD 4K
010D E6 30             1386          ANI     30H     WRAP AROUND
010F 32 95 01          1388          STA     PAGE    SAVE
0112 F1                1390          POP     PSW     RESTORE
0113 C9                1392          RET     .       AND RETURN
                       1394 *
     0114              1396 GETPG    EQU     $       **** GET PAGE ****
0114 F5                1398          PUSH    PSW     SAVE
0115 3A 95 01          1400          LDA     PAGE    GET PAGE NUMBER
0118 2A 8D 01          1402          LHLD    BDADR   BOARD ADDRESS
011B 84                1404          ADD     H       ADD PAGE #
011C 67                1406          MOV     H,A     SET PAGE ADDRESS
011D F1                1408          POP     PSW     RESTORE
011E C9                1410          RET     .       RETURN
                       1412 *
     011F              1414 TEST     EQU     $       **** TEST ****
011F CD 29 01          1416          CALL    WRITE   WRITE TEST PATTERN
0122 CD 42 01          1418          CALL    READ    AND READ IT BACK
0125 CD 04 01          1420          CALL    NXTPG   BUMP PASE POINTER
0128 C9                1422          RET     .       THEN RETURN
                       1424 *
     0129              1426 WRITE    EQU     $       **** WRITE ****
0129 F5                1428          PUSH    PSW     SAVE
012A CD 14 01          1430          CALL    GETPG   GET PROPER HL
012D 16 10             1432          MVI     D,10H   COUNT 4K
                       1434 *
     012F              1436 WRIT1    EQU     $       **** WRITE 1 ****
012F F5                1438          PUSH    PSW     SAVE WORKING PATTERN
0130 77                1440          MOV     M,A     TRY TO STORE
0131 AE                1442          XRA     M       IS DATA GOOD?
0132 C4 5A 01          1444          CNZ     BITER   RECORD BIT IF NOT
0135 F1                1446          POP     PSW     RESTORE PATTERN
0136 17                1448          RAL     .       PERMUTE
0137 2C                1450          INR     L       BUMP STORAGE ADDRESS
0138 C2 2F 01          1452          JNZ     WRIT1
013B 24                1454          INR     H       BUMP BY 256
013C 15                1456          DCR     D       ENOUGH FOR 4K
013D C2 2F 01          1458          JNZ     WRIT1
0140 F1                1460          POP     PSW     RESTORE
0141 C9                1462          RET     .       AND RETURN
```

https://doi.org/10.18452/21165

```
                        1464 *
     0142               1466 READ   EQU   $       **** READ ****
0142 F5                 1468        PUSH  PSW     SAVE
0143 CD 14 01           1470        CALL  GETPG   GET PROPER HL
0146 16 10              1472        MVI   D,10H   COUNT 4K
                        1474 *
     0148               1476 READ1  EQU   $       **** READ 1 ****
0148 F5                 1478        PUSH  PSW     SAVE WORKING PATTERN
0149 AE                 1480        XRA   M       IS DATA STILL GOOD ?
014A C4 5A 01           1482        CNZ   BITER   ACCUMULATE ERRORS
014D F1                 1484        POP   PSW     RESTORE PATTERN
014E 17                 1486        RAL   .       PERMUTE
014F 2C                 1488        INR   L       BUMP STORAGE ADDRESS
0150 C2 48 01           1490        JNZ   READ1
0153 24                 1492        INR   H       BUMP BY 256
0154 15                 1494        DCR   D       ENOUGH FOR 4K
0155 C2 48 01           1496        JNZ   READ1
0158 F1                 1498        POP   PSW     RESTORE
0159 C9                 1500        RET   .       AND RETURN
                        1502 *
     015A               1504 BITER  EQU   $       **** BIT ERROR ****
015A E5                 1506        PUSH  H       SAVE REGS
015B 47                 1508        MOV   B,A     ERROR BIT
015C 21 91 01           1510        LXI   H,BITS  ERROR BIT TABLE
015F 3A 95 01           1512        LDA   PAGE    GET CURRENT PAGE
0162 07                 1514        RLC   .       SHIFT TO
0163 07                 1516        RLC   .       LOW ORDER
0164 07                 1518        RLC   .       TWO BITS
0165 07                 1520        RLC   .
0166 85                 1522        ADD   L       DISPLACE BY PAGE #
0167 6F                 1524        MOV   L,A     INTO BIT TABLE
0168 7E                 1526        MOV   A,M     GET BITS ACCUMULATED SO FAR
0169 B0                 1528        ORA   B       ADD NEW ONES
016A 77                 1530        MOV   M,A     AND PUT IN TABLE
                        1532 *
016B 2A 8F 01           1534        LHLD  COUNT   ERROR COUNT
016E 23                 1536        INX   H       BUMP
016F 22 8F 01           1538        SHLD  COUNT
0172 7C                 1540        MOV   A,H     HAS COUNT
0173 B5                 1542        ORA   L       GONE AROUND TO 0?
0174 CA E6 00           1544        JZ    ABORT   YES, TERMINATE TEST
0177 E1                 1546        POP   H       RESTORE
0178 C9                 1548        RET   .       AND RETURN TO TEST
                        1550 *
0179 43 4F 4D 50        1552 CMPLT  ASC   'COMPLETED'
     4C 45 54 45
     44
0182 0D                 1554        DB    0DH
0183 41 42 4F 52        1556 TERM   ASC   'ABORTED'
     54 45 44
018A 0D                 1558        DB    0DH
                        1560 *
     0019               1562 SOUT   EQU   19H     DISPLACEMENT TO JUMP
     001F               1564 SINP   EQU   1FH     DISPLACEMENT TO JUMP
                        1566 *
     018B               1568 RAM    EQU   $       DEFINE WRITABLE STORAGE AREA
                        1570 *
018B                    1572 IOADR  DS    2       ADDRESS OF CALLER'S JUMP TABLE
018D                    1574 BDADR  DS    2       ADDRESS OF 16KRA UNDER TEST
018F                    1576 COUNT  DS    2       ERROR COUNT
     0191               1578 BITS   EQU   $       CHIP MAP, MUST NOT CROSS 256 BYTE BOUNDR
0191                    1580 ROW1   DS    2       BIT MAP FOR BITS IN ROW 1
0193                    1582 ROW2   DS    2       BIT MAP FOR BITS IN ROW 2
0195                    1584 PAGE   DS    1       CURRENT PAGE
0196                    1586 FILL   DS    1       STATIC TEST BYTE
```

B. Interview mit Marius Groth[186]

Stefan Höltgen: Hast du das Programm (vgl. Anhang A), das du eingetippt hast, oder Teile davon verstanden? Soweit ich weiß, programmierst du Assembler ja nicht selbst, aber hast du während oder nach der Eingabe ein Teilverständnis des Programms bekommen?

Marius Groth: Nicht wirklich ... ich habe zwar ab einem bestimmten Punkt verstanden, welche Hexcodes für „bekannte“ Anweisungen (NOP, JMP) stehen und wie ASCII-Zeichen eingebunden werden, aber vom eigentlichen Ablauf habe ich kein Verständnis entwickelt (ich könnte dir nicht einmal sagen wie genau das RAM getestet wird [also ob ein fixer oder zufälliger Wert geschrieben oder dieser direkt oder verzögert getestet wird]).

Stefan Höltgen: Seit wann reparierst du historische Computer? Wie hat sich das entwickelt?

Marius Groth: Das Hobby Retrocomputing hat bei mir eigentlich unmittelbar mit der Notwendigkeit von Reparaturen angefangen, bei den ersten 2 C64 die ich mir 2006 gekauft habe war einer defekt, den habe ich dann (erfolglos) versucht mit meinem Vater zu reparieren. Später folgte dann ein ZX Pentagon 1024SL 2.2 aus Russland den es nur als Bausatz gab - mit ordentlich Respekt habe ich mich dann daran getraut - ungewiss ob das Vorhaben den Rechner zusammen zu bauen funktionieren wird. Löten hatte ich zwar kurz und auf einfachstem Niveau im Rahmen meiner Ausbildung gelernt, aber wirklich in mein Können vertraut habe ich nicht (und in der Ausbildung wars auch eher nervig). Der Rechner lief natürlich nicht komplett auf Anhieb, gab aber zumindest ein Bild aus, so dass Fehlersuche angesagt war - ein paar Tage später lief er dann auch und ich stellte fest, dass das Spaß macht. Inzwischen hat sich daraus ein richtiges Hobby entwickelt, da mir das Handwerk spaß macht und ich inzwischen aus Überzeugung lieber kaputte Rechner kaufe als funktionale teuer zu bezahlen bzw. fremde Rechner repariere als eine Empfehlung auszusprechen etwas neu zu kaufen.

Stefan Höltgen: Welche Geräte reparierst du am liebsten und warum?

Marius Groth: Am liebsten sind mir entweder populäre 8Bit Computer der 80er mit mittlerem Integrationsniveau (C64, ZX Spectrum, Atari XL, Oric, TI-99, etc.) oder obskure Geräte von denen es nur Wenige gibt. Die Motivation ist bei beiden Schwerpunkten sehr unterschiedlich. Die 8-Bit-Mikrocomputer sind in ihrer Bauart noch sehr gut nachvollziehbar, von der Komplexität überschaubar (im Gegensatz zu früheren überwiegend TTL-basierten System) und meist sehr gut dokumentiert oder befinden sich im Besitz von vielen anderen Hobbykollegen was einen Austausch von Erfahrungen ermöglicht. Seltene/obskure Geräte sind eher Herausforderungen, weil der Austausch wegfällt bzw. man teilweise „Pionierarbeit“ tätigt, weil Dokumentationen fehlen.

186 E-Mails vom 24.01.2017 und 01.02.2018

Stefan Höltgen: Welche Wissensquellen nutzt du insgesamt (Gern eine Liste mit absteigender Priorität)?

Marius Groth: Erfahrungen aus eigenen/fremden vollzogenen Reparaturen, Schaltpläne/ Datenblätter vom Hersteller (über's Internet, selten über physikalische, historische Ausdrucke), Internetforen zwecks Austausch, Persönlicher Kontakt mit anderen Hobbykollegen, Eigenes herleiten von Funktionsweisen/Elektrotechnisches Wissen aus der Ausbildung.

Stefan Höltgen: Wie gehst du mit Informationen um, die du nicht verstehst (etwa Schaltpläne, Software, ...) aber für deine Reparatur benötigst?

Marius Groth: Bei Informationen die ich nicht verstehe versuche ich entweder diese mir selbst anzueignen, sofern mich diese persönlich interessieren bzw. dies in einem angemessenen Zeitrahmen möglich ist oder kann inzwischen auf einen „Pool“ an Kontakten zurückgreifen die andere Schwerpunkte haben und mir jene Informationen verständlich/ergebnisorientiert erklären können.

Stefan Höltgen: Bei welchen Schäden/Defekten hast du eine Reparatur abgelehnt oder abgebrochen, weil sie sich nicht gelohnt hat oder unmöglich war?

Marius Groth: Prinzipiell lehne ich fast jede „größere“ Reparatur an Monitoren ab, da ich von analoger Videotechnik kaum Ahnung habe und dort mit Stromstärken gearbeitet wird die lebensgefährlich sind (Bildjustierung und digitale Spannungsversorgung ist das Maximale, was ich dort mache). Ich weiss zwar wie ich dort Gefahren umgehen kann bzw. was ich in keinem Fall tun sollte - das sind aber definitiv Arbeiten die ich lieber anderen überlasse die davon Ahnung haben statt mich durch Unwissenheit Gefahren auszusetzen oder das Gerät weiter zu beschädigen. Eine Reparatur die ich abgebrochen habe war das LCD-Display eines Taschenrechners, da dort zum einen Bauteile verklebt waren und diese außerhalb von Reinräumen nur mit extrem hohem Zeitaufwand, nicht vorhandenem Spezialwissen und Spezialwerkzeug eine Reparatur möglich gewesen wäre - in dem Fall wäre es sinnvoller gewesen einen weiteren, defekten Taschenrechner als Ersatzteilspender zu nehmen. Insgesamt sind Reparaturen mit Spezialbauteilen die kaum noch beschaffbar sind immer sehr schwierig, teilweise kann man zwar improvisieren und es besteht zwar die Möglichkeit digitale Schaltungen nachzubauen, letzteres umfasst aber meistens ein Zeitvolumen von mehreren 100 Stunden und erfordert Kompetenzen die ich nicht habe (und bislang auch noch nicht die absolute Notwendigkeit bestand mir diese anzueignen).

Stefan Höltgen: Hast du ein Ziel, das du aus deiner Reparatur-Tätigkeit ansteuerst? (Geldverdienen, persönlicher Erkenntnisgewinn oder soziale Kontakte, ...) Und welche(n) Nutzen ziehst du aus der Reparatur von historischen Computern?

Marius Groth: Meine Motivation lässt sich in 2 Punkten zusammenfassen: Einerseits gehts mir darum, dass die Computer weiterhin erlebbar bleiben, Gehäuse mit totem Innenleben

sind langweilig - da ist es mir lieber zu sehen, dass Menschen an den Computern weiterhin Spaß haben (spielen, Nostalgie) oder für zeitgenössische kreative Zwecke nutzen und Grenzen des technisch machbaren neu ausloten, an deren Ergebnissen ich dann wieder Spaß habe (Demoscene, Chiptunes). Zum anderen stelle ich inzwischen zunehmend fest, dass der Markt für klassische Computer zunehmend als Wertanlage missbraucht wird (Preise werden in die Höhe getrieben, Rechner in Vitrinen/Schließfächern verstaut) - mit meinen Reparaturen möchte ich den Markt ein wenig entzerren bzw. ein Bewusstsein entwickeln, dass auch defekte Maschinen alles andere als schrottreif sind und sich die Materialkosten in sehr vielen Fällen im Cent bis wenige Euro-Bereich bewegt und damit ein Bewusstsein für ein gerne übersehenes Marktsegment erzeugen. Klassische Computer lassen sich im Gegensatz zu aktuellen Computern nämlich noch reparieren und sind häufig auch „nur“ Massenprodukte.

Marius Groth: Geld ist natürlich immer ein nettes Beiwerk, allerdings nicht ausreichend um damit den Lebensunterhalt bestreiten zu können - würde mir der Reparaturprozess selbst keinen Spaß machen, würde ich dies vermutlich auch nicht gegen Geld machen. Sich neues Wissen zu erarbeiten und auch anwenden zu können ist natürlich auch interessant und bestärkend, aber eher ein Mittel zum Zweck.

C. Sourcecode des Programms Game of Memories

```
; Game of Memories V1.0
; Stefan Höltgen & Thomas Nückel, 2015
; Sprache: Assembler 2650
;
; Beschreibung:
;
; Dieses Programm ist ein Game of Life nach dem Regelsatz von John Horton Conway.
; Es wurde für den Emulator des Signetics Instructor 50 geschrieben. Die dabei
; verwendete Simulationssoftware ist Winarcadia 22.74. Da der Instructor 50 nur über
; eine rudimentäre Ausgabe verfügt, wurde das Spielfeld für das Game of Life
; in einen 16x16 Byte großen Speicherbereich gelegt, den der Emulator graphisch
; darstellen und so dem Betrachter zugänglich machen kann - wodurch sich der Name des
; Programms erklärt: Game of Memories.
;
; ############################ Technische Erläuterungen ##############################
;
;
; Benutzte Variablen:
;
; R0 = Inhaltsregister für geladene Werte (Feldwert)
; R1 = Adressregister zur indexierten Ansteuerung von Adressen in den Feldern
; R2 = Register für Spielarithmetik (Umfeldsumme)
; R3 = Allzweckregister, u. a. zum Zwischenspeichern
; R2 und R3 werden außerdem als Register für die Warteschleife verwendet
;
; Vom Programm benutzte Adressen:
;
; 0000h-00FFh = Adressen für das Spielfeld
; 0E00h-0EFFh = Adressen für das Arbeitsfeld
; 0F00h-0F07h = Adressen für Arithmetik-Memory
; 0F09h-0F0Ah = Adressen für Generations-Zahl (dezimal)
; Kommentar: Für jedes Game of Life sind - solange kein anderer Buffer verwendet
; wird - zwei Felder notwendig. Bei Verwendung von nur einem Feld würden sonst
; Zellen überschrieben werden, die noch für die weitere Analyse notwendig sind.
; Deswegen gibt es zwei Felder: Ein Spiel- und ein Arbeitsfeld.
;
; Programmstruktur (mit Labeln und Adressen):
;
```

```
; INIT (1780h): Initialisierung von Spiel- & Arbeitsfeld
;       CPA2B   (178Bh): Kopieren ...
;       17PROOF (179Dh): ... des Spielfeldes ...
;       A2BEND  (17A2h): ... ins Arbeitsfeld.
; MAIN (17A5h): Hauptschleife
;       ANAANF (17ACh): Analyse des Spielfeldes
;               SOND (0180h): Test der Spielfeldecken und -ränder
;                       LOE (02D0h): Sonderfall: Linke obere Ecke
;                       ROE (02F0h): Sonderfall: Rechte obere Ecke
;                       LUE (0310h): Sonderfall: Linke untere Ecke
;                       RUE (0330h): Sonderfall: Rechte untere Ecke
;                       RO  (0350h): Sonderfall: Oberer Rand
;                       RU  (0380h): Sonderfall: Unterer Rand
;                       RR  (03E0h): Sonderfall: Rechter Rand
;                       LR  (03B0h): Sonderfall: Linker Rand
;               NORM (0260h): Adress-Arithmetik (Analyse der jeweiligen Nachbarzellen)
;       REG (0410h): Regeln anwenden
;               REG1 (0413h): Regel 1
;               REG2 (0425h): Regel 2
;               REG3 (0432h): Regel 3
;       REGEND (043C): Ende der Regelprüfung
; LOOPEND (043Fh): Ende der Hauptschleife
; Subroutinen:
;       ADDFLD (02B0h): Subroutine: Addition der Umfelder
;       CLRART (0150h): Subroutine: Löschen des Arithmetikfeldes
;       CLRFLD (0102h): Löschen von Arbeits- und Spielfeld.
;       SEED   (0120h): Spielfeld ...
;               LOOP  (0126h): ... zufällig ...
;               NOERR (0131h): ... mit Zellen füllen.
;
; ################################ Der Programmcode ##################################
;
;ADR    LABEL           MNEMONICS              OPCODES                KOMMENTAR
;
,
; INITIALISIERUNG:
;
1780    INIT            BCTA,UN CLRFLD 1F 01 02                       ; Springe zu Subroutine
CLRFLD
1783                    BCTA,UN SEED   1F 01 20                       ; Springe zu Subroutine
SEED
1786                    PPSL 20                77 20                          ; Compare-Bit auf
logisch stellen
1788                    BCTA,UN CPA2B  1F 17 8B                       ; Springe zu CPA2B
; Kopie des Spielfeldes erzeugen:
178B    CPA2B           LODI,R1 00             05 00                          ; Lade R1 mit 0
178D                    LODA,R1 0900   0D 6E 00                       ; Kopiere den Wert aus der
R1-ten
; Zelle des Spielfelds nach R0:
1790                    COMI,R0 15             E4 15                          ; Ist R0 gleich
15h?
1792                    BCFA,2 17PROOF 9C 17 9D                       ; Nein, dann springe nach
17PROOF
1795                    LODI,R0 00             04 01                          ; Ja, dann lade R0
mit 1 und ...
1797                    STRA,R1 6000   CD 60 00                       ; ... besetze das R1-te
Feld auf dem
                                                                              ; ... Arbeitsfeld
mit dem Wert aus R0 ...
179A                    BCTA,UN A2BEND 1F 17 A2                       ; ... springe nach A2BEND
179D    17PROOF         LODI,R0 01             04 00                          ; Lade R0 mit 0 und
...
179F                    STRA,R1 6000   CD 60 00                       ; ... besetze das R1-te
Feld auf dem
                                                                              ; ... Arbeitsfeld
mit dem Wert aus R0
17A2    A2BEND  BIRA,R1 CPA2B  DD 17 8D                     ; Inkrementiere R1 und springe
nach 178D,
                                                                              ; solange, bis R1
gleich 0 geworden ist.
                                                                              ; Dann weiter.
;
; HAUPTSCHLEIFE:
;
```

```
17A5    MAIN            BCTR,UN CLRART 1F 01 50                     ; Springe zu Subroutine
CLRART
17A8                    LODI,R0 00             04 00                        ; Lade R0 mit 0
(Arbeitsfeldadresse)
17AA                    LODI,R2 00             06 00                        ; Lade R2 mit 0
(für Spielarithmetik)
;
; Feld analysieren:
;
17AC    ANAANF  WRTD,R0         F1                              ; Analysefortschritt auf LEDs
ausgeben
17AD                    BCTA,UN SOND   1F 01 7E                     ; Springe zur Analyse der
Felder
;
; Die normale Spielfeldarithmetik prüft alle Zellen der Moore-Nachbarschaft um die für
; die nächste Generation zu bestimmende Zelle herum - also 8 Zellen, vier orthogonal und vier
; diagonal Anliegende. Das würde aber im Falle einer Eck- oder Randzelle zu falschen
; Ergebnissen führen.
; Darum wird im Folgenden getestet, was für eine Zelle konkret vorliegt, um zu erfahren,
; wieviele Nachbarzellen sie hat und wo diese von ihr aus gesehen liegen.
; Dabei wird zuerst geprüft, ob es sich um eine der 4 Spielfeldecken handelt (die jeweils nur
; 3 Nachbarfelder haben), und wenn nicht, dann, ob die Zelle am Spielfeldrand liegt
; (und damit nur 5 Nachbarn hat). Durch diese Reihenfolge wird auch vermieden, dass
; eine Spielfeldecke fälschlich als Randzelle behandelt wird.
; Handelt es sich auch nicht um eine Randzelle, dann bleibt nur noch, dass es sich um eine
; normale Zelle aus dem Spielfeld handelt.
;
; Auf Spielfeld-Ecken testen:
017E                    PPSL 02        77 02                        ; PPSL umstellen
0180    SOND            COMI,R1 00             E5 00                        ; Wenn linke obere
Ecke (wenn R1=00) ...
0182                    BCTA,0 LOE             1C 02 D0                     ; ... dann springe
nach LOE
0185                    COMI,R1 0F             E5 0F                        ; Wenn rechte obere
Ecke (wenn R1=0F) ...
0187                    BCTA,0 ROE             1C 02 F0                     ; ... dann springe
nach ROE
018A                    COMI,R1 F0             E5 F0                        ; Wenn linke untere
Ecke (wenn R1=F0) ...
018C                    BCTA,0 LUE             1C 03 10                     ; ... dann springe
nach LUE
018F                    COMI,R1 FF             E5 FF                        ; Wenn rechte
untere Ecke (wenn R1=FF) ...
0191                    BCTA,0 RUE             1C 03 30                     ; ... dann springe
nach RUE
0194                    BCTA,UN        1F 02 3A                     ; Wenn keine Ecke, springe
nach Rand
                                                                    ;       testen.
;
; Wenn es für die hier zu bestimmende Zelle bis jetzt keine Übereinstimmung gegeben hat, dann
; ist sie keine Eckzelle. Sie kann also nur noch eine Rand- oder eine normale Zelle im Feld
sein.
; Als nächstes wird darum geprüft, ob es sich bei ihr um eine Randzelle handelt.
;
; Auf Spielfeld-Rand testen:
023A                    COMI,R1 10             E5 10                        ; Wenn oberer Rand
(wenn R1 < 10) ...
023C                    BCTA,2 RO              1E 03 50                     ; ... dann springe
nach RO
023F                    COMI,R1 EF             E5 EF                        ; Wenn unterer Rand
(wenn R1 > EF) ...
0241                    BCTA,1 RU              1D 03 80                     ; ... dann springe
nach RU
0244                    TMI,R1 0F              F5 0F                        ; Wenn rechter Rand
(wenn R1=xF
                                                                    ; (xxxx1111)) ...
0246                    BCTA,0 RR              1C 03 E0                     ; ... dann springe
nach RR
0249                    LODZ,R1        01                           ; Kopiere den Wert aus R1
nach R0 ...
024A                    STRZ,3         C3                           ; ... Kopiere den Wert aus
R0 nach R3 ...
024B                    ADDI,R3 0F             87 0F                        ; ... Addiere 0F zu
dem Wert in R3 ...
```

```
024D                TMI,R3 0F           F7 0F                       ; ... ist R3 jetzt
gleich 0F, dann linker
                                                                    ; Rand ...
024F                BCTA,0 LR           1C 03 B0                    ; ... dann springe
nach LR
0252                BCTA,UN NORM   1F 02 60                  ; Wenn kein Rand, springe
nach Normalfeld
                                                                    ; auslesen.
;
; Wenn es auch bis hierhin keine Übereinstimmung für die zu bestimmende Zelle gegeben hat,
dann ist
; sie auch keine Randzelle. Sie kann also nur noch eine normale Zelle im Feld sein. Eine
weitere
; Prüfung ist daher nicht notwendig, sondern es kann gleich zur Adress-Arithmetik übergegangen
; werden.
;
;
; ADRESS-ARITHMETIK:
;
; Die arithmetischen Beziehungen zwischen der gerade untersuchten Zelle X und ihren
Nachbarzellen
; stellen sich wie folgt dar (innerhalb der Nachbarzellen ist der relative Abstand zu X
angegeben;
; außerhalb der relative Abstand der Nachbarzellen zueinander):
;
;             +1/1h  +1/1h
;             ---->  ---->
;        +------+------+------+
;        | -17/ | -16/ | -15/ |
; -16/ ^ | -11h | -10h | -Fh  | | +16/
; -10h | +------+------+------+ | +10h
;      | | -1/  |  X   |  +1/ | v
; -16/ ^ | -1h  |      |  +1h | | +16/
; -10h | +------+------+------+ | +10h
;      | | +15/ | +16/ | +17/ | v
;        | +Fh  | +10h | +11h |
;        +------+------+------+
;            <----   <----
;            -1/-1h  -1/-1h
;
; Für die Sonderfälle der Eck- und Randzellen gelten die selben Abstände, nur dass diese
; entsprechend weniger Nachbarn aufweisen. Im Falle der vier Eckzellen jeweils drei Nachbarn,
im
; Falle er 56 Randzellen fünf Nachbarn und acht bei den normalen Zellen im Spielfeld.
;
; Auf Basis dieser Überlegungen können nun die Nachbarn der für die nächste Generation
; zu bestimmenden Zelle angesteuert und ihre Werte für die spätere Verwendung gespeichert
werden.
; Dabei muss es für jede der 4 Eckzellen, für jeden der 4 Ränder, und für die normalen Zellen
; im Feld eine eigene Adressarithmetik geben, im Ganzen also 9 Adressarithmetiken.
;
; Normalfeld-Adressearithmetik:
; Summe für normale Zelle im Feld erzeugen:
0260    NORM        LODZ,R1         01                          ; Kopiere den Wert aus R1
nach R0
0261                STRZ,R3         C3                          ; Kopiere den Wert aus R0
nach R3
0262                SUBI,R1 01          A5 01                       ; R1 = R1 - 1 (für
Zelle links)
0264                LODA,R1 0A 00   0D 60 00                    ; Zellwert aus 0000
indiziert über R1 in
                                                                    ; R0 laden
0267                STRA,R0 0F 00   CC 0F 00                    ; R0 in 0F00h speichern
026A                SUBI,R1 01          A5 10                       ; R1 = R1 - 10 (für
Zelle links oben)
026C                LODA,R1 0A 00   0D 60 00                    ; Zellwert aus 0000
indiziert über R1 in
                                                                    ; R0 laden
026F                STRA,R0 0F 01   CC 0F 01                    ; ... in 0F01h speichern
0272                SUBI,R1 11          85 01                       ; R1 = R1 + 1 (für
Zelle oben)
0274                LODA,R1 0A 00   0D 60 00                    ; Zellwert aus 0000
indiziert über R1 in
                                                                    ; R0 laden
```

```
0277                  STRA,R0 0F 02  CC 0F 02                     ; ... in 0F02h speichern
027A                  SUBI,R1 10             85 01                       ; R1 = R1 + 1 (für
Zelle rechts oben)
027C                  LODA,R0 06 00  0D 60 00                    ; Zellwert aus 0000
indiziert über R1 in
                                                                         ; R0 laden
027F                  STRA,R0 0F 03  CC 0F 03                     ; ... in 0F03h speichern
0282                  ADDI,R1 01             85 10                       ; R1 = R1 + 10 (für
Zelle rechts)
0284                  LODA,R0 06 00  0D 60 00                    ; Zellwert aus 0000
indiziert über R1 in
                                                                         ; R0 laden
0287                  STRA,R0 0F 04  CC 0F 04                     ; ... in 0F04h speichern
028A                  ADDI,R1 11             85 10                       ; R1 = R1 + 10 (für
Zelle rechts unten)
028C                  LODA,R0 06 00  0D 60 00                    ; Zellwert aus 0000
indiziert über R1 in
                                                                         ; R0 laden
028F                  STRA,R0 0F 05  CC 0F 05                     ; ... in 0F05h speichern
0292                  ADDI,R1 10             A5 01                       ; R1 = R1 - 1 (für
Zelle unten)
0294                  LODA,R0 06 00  0D 60 00                    ; Zellwert aus 0000
indiziert über R1 in
                                                                         ; R0 laden
0297                  STRA,R0 0F 06  CC 0F 06                     ; ... in 0F06h speichern
029A                  ADDI,R1 0F             A5 01                       ; R1 = R1 - 1 (für
Zelle links unten)
029C                  LODA,R0 06 00  0D 60 00                    ; Zellwert aus 0000
indiziert über R1 in
                                                                         ; R0 laden
029F                  STRA,R0 0F 07  CC 0F 07                     ; ... in 0F07h speichern
02A2                  BCTA,UN ADDFLD 1F 02 B0                     ; Zur Additions-Subroutine
;
; An dieser Stelle kommt jede Adressarithmetik (auch die noch folgenden) an,
; nachdem sie die Additions-Subroutine aufgerufen hat:
;
; Registerrücktausch - Die Werte in den Registern werden für den nächsten Schritt ausgetauscht
02A5    ENDAA         LODZ,R3        03                           ; R3 in R0 zurückspeichern
02A6                  STRZ,R1        C1                           ; R0 in R1 zurückspeichern
02A7                  BCTA,UN REG    1F 04 10                     ; unbedingt zur
Regelanwendung
;
; Spielfeldecken-Adressarithmetik
;
; Summe für linke obere Ecke erzeugen:
02D0    LOE           STRZ,R1        01                           ; Kopiere den Wert aus R1
nach R0
02D1                  STRZ,R3        C3                           ; Kopiere den Wert aus R0
nach R3
02D2                  ADDI,R1 01             85 01                       ; R1 = R1 + 1 (für
Zelle rechts)
02D4                  LODA,R0 06 00  0D 60 00                    ; Zellwert aus 0000
indiziert über R1 in                                                                  ; R0
laden
02D7                  STRA,R0 0F00   CC 0F 00                     ; R0 in 0F00h sichern
02DA                  ADDI,R1 10             85 10                       ; R1 = R1 + 10 (für
Zelle rechts unten)
02DC                  LODA,R0 06 00  0D 60 00                    ; Zellwert aus 0000
indiziert über R1 in
                                                                         ; R0 laden
02DF                  STRA,R0 0F01   CC 0F 01                     ; R0 in 0F01h sichern
02EB                  SUBI,R1 01             A5 01                       ; R1 = R1 - 1 (für
Zelle unten)
02E4                  LODA,R0 06 00  0D 60 00                    ; Zellwert aus 0000
indiziert über R1 in
                                                                         ; R0 laden
02E7                  STRA,R0 0F02   CC 0F 02                     ; R0 in 0F02h sichern
02EA                  BCTA,UN ADDFLD 1F 02 B0                     ; Zur Additions-Subroutine
;
; Summe für rechte obere Ecke erzeugen:
02F0    ROE           STRZ,R1        01                           ; Kopiere den Wert aus R1
nach R0
02F1                  STRZ,R3        C3                           ; Kopiere den Wert aus R0
nach R3
```

```
02F2                  ADDI,R1 10            85 10                       ; R1 = R1 + 10 (für
Zelle unten)
02F4                  LODA,R0 06 00  0D 60 00                   ; Zellwert aus 0000
indiziert über R1 in
                                                                        ; R0 laden
02F7                  STRA,R0 0F00   CC 0F 00                   ; R0 in 0F00h sichern
02FA                  SUBI,R1 01            A5 01                       ; R1 = R1 - 1 (für
Zelle links unten)
02FC                  LODA,R0 06 00  0D 60 00                   ; Zellwert aus 0000
indiziert über R1 in
                                                                        ; R0 laden
02FF                  STRA,R0 0F01   CC 0F 01                   ; R0 in 0F01h sichern
0302                  SUBI,R1 10            A5 10                       ; R1 = R1 - 10 (für
Zelle links)
0304                  LODA,R0 06 00  0D 60 00                   ; Zellwert aus 0000
indiziert über R1 in
                                                                        ; R0 laden
0307                  STRA,R0 0F02   CC 0F 02                   ; R0 in 0F02h sichern
030A                  BCTA,UN ADDFLD 1F 02 B0                   ; Zur Additions-Subroutine
;
; Summe für linke untere Ecke erzeugen:
0310   LUE            STRZ,R1        01                         ; Kopiere den Wert aus R1
nach R0
0311                  STRZ,R3        C3                         ; Kopiere den Wert aus R0
nach R3
0312                  SUBI,R1 10            A5 10                       ; R1 = R1 - 10 (für
Zelle oben)
0314                  LODA,R0 06 00  0D 60 00                   ; Zellwert aus 0000
indiziert über R1 in
                                                                        ; R0 laden
0317                  STRA,R0 0F00   CC 0F 00                   ; R0 in 0F00h sichern
031A                  ADDI,R1 01            85 01                       ; R1 = R1 + 1 (für
Zelle rechts oben)
031C                  LODA,R0 06 00  0D 60 00                   ; Zellwert aus 0000
indiziert über R1 in
                                                                        ; R0 laden
031F                  STRA,R0 0F01   CC 0F 01                   ; R0 in 0F01h sichern
0322                  ADDI,R1 10            85 10                       ; R1 = R1 + 10 (für
Zelle rechts)
0324                  LODA,R0 06 00  0D 60 00                   ; Zellwert aus 0000
indiziert über R1 in
                                                                        ; R0 laden
0327                  STRA,R0 0F02   CC 0F 02                   ; R0 in 0F02h sichern
032A                  BCTA,UN ADDFLD 1F 02 B0                   ; Zur Additions-Subroutine
;
; Summe für rechte untere Ecke erzeugen:
0330   RUE            STRZ,R1        01                         ; Kopiere den Wert aus R1
nach R0
0331                  STRZ,R3        C3                         ; Kopiere den Wert aus R0
nach R3
0332                  SUBI,R1 01            A5 01                       ; R1 = R1 - 1 (für
Zelle links)
0334                  LODA,R0 06 00  0D 60 00                   ; Zellwert aus 0000
indiziert über R1 in
                                                                        ; R0 laden
0337                  STRA,R0 0F00   CC 0F 00                   ; R0 in 0F00h sichern
033A                  SUBI,R1 10            A5 10                       ; R1 = R1 - 10 (für
Zelle links oben)
033C                  LODA,R0 06 00  0D 60 00                   ; Zellwert aus 0000
indiziert über R1 in
                                                                        ; R0 laden
033F                  STRA,R0 0F01   CC 0F 01                   ; R0 in 0F01h sichern
0342                  ADDI,R1 01            85 01                       ; R1 = R1 + 1 (für
Zelle oben)
0344                  LODA,R0 06 00  0D 60 00                   ; Zellwert aus 0000
indiziert über R1 in
                                                                        ; R0 laden
0347                  STRA,R0 0F02   CC 0F 02                   ; R0 in 0F02h sichern
034A                  BCTA,UN ADDFLD 1F 02 B0                   ; Zur Additions-Subroutine
;
; Spielfeldrand-Adressarithmetik:
;
; Summe für oberen Rand erzeugen:
```

```
0350   RO             STRZ,R1               01                        ; Kopiere den Wert
aus R1 nach R0
0351                  STRZ,R3        C3                         ; Kopiere den Wert aus R0
nach R3
0352                  SUBI,R1 01             A5 01                       ; R1 = R1 - 1 (für
Zelle links)
0354                  LODA,R0 06 00  0D 60 00                   ; Zellwert aus 0000
indiziert über R1 in
                                                                        ; R0 laden
0357                  STRA,R0 0F 00  CC 0F 00                   ; R0 in 0F00h speichern
035A                  ADDI,R1 02             85 02                       ; R1 = R1 + 2 (für
Zelle rechts)
035C                  LODA,R0 06 00  0D 60 00                   ; Zellwert aus 0000
indiziert über R1 in
                                                                        ; R0 laden
035F                  STRA,R0 0F 01  CC 0F 01                   ; R0 in 0F01h speichern
0362                  ADDI,R1 10             85 10                       ; R1 = R1 + 10 (für
Zelle rechts unten)
0364                  LODA,R0 06 00  0D 60 00                   ; Zellwert aus 0000
indiziert über R1 in
                                                                        ; R0 laden
0367                  STRA,R0 0F 02  CC 0F 02                   ; R0 in 0F02h speichern
036A                  SUBI,R1 01             A5 01                       ; R1 = R1 - 1 (für
Zelle unten)
036C                  LODA,R0 06 00  0D 60 00                   ; Zellwert aus 0000
indiziert über R1 in
                                                                        ; R0 laden
036F                  STRA,R0 0F 03  CC 0F 03                   ; R0 in 0F04h speichern
0372                  SUBI,R1 01             A5 01                       ; R1 = R1 - 1 (für
Zelle links unten)
0374                  LODA,R0 06 00  0D 60 00                   ; Zellwert aus 0000
indiziert über R1 in
                                                                        ; R0 laden
0377                  STRA,R0 0F 04  CC 0F 04                   ; R0 in 0F05h speichern
037A                  BCTA,UN ADDFLD 1F 02 B0                   ; Zur Additions-Subroutine
;
; Summe für unteren Rand erzeugen:
0380   RU             STRZ,R1        01                         ; Kopiere den Wert aus R1
nach R0
0381                  STRZ,R3        C3                         ; Kopiere den Wert aus R0
nach R3
0382                  SUBI,R1 01             A5 01                       ; R1 = R1 - 1 (für
Zelle links)
0384                  LODA,R0 06 00  0D 60 00                   ; Zellwert aus 0000
indiziert über R1 in
                                                                        ; R0 laden
0387                  STRA,R0 0F 00  CC 0F 00                   ; R0 in 0F00h speichern
038A                  SUBI,R1 10             A5 10                       ; R1 = R1 - 10 (für
Zelle links oben)
038C                  LODA,R0 06 00  0D 60 00                   ; Zellwert aus 0000
indiziert über R1 in
                                                                        ; R0 laden
038F                  STRA,R0 0F 01  CC 0F 02                   ; R0 in 0F01h speichern
0392                  ADDI,R1 01             85 01                       ; R1 = R1 + 1 (für
Zelle oben)
0394                  LODA,R0 06 00  0D 60 00                   ; Zellwert aus 0000
indiziert über R1 in
                                                                        ; R0 laden
0397                  STRA,R0 0F 02  CC 0F 02                   ; R0 in 0F02h speichern
039A                  ADDI,R1 01             85 01                       ; R1 = R1 + 1 (für
Zelle rechts oben)
039C                  LODA,R0 06 00  0D 60 00                   ; Zellwert aus 0000
indiziert über R1 in
                                                                        ; R0 laden
039F                  STRA,R0 0F 03  CC 0F 03                   ; R0 in 0F03h speichern
03A2                  ADDI,R1 10             85 10                       ; R1 = R1 + 10 (für
Zelle rechts)
03A4                  LODA,R0 06 00  0D 60 00                   ; Zellwert aus 0000
indiziert über R1 in
                                                                        ; R0 laden
03A7                  STRA,R0 0F 04  CC 0F 04                   ; R0 in 0F04h speichern
03AA                  BCTA,UN ADDFLD 1F 02 B0                   ; Zur Additions-Subroutine
;
; Summe für linken Rand erzeugen:
```

```
03B0   RR             STRZ,R1        01                      ; Kopiere den Wert aus R1
nach R0
03B1                  STRZ,R3        C3                      ; Kopiere den Wert aus R0
nach R3
03B2                  SUBI,R1 10            A5 10                    ; R1 = R1 - 10 (für
Zelle oben)
03B4                  LODA,R0 06 00  0D 60 00                ; Zellwert aus 0000
indiziert über R1 in
                                                                ; R0 laden
03B7                  STRA,R0 0F 00  CC 0F 00                ; R0 in 0F00h speichern
03BA                  ADDI,R1 01            85 01                    ; R1 = R1 + 1 (für
Zelle rechts oben)
03BC                  LODA,R0 06 00  0D 60 00                ; Zellwert aus 0000
indiziert über R1 in
                                                                ; R0 laden
03BF                  STRA,R0 0F 01  CC 0F 01                ; R0 in 0F01h speichern
03C2                  ADDI,R1 10            85 10                    ; R1 = R1 + 10 (für
Zelle rechts)
03C4                  LODA,R0 06 00  0D 60 00                ; Zellwert aus 0000
indiziert über R1 in
                                                                ; R0 laden
03C7                  STRA,R0 0F 02  CC 0F 02                ; R0 in 0F02h speichern
03CA                  ADDI,R1 10            85 10                    ; R1 = R1 + 10 (für
Zelle rechts unten)
03CC                  LODA,R0 06 00  0D 60 00                ; Zellwert aus 0000
indiziert über R1 in
                                                                ; R0 laden
03CF                  STRA,R0 0F 03  CC 0F 03                ; R0 in 0F03h speichern
03D2                  SUBI,R1 01            A5 01                    ; R1 = R1 - 1 (für
Zelle unten)
03D4                  LODA,R0 06 00  0D 60 00                ; Zellwert aus 0000
indiziert über R1 in
                                                                ; R0 laden
03D7                  STRA,R0 0F 04  CC 0F 04                ; R0 in 0F04h speichern
03DA                  BCTA,UN ADDFLD 1F 02 B0                ; Zur Additions-Subroutine
;
; Summe für rechten Rand erzeugen:
03E0   LR             STRZ,R1        01                      ; Kopiere den Wert aus R1
nach R0
93E1                  STRZ,R3        C3                      ; Kopiere den Wert aus R0
nach R3
03E2                  ADDI,R1 10            85 10                    ; R1 = R1 + 10 (für
Zelle unten)
03E4                  LODA,R0 06 00  0D 60 00                ; Zellwert aus 0000
indiziert über R1 in
                                                                ; R0 laden
03E7                  STRA,R0 0F 00  CC 0F 00                ; R0 in 0F00h speichern
03EA                  SUBI,R1 01            A5 01                    ; R1 = R1 - 1 (für
Zelle links unten)
03EC                  LODA,R0 06 00  0D 60 00                ; Zellwert aus 0000
indiziert über R1 in
                                                                ; R0 laden
03EF                  STRA,R0 0F 01  CC 0F 01                ; R0 in 0F01h speichern
03F2                  SUBI,R1 10            A5 10                    ; R1 = R1 - 10 (für
Zelle links)
03F4                  LODA,R0 06 00  0D 60 00                ; Zellwert aus 0000
indiziert über R1 in
                                                                ; R0 laden
03F7                  STRA,R0 0F 02  CC 0F 02                ; R0 in 0F02h speichern
03FA                  SUBI,R1 10            A5 10                    ; R1 = R1 - 10 (für
Zelle links oben)
03FC                  LODA,R0 06 00  0D 60 00                ; Zellwert aus 0000
indiziert über R1 in
                                                                ; R0 laden
03FF                  STRA,R0 0F 03  CC 0F 03                ; R0 in 0F03h speichern
0402                  ADDI,R1 01            85 01                    ; R1 = R1 + 1 (für
Zelle oben)
0404                  LODA,R0 06 00  0D 60 00                ; Zellwert aus 0000
indiziert über R1 in
                                                                ; R0 laden
0407                  STRA,R0 0F 04  CC 0F 04                ; R0 in 0F04h speichern
040A                  BCTA,UN ADDFLD 1F 02 B0                ; Zur Additions-Subroutine
;
,
```

```
; SPIELREGELN ANWENDEN:
;
; Entscheidend für den Status der Zelle in der nächsten Generation - ob sie tot oder lebendig
sein
; wird - ist der momentane Status der Zelle sowie die Anzahl ihrer lebenden Nachbarn. Letztere
; wurde als die Summe der Werte aus den Nachbarzellen durch die Adress-Arithmetik in R2
gespeichert.
; Mit dem momentanen Zustand der Zelle in R0 und diesem Wert in R2 können die Regeln angewandt
; werden Ausgelesen wird der alte Status einer Zelle jeweils aus dem Arbeitsfeld. Geschrieben
wird
; der neue Status derselben Zelle in den "Zwilling" der Zelle auf dem Spielfeld. So wird das
; erwähnte Problem des Überschreibens umgangen.
; Da es mit dem hier verwendeten Assembler nicht möglich ist, gleichzeitig zu testen, ob die
Summe
; kleiner als 2 und größer als 3 ist, muss dieser Test in zwei getrennten Regelschritten
erfolgen.
; Weiterhin ist zu beachten, dass nur das Arbeitsfeld mit 0en und 1en gefüllt ist. Während
also der
; Ist-Zustand einer Zelle auf 0 und 1 hin befragt werden muss, müssen beim Einschreiben des
neuen
; Status der Zelle in das Spielfeld nicht 0 und 1, sondern 17 für tot und 15 für lebendig
verwendet
; werden.
;
0410    REG             LODA,R1 6A00    0D 60 00                    ; Hole den Wert der Zelle
aus dem
                                                                    ; Spielfeld ...
                                                                    ;  ... indiziert
über R1 in R0.
; Der Wert in R1 bestimmt also, welche Zelle dabei angesteuert wird. R0 enthält danach den
; momentanen Wert der Zelle, deren nächster Zustand bestimmt werden soll.
;
; Regel 1 - WIEDERGEBURT:
; Eine tote Zelle mit 3 lebenden Nachbarn wird neu geboren;
; Wenn ( R0 = 0 und R2 = 3 ) dann R0 = 15; und dieser Wert aus R0 wird dann in den "Zwilling"
der
; Zelle auf dem Spielfeld geschrieben:
;
0413    REG1            COMI,R0 00              E4 00                       ; Ist die Zelle
momentan tot? (Ist R0 = 0)
0415                    BCFA,0 REG2             9C 04 25                    ; Wenn sie lebt,
dann springe zu Regel 2
0418                    COMI,R2 03              E6 03                       ; Sonst: Ist die
Nachbaranzahl gleich 3?
                                                                            ; (Ist R2 = 3)
041A                    BCFA,0 REGEND   9C 04 3C                    ; Wenn das falsch ist,
springe zum
                                                                            ; Regelprüfung-Ende
041D                    LODI,R0 01              04 15                       ; Sonst: Den Wert
für lebende Zelle (15)
                                                                            ; in R0 laden ...
041F                    STRA,R0 0E00    CD 6E 00                    ; ... und damit die Zelle
im Spielfeld
                                                                            ; wieder beleben
0422                    BCTA,UN REGEND 1F 04 3C                     ; Unbedingt nach
Regelprüfung-Ende
;
; Es muss nicht noch einmal geprüft werden, ob die Zelle tot ist. Das hat sich zwangsläufig
schon
; ergeben, wenn Regel 1 nicht appliziert wurde, sondern direkt von dort aus nach Regel 2
geprungen
; wurde.
;
; Regel 2 - TOD DURCH ISOLATION:
; Eine lebende Zelle mit weniger als 2 Nachbarn stirbt;
; wenn ( R0 = 1 und R2 < 2 ) dann R0 = 17;  und dieser Wert aus R0 wird dann in den "Zwilling"
der
; Zelle auf dem Spielfeld geschrieben:
;
0425    REG2            COMI,R2 02              E6 02                       ; Ist die
Nachbaranzahl kleiner als 2?
                                                                            ; (Ist R2 < 2)
0427                    BCFA,2 REGEND   9E 04 32                    ; Wenn kleiner, dann nach
Regel 3
```

```
042A                    LODI,R0 17              04 17                           ; Sonst: Den Wert
für tote Zelle (17) in
                                                                                ; R0 laden ...
042C                    STRZ,R1 0E00    CD 6E 00                        ; ... und damit die Zelle
im Spielfeld
                                                                                ; töten
042F                    BCTA,UN REGEND 1F 04 3C                         ; Unbedingt nach
Regelprüfung-Ende
;
; Regel 3 - TOD DURCH ÜBERBEVÖLKERUNG:
; Eine lebende Zelle mit mehr als 3 Nachbarn stirbt;
; wenn ( R1 = 0 und R2 > 3 ) dann R0 = 17);  und dieser Wert aus R0 wird dann in den
"Zwilling" der
; Zelle auf dem Spielfeld geschrieben:
;
0432    REG3            COMI,R2 03              E6 03                           ; Ist die
Nachbaranzahl größer als 3? (Ist
                                                                                ; R2 > 3)
0434                    BCFA,2 REGEND   9D 04 3C                        ; Wenn größer, dann nach
Regelprüfung-Ende
0437                    LODI,R1 17              04 17                           ; Sonst: Den Wert
für tote Zelle (17) in
                                                                                ; R0 laden ...
0439                    STRA,R3 0E00    CD 6E 00                        ; ... und damit die Zelle
im Spielfeld
                                                                                ; töten
;
043C    REGEND  BIRA,R1 ANAANF DD 17 A5                         ; Analyseschleifen-Ende;
Inkrementiere R1
                                                                                ; um global die
nächste Zelle anzusteuern
;
; LOOP - Lädt die Register mit verschiedenen Werten, die dann dekrementiert werden, für eine
; Wartschleife von 0,5 Sekunden. Andernfalls würde die Geschwindigkeit des Generationswechsels
die
; menschlichen Wahrnehmungsschwellen unterlaufen.
;
043F    LOOPEND LODI,R2 71              06 71                           ; Lade 71 nach R2; Äußere
Warteschleife
0441                    LODI,R3 C2              07 C2                           ; Lade C2 nach R3;
Innere Warteschleife
0443                    BDRR,R2 7E              FA 7E                           ; Dekrementiere R2
bis 0
0445                    BDRR,R3 7C              FB 7C                           ; Dekrementiere R3
bis 0
0447                    BCTA,UN 1786    1F 17 86                        ; Fertig, zurück in die
INITIALISIERUNG
;
;
;
;
*******************************************************************************************
*****
;
; SUBROUTINEN
;
;
; CLRFLD - Löschen des Spielfeldes mit 17, graphisch ausgegeben als " "
;
0102                    LODI,R1 00              05 00                           ; Lade 0 nach R1
0104                    LODI,R0 17              04 17                           ; Lade 15 nach R0
0106                    STRA,R1 0E00    CD 6E 00                        ; Kopiere R0 indexiert
über R1 nach 0E00
0109                    BIRA,R1 0106    DD 01 06                        ; Inkeremtiere R1 und
springe zurück
                                                                                ; solange R0 > 0
010C                    BCTA,UN INIT    1F 17 83                        ; zurück nach INIT
;
;
; SEED - Das Spielfeld wird mit Werten gefüllt. Als Random-Seed dient die Anzahl der
abgelaufenen
; Clock-Cylcles zum Zeitpunkt des Programmstarts.
;
0120    SEED            CPSL 02         75 02                           ; COM arithmetisch
```

```
0122                  LODI,R2 A0            06 FF                         ; R2=FF
0124                  LODI,R1 FF            05 FF                         ; Beginn Schleife
(255 mal)
0126    LOOP          REDE,R0 73            54 73                         ; Zufallswert holen
0128                  RRL,R0         D0                           ; nach rechts rotieren
0129                  TPSL 04        B5 04                        ; testet Vorzeichenbit
012B                  BCFA NOERR            9C 01 31                      ; Bei 0 springen
012E                  REDE,R3 73            57 73                         ; noch einen
Zufallswert holen
0130                  EORZ,R3               22                    ; R0 XOR R3 -> R0
0131    NOERR         STRZ,R3               C3                    ; Zufallszahl in R3
ablegen
0132                  LODI,R0 15            04 15                         ; "o" ...
0134                  STRA,R3 0900   CF 6E 00                     ; ... in Feld schreiben
0137                  BDRA,R1 LOOP   FD 01 28                     ; Schleifenende
013A                  BCTA,UN 0124   FE 01 24                     ; Wiederhole Seed
013D                  BCTA,UN INIT   1F 17 86                     ; zurück nach INIT
;
;
; CLRART - Löschen der Arithmetik-Speicherzellen: Da bei der Berechnung der Sonderfelder
; (an den Ecken und Rändern) weniger Nachbarzellen in den Arithmetikspeicher übergeben werden,
; müssen evtl. vorhandene Residuen aus anderen Berechnungen mit 0 gefüllt werden, damit diese
; die Addition nicht verfälschen.
;
0150    CLRART  LODI,R3 00            07 00                         ; Lade R3 mit 0
0152                  STRA,R3 0F00   CF 0F 00                     ; Zwischenspeicher löschen
0155                  STRA,R3 0F01   CF 0F 01
0158                  STRA,R3 0F02   CF 0F 02
015B                  STRA,R3 0F03   CF 0F 03
015E                  STRA,R3 0F04   CF 0F 04
0161                  STRA,R3 0F05   CF 0F 05
0164                  STRA,R3 0F06   CF 0F 06
0167                  STRA,R3 0F07   CF 0F 07
016A                  STRA,R3 0F08   CF 0F 08
016D                  RETC,UN 17A8   1F 17 A8                     ; Fertig, zurück ins
Hauptprogramm
;
; Kommentar: Hier findet sich ein ungefährlicher "Bug": Die Adresse 0F08 wird nie von der
ADDFLD
; angesteuert, das Löschen dort und damit die ganze Adresszeile 016A sind unnötig.
;
; ADDFLD - Addition der Umfeld-Speicherzellen-Inhalte: Hier werden die Inhalte der
Nachbarzellen
; miteinander addiert, um dann für die Spielregeln ausgewertet zu werden. In den
Speicherzellen
; stehen 15h ("o") oder 17h (" "), die arithmetisch in 1 und 0 umgewandelt werden.
; Die in R2 abgelegte Summe liegt dann zwischen 0 und 8.
;
02B0    ADDFLD  LODI,R2 00            06 00                         ; R2 löschen
02B2                  ADDA,R2 0F01   8E 0F 00                     ; (0001h) zu R2 addieren
02B5                  ADDA,R2 0F02   8E 0F 01                     ; (0002h) zu R2 addieren
02B8                  ADDA,R2 0F03   8E 0F 02                     ; (0003h) zu R2 addieren
02BB                  ADDA,R2 0F04   8E 0F 03                     ; (0004h) zu R2 addieren
02BE                  ADDA,R2 0F05   8E 0F 04                     ; (0005h) zu R2 addieren
02C1                  ADDA,R2 0F06   8E 0F 05                     ; (0006h) zu R2 addieren
02C4                  ADDA,R2 0F07   8E 0F 06                     ; (0007h) zu R2 addieren
02C7                  ADDA,R2 0F08   8E 0F 07                     ; (0008h) zu R2 addieren
02CA                  BCTA,UN ENDAA  1F 02 A5                     ; Fertig und zum
Registerrücktausch
```

D. Interview mit Martin Wendt (enthusi)

Stefan Höltgen: Seit wann entwickelst du Computerspiele für 8- und 16-Bit-Systeme?

Martin Wendt: Dieses Hobby begann bereits mit meinem ersten Heimcomputer Anfang der 90er, aber natürlich eher für einen selbst oder den Freundeskreis. Einfache Textadventures und ein kleines Rollenspiel fallen mir noch ein. Ein Spiel welches ich schließlich für meinen ersten PC portierte, landete 1998 auf der PC-Action CD als

Shareware. Dem folgte aber eher die Programmierung von Effekten und sogenannten Demos, bis ich schließlich 2008 das erste C64-Spiel veröffentlichte, das dann mit Anleitung und Box über einen britischen Publisher angeboten wurde.

Stefan Höltgen: Betreibst du die Entwicklung ausschließlich als Hobby?

Martin Wendt: Das ist ausschließlich ein Hobby von mir. Statt lange Videospiele zu spielen, habe ich immer lieber programmiert und mich dabei stets auch für Nischensysteme interessiert. Am C64 mit der bemitleideten Datassette beispielsweise. Der Markt – so man denn überhaupt davon sprechen kann – ist sehr sehr klein. Spiele, die als Box mit Anleitung und Datenträger verkauft werden für solche Systeme, erwirtschaften nur selten mehr als die reinen Materialkosten. Am meisten Geld spielten Auktionen zugunsten von Wohltätigen Organisationen ein, bei denen wir Prototypen oder dergleichen versteigerten. Der Erlös ging vollständig an *Ärzte ohne Grenzen*. Das fand ich immer am befriedigendsten. Man investiert sein Hobby in etwas, an dem Leute im Idealfall Spaß haben und generiert somit quasi aus dem Nichts einen Wert, der Menschen anderswo zu Gute kommt.

Stefan Höltgen: Für welche historischen Systeme hast du bislang Computerspiele entwickelt?

Martin Wendt: Veröffentlicht wurden Spiele für C64 und Atari 2600, während ich auch für das Atari Lynx, den Watara Supervision, ZX Spektrum und den Gameboy Advance (GBA) entwickele. Bis auf den GBA und dem ZX Spektrum laufen alle Systeme auf eng miteinander verwandten Prozessoren der 6502 Familie.

Stefan Höltgen: In welchen Programmiersprachen hast du für diese Systeme Computerspiele entwickelt?

Martin Wendt: Immer in Assembler, das macht für mich den Reiz aus, so nah wie möglich an der Hardware selbst zu arbeiten. Ich versuche dabei jeder Hardware etwas zu entlocken, was spezifisch für die Designentscheidungen des jeweiligen Systems war. 8 Bit und Assembler gehören für mich zusammen. Erst am GBA programmiere ich große Teile in C.

Stefan Höltgen: Welche Tools setzt du bei der Entwicklung ein?

Martin Wendt: Da ich immer einen Grundüberblick über ein Programm und das jeweilige Timing haben möchte, greife ich auch nie auf vorhandene Programmierbibliotheken oder Oberflächen zurück. Ich erfinde das Rad im Zweifel lieber neu und dann jedes mal ein wenig optimierter oder spezifischer auf die Bedürfnisse angepasst. Der Weg ist schließlich Teil des Ziels. Man lernt nie aus und verbessert sich immer ein wenig. Gerade auf 8-Bit-Systemen sind dadurch gewisse Dinge überhaupt erst denkbar. Eine generische Routine, die auf diversen Systemen gleich läuft, kann per se nicht optimiert sein. Ich greife immer auf einen beliebigen Text-Editor zurück und suche mir einen passenden

[Cross]Assembler, wobei ich meist einen sehr einfachen bevorzuge. Für C64, Atari2600, Lynx und Supervision verwende ich den selben Assembler, obwohl sich für die jeweiligen Systeme ganz eigene Subkulturen herausgearbeitet haben. Da ich aber nie auf vorhandene Bibliotheken zurückgreife, stört mich diese Inkompatibilität der Assembler nicht weiter. Im Gegenteil, dadurch lernt man etwas erst wirklich im Detail kennen.

Stefan Höltgen: Wie erledigst du ohne IDE das Debugging?

Martin Wendt: Wann immer möglich bevorzuge ich, debug-Informationen auf der echten Hardware anzeigen zu lassen. Das kann ein einfaches Zeichen am C64 sein, oder, wie teilweise bei Caren geschehen, eine komplexe Anzeige diverser Speicheradressen in Echtzeit mit einem Zeichensatz der die Werte von 0x00 bis 0xff darstellt. Ein einfaches `LDA timer_A; STA SCREEN+40*24+0` reicht dann, um den 8-Bit-Wert am Bildschirm darstellen zu können, wenn man zuvor auf den richtigen Zeichensatz geschaltet hat (siehe Abb. A.1).

Abb. A.1: Caren and the Tangled Tentacles mit Debugging-Informationen im unteren Bildbereich

Dies hat aber entscheidende Nachteile – vor allem auf kleineren Konsolen. Zum einen verbraucht hier ein eigener Zeichensatz zusätzlichen Speicher (im Fall von Caren spielt im debug-mode keine Musik und der Zeichensatz belegt den Speicher den sonst das SID hätte), zum anderen benötigt der Code zur Anzeige in der Regel kritische CPU Zeit. Auf so mancher Hardware ist dieser Ansatz sogar kaum möglich, wie beispielsweise am Atari VCS/2600 der keinen Framebuffer besitzt. Man kann zwar einzelne Werte über das Spriteregister anzeigen lassen, aber meist ist es ja gerade eine Spriteroutine, welche man genau timen möchte. Das Ändern einer dominanten (Hintergrund)Farbe funktioniert hingegen fast auf jedem System. Wechselt man vor und nach einer Routine diese Farbe und synchronisiert das zudem mit dem Bildaufbau, so hat meine einen recht genauen visuellen Eindruck der Rasterzeit, die jene Routine verbraucht hat. Es bietet sich sich – je nach System – auch an, komplexere Informationen in diese Farbe zu legen. Beim C64 mit seinen 16 Farben lässt sich über die Rahmenfarbe das High Nibble einer 8 bit Variable ausgeben. Das hilft oft schon viel. Der Watara Supervision hat allerdings keine Mög-

lichkeit die Bildschirmanzeige zu synchronisieren. Auch gibt es keine Farbregister, die man ändern könnte. Hier bin ich teilweise den erstaunlich praktischen Weg über die Audioregister gegangen. Nahezu jede 8-Bit-Konsole besitzt 8- oder 16- Bit-Register zur Festlegung einer Frequenz oder eines Frequenzteilers für einen Tongenerator. Der Wert eines Timers lässt sich in dieses Register kopieren und man hört unmittelbar wie schnell der Wert sich ändert. Am Supervision, der ein echtes Stereo-Panning erlaubt, lassen sich so sogar in Echtzeit zwei unterschiedliche Werte vergleichen (eine Ausgabe auf den linken Lautsprecher, eine weitere auf den Rechten). Das wird erst mit Kopfhörern ‚familienfreundlich' (Frequenz-Sweeps sind nicht so angenehm für das Ohr), aber das menschliche Ohr ist erstaunlich empfindlich. 256 verschiedene Frequenzen lassen sich deutlich besser unterscheiden als 256 Grautöne! Ob der Timer als Ton auf dem linken Ohr den Grenzwert, der auf dem rechten Ohr zu hören ist, erreicht hat, oder wie nah er ihm kommt, ist gut rauszuhören (im direkten Vergleich). Absolute Werte sind hingegen für den Normalmenschen kaum auszumachen. Ganz konkret hatte ich einmal einen Stack-Underflow und eine einfache Routine im NMI (non-maskable interrupt). Ich legte mir den Wert des Stackpointers permanent auf ein Ohr. Es war sofort zu hören, an welcher Stelle im Spiel der Ton und somit der Stackpointer in die Knie ging. Audio-Debugging bietet sich auch für sehr schnelle Änderungen an, während die Darstellung über den Bildschirm nie schneller als die Bildfrequenz sein kann und auch dann schwer zu lesen ist.

Stefan Höltgen: Setzt du eigene Tools bei der Entwicklung ein?

Martin Wendt: Mein Assemblerquelltext ist meist ziemlich direkt. Ich verwende nicht einmal Makros, wie sie teilweise für 16-Bit-Operationen üblich sind. Allerdings greife ich ausführlich auf Python als moderne Hochsprache zurück, wenn es darum geht, Daten passend aufzubereiten. Bei zeitkritischen Prozessen, spielt oft die Datenstruktur eine maßgebliche Rolle. Komplexe Tabellen oder ähnliches bereite ich also in Python vor, um die wenigen Befehle eines 65C02 optimal einsetzen zu können. Das artet teilweise in sehr abenteuerliche Verschachtelungen aus.

Stefan Höltgen: Welche Emulatoren setzt du ein?

Martin Wendt: Am C64 ist der offene VICE-Emulator, den ich unter Linux verwende, besonders genau. Er hat den Ruf etwas benutzerunfreundlich zu sein, aber die reine Präzision der Emulation dürfte die der meisten Systeme deutlich übersteigen. Für den Atari 2600 hat sich der Stella-Emulator heraus kristallisiert. Er ist ebenfalls quelloffen und unter Linux verfügbar. Neben der herausragenden Debug-Oberfläche, ist er in den letzten Jahren auch zunehmend genauer geworden. Als ich anfing war der Z26-Emulator noch genauer. Ebenfalls quelloffen ist der Free Unix Spectrum Emulator (FUSE). Der ZX Spectrum ist etwas einfacher zu emulieren, da es weniger Seiteneffekte gibt, die zu berücksichtigen sind. Für den GBA sind die besseren Emulatoren leider nur unter Windows verfügbar, sodass ich teilweise unter Linux einen Fork des VisualBoyAdvance verwende, und teilweise über die Windows-Schnittstellenemulation WINE den

hervorragenden NO$GBA-Emulator nutze. Für den Watara SuperVision gibt es überhaupt erst seit kurzem eine nennenswerte Emulation. Für einfache Logiktests habe ich einen bestehenden Emulator (Potaro) auf mein System portiert, der läuft jetzt unter Linux/SDL. Das echte Spiel teste ich dann über Wine im Wataroo-Emulator und vor allem Sound- und besondere Grafik-Effekte an echter Hardware über ein gebasteltes EPROM-Board (siehe Abb. A.2), wobei ich keine EPROMs löschen und brennen muss, sondern eine Hardware verwende, die ein EPROM mit SRAM emuliert und welche ich direkt vom PC aus bespielen kann mit meinem EPROM-Image. Das nur unter Windows, aber auch unter WINE laufende Wataroo. Dessen Autor ist selbst maßgeblich am Reverse Engineering der Hardware beteiligt, sodass es der mit Abstand beste Emulator ist (der aber längst nicht alle Aspekte der Geräte vollständig widerspiegelt). Zusätzlich nutze ich meine eigene Variante des Potator-Emulators. Am weitesten hinter der echten Hardware bleiben sämtliche Atari-Lynx-Emulatoren zurück. Spezifische Effekte des Timings oder der Grafik werden intern gar nicht erst abgebildet. Hier lag der Fokus ganz klar darauf, den vorhandenen Softwarekatalog zu emulieren und weniger die Hardware selbst vollständig abzubilden. Das ist eine ganz neue Herausforderung.

Stefan Höltgen: Welchen Anteil haben die Originalplattformen bei der Entwicklung?

Martin Wendt: Immer wenn ich mich in ein System und seine Spezifikationen einlese, steht recht früh am Anfang auch die echte Hardware, um ein Gespür dafür zu bekommen. Ich spiele selbst sehr wenig, und lerne so immer mal etwas Neues kennen, wie beim Atari Lynx. Erst dann merkt man, was mit der Hardware möglich ist, oder auch welche Möglichkeiten vielleicht noch gar nicht ausgereizt oder bedacht wurden. Und natürlich müssen gerade neue Effekte recht kleinschrittig auf echter Hardware getestet und entwickelt werden. Am Lynx gibt es nur sehr wenige technisch beeindruckende Homebrew-Projekte, am Supervision dürfte unser Assembloids[187] das erste nicht-kommerzielle Spiel überhaupt werden.

187 In einem YouTube-Video beschreibt der Entwickler sein Setup für die Programmierung des Spiels Assembloids für die Atari 2600/VCS: https://www.youtube.com/watch?v=RaE7qcRDivo [letzter Abruf: 15.03.2019].

Abb. A.2: Martin Wendt: Die Module für Watara Supervision enthalten lediglich ein einfaches 27512 EPROM, also 64 KB Speicher ohne weitere Schaltlogik. Bei den originalen Spielen verbirgt sich dieses EPROM in SMD Baugröße unter einem 'Blob' aus Epoxidharz. Über eine Streifenrasterplatine habe ich alle Leitungen vom Modulport herausgeführt und mit einem IC-Sockel verdrahtet. In diesem ließe sich ein 27512 setzen, welches man allerdings jedes mal neu löschen und brennen müsste. Stattdessen verwende ich einen sogenannten EPROM-Emulator. Ein SRAM (flüchtiges RAM) welches sich bequem vom PC aus via USB beschreiben lässt und sich wie ein solches EPROM verhält. Innerhalb einer Sekunde lässt sich so eine Software auf echter Hardware testen, so wie im Bild eine Interlacegrafik die mit 30 Hz zwei 4-Farbbilder wechselt, um so bis zu 7 Farben darzustellen. Das LCD Display hat extrem langsame Refreshzeiten, was dazu führt, dass am Supervision nicht das geringste Flackern wahrzunehmen ist. Ein Novum am Supervision was durch die fehlende Synchronisation mit dem Display etwas komplizierter ist.

Stefan Höltgen: Hast du bei der Entwicklung von Computerspielen schon einmal auf Hardware-Glitches der Plattform zurückgegriffen (Designfehler, illegale Opcodes, …)?

Martin Wendt: Beim C64 sind die sogenannten illegalen Opcodes teilweise sehr naheliegend, auch wenn ich nicht explizit dahingehend versuche zu optimieren. Der Atari 2600 hat viele Glitches, die man alle berücksichtigen muss, sobald man vom ‚handbuchartigen' Weg der braven Programmierung abweicht. Es gibt dort etliche Timing-kritische Effekte, die sich nicht aus der Hardware selbst ableiten lassen aber zunehmend besser verstanden werden und auch gezielt eingesetzt werden. Beim Atari Lynx zum Beispiel gelang es mir zum ersten mal überhaupt die Farbpalette zyklenexakt pro Bildschirmzeile zu ändern für farbigere Grafiken ohne Flackern. Die vorhandenen Emulatoren berücksichtigen Zyklen im Timing überhaupt nicht und Der Bildschirmaufbau findet zeilenweise statt punktweise statt. Ein solcher Effekt war mit der Emulation also gar nicht erst testbar und auch in den Hardwarespezifikationen nicht vorgesehen. Kein Designfehler, aber zumindest eine Kuriosität ist die Option am C64 Sprites zwar über anderen Sprites aber hinter der Hintergrundgrafik darstellen zu können. Dadurch kann man unerwartete Löcher im Sprite provozieren, die wir bei Caren and the Tangled Tentacles gezielt zur Maskierung verwenden. Das gab es bei Spielen bis dato nicht.

Stefan Höltgen: Nutzt du moderne Hardware-Ergänzungen der Originalsysteme bei der Entwicklung deiner Spiele?

Martin Wendt: An moderner Hardware benutze ich lediglich vorhandene Lösungen, um Software auf den Geräten laufen zu lassen, ohne dafür eigene EPROM-Karten herstellen zu müssen. Seien es SD-Karten oder Kabelverbindungen zum PC. Von Erweiterungen, die dem System selbst neue Eigenschaften beschweren, halte ich wenig und nehme gezielt Abstand davon. Mich reizt jeweils nur das Vanilla System wie es seinerzeit verbreitet war. Also von Speichererweiterungen und extra Chips auf den Steckkarten mache ich nie Gebrauch.

Stefan Höltgen: Hat dir bei der Entwicklung von Spielen schon einmal ein Emulator Probleme bereitet, weil er sich anders als das Originalgerät verhielt?

Martin Wendt: Ja, bei jedem der Emulatoren war das bisher so. Selbst VICE ließ unter bestimmten Bedingungen einen Speicher-Modus zu, den es am echten C64 nicht gab. Das passierte selten, aber natürlich [passierte es] mir und wurde erst kürzlich gefixt. Die Emulationsentwickler sind sehr dankbar für Software, die Schwächen aufzeigt oder überprüfbar macht. Zumindest in der C64-Szene ist das ein konstruktiver Wettlauf zwischen Demo-Programmierern und Emulator-Entwicklern. Der Atari 2600 zeigt unzählige Besonderheiten, die ich quasi alle am echten Gerät kennenlernen musste, als ich damit etwa 2012 anfing. Inzwischen hat der Emulator gut aufgeholt und es reicht immer mal wieder am echten Gerät zu testen. Beim Lynx ist es nach wie vor so, dass die

Emulation kein Ersatz für die echte Hardware sein kann, sobald man sie ein wenig fordert. Gerade das komplexe Zusammenspiel der Grafikeinheit mit der CPU wird teilweise um Faktor 2 zu schnell oder gar nicht richtig emuliert. Von der Audiohardware ganz zu schweigen. Wie sich die Ausgabe verhält, wenn man Frequenzen und Wellenformen mehrfach pro Frame wechselt, hatte niemanden zuvor wirklich interessiert. Ein C64-Musiker arbeitet aber durchaus damit, so dass wir uns das von Null an erarbeiten mussten für unser Audiosystem. Der Supervision-Handheld selbst kann von Haus aus fast gar nichts besonderes, aber Details vor allem im Audio-System und bei taktkritischen Operationen werden erst seit 2018 überhaupt nachgebildet. Selbst der inzwischen sehr gute Emulator Wataroo ignoriert die Tatsache, dass sich das Bild nicht für jeden Frame aufbaut, sondern lediglich eines von zwei bits pro Frame an das Display transferiert werden. Ein Effekt, der dadurch für das kommende Assembloids möglich ist, ließ sich nur an echter Hardware testen.

Stefan Höltgen: Entstehen deine Projekte in der Regel als Einzel- oder Gruppenarbeiten?

Martin Wendt: Es beginnt eigentlich immer damit, dass entweder im Gespräch oder als einfacher Gedanke eine Idee aufkommt, wie man etwas besonders raffiniert, eindrucksvoll oder optimal programmieren könnte in Hinblick auf die vorhandenen Restriktionen der Hardware. Am Anfang steht eine Machbarkeitstudie, die meist aus reiner Programmierarbeit besteht. Grafische Elemente oder gar Musik kommen meist später hinzu. Es kommt aber durchaus auch vor, dass eine besonders ausgefeilte Audioroutine oder grafische Perspektive der Kern eines Projektes wird.

Stefan Höltgen: Kannst du den Entwicklungsprozess von Caren and the Tangled Tentacles skizzieren? – Von der Planung über die Aufgabenverteilung, die Programmierung bis hin zum Test und dem Versioning?

Martin Wendt: Oliver Lindau (der Grafiker) und ich sind seit jeher am Genre der Grafikadventures interessiert und in der Vergangenheit habe ich mir die ein oder andere Engine eines solchen Spieles auf unterschiedlichsten Plattformen im Detail angeschaut. Anstoß für das Projekt Caren and the Tangled Tentacles war schließlich die Ankündigung eines Programmierwettbewerbs für genau jenes Genre am C64. Somit war ein klarer Start (und Ende) vorgegeben. Ich kontaktierte sofort Oliver, weil ich wusste, dass wir gut zusammenarbeiten würden können bei so ähnlichen Vorstellungen vom Spieledesign. Anfangs diskutierten wir fast täglich welche Ideen wir jeweils auf jeden Fall in so einem Projekt umsetzen wollen würden. Das verlief absolut gleichberechtigt. Ich gab grafische Limitierungen vor, um neue Programmiertricks umsetzen zu können und er legte an anderer Stelle besonderen Wert auf Details wie die separate Bewegung von Beinen und Armen oder das Auf- und Abwippen der Köpfe beim Laufen. Grafik und Code liefen lange parallel zueinander, bis die Engine schließlich so weit stand, dass die Raumgrafiken voll eingebunden werden konnten und ich mit dem Skripten der Räume begann. Kamil Wolnikowski kam mit der Musik erst später dazu. Ihm wurde aber als Profi auf seinem

Gebiet quasi die gesamte Audiodirektion übertragen. Es gab also de facto drei Projektleiter – jeweils in ihrem Metier. Etwas, das es wohl nur noch auf 8-Bit-Rechnern aus den 80ern zu finden gibt. Danach wurden die Datenmengen, der Spielumfang und somit gezwungenermaßen das ganze Team ungleich größer. So lautete auch in etwa das Feedback welches uns von Spieleentwicklern aus der C64-Hochzeit erreichte. Man sehnte sich nach den Zeiten der ‚guten alten 3-Mann-Teams', vermisste die widrigen Umstände der Entwicklung auf 8-Bit-Systemen aber in keiner Weise. Zum Wettbewerbsende testeten wir das Spiel weitestgehend selbst, allen voran der Musiker dessen Arbeit als erstes abgeschlossen war. Später bot ein professioneller Spieletester seine unschätzbare Mithilfe an. Mit Robert Megone stehen wir nach wie vor in Kontakt für die Weiterführung dieses besonderen Abenteuers.

Stefan Höltgen: Gab es so etwas wie eine alpha- und beta-Version, die von außenstehenden Spielern getestet wurden? Und die späteren Versionen von „Caren" - enthielten die auch Bug Fixes oder wodurch waren die motiviert?

Martin Wendt: Von Caren gibt es im Prinzip 3 Versionen. Zum einen die Wettbewerbsversion die mit extrem heisser Nadel am Ende gestrickt wurde. Einige Elemente wurden buchstäblich in letzter Minute oder gar Sekunde implementiert. Getestet haben das quasi nur ich und Oliver im Wechsel und es haben sich auch einige kleine Fehler in die Version geschlichen. Viele der Skripte für die Räume sind stark miteinander verknüpft und es geht schnell eine Abhängigkeit verloren. Etwa einen Monat später haben wir eine Version 1.1 publiziert die komplett fehlerbereinigt war und auch an der ein oder anderen Stelle mehr Umfang aufwies. Hier war schon unser Profitester Rober Megone involviert. Das war ungemein hilfreich, da nun zum ersten mal nicht nur ein erfolgreiches Durchspielen getestet wurde, sondern auch alle möglichen Aktionen die einem Spieler so in den Sinn kommen könnten. Dialoge wurden dadurch ungleich umfangreicher aber auch insgesamt schlüssiger. Seit dem arbeiten wir weiter an der Story. Es gab eine Version die exklusiv einem Grafikbuch bei Kickstarter beigelegt wurde beispielsweise. Den vorerst letzten publizierten Stand stellt dann die Thank you! Edition 2018 dar. Hier wurde sehr vieles überarbeitet. Etliche neue Räume und Nebenhandlungen sind dazu gekommen und auch das Bedieninterface wurden überarbeitet. Diese Version ist mehr als doppelt so groß wie die Wettbewerbsvariante und stellt de facto die finale Version der Engine dar. Für ein anschließendes Projekt arbeiten wir also vor allem an Story und Spieldesign.

E. Interview mit James Jacobs

Stefan Höltgen: Please tell me about your background in computing/computer science (as a hobbyist and a professional) and about your '2650 history'.

James Jacobs: We had a Tempest MPT-03 system as a child with about eight games. I did not find out about anything about Signetics or even Emerson until much later. I have been a developer for Windows and AmigaOS since the 1990s. I also have been a user of the TRS-80, Commodore 64, etc. I added the other systems to WinArcadia as soon as I had enough information about them. It has been fascinating to learn about them.

Stefan Höltgen: When and why did you start to work on WinArcadia?

James Jacobs: In 2006; I was a user of emulators for many systems, but noticed the Arcadia 2001 emulators at the time were not very good. I was having to patch games to work around bugs in the MESS emulator. The platform seemed neglected.

Stefan Höltgen: Which sources (hardware specs, ROMs, other) did you use for implementing the emulators? (for example the VC4000)

James Jacobs: Datasheets, schematics, video/screenshots of the real machines, examining listing of games and BIOSes, disassembling games/BIOSes, running games on the real machine, etc. I wrote a few test programs which have been included on the Arcadia and Interton multicarts, but having a RAM cart where I could just send code to the real device at will would help. A logic probe would probably also be useful.

Stefan Höltgen: What programming language and IDE or other tools did you use for developing WinArcadia?

James Jacobs: It's written in Microsoft Visual C. I used to use the VACS assembler, DASMx disassembler and my Annotate auto-commenter for working with 2650 code until I integrated those functions into WinArcadia. Also my HowDif utility for comparing dumps, and Audacity and my Taper utility for transforming damaged cassette tape waveforms into loadable game files.

Stefan Höltgen: What „outside environment" did you try emulate (e.g. specs of CRT monitors, sounds from peripherals, ...) What were the problems/benefits of such emulations?

James Jacobs: There is an „ambient sounds" option that will emit teletype noise for the appropriate machines, coin tray sounds for coin-ops, etc. I didn't add any blurring effects as I hate blurry emulators; there are scanlines available for those who want them though. The CRTs emulated are the most commonly used and best documented ones for those systems (the Electronics Australia Low Cost VDU for PIPBUG-based) machines, but they typically were not supplied with any I/O peripherals, so there were a wide variety of input and output devices used with them.

Stefan Höltgen: Did you have contact to the hardware and software developers of the original systems? What did you learn from them?

James Jacobs: I have been in contact with Derek Andrews, programmer of the Leapfrog game for the Voltmace Database (VC 4000 compatible) and Tom Pittman, programmer of the Grand Slam Tennis game for the Emerson Arcadia 2001. They were able to give some historical context, but unfortunately did not have much of their listings, documentation, etc. Also Mr Kessler, the designer of the 2650, is active on some Signetics forums.

Stefan Höltgen: (How) did the retrocomputing community participate to the WinArcadia project?

James Jacobs: Various people have kindly provided hardware, dumps of cassette tapes and ROMs, bug reports, scans of listings/documentation, feature requests, etc., there is a list of the major contributors in the manual. I was hoping others might also work with me on the source code but that has not eventuated to any significant degree yet.

Stefan Höltgen: Please tell me about 1 or 2 glitches of WinArcadia that stem from differences between the emulator and the original systems?

James Jacobs: Envelope generation for the Programmable Sound Generators found in eg. the expanded Elektor TVGC is not implemented yet; very few programs make use of it. The 2636 PVI emulation is still not perfect so there are many games, especially for Interton VC 4000, which have still have graphical glitches.

Stefan Höltgen: Why did you implement additional features like the memory map monitor or the output of the CPU registers?

James Jacobs: It was always intended to be a full-featured emulator, easy to use for gaming yet having all the features needed for developers. Those developer features are useful for other purposes than just writing new games, eg. creating trainers for existing games, disassembling games/BIOSes to investigate compatibility issues, etc. And it makes development of the emulator easier if I have all those tools to see exactly what is happening.

Stefan Höltgen: How close is WinArcadia to the hardwares of the original systems (in terms of [Tijms 2000] accuracy levels)? For example the accuracy of the VC4000 emulation.

James Jacobs: Between levels 2 (Cycle accuracy) and 3 (Instruction level accuracy). It is instruction level accurate as per your definition except that the timing of CPU instructions *is* emulated precisely according to how many cycles they would take on the real machine. The synchronization between CPU and other chips is line-synced on the game systems and frame-sync on the others (PIPBUG, etc.) where there is not enough information available and no programs depend on it. A certain amount of high level emulation for PIPBUG and similar serial-based systems to handle I/O in a flexible and

efficient manner. I do have an experimental build of WinArcadia which uses pixel-synced emulation of the 2636 PVI; I did not see much improvement in compatibility though. I will probably integrate it into the emulator, perhaps as an option, after further development and testing.

Kurzzusammenfassungen

Deutschsprachige Zusammenfassung

Wie ist eine gegenstandsadäquate Erfassung historischer Computer aus informatischer Perspektive möglich? Diese Frage wird unter Berücksichtigung der Medienarchäologie, die die Operativität und damit Ahistorizität technischer Medien betont, gestellt, woraus sich die Computerarchäologie als interdisziplinäre Theorie und Methode ergibt. Die Arbeit erfasst zunächst die Problem technisch defizitärer, widersprüchlicher und idiosynkratischer Computerhistoriografie an ausgewählten Beispielen, um diese geschichtskritisch (H. White, R. G. Collingwood), diskurs- (M. Foucault) und medienarchäologisch (W. Ernst) zu dekonstruieren. Unter dem Begriff der Archäographie werden sodann Werkzeuge und Methoden zusammengestellt, die es ermöglichen operative 'alte' Computer techniknah zu untersuchen und zur Entwicklung einer Theorie mittlerer Reichweite zu beschreiben. Dabei suspendieren Methoden der Informatik (insbesondere der theoretischen, praktischen und technischen Informatik), Elektrotechnik, Logik, Mathematik und Diagrammatik die hermeneutischen Beschreibungsverfahren der Historiografie. Zusätzliche Methoden der Medienwissenschaft und anderer Disziplinen (Re-Enactment, Demonstration, Computerphilologie) ergänzen dieses Methodenset. Frühe Mikrocomputer (1975-1995) bilden den Gegenstand der nachfolgenden Untersuchung.

Retrocomputing, das als eine hobbyistische Form der Computerarchäologie bereits seit mehreren Jahrzehnten betrieben wird, bildet den Rahmen für vier computerarchäologische Projekte, die u. a. im Rahmen medienwissenschaftlicher Lehrveranstaltungen durchgeführt wurden: 1. die Analyse einer 'traditionellen' Computer-Demonstration (Simulation eines springenden Balls) auf unterschiedlichen Plattformen und verschiedenen Implementierungsmethoden/-sprachen seit den 1960er Jahren; 2. die Entwicklung eines "Game of Life" auf einer 8-Bit-Computerplattform unter besonderer Berücksichtigung der simulativen Möglichkeiten zellulärer Automaten für didaktische und 'historiografische' Zwecke; 3. die Entwicklung eines neuen Computerspiels sowie eines modernen Massenspeichers für eine Spielkonsole von 1978 in Hinblick auf die Unterschiede von Software-Emulation und realer Hardware; und 4. die Reparatur eines 8-Bit-Computers von 1977 durch einen Hardware-Hacker vor dem Hintergrund der eingesetzten Werkzeuge, Informationsquellen und nicht-professionellen Vorgehensweisen.

Diese Projekte werden im Anschluss durch eine didaktische Betrachtung gerahmt, bei der der modus operandi des Retrocomputing analysiert wird. Theoretisches, historisches und praktisches Wissen werden hierbei autodidaktisch durch Trial and Error, Gamification und E-Learning im "Learning by Doing"-Verfahren erworben. Retrocomputing verfährt damit auf ähnliche Weise

wie ‘Homecomputing’ (ab den späten 1970er Jahren). Dies wird durch die Gegenüberstellung von Beispielen aus beiden Bereichen veranschaulicht. Die didaktische Reflexion der Retrocomputing-Projekte mündet in den Vorschlag einer Retro-Didaktik, die geeignet wäre durch Komplexitätsreduktion (Beschäftigung mit operativen frühen Mikrocomputern) breitenwirksam Kenntnisse über aktuelle Informatik-Systeme zu vermitteln und zugleich ein historisch-kritisches Bewusstsein der Computerkultur zu generieren.

English Abstract

How can historical computers be described properly from the viewpoint of computer science? By considering media archaeology’s theory of operative and thus a-historical media computer archaeology combines an interdisciplinary set of theories and methods to answer this question. At first, the problems of computer historiography (technical inaccuracy, inconsistency, and idiosyncrasy) will be deconstructed with the help of history criticism (H. Whyte, R. G. Collingwood), discourse archaeology (M. Foucault), and media archaeology (W. Ernst). Following that, technology–oriented tools and methods are gathered for describing ‘old’ computers within an ‘archaeography’ and analyzing them within a mid-range theory. Methods of computer science (from theoretical, practical, and technical c.s.), electronics, logics, mathematics, and diagrammatics supersede hermeneutical methods of historiography. Additional tools (re-enactment, demonstration, computer philology) from media science and other disciplines complement this set of methods. The objects of the following analyzation are early microcomputers (1975-95).

Retro computing, that has been practiced for decades as a hobbyistic way of computer archaeology, sets the frame for four computer archaeological projects that had been implemented within media scientific seminars (and other occasions): 1. a computer philological analysis of a ‘traditional’ computer demo (simulating a jumping ball) on different platforms and in different programming languages since the 1960s; 2. the development of a “Game of Life” on an 8-bit platform to examine the didactical and ‘historiographical’ potentialities of cellular automata; 3. the development of a new computer game for a 1978 gaming console to examine the differences between software emulation and material hardware; 4. the reparation of an 8-bit computer from 1976 done by a hardware hacker to analyze the tools, methods and knowledge-gaining process of such a non-professional approach.

These projects are discussed afterwards to gain the specific didactical modus operandi of retro computing hobbyists. Just like historical home computing (starting from the late 1970s) retro computing autodidactically gathers theoretical, historical, and practical knowledge by trial and

error, gamification, and e-learning through a “learning by doing” procedure. The confrontation of three historical examples with three actual retro computing projects will prove this. The didactical reflection of retro computing projects describes a ‘retro didactic’ that would be useful for a broad application of historic sensitive, computer scientific knowledge with the help of less complex systems (like early microcomputers are).