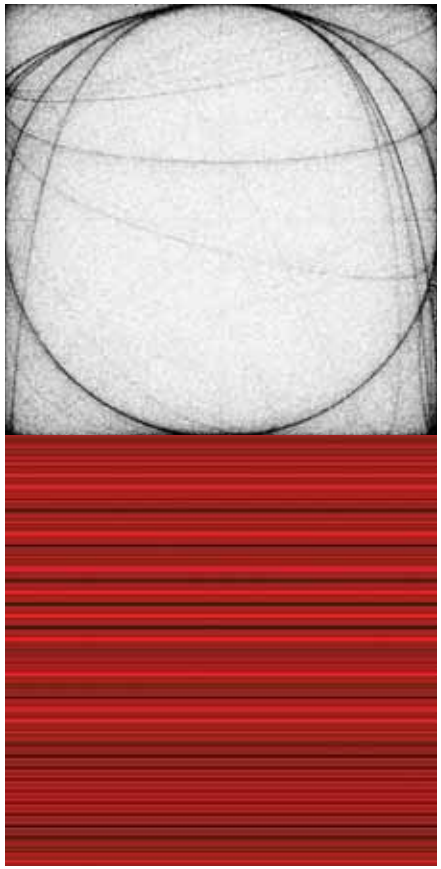
MAGISTERARBEIT

KUNSTINFORMATIK



Ausgeführt am Institut für

Computergraphik und Algorithmen

der Technischen Universität Wien

unter der Anleitung von

Prof. Dr. Werner Purgathofer

durch

Bakk. Bakk. Michael Bliem

Porzellangasse 49/1/4 1090 Wien

Wien, am 24. August 2005 Unterschrift

Inhaltsverzeichnis

Vorwort 5

Einleitung 20

Punkte 24
Grafik mit Java 24
Optimierung 25
MemoryImageSource 29
Pixel 30

Linien 35
Bresenham Algorithmus 35
Implementierung 37
Antialiasing 38
Implementierung 39
Rechtecke 41
Ovale 41

Sound Synthese 43
Java Sound API 43
AudioInputStream 45
Wellenformen 46
Additive und subtraktive Soundsynthese 47
Elektronische Bauteile im analogen Synthesizer 48
Modulationsformen 49
Sampling 52
Wavetables 53

Imagedriven Music 55

Music Instrument Digital Interface 55
MIDI-Channel 56
Control Change 58
Pitch Bend 59
Program Change 59
Erweiterte MIDI-Messages 61
Interaktive MIDI-Verarbeitung 62
Interaktive Soundsynthese 63
Maschinengrenzen (1992) 63
Doppelpufferung 64

Konstruktionsprinzipien 66

Schwarz-Weiß-Grafik 67
Liniengrafik 67
Iterative Systeme 68
Das Apfelmännchen 70
Fraktale Systeme 70
Modulo-Arithmetik 72
Lissajousche Figuren 72
Zeichenautomaten 73
Vergleichende Bildsynthese 74
Motion Capturing 75

Fourier Analyse 77

Fast Fourier Transformation 78
Mehrdimensionale FFT 81

Schlußwort 83

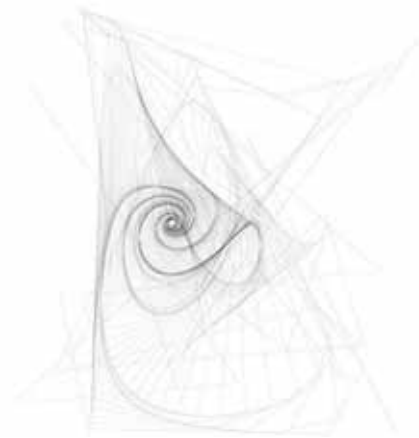
Anhang 85

Code 85
MIDI-Tabellen 140
Links und Literatur 144

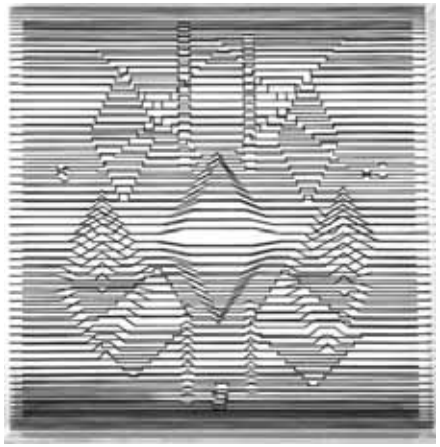
Danke an

meine Eltern für die mentale Unterstützung
Mag. Agnieszka Raganowicz für die Inspiration
Dr. Helwig Hauser für die Klärung grundsätzlicher Fragen zur Diplomarbeit
Prof. Dr. Werner Purgathofer für konstruktive Kritik und die Anregung zum Schreiben des Vorwortes
alle die mich in der Zeit des Schreibens erduldet und unterstützt haben

Vorwort



Da ich bei der Verfassung meiner Diplomarbeit *Kunstinformatik* auf einige Missverständnisse gestoßen bin, sowohl das Thema als auch die Sinnhaftigkeit betreffend, habe ich mich dafür entschieden, in einem persönlich gehaltenen Vorwort, auf die offenen Fragen so weit als möglich einzugehen.

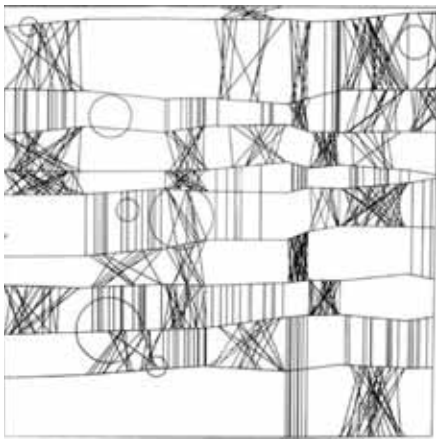


Victor Vasarely
Ibadan A (1965)

Ich selbst habe das Informatikstudium begonnen, da ich, wie viele experimentierfreudige Künstler, auf der Suche nach neuen Ausdrucksformen war. Der Computer mit seinen ungeahnten Möglichkeiten schien mir gerade dazu prädestiniert zu sein, mir dieses Ziel langfristig erfüllen zu können. Ich war begeistert von den Grafiken und Animationen die diese Geräte auf den Bildschirm zauberten. Demos wurden die Programme genannt, die von kleinen Gruppen, bestehend aus Musikern, Grafikern und Programmierern, erstellt wurden und auf ästhetisch reizvolle Weise demonstrierten, was mit der gängigen Hardware alles zu machen ist.

Einige diese Gruppen wurden über die Hackerszene hinaus bekannt. Allerdings gab es nie einen Markt für solche Demos, diese konnten nicht gekauft werden, sondern wurden kostenlos weitergegeben. Obwohl es für mich als Jugendlicher völlig unverständlich war, wie diese Demos erzeugt wurden, übten sie doch eine große Faszination auf mich aus und begleiten mich nun schon zwei Jahrzehnte lang. Dabei demonstrierten sie mir die Fähigkeiten eines Commodore 128, Amiga 500, Atari 1040, Atari Falcon und später der Intel-PCs.

Auf allen diesen Rechnern wurde auch Kunst gemacht. Die Commodorerechner eignete sich vor allem für die Arbeit mit Video, während Atari sich am Musikmarkt behauptete. Doch für beide Betriebe war der Markt nicht groß genug um finanziell überleben zu können und so verschwanden beide langsam von der Bildfläche. Für die ehemalige Klientel wurde der Macintosh zur neuen Plattform, welche sich mit den Bereichen Grafik und Layout ein festeres Standbein in der Computerindustrie verschaffen hatte. Dies lag zum einen daran, dass die meisten Softwarehersteller für Commodore- und Atarirechner, ihre Software auf den Macintosh portierten, zum anderen an der intuitiv benutzbaren grafischen Oberfläche des MacOS, das zusätzlich durch die ansprechende Optik zu überzeugen wusste.

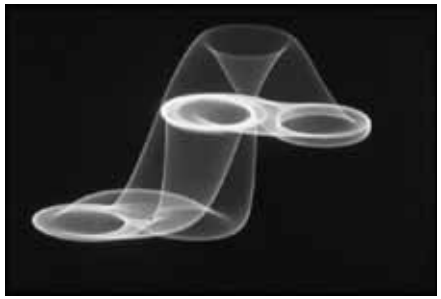


Frieder Nake
Hommage a Paul Klee (1965)

Die Microsoft-Intel-Plattform war für Künstler lange Zeit völlig uninteressant, orientierte sich diese doch weitestgehend an den Bedürfnissen der Wirtschaft und ignorierte die Wünsche der kreativen Sparten. Als Ausnahme muss die Unterhaltungsindustrie erwähnt werden, die vor allem durch Computerspiele einen neuen Absatzmarkt auf diesem Computer-System vorfand.

Die sehr technisch ausgerichteten Unix-Systeme waren zu komplex und undurchschaubar um Künstlern als Werkzeug dienen zu können. Einzige Ausnahme bildete das Next-System (später Open-Step). Die Software war weitestgehend frei zugänglich, das GUI intuitiv bedienbar und der Rechner besaß neben der CPU einen eingebauten DSP der Echtzeitberechnungen ermöglichte. Das Institut IRCAM in Paris entwickelte Anfang der 90er Jahre ausgereifte Einschubkarten für dieses System, die es Musikern erlaubte in Echtzeit äußerst komplexe Soundszenarien zu generieren. Bedient wurde diese Hardware mit der Programmiersprache MAX, bei der nicht mit Text codiert wird, sondern mit Hilfe grafischer Symbole ein Programm erstellt wird. Diese Sprache ist inzwischen die meistgenutzte Programmiersprache von Künstlern und seit einigen Jahren auch in der Lage Video zu verarbeiten. Allerdings wird auch diese Software nur noch auf der Macintoshplattform verwendet, da sich auch Next nicht am Markt behaupten konnte. Die freie Version dieser Sprache für Windows-Systeme (genannt *pd*) funktioniert zwar und wird auch benutzt, allerdings wird diese Software weniger gepflegt und hinkt in ihren Möglichkeiten der Version für MacOS hinterher.

Ich werde gleich noch einmal auf die Programmiersprache MAX zurückkommen, möchte jetzt aber die Aufmerksamkeit wieder auf die Demoprogramme richten. Diese Programme liefen schon auf Rechnern deren Leistungsfähigkeit weit unter derer heutiger Systeme liegen. Trotz dieser Tatsache konnten die Programmierer solcher Demos mit flüssig laufenden Animationen und komplexen mathematischen Berechnungen überzeugend aufzeigen, wie schnell diese Maschinen arbeiten können. Diese oftmals verblüffenden Ergebnisse konnten nur unter Ausnutzung aller Optimierungsmöglichkeiten erreicht werden.

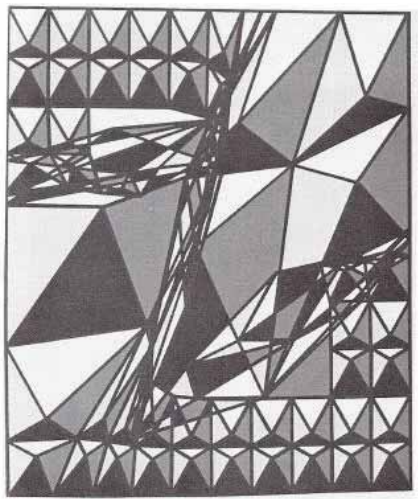


Herbert W. Franke
Electronic Graphics (1962)

Eine der effizientesten Möglichkeiten das Laufzeitverhalten zu verbessern, ist das Programmieren in Assembler, also direkt in Maschinensprache. Dabei befindet man sich so nahe an der Hardware, dass praktisch jeder Prozessortakt optimal ausgenutzt werden kann. Überdies können so auch die Grenzen des Betriebssystems umgangen werden. Zugriffe auf die Grafikhardware müssen nicht über spezielle Routinen, sondern können direkt durch Setzen der Bits im Bildschirmspeicher geschehen. Die Tricks der Hacker kennen dabei keine Grenzen. Als Hardwareexperten schaffen sie es, das System teilweise weiter auszureizen, als dies von den Entwicklern der Hardware vorgesehen war. Zusammenfassend kann man sagen, dass Democoder und Hacker Meister im Umgang mit Hardware (bzw. technisch korrekt: Meister der hardwarenahen Programmierung) sind.

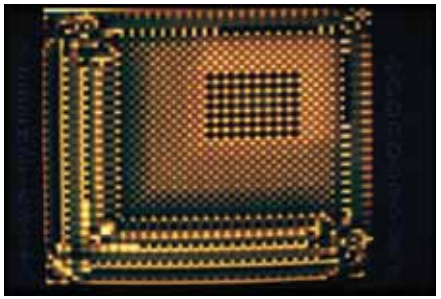
A. Michael Noll
Computer Comp. with Lines (1964)

Wenn wir den Programmierstil der Hacker in Assembler mit jenem der Künstler in MAX vergleichen, finden wir nicht sehr viele Gemeinsamkeiten. MAX ist von der Hardware völlig abstrahiert. Der Programmierer verbindet Symbole nach einer Logik, die der Hardware in keiner Weise entspricht. Hinter diesen Symbolen steckt Code, der die Betriebssystemroutinen optimal ausnutzt. Die Betriebssystemroutinen greifen dann auf die Hardware selbst zu. Will man ein in MAX geschriebenes Programm optimieren, muss man diese Abläufe, die hinter der Sprache vonstatten gehen, sehr gut verstehen. Doch nicht immer ist es so möglich ein befriedigendes Laufzeitverhalten zu erreichen. Notfalls kann ein neues Symbol in der Sprache C geschrieben werden, welches die nötige Performance bietet. Dazu sind aber nur die wenigsten MAX-Programmierer in der Lage. In der Praxis sieht es eher so aus, dass Künstler entweder mit Technikern zusammenarbeiten, die die Fähigkeiten besitzen Laufzeitprobleme zu beseitigen oder auf die tatsächlichen Möglichkeiten die die Hardware eigentlich bietet verzichten. Zusammenfassend können wir hier sagen, dass MAX-Programmierer in der Regel keine Meister der Hardware sind und das vorhandene Werkzeug bzw. Instrumentarium nur mit Hilfe von Technikern wirklich ausreizen können.

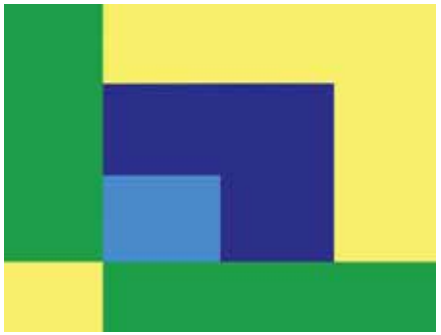


Ruth Leavitt
Prismatic Variation II (1976)

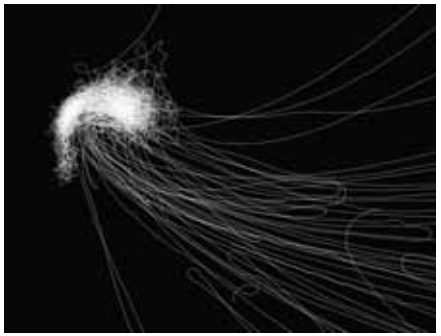
Behalten wir diese Tatsachen im Auge und wenden wir uns nun mehr der Kunst zu. Kunst zu definieren ist meiner Meinung nach nicht möglich und der Versuch selbiges zu tun bleibt Kunsthistorikern und Kunstwissenschaftlern vorbehalten. Wenn ich also hier von Kunst und Künstlern rede, so vertrete ich lediglich meine eigene momentane Ansicht und meine derzeitigen künstlerischen Vorstellungen, die keinen Anspruch auf Richtigkeit erheben können und sollen. Wenn ich Erfahrungen schildere und diese interpretiere, so entsprechen die Schlüsse die ich ziehe ausschließlich meiner subjektiven Sicht. Die selben Erfahrungen würden von anderen Künstlern mit großer Wahrscheinlichkeit vollkommen anders aufgefasst und bewertet werden.



Herbert W. Franke
Green Series (1975)



Edward Zajec
Prismiance (Animation 1984)



Golan Levin
Floccus

Kunst zu schaffen ist ein unkontrollierter, chaotischer, intuitiver und tief emotionaler Akt, der sich der Logik und Realität weitestgehend entzieht. Wann eine solche Schaffensphase eintritt ist unabsehbar, ebenso das Resultat. Regeln gibt es keine, auch nicht die, dass es keine Regeln gibt. Kunst kann sich Themen widmen, Ideen bearbeiten, Ansichten einnehmen. Kunst drückt sich durch den Künstler aus, nicht umgekehrt. Manchmal aber doch umgekehrt. Kunst bleibt ein unentdecktes, rätselhaftes Geheimnis wenn sie lebt und verkommt zu stumpfer Einfalt, wenn sie stirbt. Kunst ist aber unsterblich. Will man Kunst wirklich verstehen, muss man sie leben. Keine Theorie der Welt kann den Besuch eines Museums ersetzen. Ohne wirkliche Hingabe an die Kunst, kann man ihr nur verständnislos gegenüberstehen.

Kunst ist prinzipiell überall und zu jeder Zeit möglich, kann Publikum haben, muss es aber nicht. Es gibt Künstler die ihre Werke nicht der Öffentlichkeit präsentieren wollen oder nur in ganz bestimmten Kreisen, in denen sie den geeigneten Rahmen für ihre Arbeiten sehen. Andere Künstler sind mit ihren Werken freizügiger und suchen die Öffentlichkeit geradezu auf. Beide können dabei erfolgreich sein, wobei Erfolg sich hier nicht marktwirtschaftlich messen lässt. Erfolg kann bedeuten, dass ein Künstler mit seinem Werk zufrieden ist. Erfolg kann heißen, dass die Werke eines Künstlers in weiten Kreisen bekannt werden. Der größte Erfolg ist wohl, wenn ein Künstler in die Kunstgeschichte eingeht, was manchmal auch erst viele Jahre nach der Schaffung der Werke passieren kann. Finanzieller Erfolg ist auch für Künstler nicht unbedeutend, aber meistens zweitrangig.

Ein wichtiger Begriff, der oft im Zusammenhang mit Künstlern erwähnt wird, ist die Genialität. Da Künstler aus rein marktwirtschaftlicher Sicht kaum etwas für die Gesellschaft leisten und deren Arbeit daher wenig geschätzt wird, werden diese oft nur unter dem Gesichtspunkt der Genialität toleriert. Ein Mensch der sich als Künstler definiert steht sofort unter dem gesellschaftlichen Zwang sich in irgendeiner Form als genial beweisen zu müssen. Genial sind aber nur die wenigsten Menschen und es ist ein sehr menschliches Gefühl nicht gerne auf die nicht vorhandene Genialität hingewiesen zu werden. Wird von einem Menschen aber Genialität erwartet, wo diese nicht vorhanden ist, stellt man ihn damit bloß. Um sich vor dieser Bloßstellung zu schützen, nehmen viele Künstler anderen gegenüber eine vorsichtige, distanzierte Haltung ein, die leider sehr oft als Arroganz aufgefasst wird. Tatsächlich haben Künstler aber oft erstaunlich viele Talente, die beachtet werden wollen. Umso mehr schmerzt es, wenn durch das nicht Beweisen können der Genialität, die wahren Begabungen übersehen werden.

Was von Künstlern aber zu Recht verlangt wird, ist die Beherrschung ihrer Werkzeuge und Ausdrucksmittel. Obwohl auch diese Regel nicht ohne Ausnahme existiert, kann ernsthafter Umgang mit Medien und Geräten nur dann glaubhaft wirken, wenn der Künstler sein Handwerk wirklich versteht und mit seinem Instrumentarium umzugehen weiß. Man darf von einem Künstler erwarten, dass seinen Arbeiten eine intensive Beschäftigung mit der behandelten Materie vorausging. So kann ein Pianist, der mit einer einzigen Taste spielt, durchaus als Künstler angesehen werden. Allerdings muss er damit überzeugen können und das ist in der Regel nur möglich, wenn er sich zuvor intensiv dem Spiel mit einer Taste gewidmet hat. Wirklich glaubhaft wird dieser Pianist, wenn er nicht nur diese eine Taste beherrscht, sondern auch mit den anderen Tasten des Klaviers umzugehen weiß. Hat er das aber einmal bewiesen, wäre es eine Zumutung, diesen Beweis immer wieder bringen zu müssen. Gerade in Zeiten der Beschäftigung mit einer

Taste, wird der Umgang mit den anderen Tasten vernachlässigt werden. Zusammenfassend behaupte ich, dass ein Künstler im traditionellen Sinn, ein Meister seines Mediums ist und im schaffenden, kreativen Akt dieses dann intuitiv beherrscht.

Laurence Gartel
Catacomb for a Princess (1986)

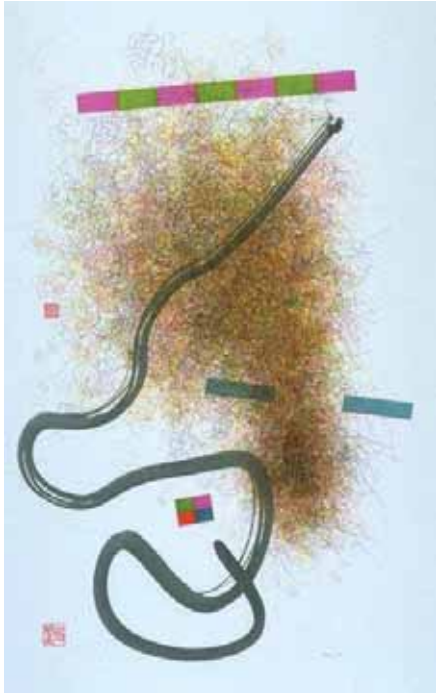
Wenden wir uns nun der Computerkunst zu. Der Begriff ist relativ neu, der Terminus wird zwar benutzt ist aber kunsthistorisch betrachtet wenig geeignet um eine Kunstgattung zu beschreiben. Computerkunst wird der Medienkunst zugeschrieben, wobei auch diese Bezeichnung von Kunsthistorikern noch diskutiert wird. Ungeachtet der Terminologie hat sich die Medienkunst ihren Rang in der Kunstwelt erobert und zählt zu den etablierten Kunstrichtungen der Gegenwart. Stärkster Vertreter dieser Richtung ist die Videokunst, Computerkunst gewinnt immer mehr an Beachtung, steht aber noch am Anfang ihrer Entwicklung und macht deshalb nur einen sehr kleinen Teil innerhalb der Medienkunst aus. [1]

Computerkunst (welche oft synonym mit digitaler Kunst verwendet wird) kann sich primär mit dem Computer selbst auseinandersetzen und diesen als Medium benutzen oder der Computer dient als Werkzeug, wie bei den Gattungen digitale Malerei, 3D-Kunst, Fotomanipulation, mathematische Kunst, Netzkunst, interaktive Kunst, digitale Musik, Multimedia-Kunst und Mischformen aus den genannten Bereichen. Von allen diesen Richtungen wird der Netzkunst derzeit von Kunsthistorikern die meiste Aufmerksamkeit geschenkt, betretet diese doch ein so fremdes Terrain, indem das Internet als Medium angesehen wird, was völlig neue Kunstkonzepte notwendig macht. Die Videokunst zählt zwar nicht zur Computerkunst kann aber - aufgrund des Wandels der Werkzeuge von analog zu digital - immer schwerer von der Computerkunst differenziert werden. Es gibt also sehr viele Kunstrichtungen, die sich in hohem Maße des Computers bedienen, welche wir hier zusammenfassend als Computerkunst bezeichnen wollen.

Ich möchte nun kurz aufzeigen, wie die Ausbildung zum Künstler in der Regel vonstatten geht und wie die Ausbildung zum Computerkünstler im speziellen aussieht bzw. aussehen könnte. Kunst ist eine der ältesten Wissenschaften überhaupt und so durchläuft auch ein Künstler eine akademische Ausbildung. Für die Zulassung zu einem Kunststudium ist es notwendig sein künstlerisches Talent unter Beweis zu stellen. Musiker müssen bei dieser Zulassungsprüfung ihr Instrument bereits weitgehend beherrschen. Das Musikstudium selbst widmet sich dann der künstlerischen Weiterentwicklung. Ein Maler muss ebenfalls schon in der Lage sein technisch sehr gut Zeichnen zu können, bevor er sich im Studium künstlerischen Fragen widmen kann. Als Voraussetzung für die künstlerische Ausbildung steht im klassischen Sinn also auch die technische Kompetenz. Strebt man die Ausbildung zum Computerkünstler an, stellt sich also zuallererst die Frage nach der technischen Ausbildung.

Yves Netzhammer
(Animation 1999)

Als ich 1992 zu studieren begann, gab es in Wien auf der Universität für angewandte Kunst bereits das Studium visuelle Mediengestaltung, welches sich aber nur am Rande mit Computerkunst beschäftigte. Der Umweg über das Informatikstudium an der technischen Universität erschien daher lohnender. Das Informatikstudium stand damals noch in keinster Weise unter dem Aspekt von Kunst sondern war rein naturwissenschaftlich geprägt. Durch den großen Erfolg der Informationstechnologie in der Wirtschaft, wurden die Studieninhalte immer mehr den Bedürfnissen der



Roman Verostko
Carnival (1989)

Joan Truckenbrod
Kinetic Discharge (1991)

Industrie angepasst. Seit dem neuen Studienplan aus dem Jahre 2003, in dem das Informatikstudium in verschiedene Einzelstudien unterteilt wurde, um diese Bedürfnisse gezielter befriedigen zu können, gibt es auch das Studium Medieninformatik. Dieses behauptet von sich - unter anderem - eine Brücke zur Kunst schlagen zu wollen, schafft dies bisher aber nur in Zusammenarbeit mit der Architektur, was daran liegen könnte, dass Architektur seit jeher zweckgebundener ist als andere Kunstrichtungen.

Die Universität für angewandte Kunst hat auf den Erfolg der Medienkunst mit der Einführung dreier medienübergreifender Studienrichtungen reagiert, wobei sich eines dieser Studien der digitalen Kunst widmet. Bei diesem Studium wird bei der Zulassungsprüfung nur ein begrenztes technisches Vorwissen und Können vorausgesetzt. Das Studium selbst widmet sich daher nicht ausschließlich der künstlerischen Weiterentwicklung sondern im Besonderen auch der technischen Qualifikation. Meiner Meinung nach leidet die Qualität des Studiums unter dieser Lösung. Einerseits ist eine wirklich fundierte technische Ausbildung innerhalb des zeitlichen Rahmens von vier Jahren kaum möglich, andererseits kommt durch die - im Vergleich zu anderen Kunststudien - starke technische Prägung die künstlerische Entwicklung zu kurz.

Prinzipiell ist es auch als Autodidakt möglich eine Karriere als Künstler einzugehen. Akademische Abschlüsse zählen in der Kunstwelt weit weniger als in der Wirtschaft. Allerdings reduziert eine offizielle, künstlerische Ausbildung den gesellschaftlichen Druck auf den Künstler, da weithin bekannt ist, wie selektiv die Zulassungsprüfungen sind. Durch eine solche Zulassung wird die künstlerische Begabung daher weniger stark in Frage gestellt. Eine fundierte Ausbildung zum Computerkünstler gibt es aber derzeit noch nicht, was ein teilweise autodidaktisches Aneignen der dafür nötigen Kompetenzen erzwingt.

Kann ein Informatikstudium die technische Grundlage eines Computerkünstlers sein? Diese Frage möchte ich nun an einigen Überlegungen erörtern. Prinzipiell beschäftigt sich kein Studium so sehr mit den Möglichkeiten eines Computers, wie das Informatikstudium. Da dieses sich aber nur sehr langsam auch Kunstströmungen öffnet, ist zu klären, inwieweit ein werdender Künstler von den angebotenen Lehrveranstaltungen auch wirklich profitieren kann. Betrachten wir das Studium zunächst unter dem Gesichtspunkt, welcher weiter oben erörtert wurde, nämlich, dass ein Künstler sein Medium zu beherrschen und dadurch intuitiv zu bedienen vermag. Wir haben bereits festgestellt, dass Hacker und Demoprogrammierer Spezialisten der hardwarenahen Programmierung sind. Sehen wir die Hardware als das Medium eines Computerkünstlers, so wird man durch Kenntnisse der technischen Informatik und der Programmierung in Assembler zum Meister im Umgang mit diesem Medium. Ob dieser Umgang auch intuitiv erfolgen kann ist allerdings höchst fraglich und hängt stark von der technischen Begabung des betreffenden Künstlers ab.

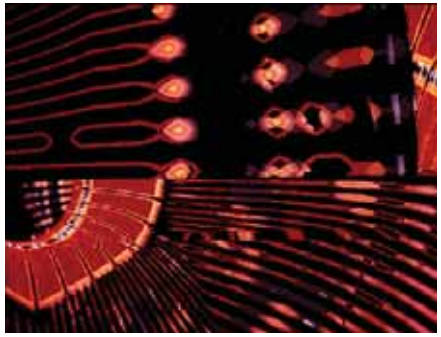
Im Allgemeinen kann man intuitive Assemblerprogrammierung wohl kaum als Ausbildungsziel erwarten. Da aber alle modernen Hochsprachen die Einbindung von Maschinencode unterstützen und Laufzeitprobleme oft dadurch beseitigt werden können, indem zeitkritische Routinen in Assembler geschrieben werden, kann das Ausbildungsziel dahin-

gehend abgeändert werden, dass der nichtintuitive Einsatz von Assembler - bei Bedarf - beherrscht wird. Lehrveranstaltungen, die sich verstärkt der Hardware und der Assemblerprogrammierung widmen, können daher als essentiell für die Ausbildung zum Computerkünstler angesehen werden.

Aus obigen Überlegungen wird ersichtlich, dass ein Computerkünstler, der sich primär dem Medium Computer zuwendet, als eine Art Programmierkünstler angesehen werden kann, während Computerkünstler, die den Computer als Werkzeug benutzen, indem die vorhandene Software ausgereizt wird, als Anwendungskünstler bezeichnet werden können. Das Ziel, das es zu erreichen gilt, ist in beiden Fällen Kunst. Technisch gesehen handelt es sich beim Programmierkünstler um einen Softwareentwickler und beim Anwendungskünstler um einen Softwareanwender. Das Informatikstudium richtet sich weitgehend an Softwareentwickler, mit dem Ziel, diese mit den Fähigkeiten auszustatten, die notwendig sind, um Software zu erstellen, die die Bedürfnisse der Softwareanwender befriedigt. Dieses Ziel entspricht aber in keiner Weise jenem eines Programmierkünstlers, dessen Ziel nicht die Schaffung von Software, sondern die von Kunst ist.

Für Computerkünstler, die sich auf Anwendungen spezialisieren, ist das Informatikstudium kaum als technische Basis geeignet. Die Fähigkeit mit Anwendungssoftware umgehen zu können, wird von Informatikstudenten stillschweigend vorausgesetzt. Ein werdender Informatiker muss technisch bereits so versiert sein, dass ihm die Einarbeitung in neue Softwareprodukte keine Mühe mehr kostet. Die Arbeit mit der fertigen Software bezeichnet ein Informatiker als leicht. Allerdings ist sie das nur technisch betrachtet. Der kreative, intuitive Umgang mit Software wird nicht selten von der Benutzeroberfläche und dem eingeschränkten Interface Maus, Keyboard und Monitor stark beeinträchtigt. Tastaturkürzel intuitiv zu beherrschen ist für Anwendungskünstler in etwa dasselbe, wie für Programmierkünstler die intuitive Assemblerprogrammierung. Aus diesem Grund ist vielen Werken von Computerkünstlern die technische Herkunft sehr stark anzumerken (was völlig wertfrei zu verstehen ist). Als Beispiel sei hier auf den Computerkünstler James Faure Walker verwiesen, welcher sich unter anderem den Funktionen Cut, Copy und Paste widmete und damit großartige Bilder schuf. Ich sehe es als eine Aufgabe der Medieninformatik, sich auch den Bedürfnissen der Kunst verstärkt zuzuwenden und Software zu entwickeln, die es Computerkünstlern ermöglicht ihre Visionen zu verwirklichen ohne dabei auf Softwareprodukte angewiesen zu sein, die eigentlich für Anwender aus anderen Berufssparten entwickelt wurden, wie dies z.B. bei Photoshop der Fall ist, das primär nicht für Künstler sondern Grafiker entwickelt wurde.

Für Computerkünstler, die sich dem Medium Computer selbst zuwenden, bietet das Informatikstudium aber trotz der unterschiedlichen Zielsetzung, eine große Auswahl an nützlichen Lehrveranstaltungen. Zum einen vermittelt die technische Informatik, wie gesagt, das Basiswissen um die Funktionsweise des Instruments Computer überhaupt verstehen zu können. Kenntnisse, die Netzwerktechnik betreffend, sind angesichts der sich rasch verbreitenden Netzkunst von Bedeutung. Weiters empfehlen sich die Lehrveranstaltungen aus Softwareengineering, die sich verstärkt mit Softwareentwicklung und Programmierparadigmen auseinandersetzen. Zwar ist gerade dort das auf Software als Endprodukt ausgerichtete Ziel besonders spürbar, andererseits bieten Themen wie Compilerbau,

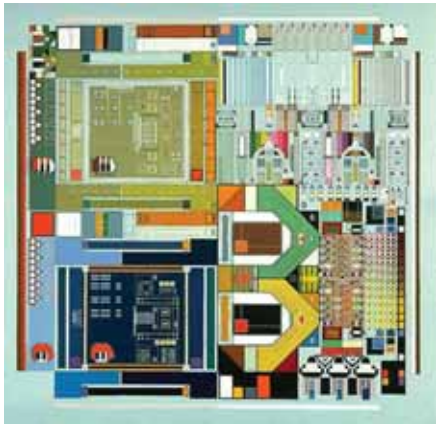


David Em
Navajo (1978)

Laurence Gartel
Banana and Wine (1978)

James Faure Walker
Forms in Motion: Buster (2001)

Marius Watz
Interaktive Software (2002)



Mark Wilson
untitled light gray ground (1973)

Gerhard Mantz
master and slave (2003)

Codewiederverwendung und Softwareanalyse nützliche Einblicke, die einem Programmierkünstler dabei helfen können seinen eigenen Programmierstil zu entwickeln, der weitestgehend intuitiv erfolgen kann.

Doch gerade die Vermeidung dieser Individualisierung des Programmierens ist eines der Hauptanliegen der Industrie an die Softwareentwicklung und wird daher entsprechend gefördert. Dieser Umstand macht es Programmierkünstlern besonders schwer, aus den angebotenen Inhalten die für ihn relevanten Informationen herauszufiltern. Ansonsten empfehlen sich natürlich viele Lehrveranstaltungen aus Medieninformatik, welche trotz der noch immer vorhandenen Kluft zur Kunst, dieser von allen Informatikstudien am nächsten steht. Die dort angebotenen Lehrinhalte können - je nach Interesse - zur Spezialisierung auf einzelne Medien dienen, welche, durch die Digitalisierung derselben, auch nach und nach Bestandteile der Computerkunst werden. Aus der Sicht des Programmierkünstlers kann man sich von den dort angebotenen Lehrveranstaltungen vor allem Kenntnisse bezüglich der Programmierung von Hardware erwarten, welche die geschaffenen Werke in irgendeiner Form zur Geltung bringen kann, z.B. akustisch mit Hilfe einer Soundkarte oder optisch über Grafikkarte und Monitor.

Eine Spezialisierung auf alle digitalen Medien ist unmöglich. Zum einen gibt es bereits eine Vielzahl an Ein- und Ausgabemöglichkeiten, von denen viele als Medium im künstlerisch experimentellen Sinn nutzbar sind, zum anderen werden es durch die fortschreitende Digitalisierung immer mehr. Ich möchte hier kurz die gängigen Geräte aufzählen mit denen ich bisher künstlerisch gearbeitet habe, um zu zeigen, wie groß diese Anzahl bereits für akustische und optische Medien ist. Für die akustische Umsetzung verwendete ich DAT, CD, Soundkarte, Mischpult, Effektgeräte, Hall, Synthesizer, Sampler, Drummachines, MIDI-Mixer, Sequenzer, Tonbandmaschinen, Mehrkanalmaschinen, ADAT, PC-Lautsprecher, u.a. Für die optische Umsetzung Grafikkarte, Monitor, Tintenstrahldrucker, Photo-CD, Video-CD, DVD, Scanner, Digitalkamera, DV-Kamera, Photodrucker und Videobeamer. Die analogen Geräte die ebenfalls mit PCs verbunden werden können, wie z.B. TV und LP wurden dabei nicht berücksichtigt.

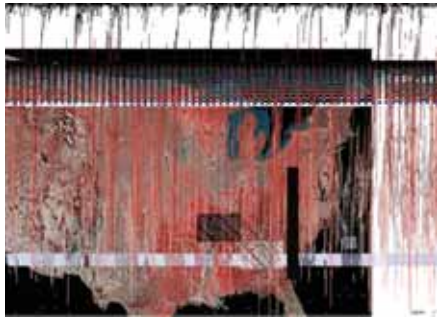
Zusätzlich wird gerade im Kunstbereich verstärkt an der Entwicklung neuer Interfaces geforscht, welche die Grenzen der bisherigen überschreiten und einen neuen für Künstler ansprechenderen Zugang zum Computer bieten sollen. Zu nennen wären kommerzielle Produkte wie MIDI-Saxophon, Masterkeyboard, MIDI-Fader, Tabletts, Joysticks, Joypads sowie jene, welche nicht in Produktion gingen, sondern Einzelanfertigungen waren und sind. Im Musikbereich ist das z.B. der Sentograph (Tamas Ungvary), welcher feine Nuancen von Fingerbewegungen diskretisiert und die so gewonnenen Daten von einem Computer zu Klang verarbeitet werden können. In der Malerei kann ein Plotter mit aufgesetztem Pinsel (Roman Verostko) als Beispiel dienen. Tendenzen, in welche Richtung sich diese Entwicklungen hinbewegen, sind aufgrund der Fülle von Künstlern die sich der Entwicklung solcher Geräte widmen, nicht auszumachen. Es ist ebenfalls nicht abzusehen ob und welchen Geräten und Medien sich die Computerkunst in Zukunft verstärkt zuwenden wird.

In der technischen Ausbildung zum Computerkünstler erscheint es angesichts der großen Anzahl und dem schnellen Wandel der Medien sinnvoll, sich den Daten selbst, anstatt den, die Daten tragenden, Medien, zuzuwenden. Ich

möchte diese Aussage anhand vierer Beispiele, die den Großteil der in der Computerkunst anfallenden Daten abdecken, kurz näher erklären. Als erstes Beispiel dient digitalisierter Klang, der in Form von Samples vorliegt. Ein Programmierkünstler sollte in erster Linie in der Lage sein, mit diesen Samples intuitiv umgehen zu können, sei es indem diese weiter verarbeitet oder überhaupt erst erzeugt werden. Diese Rohdaten eignen sich aber nicht direkt für die Übertragung auf ein Medium, sondern müssen zuerst durch entsprechende Komprimierung, Anordnung und Beschreibung in einer Datei mit entsprechendem Dateiformat untergebracht werden. Mit diesem rein technischen Vorgang sollte ein Programmierkünstler, so weit es geht, aber nicht konfrontiert werden. Selbiges gilt für MIDI-Daten, die in Form von MIDI-Messages vorliegen. Zwar sollen diese Messages kreativ erzeugt und verarbeitet werden können, wie diese aber in einem MIDI-File gespeichert werden, ist für einen Künstler nicht weiter von Interesse. Der Umgang mit Pixel einer Grafik muss ohne geistige Anstrengung beherrscht werden, nicht aber die Fähigkeit die verschiedensten Grafikformate zu verstehen. Die Arbeit mit Video und Animation verlangt nach gefühlsbetontem Umgang mit Frames. Kenntnisse bezüglich der Codierung und Decodierung können aber nicht vorausgesetzt werden.

Harold Cohen
Meryl

Die Fähigkeit Daten von und auf ein entsprechendes Medium - bei Bedarf - zu übertragen, sollte jedoch vorhanden sein. Dabei muss aber grundsätzlich zwischen der Aufzeichnung und der Schaffung von Kunst unterschieden werden. Die Aufzeichnung von Kunst auf ein Medium ist im traditionellen Sinn nicht die Aufgabe des Künstlers selbst, sondern die von, zu diesem Zweck ausgebildeten, Technikern. In der Musik z.B. sind dies die Tonmeister, in der Malerei Experten des Gemäldedrucks. Gehört der Prozess der Herstellung eines Mediums zur Schaffung eines Werkes, so liegt diese Aufgabe beim Künstler selbst. Als Beispiel aus der bildenden Kunst sei hier auf die Photografie verwiesen, bei der die Entwicklung eines Fotos sehr wohl dem Photografen selbst unterliegt.



Mark Napier
Feed USA (2003)

Bevor ich den Begriff Kunstinformatik zu definieren versuche, möchte ich noch eine Stellungnahme zu den unterschiedlichen Auffassungen von Präsentation im technischen sowie im künstlerischen Sinn abgeben. Präsentation im technischen Sinn ist ein Aufzeigen von Sachverhalten, Problemen und Lösungen zu einem Thema, wobei der Vortragende meist die Person ist, die sich genau diesem Thema entweder forschend oder analysierend angenähert hat. Der Vortragende distanziert sich dabei persönlich weitgehend von der vorgetragenen Materie. Die Beschäftigung mit derselben lässt sich aber anhand der geistigen und praktischen Kompetenz - bezogen auf das Thema - feststellen. Präsentation im künstlerischen Sinn bedeutet das zur Schau stellen der Kunstwerke in einem, den Werken angemessenen, Rahmen, wobei der Künstler selbst nur selten diejenige Person ist, die diese Arbeiten vorstellt. Der Künstler distanziert sich dabei nicht persönlich von seinen Werken, sondern stellt sich bewusst den Reaktionen des Publikums auf seine Arbeit. Zwar widmet sich auch das Informatikstudium verstärkt der Präsentation, allerdings bisher nur im technischen Sinn. Ob es in Zukunft auch künstlerisch gehaltene Präsentationsformen geben wird, was die Attraktivität des Studiums für werdende Computerkünstler weiter steigern würde, bleibt abzuwarten.

Ich habe mich dazu entschlossen sowohl mein Studium, welches ich individuell aus den verschiedensten Studienrichtungen wie elektroakustische Komposition, Musikwissenschaft, Publizistik aber hauptsächlich Informatik zusammengestellt habe, als auch die Diplomarbeit für dieses Studium *Kunstinformatik* zu nennen. Der Begriff wird -

Wassily Kandinsky
Composition 8 (1923)

Kasimir Malewitsch
Suprematist Painting (1915/16)

sehr weit gefasst - vom interdisziplinären Arbeitskreis Musik- und Kunstinformatik der Johannes Gutenberg-Universität in Mainz bereits verwendet. Ich möchte diesen allerdings enger fassen, um Überschneidungen zur Medieninformatik weitestgehend zu vermeiden. Kurz gesagt, sehe ich in der Kunstinformatik den technischen Aspekt der Computerkunst. Das Studium Kunstinformatik soll werdenden Computerkünstlern die Technik zur Ausübung ihres Handwerks vermitteln. Darunter fällt in erster Linie die Erarbeitung eines intuitiven Programmierstils unter Ausreizung aller technischen Möglichkeiten, aber auch die Fähigkeit die Rahmenbedingungen für die künstlerische Arbeit selbständig herstellen zu können. So soll ein Programmierkünstler in der Lage sein effiziente Methoden entwickeln zu können, welche in der schaffenden Phase intuitiv einsetzbar sind.

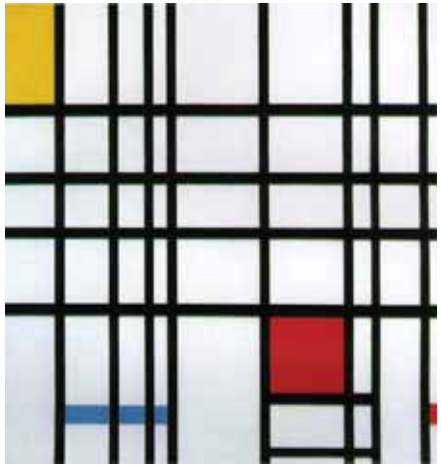
Nach dieser Definition gehört auch die Ausbildung zum intuitiven Umgang mit Anwendungssoftware zum Studium Kunstinformatik. Dabei spielt Übung und Praxis im Umgang mit der Software eine wesentliche Rolle, welche lehrend aber nicht übermittelt werden kann. Diese Fähigkeiten zählen somit zu jenen, welche selbständig, parallel zum Studium, erarbeitet werden müssen. Das Angebot an verwendbarer Software wächst ständig und möglicherweise wird es in Zukunft - wenn die Medieninformatik die Brücke zur Kunst tatsächlich schließt - auch Software geben, die den Bedürfnissen von Computerkünstlern stärker entgegenkommt.

Wie eingangs gesagt, übten Computerdemos eine große Faszination auf mich aus und waren ein Motivationsgrund das Informatikstudium zu beginnen. Wie man solche Demos macht, habe ich bis zuletzt nicht gelernt. Aus diesem Grund wollte ich zum Abschluss meines Studiums - mit der Diplomarbeit - die verbliebenen Wissenslücken schließen. Computerdemos werden langsam auch von Kunsthistorikern untersucht und gelten in weniger traditionellen Kunstkreisen schon als künstlerische Werke. Meiner Meinung nach gehören diese nicht zur Computerkunst, da keine künstlerische Absicht hinter der Erzeugung dieser Programme steht. Diese sind zwar ästhetisch betrachtet ansprechend, doch Ästhetik ist seit den verschiedensten Kunstströmungen der Moderne kein wesentliches Kennzeichen von Kunst mehr. Was diese Programme aber aufzeigen sind die technischen Möglichkeiten und Grenzen die das Medium Computer bietet und umschließt. Und genau innerhalb dieser Grenzen und mit den vorhandenen Möglichkeiten soll ein Computerkünstler, der Meister seines Instruments ist, Spielen, Experimentieren und letztendlich Schaffen können.

Die Diplomarbeit *Kunstinformatik* ist lehrbuchartig aufgebaut um denjenigen Lesern, welche vielleicht ebenfalls den Weg zur Computerkunst suchen, Hilfe zur Selbsthilfe bieten zu können. Anders als bei der rein wissenschaftlichen Informatik ist das Ziel nicht die Fähigkeit funktionierenden Code schaffen zu können, sondern in kurzer Zeit schaffend - durch geeigneten Code umgesetzt - Ideen zu verwirklichen. Zwar können und sollen die Implementierungen, welche in der folgenden Arbeit aufscheinen, auch eingesetzt werden, in erster Linie soll die Diplomarbeit aber aufzeigen, wie in der Computerkunst programmiert werden kann. Ein ausgebildeter Kunstinformatiker soll technisch gesehen in der Lage sein mit Leichtigkeit anspruchsvoll Programmieren zu können, um sich im Folgenden - mit dem technischen Handwerk ausgerüstet - künstlerischen Fragen und der künstlerischen Weiterentwicklung vollständig widmen zu können. Zwar richtet sich die Diplomarbeit in erster Linie an (werdende) Computerkünstler, ich hoffe aber mit der

Arbeit auch kunstinteressierte Informatiker und informatikinteressierte Künstler ansprechen zu können.

Zum Abschluss des Begleittextes möchte ich noch kurz auf die Geschichte der Computerkunst eingehen. Ihren Bezug zur traditionellen Kunst und Musik findet die Computerkunst im Futurismus, dem Kubismus und der geometrisch abstrakten Kunst des Expressionismus. Schon W. A. Mozart entwarf einen Algorithmus zur Erzeugung von Musik. Ernsthafte algorithmische Komposition wurde allerdings erst Anfang des 20. Jahrhunderts betrieben. Tabelle 1.1 stellt einige Künstler vor, deren Werke bereits vor Erscheinen der Computerkunst Elemente enthalten, die an die Möglichkeiten und die Strukturen von computergenerierter Grafik und Musik erinnern.



Piet Mondrian
Composition with Red, Yellow and Blue (1921)

Wassily Kandinsky geb. 1866 in Moskau (RUS), lebte und arbeitete in Moskau, München und Neuilly sur Seine (F), gest. 1944
Kasimir Malewitsch geb. 1878 nahe Kiev (RUS), lebte und arbeitete in Moskau und Leningrad, gest. 1935
Piet Mondrian geb. 1872 in Amersfoort (NL), lebte und arbeitete in New York, gest. 1944
Victor Vasarely geb. 1908 in Pecs (HU), lebte und arbeitete in Paris, gest. 1997 [2]
Iannis Xenakis (Komponist) geb. 1922 in Braila (RUM), lebte und arbeitete in Paris, gest. 2001

Tabelle 1.1

Victor Vasarely
Vega Sakk (1968/69)

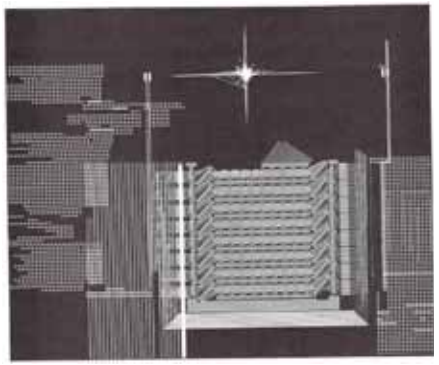
Als Geburtsstunde der Computerkunst kann das Jahr 1965 gelten, als die drei Mathematiker Georg Nees und Friedrich Nake in Stuttgart sowie A. Michael Noll in New York ihre öffentliche Präsentation von Computergrafiken als Computerkunst deklarierten. Tabelle 1.2 präsentiert einige Ergebnisse der Pioniere der Computerkunst. Wurde die Computerkunst in ihren Anfängen noch als Hobby von Mathematikern und Technikern belächelt, so gewannen später mehr und mehr Künstler die Oberhand. In den ersten Jahren wurden die von Computern generierten Bilder mit mechanischen Plottern auf Karton gezeichnet oder auf Endlospapier ausgedruckt. Spätestens seit der 1979 erstmals in Linz veranstalteten ars electronica, verlagerte sich das Interesse auf die Ausgabe auf den Bildschirm. [10]

Diese Grafiken und Animationen waren zunächst einfache Schwarz-Weiß-Bilder, da die damaligen Monitore monochrom waren. Es gab also keine Möglichkeit Pixel anders einzufärben als entweder mit Schwarz oder mit Weiß. Damalige Rechner hatten eine maximale Auflösung von 320 mal 256 Punkten. Doch gerade durch die sichtbaren Pixel und die geringe Farbtiefe entstand mit der Zeit eine eigene Ästhetik, eine direkt mit Computern verbundene

Assoziation, die eine neue und bisher einzigartige Bildgestaltung ermöglichte. So unnatürlich diese kleinen Rechtecke auch erscheinen mögen, sie sind untrennbar mit Computern verbunden. Ein Pixel - vor allem in einem reinen Schwarz-weißbild - symbolisiert das Phänomen Digital, die Eindeutigkeit eines Bits, die Kompromisslosigkeit einer Maschine für die es nur 'entweder - oder' gibt, dazwischen ist Leere.

Kenneth Knowlton
Nudebild (1966)

Die digitale Revolution hat inzwischen auch Einzug in die Fotographie und Videotechnik gehalten, die Auflösung moderner Monitore hat die 2 Millionen-Pixel-Grenze überschritten, der Farbraum hat sich auf über 16 Millionen Farben ausgeweitet. Computergrafik ist längst photorealistisch geworden. Und im Schatten dieser großartigen technischen Errungenschaften feiert das alte Bild der Computergrafik vor allem im Kunstbereich aber auch in Werbung, Grafik und Layout sein Comeback.



Lillian F. Schwartz
Night Scene (1975)

Herbert W. Franke, geb. 1927 in Wien (A), lebt und arbeitet in München [3]
Frieder Nake, geb. 1938 in Stuttgart (D), lebt und arbeitet in Bremen (D)
Charles Csuri, geb. 1922 (USA)
A. Michael Noll, geb. 1939 (USA)
Edward Zajec, geb. 1938 in Triest (IT), lebt und arbeitet in USA
Georg Nees, geb. 1926 Nürnberg (D), lebt und arbeitet in Erlangen (D)
Kenneth Knowlton, geb. 1931 (USA)
Ruth Leavitt, geb. 1944 (USA)
Lillian F. Schwartz, geb. 1927 (USA)
Vera Molnar, geb. 1924 in Budapest (HU), lebt und arbeitet in Frankreich

Tabelle 1.2

Charles Csuri
SineScape II (1967)

Die Computerkunst wurde lange Zeit weder toleriert noch beachtet. Nur wenige Künstler erreichten zumindest ein so großes Publikum, dass sie die Aufmerksamkeit von Kunsthistorikern wecken konnten. Eine Ausnahme bildet der Maler Harold Cohen, der mit seiner Software AARON für großes Aufsehen sorgte. Diese Software ist in der Lage selbständig Gemälde mit Personen und Pflanzen zu erstellen. Beachtlich daran ist, dass die Bilder nicht computer-generiert wirken und die Strichführung sehr ausdrucksstark ist. Der Sourcecode wurde bis heute nicht freigegeben, dafür aber die Software, welche inzwischen nicht mehr weiterentwickelt wird. Tabelle 1.3 listet einige Computer-künstler auf, die in der Kunstwelt als etabliert bezeichnet werden können.

Obwohl die Computerkunst inzwischen ein festes Standbein in der Kunst hat, lebt die Szene nach wie vor vom gegenseitigen Austausch im Internet. Es gibt aber auch schon echte Museen die sich ausschließlich der Computerkunst widmen, wie z.B. das Digital Art Museum [8] in Berlin. Die ars electronica hat ihre Bedeutung in der Kunstwelt dagegen weitestgehend verloren. Diese ist inzwischen zu einem stark von der Industrie geprägten Festival für Techniker und Designer geworden. Für Computerkünstler ist sie aber dennoch interessant, da gerade dort wichtige Informationen zu den gegenwärtigen Möglichkeiten der Medieninformatik zugänglich sind. Andere Festivals haben dagegen für die gesamte Medienkunst sehr an Bedeutung gewonnen. Wichtigste Vertreter sind dabei documenta und Biennale. Die Tabellen 1.4 - 1.6 zeigen gegenwärtige Computerkünstler, Netzkünstler und künstlerisch ambitionierter Grafiker und Techniker, welche teilweise im Sinne der Computerkunst mit dem Medium Computer arbeiten, wenn auch nicht im künstlerisch schaffenden Sinn.

Manfred Mohr, geb. 1938 in Pforzheim (D), lebt und arbeitet in Paris, New York

Harold Cohen, geb. 1928 (UK)

Joan Truckenbrod, geb. 1945 in Greensboro (USA)

Yoichiro Kawaguchi, geb. 1952 auf Tanegashima Island (J), lebt und arbeitet in Tokio

Mark Wilson, geb. 1943 (USA)

Laurence Gartel, geb. 1956 (USA)

Sue Gollifer, geb. 1944 (UK)

Jean-Pierre Hébert, geb. 1939 (USA), lebt und arbeitet in Santa Barbara (USA)

Roman Verostko, geb. 1929 in Pennsylvania (USA)

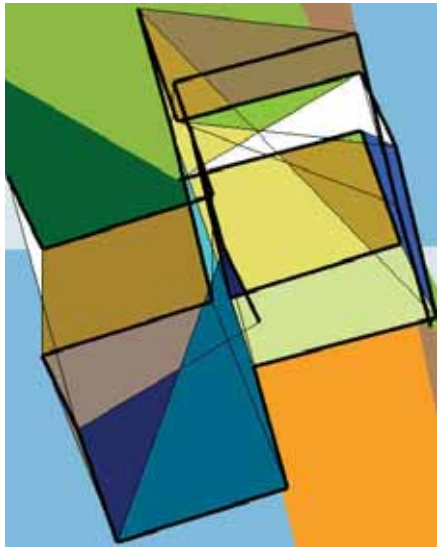
David Em, geb. 1953 (USA)

Yoshiyuki Abe, geb. 1947 in Gunma (J)

Paul Brown, geb. 1947 (UK), lebt und arbeitet in Australien

Rejane Spitz, geb. 1956 (BR)

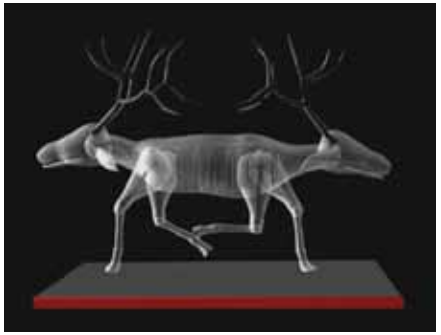
Tabelle 1.3



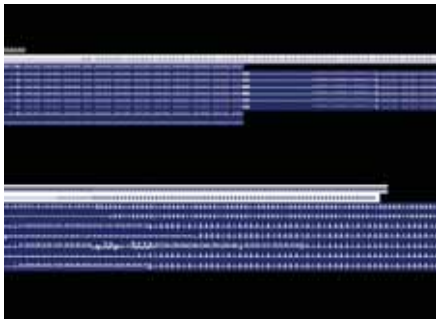
Manfred Mohr
P-701/B space color (2000)

Yoichiro Kawaguchi
Wriggon

James Faure Walker
Colour Study (1990)



Yves Netzhammer
(Animation 1999)



Jodi
ASDFG (2000)

Computerkünstler

James Faure Walker, geb. 1948 (UK)

Mike King, geb. 1953 in London (UK), lebt und arbeitet in London

Gerhard Mantz, geb. 1950 in New-Ulm (D), lebt und arbeitet in Berlin [4]

Yves Netzhammer, geb. 1970 in Affoltern (CH), lebt und arbeitet in Zürich [4]

Olga Tobreluts, geb. 1970 in Leningrad (RUS)

Kerry John Andrews, geb. 1956 (UK)

Tabelle 1.4

Netzkünstler

Jodi geb. 1968 (NL), 1965 (BEL), leben und arbeiten in Barcelona und Amsterdam [5]

Wolfgang Staehle geb. 1950 in Stuttgart (D), lebt und arbeitet in New York [6]

Mark Napier, geb. 1961 in New Jersey (USA), lebt und arbeitet in New York

Nam June Paik, geb. 1932 in Seoul (KOR), lebt und arbeitet in New York

Alexei Shulgin, geb. 1963 in Moskau (RUS), lebt und arbeitet in Moskau

Robert Adrian X, geb. 1935 in Toronto (CAN), lebt und arbeitet in Wien

etoy, anonym, leben und arbeiten in der Schweiz

0100101110101101.org, anonym, leben und arbeiten in Österreich [7]

Tabelle 1.5

Künstlerisch ambitionierte Grafiker und Techniker

John Maeda, geb. 1966 in Seattle (USA), lebt und arbeitet in Massachusetts (USA)

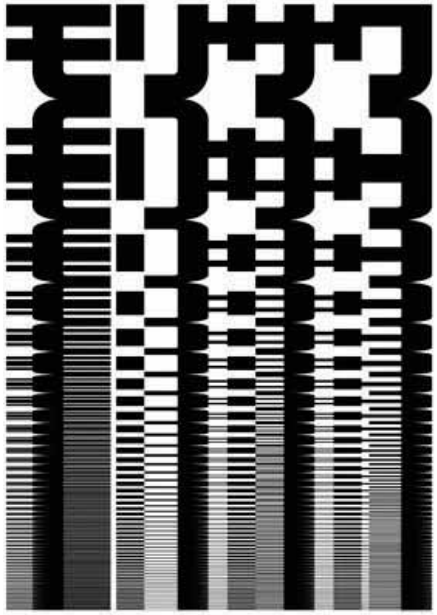
Casey Reas (USA), lebt und arbeitet in USA [11]

Golan Levin, geb. 1972 (USA)

Marius Watz, geb. 1973 in Oslo (NOR) [12]

Tabelle 1.6

Ich hoffe hiermit alle offenen Fragen und Missverständnisse beseitigt, den Sinn der folgenden Arbeit ausreichend verständlich gemacht und einen kleinen Einblick in die Computerkunst vermittelt zu haben und wünsche viel Spaß und Interesse beim Lesen meiner Diplomarbeit.



John Maeda
The 10 Morisawa Posters (1996)

Einleitung

"Kunst und Technik sind zwei grundlegende Erscheinungsformen unserer Kultur. Ursprünglich gab es keine klare Unterscheidung - der Ausdruck 'techne' schließt sowohl Technik wie auch Kunst in sich ein. Im Laufe der Entwicklung rückten beide mehr und mehr auseinander - Kunst und Technik haben andere Methoden, Denkweisen und Ausdrucksformen entwickelt, ebenso weichen ihre Zielsetzungen voneinander ab. In den letzten Jahrzehnten hat sich dadurch eine Kluft des Missverstehens und der Ablehnung zwischen beiden Bereichen gebildet.

Der Techniker steht den meisten Resultaten der modernen Kunst verständnislos gegenüber. Sie entsprechen weder seinen Erwartungen noch seinen Wertansprüchen. Mit den Erklärungen, die man ihm anbietet, weiß er nichts anzufangen; verglichen mit der Logik und Prägnanz, die er von Aussagen aus seinem Fachbereich fordert, erscheinen ihm die meisten Deutungsversuche als inhaltsloses und pseudophilosophisches Geschwätz. Ähnlich vernichtend ist sein Urteil über die gestellten Ansprüche, wie Veränderung der Welt durch Kunst usw., die ihm irrealistisch und größenwahnsinnig vorkommen.

Der Künstler ist mit den Ergebnissen der Technik ebensowenig zufrieden. Ein großer Teil der Schwierigkeiten, mit denen wir es in unserer Welt zu tun haben, geht direkt oder indirekt auf den technischen Fortschritt zurück. Die Technisierung führt zur Zerstörung der natürlichen Umwelt. Mit dem neuen Lebensstil in einer synthetischen Umgebung kommt der Trend zur Reglementierung, zur Kontrolle, zur Uniformierung - kurz zur Beeinträchtigung der Freiheit. Viele Künstler sehen ihren Aufgabenbereich als einen Freiraum an, den es gegen technische Ansprüche zu verteidigen gilt."

Herbert W. Franke, 1978.

Dabei sind die Unterschiede zwischen Technik und Kunst gar nicht so groß wie man meinen sollte! Getrennt voneinander befragt, werden wohl beide klare Antworten bezüglich der Welt haben. Oben ist oben, unten ist unten. Anerkennende Zustimmung von beiden Seiten. Doch stecke man beide einmal in ein gemeinsames Bild?

Sind diese Ansichten verträglich? Wo sie sich doch so zu widersprechen scheinen? Der Technik wird es nicht schwer fallen der Kunst recht zu geben, ist ihr doch die Gravitation und ihre Auswirkung auf die Wahrnehmung bewusst. Und die Kunst wird die Ansichten der Technik höchstens milde belächeln, besitzt sie doch die Freiheit die Wahrheit auch dort und gerade dort zu finden, wo selbst die Gravitation ihre Gültigkeit verliert.

Wenn sich Technik und Kunst also treffen wollen, hat es keinen Sinn nach einem gemeinsamen Kontext zu suchen, diesen gibt es schlicht und einfach nicht. Beide sehen sich, beide können einander akzeptieren, möglicherweise brauchen sie sich sogar. Aber jede Aussage die über das Bündnis getroffen wird, kann nur von einem der beiden Partner als richtig angesehen werden. Dort wo das Nein der Technik anfängt, beginnt für die Kunst das Ja. Wo die Technik zustimmend nickt, gähnt sich die Kunst vor Langeweile zu Tode.

Tatsache ist, dass die Kunst zumindest eine banale Technik braucht, um sich ausdrücken zu können. Die Technik andererseits ist - rein gravitativ betrachtet - auf die Kunst nicht angewiesen. So groß ist ihr Nutzen, dass sie sich ihre Partner gelassen aussuchen kann. Medizin, Industrie, Militär und auch Unterhaltung sind äußerst willige und profitable - beinahe sogar abhängige - Mitstreiter. Wieso also eine Bindung mit der undankbaren Kunst eingehen?

Weil vielleicht die Kunst in der Lage ist, der Technik einen angemessenen Platz in der Gesellschaft zuzuschreiben, der nicht nur zweckgebunden ist. Auch wenn die Technik noch so sehr gebraucht wird, so steht sie doch nie im Rampenlicht und wird, sobald sie ihren Dienst geleistet hat, nicht mehr weiter beachtet. Schlimmer noch, sie wird höchstens gebilligt, kaum geschätzt und überhaupt nicht geliebt. Kann sie ihre Aufgabe einmal nicht erfüllen, wird sie verflucht und verachtet. Ist sie stolz auf ihre Leistungen, erntet sie spottenden Hohn. Wie wünschenswert wäre daher ein Partner, der der Technik eine Stellung einräumen kann, der sie zu einem fest integrierten Bestandteil der Gesellschaft mit ideologischem Hintergrund und Ansehen macht? Das ist Geschmacksache! Die Technik würde dazu sagen: "*Ich brauche diese Stellung doch nicht!*", die Kunst: "*Die braucht doch diese Stellung nicht!*" Und sie würden mit Sicherheit nicht dasselbe damit meinen.

Nun soll nicht weiter diskutiert werden ob und wie die Kunst und die Technik zueinander finden können oder sollen, das Bündnis existiert bereits, die Kunst hat den Computer gefunden und die Beziehung scheint immer besser zu funktionieren. Was geblieben ist, sind die unterschiedlichen Ansichten über ein und denselben Sachverhalt. Diese scheinbar verkehrten Welten sollen an folgendem Beispiel kurz demonstriert werden.

Fehlerhafte Überläufe

"*In Java ist es möglich direkt in den Bildschirmspeicher zu schreiben, was aber zu störenden Fehlern beim Bildaufbau führen kann, da das Zeichnen nicht synchron zur Bildwiederholfrequenz des Monitors passiert. Das führt zu dargestellten Bildern, die nicht komplett fertig gezeichnet wurden. Wahrnehmbar werden solche Fehler durch ein mehr oder weniger starkes Flimmern bzw. Streifen die durch das Bild wandern. Diese Effekte kann man umgehen, indem das Bild zuerst im Speicher aufbereitet wird und erst dann in den Bildspeicher geschrieben wird.*"

Diesmal hat zuerst die Kunst das Wort und erklärt, welche Informationen das Geschriebene enthält: "*Der Blick fällt auf das Wort* 'möglich'*, Möglichkeiten sind da um genutzt zu werden. Was ist also möglich? Es können* 'störende Fehler' *erzeugt werden, die sich als* 'Flimmern' *bemerkbar machen. Das ist sicher brauchbar, sehr interessant. Aber* 'kann man umgehen' *klingt weniger erfreulich. Heißt das doch, dass die Möglichkeit in Wahrheit der Normalzustand zu sein scheint und der Normalzustand viel aufwendiger herzustellen ist. Das klingt nach Einarbeitung in komplexe technische Zusammenhänge, der Text zwischen* 'Fehler beim Bildaufbau' *und* 'starkes Flimmern' *scheint also doch nicht so unwichtig zu sein und bedarf genauerer Untersuchung! Worte wie* 'Bildschirmspeicher' *und* 'Bildwiederholfrequenz' *warten darauf mit Sinn gefüllt zu werden! Womit haben es sich diese Worte verdient so genau betrachtet zu werden? Was gibt diesen Worten das Recht mir die Zeit zu stehlen? Wofür braucht man diese Worte? Der Sinn braucht diese Worte nicht!*"

Das Wort möglich ist auch für die Technik von Bedeutung: "*Möglichkeiten sind da um zu wissen was nicht möglich ist. Ist etwas möglich, ist die technische Grenze an dieser Stelle noch nicht erreicht, was gut ist. Es können* 'störende Fehler' *auftreten, was schlecht ist und natürlich verhindert werden muss! Ist es überhaupt möglich, diese Fehler zu verhindern? Ja, zum Glück, in Java ist Doppelpufferung möglich! Gut zu wissen! Was steht da sonst noch? Erklärung wie es zu den Fehlern kommt. Das ist aber eigentlich sowieso klar,* 'Bildschirmspeicher'*,* 'Bildwiederholfrequenz'*. Der Sinn braucht diese Worte nicht!*"

Die Herangehensweise an die Technik aus der Sicht der Kunst kann theoretisch nur bedingt erklärt werden, weswegen dieser nicht weiter Beachtung geschenkt werden soll. Anders sieht es bei der Annäherung der Technik an die Kunst aus. Löst man die traditionelle Vorstellung von der Technik - einem Zweck zu dienen - auf, eröffnen sich ungeahnte neue Türen. Fehler des Zwecks werden zu Schätzen der Kunst. Gerade dort, wo der Plan versagt, ergeben sich neuartige Strukturen zwischen Zufall, Chaos und Regulativ. Die Aufgabe der Kunst ist es, in diesen Bereichen eine Ordnung zu finden, diesen einen Sinn zu geben. Die Technik liefert das Material dazu. Tut sie dies zweckgebunden, liegt ein Teil der Lebendigkeit des betreffenden Mediums brach. Versinkt sie im reinen Chaos, muss selbst die Kunst von ihr weichen.

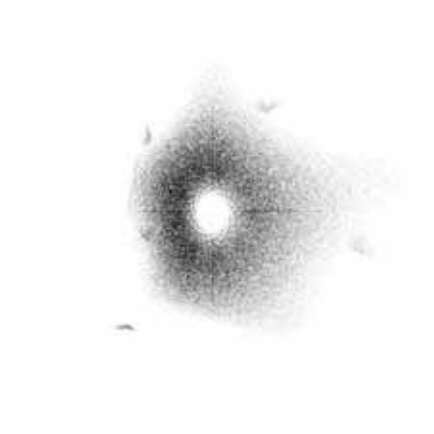
Es gibt kein anderes technisches Gerät, das sich innerhalb dieser Grenzen - Ordnung und Chaos - so flexibel und dynamisch verhalten kann und steuern lässt, wie der Computer. Zwar verhält sich die Hardware selbst in keiner Weise unkontrolliert, dafür aber die Ergebnisse, die diese Hardware mit Hilfe von Software erzeugt. Und gerade diese Tatsache, dass die Maschine Computer kein autonom funktionierender Automat ist, sondern ihr Verhalten von Bits

und Bytes gesteuert wird, die sich ebenfalls auf dieser Maschine befinden und somit Teil von dieser sind, machen sie zu einem der interessantesten Werkzeuge für schaffende Menschen, da die Grenzen des Möglichen nicht absehbar, die Funktionsweise aber voraussehbar und festlegbar ist.

So sehr diese Offenheit an Möglichkeiten auch wünschenswert ist, wird diese um den Preis der Komplexität erkauft. Hinter einer Linie, die am Monitor gezogen wird, steckt Mathematik, Logik und viel Code. Je ausgefallener diese Linie ist, umso aufwendiger ist auch die dahinterstehende Technik. Und diese Technik richtet sich nicht nach den Bedürfnissen der Kunst. Das Gegenteil ist viel mehr der Fall. So sehr sich ein Computer auch zum Experimentieren eignet, die Grundlage für einfachste Experimente bedarf perfekter (da sonst nicht funktionierender) Algorithmen und aufwendiger Einbindung von Ein- und Ausgabegeräten. Sind diese einmal geschaffen, können sie immer wieder in unterschiedlichsten Kombinationen verwendet werden, allerdings nur für die konzipierte Funktionalität. Sobald anderes Verhalten von einer Routine gewünscht wird, muss diese entweder abgeändert oder neu erstellt werden. Beides kommt gerade im experimentellen Umgang mit dem Medium Computer sehr häufig vor.

Eine Library für Computerkunst kann es schon theoretisch nicht geben, da eine Library im Prinzip immer eine Einschränkung aller Möglichkeiten auf die notwendigen darstellt und im Sinne der Kunst gerade diese Beschneidung des Möglichen die Sinnhaftigkeit einer solchen Library ad absurdum führt. Brauchbarer ist eine ständig erweiterbare Ansammlung von Methoden, welche immer wieder abgeändert werden können. Prinzipiell können nur grundlegende Methoden, wie z.B. das Setzen von Pixel im Bildspeicher, wirklich unverändert eingesetzt werden und sogar diese nicht immer. Sobald komplexere Methoden implementiert werden, gilt es Entscheidungen zu treffen, die jeder Künstler für sich alleine fällen wollen wird und auch soll. Aus diesem Grund ist es auch in der Praxis nicht möglich eine alle Computerkünstler befriedigende Library zu erstellen.

So kann sich die Informatik der Kunst hauptsächlich erklärend nähern, indem aufgezeigt wird, wie Algorithmen entworfen, Methoden erstellt und angepasst werden, welchen Stellenwert Mathematik dabei hat und wie die Daten der unterschiedlichen Medientypen aufgebaut sind und erzeugt werden können. Da bei der Generierung totales Chaos vermieden werden können soll, bedarf es auch der Erklärung, wie Daten erzeugt werden können, die nicht auf Zufall basieren aber dennoch komplex genug sind um totale Vorhersehbarkeit zu vermeiden. Diesen (und noch einigen anderen) Themen widmen sich daher die nun folgenden Kapitel.



Punkte

Grafik mit Java

Java eignet sich ideal als Programmiersprache für Künstler, da sie frei erhältlich ist, auf allen Computerplattformen und auch anderen Geräten wie z.B. Handys lauffähig ist und es möglich ist Java Applets in Webbrowsern laufen zu lassen, was besonders für die Netzkunst von Bedeutung ist. Allerdings sind mit Java auch einige Einschränkungen verbunden. Java wird von den gängigen Browsern nur bis zur Version 1.1 unterstützt, für spätere Versionen benötigt man ein Java-Plugin von dem man nicht ausgehen kann, dass dieses von sehr vielen Usern verwendet wird. Will man also Software in Java für Browser schreiben die auf möglichst vielen Rechnern funktioniert, muss man sich mit den Routinen bis zur Version 1.1, genauer 1.1.8, begnügen. Allerdings bietet diese Version bereits alles was nötig ist, um Grafik flexibel gestalten zu können. Leider gibt es immer wieder Versionen vom Internet Explorer und auch anderen Browseranbietern die Java nicht standardmäßig unterstützen.

Java ist eine Interpretersprache und daher entsprechend langsam sofern keine Tools wie Just-in-Time-Compiler eingesetzt werden. Dies gleicht die inzwischen sehr leistungsfähige Hardware der meisten Computersysteme aber teilweise wieder aus. Gerade in der Kunst, z.B. Musik und Interaktive Kunst, wünscht man sich sehr oft Echtzeitsysteme, welche spontan auf Ereignisse reagieren können. Solche Systeme können gar nicht schnell genug sein, weshalb es nötig ist die Software zu optimieren, d.h. komplexe Probleme mit so wenig Befehlen wie nur nötig zu lösen. Dies ist besonders bei einer Sprache wie Java notwendig, die generell schon langsamer ist als z.B. C++.

Optimierung

Dieses Kapitel beschäftigt sich daher nicht nur mit der Darstellung von Bildpunkten, sondern auch mit den Möglichkeiten die Java bis zur Version 1.1 bietet, um effizienten Code zu ermöglichen. Prinzipiell müssen im Code die Stellen optimiert werden, die am häufigsten ausgeführt werden. Dafür kann man auch in Kauf nehmen, dass z.B. andere Stellen etwas mehr Befehle beanspruchen, wenn dafür die gesamte Abarbeitung beschleunigt wird.

Folgende Punkte tragen zu einer Verbesserung des Laufzeitverhaltens bei:
1. Variablen vom Typ int anstelle von double
2. Shift und logische Befehle statt mathematischer Operationen
3. Berechnete Werte zwischenspeichern
4. Mathematische Umformungen
5. Komplexität wo möglich vereinfachen
6. Nur wirklich mögliche Situationen berücksichtigen
7. Methodenaufrufe reduzieren

Benötigt man eine sehr exakte Darstellung von Fließkommazahlen, kann man auf den Einsatz von Variablen und Operationen des Typs double nicht verzichten. Variablen vom Typ double benötigen aber mehr Speicher als jene vom Typ int. Dies ist wichtig für die Zwischenspeicherung großer Datensätze. Außerdem sind mathematische Operationen auf den Typ double komplexer als solche auf den Typ int. Moderne Prozessoren können zwar double Operationen teilweise schon schneller ausführen als jene auf int, allerdings erreichen auch diese nicht die Effizienz eines einfachen Shift-Befehls. Daher ist es wünschenswert Fließkommazahlen auf Ganzzahlen abzubilden, was auch mit gewissen Einschränkungen möglich ist.

Im Prinzip basiert die Simulation von Fließkommazahlen auf der Tatsache, dass Fließkommazahlen auch als Brüche darstellbar sind. Zwar sind nicht alle reellen Zahlen auch Brüche, doch lassen sich bis zu einer gewissen Genauigkeit alle reellen Zahlen annähern. Benötigt man z.B. nur drei Nachkommastellen, so kann die gegebene Zahl mit Tausend multipliziert werden und die restlichen Nachkommastellen werden ignoriert. So erhält man eine ganze Zahl die unter

dem Wissen, dass sie vertausendfacht worden ist, eine Fließkommazahl repräsentiert. Somit kann man diese Zahl mit anderen Zahlen, welche ebenfalls zuvor mit 1000 multipliziert wurden, addieren und subtrahieren und erhält am Ende ein korrektes Ergebnis.

Problematischer sind Multiplikationen und Divisionen. Bei der Multiplikation wird auch der Faktor 1000 mitmultipliziert. Das Ergebnis muss daher wieder durch 1000 dividiert werden. Bei der Division ist es umgekehrt, dabei gehen die virtuellen Nachkommastellen verloren. Für ein korrektes Ergebnis muss der Zähler vor der Division mit 1000 multipliziert werden. Diese Einschränkungen scheinen einen Mehraufwand darzustellen, allerdings nur solange man sich im Dezimalsystem befindet. Im Binärsystem sind die Nachkommastellen nichts anderes als Nachkommabits. Verwendet man daher statt dem Faktor 1000 eine Zweierpotenz wie 1024, was zehn Bits hinter dem Komma entspricht, lassen sich die zusätzlichen Multiplikationen und Divisionen durch einfache Shifts realisieren.

In Java gibt es drei Shift-Befehle, die alle von Bedeutung sind. Shifts sind Verschiebungen der Bits um eine gewisse Anzahl von Stellen entweder nach links oder nach rechts. Ein Shift nach links (<<) entspricht mathematisch einer Multiplikation mit zwei. Ein Shift nach rechts (>>) einer Division durch 2. Beim Linksshift wird das frei werdende Bit auf Null gesetzt. Beim Rechtsshift bleibt das Most-Signifikant-Bit, also das vorderste Bit, erhalten, da dieses für die Bestimmung des Vorzeichens herangezogen wird. Will man dies verhindern und soll stattdessen ebenfalls das frei werdende Bit auf Null gesetzt werden, benötigt man das nicht Vorzeichen behaftete Rechtsshift (>>>).

Weitere wichtige Bit-Operationen sind das UND (`&`) und das ODER (`|`). Ein Pixel wird z.B. immer durch einen int-Wert repräsentiert, wobei die ersten acht Bit für den Alpha-Wert bzw. die Transparenz, die zweiten acht Bit für den Rot-Wert, die nächsten acht für den Grün-Wert und die letzten acht für den Blau-Wert reserviert sind. Benötigt man den Rot-Wert müssen alle anderen Werte aus dem Farbwert gelöscht werden. Dies macht man mit einer UND-Operation des Wertes `0x00ff0000` auf den ursprünglichen int-Wert. Tabelle 3.1 zeigt alle Bit-Operationen die mit Java möglich sind. [13]

&	bitweise und Verknüpfung	`0&0 = 0`	`0&1 = 0`	`1&0 = 0`	`1&1 = 1`
|	bitweise oder Verknüpfung	`0|0 = 0`	`0|1 = 1`	`1|0 = 1`	`1|1 = 1`
^	bitweise exklusive oder Verknüpfung	`0^0 = 1`	`0^1 = 0`	`1^0 = 0`	`1^1 = 1`
~	bitweises Komplement (1-Komplement)			`~0 = 1`	`~1 = 0`
<<	bitweise links schieben				
		`1<< 1 = 2`	`1<< 2 = 4`	`1<< 3 = 8`	`3<< 1 = 6`
>>	bitweise rechts schieben				
	links das Vorzeichen nachziehen	`1>> 1 = 0`	`2>> 1 = 1`	`11>> 2 = 2`	`-2>> 1 = -1`
>>>	bitweise rechts schieben				
	links 0 nachziehen	`1>>>1 = 0`	`2>>>1 = 1`	`11>>>2 = 2`	`-2>>>1 = 126` (bei 8 Bit)

Tabelle 3.1

Werte, die bereits berechnet wurden, sollten wenn möglich nicht noch einmal berechnet werden. Kommen in Gleichungen Terme mehrmals vor, so kann ein solcher Unterterm zuerst berechnet und in einer Variablen zwischengespeichert werden und dann in der Gleichung verwendet werden. Komplizierter wird es, wenn sehr viele berechnete Werte immer wieder benutzt werden müssen. Da ein Prozessor nur eine begrenzte Anzahl von Registern zur Speicherung von Ergebnissen hat, müssen solche Werte im Speicher abgelegt werden. Das Lesen und vor allem das Schreiben von Daten aus und in den Speicher benötigt aber um ein vielfaches mehr an Zeit als das Lesen und Schreiben aus und in die Register. Ist die Berechnung der Werte allerdings mit einem Aufwand verbunden, der den Aufwand für das Lesen der Werte aus dem Speicher übersteigt, so bringt diese Vorgehensweise dennoch Vorteile. Einer der wichtigsten Algorithmen in der Kunstinformatik, die FFT, kann durch dieses Verfahren die diskrete Fouriertransformation so schnell ausführen, dass diese für eine Vielzahl von Anwendungen nutzbar geworden ist.

Noch vor ca. 15 Jahren waren Prozessoren in ihren Berechnungen so eingeschränkt, dass es schon bei einfachen Berechnungen wie der Sinus-Funktion einen Geschwindigkeitsvorteil brachte, wenn die Ergebnisse nicht neu berechnet wurden, sondern aus dem Speicher gelesen wurden. Heutige Prozessoren sind leistungsfähiger und werden immer leistungsfähiger. Es empfiehlt sich also bei der Entscheidung, ob Daten zwischengespeichert werden sollen oder neu berechnet werden sollen, beide Verfahren auszuprobieren. Ist die Zwischenspeicherung schnell genug, sollte man auf diese Zurückgreifen, da somit sichergestellt ist, dass auch langsamere Computer die Aufgabe schnell genug lösen können. Soll die Applikation nur auf einem bestimmten Rechner laufen, kann man sich für die schnellere Lösung entscheiden.

Beim Programmieren entstehen oft Situationen in denen Werte für einen laufenden Index berechnet werden müssen. Dabei kann es vorkommen, dass man unter Verwendung des Ergebnisses für den vorherigen Index schneller zum Ergebnis für den gegenwärtigen Index kommt, als wenn dieses für den aktuellen Index erneut Berechnet wird. Die Position eines Pixels in einem Array errechnet sich z.B. durch das Produkt des Y-Wertes mit der Breite des Bildes und der anschließenden Addition mit dem X-Wert. Dafür benötigt man also eine Multiplikation und eine Addition (O(2)). Werden alle Pixel hintereinander abgearbeitet muss dieser Index nicht jedes Mal neu berechnet werden, es reicht in diesem Fall den letzten Index zu inkrementieren (Addition mit 1). Auch das Quadrat eines laufenden Index lässt sich anstelle von einer Multiplikation mit zwei Additionen errechnen (siehe Tabelle 3.2), was nicht unbedingt einen Geschwindigkeitsvorteil bringen muss, aber aufzeigt mit welchen Mitteln sich unter Umständen Laufzeitverbesserungen erreichen lassen.

i	i²	quad	di	q+= di	di+= 2
1	1	0	1	1	3
2	4	1	3	4	5
3	9	4	5	9	7
4	16	9	7	16	9
5	25	16	9	25	11

Tabelle 3.2

Ganz allgemein lassen sich viele Berechnungen mathematisch so umformen, dass weniger Operationen notwendig sind. Als einfaches Beispiel sei hier folgende Formel genannt: `(x-1)*(x+1)` benötigt 1 Addition, 1 Subtraktion und 1 Multiplikation. Rechnet man sich das Ergebnis zuvor aus `(x*x)-1`, kann man die Addition einsparen. Hier sei wieder auf die Fast-Fourier-Transformation verwiesen, welche mit dieser Methode eine wesentliche Zeiteinsparung erreicht. Das berechnen der Wurzel einer Zahl ist besonders aufwendig und kann in manchen Fällen verhindert werden. Will man z.B. die Länge zweier Vektoren miteinander Vergleichen, benötigt den Wert der Länge aber danach nicht mehr, kann das Ziehen der Wurzel entfallen und nur die Summe der Quadrate miteinander verglichen werden:

```
if (x1² + y1² > x2² + y2²) ...
```

Die Lösung komplexer Probleme beinhaltet oft Details die nicht immer von essentieller Bedeutung sind. Zwar sollte ein Algorithmus seine Problemstellung immer so eindeutig und korrekt wie möglich lösen, allerdings zählt im Endeffekt das sichtbare bzw. hörbare Ergebnis. Liefert ein Algorithmus in kürzerer Zeit ein Ergebnis, das zwar nicht exakt, aber der Unterschied zum korrekten Algorithmus nicht wahrnehmbar ist, so ist die schnellere Lösung vorzuziehen.

Ein geschriebener Code sollte im Allgemeinen Fehlerresistent sein, d.h. er sollte in möglichst allen Situationen fehlerfreie Ergebnisse liefern. Einen Code, der jede Situation korrekt behandelt, kann man als sicher bezeichnen. Diese Sicherheit erkauft man sich jedoch oft um den Preis eines optimalen Laufzeitverhaltens. Sicherheitsabfragen sind allerdings nur dann sinnvoll, wenn das Auftreten der Bedingung überhaupt vorkommt. Weiß man, dass eine gewisse Situation nicht vorkommen wird, obwohl sie vorkommen könnte, so kann die entsprechende Abfrage eingespart werden. Der Nachteil dieses Vorgehens ist der, dass es eben doch zu der unerwünschten Situation kommt und so Fehler entstehen, die nur mehr schwer auffindbar sind. Eine Abfrage, die sich z.B. beim Setzen von Pixel einsparen lässt, ist die Kontrolle ob sich das Pixel überhaupt im Bild befindet. Tut es das nicht, so wird eine ArrayOutOfBounds-Exception geworfen, was fatale Folgen mit sich bringt, da das System den Prozess abbricht. Sorgt man andererseits dafür, dass ein Pixel niemals außerhalb der Bildschirmgrenzen gesetzt wird, so kann diese Abfrage eingespart werden.

Ein Vorteil von modernen Programmiersprachen ist die gute Überschaubarkeit und Lesbarkeit des Codes. Methoden halten den Sourcecode klein und übersichtlich, allerdings kostet ein Methodenaufruf auch Zeit. Je öfter eine Methode aufgerufen wird, desto mehr fällt dieser Overhead ins Gewicht. Da die Pixelmanipulationen bei der Arbeit mit Grafik die häufigsten Operationen sind und sich hier jede kleinste Optimierung positiv auf die gesamte Performance des Systems auswirkt, muss in manchen Fällen auf die gute Lesbarkeit verzichtet werden und der Code für das Setzen eines Pixels an jeder Stelle, an der dieser verwendet werden soll, auch tatsächlich geschrieben werden. Dieses Vorgehen bringt aber auch einen zusätzlichen Vorteil. Methoden sind oft starr und nicht für Ausnahmen geschaffen. Ist der Code dagegen an jeder Stelle ausgeschrieben, kann er auch an jeder Stelle entsprechend optimiert und angepasst werden.

MemoryImageSource

In Java bietet die Klasse Graphics eine Reihe von Funktionen zur schnellen Erstellung von Grafiken. Für viele Fälle sind diese Funktionen ausreichend, solange kein Zugriff auf die einzelnen Pixel nötig ist. Versucht man einzelne Pixel mit den vorhandenen Funktionen zu setzen, stößt man sehr rasch an die Leistungsgrenzen dieser Implementierung. Verlässt man andererseits diese Klasse, muss jede Grafikoperation eigenhändig geschrieben werden. Dazu benötigt man die Klasse *MemoryImageSource* die auch in der Version 1.1 inkludiert ist.

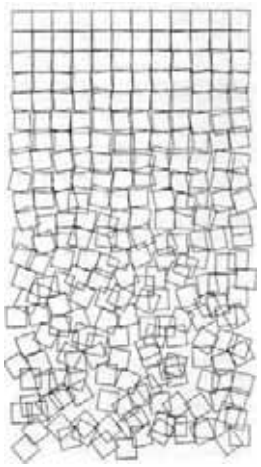
Bei der Initialisierung dieser Klasse wird der Instanz die Breite und Höhe des Bildes übergeben, außerdem das verwendete Farbmodell und ein Array mit den einzelnen Pixeln. Als Farbmodell eignen sich das ARGB- und das RGB-Modell, wobei dem RGB-Modell der Vorzug gegeben werden sollte, da die Alphaberechnungen des ARGB-Modells zeitaufwendig und nicht immer korrekt sind. Transparenz lässt sich auch mit dem RGB-Modell verwirklichen. Code 3.1 richtet ein Image auf der Basis einer MemoryImageSource ein.

```
int bgcolor;                              // backgroundcolor
int width, height;                        // width and height of image
int[] dat = new int[width * height];      // array of pixels

DirectColorModel  dcm = new DirectColorModel(32,0x00ff0000,0x0000ff00,0x000000ff,0);
MemoryImageSource mis = new MemoryImageSource(width, height, dcm, dat, 0, width);

mis.setAnimated(true);
mis.setFullBufferUpdates(true);
Image img = createImage(mis);
for(int i = 0; i < (width * height); i++) dat[i] = bgcolor;
mis.newPixels();                          // has to be called after pixel-changes
```

Code 3.1

Das ColorModel wird auf RGB gesetzt indem für den Alphawert 0 eingegeben wird. Die Anzahl der Bit ist dennoch 32, da diese Größe einem int entspricht und inzwischen auch der Standard für die Farbtiefe ist. Dabei ist allerdings darauf zu achten, dass Grafikkarten, die in einem anderen Bildmodus laufen (z.B. 256 Farben oder 16 bzw. 24 Bit), die erzeugten Bilder vor der Darstellung jedesmal konvertieren müssen, was sich negativ auf die Performance auswirkt. Auf die einzelnen Pixel kann man nun direkt durch das Array dat zugreifen, wobei jeder int-Eintrag einem Pixel im Format `0x00RRGGBB` entspricht. Für jede Primärfarbe stehen also 8 Bit zur Verfügung was eine Farbanzahl von ca. 16 Millionen entspricht. Die ersten 8 Bit werden nicht berücksichtigt und können im Code für verschiedenste andere Informationen genutzt werden, z.B. auch für Transparenz.

Die Methode *newPixels* muss spätestens dann aufgerufen werden, wenn das Bild tatsächlich verwendet werden soll, wobei die Ausführung diese Methode leider sehr viel Zeit in Anspruch nimmt, weshalb diese nur einmal pro Bildaufbau zum Einsatz kommen sollte. Erst nach diesem Aufruf wird das Bild aktualisiert dargestellt, da dabei dem System mitgeteilt wird, dass sich Pixel des Bildes verändert haben. Das Bild img kann nun mit dem *drawImage* Befehl der Klasse Graphics in den Bildschirmspeicher oder Doppelpuffer geschrieben werden. [24]



Georg Nees
Würfel-Unordnung (1968/71)

Pixel

Die einfachste und effizienteste Art ein Pixel zu setzen ist es, den Farbwert an dem entsprechenden Index zu setzen. Der Farbwert muss in diesem Fall nur einmal berechnet oder definiert werden und kann danach allen Pixel, die diese Farbe annehmen sollen, zugewiesen werden. Für diesen Vorgang benötigt man nur eine Schreiboperation, wenn man davon ausgeht, dass der Index des Pixels bereits berechnet wurde (Code 3.2).

```
int color = 0x00ff0080;                    // colorvalue: red: 255, green: 0, blue: 128
int id;                                    // index of pixel to be set

dat[id] = color;
```

Code 3.2

Soll der Alphakanal berücksichtigt werden, müssen die ersten 8 Bit des Farbwertes ausgelesen werden, um die Transparenz bestimmen zu können. Ist dieser Wert 255, so soll die Farbe, die das Pixel bereits hat, erhalten bleiben. Beträgt die Transparenz 0 so soll das Pixel seine Originalfarbe vollständig verlieren und stattdessen den Wert der Zeichenfarbe erhalten. Für Werte zwischen diesen Extremen soll entsprechend gemittelt werden. Die Berechnung der Farbe die im Endeffekt gesetzt wird hängt also vom Alphakanal, von der Farbe die das Pixel bereits hat und von der Zeichenfarbe ab. Da man davon ausgehen kann, dass der Alphakanal nur in den seltensten Fällen 0 oder 255 betragen wird (da in diesen Fällen kein Alphablending stattfindet), muss für das Setzten eines Pixels zuvor aus dem Array gelesen werden. Es ist also eine Lese- und eine Schreiboperation notwendig, was die Geschwindigkeit fast halbiert.

Zu dem Aufwand, das Pixel auslesen zu müssen, kommen noch eine mehr oder weniger große Anzahl von logischen und mathematischen Operationen, je nach Implementierung. Die korrekte und auf jeden Fall fehlerfreie Implementierung benötigt die meisten Operationen. Dabei wird der Alphawert ausgelesen und in a zwischengespeichert. Danach wird der Rest ermittelt, der entsteht, wenn man den Alphawert von 255 abzieht und in v zwischengespeichert. Nun können die Farbanteile berechnet und das Pixel auf das Ergebnis gesetzt werden (Code 3.3).

Durch die Multiplikation der Farbwerte mit a bzw. v rückt das Ergebnis um 8 Bit nach links, da nicht mit einem Faktor zwischen 0 und 1 sondern 0 und 255 multipliziert wird. Deshalb werden rr, gg und bb nach den Multiplikationen zurückgeschoben. Dieser Code benötigt neben dem Lesen und Schreiben zusätzlich noch 10 Shifts, 7 UND-Operationen, 5 Additionen, 1 Subtraktion und 6 Multiplikationen, insgesamt 29 Operationen.

```
public void alpha(int id, int c) {
     int o = dat[id];                    // current color of pixel
     int a = (c & 0xff000000) >>> 24;    // alpha calculation
     int v = 255 - a;                    // invers of alpha
     int ro = (o & 0x00ff0000) >> 16;    // old red
     int go = (o & 0x0000ff00) >>  8;    // old green
     int bo = (o & 0x000000ff);          // old blue
     int rc = (c & 0x00ff0000) >> 16;    // color red
     int gc = (c & 0x0000ff00) >>  8;    // color green
     int bc = (c & 0x000000ff);          // color blue
     int rr = (ro * a + rc * v) >> 8;    // new red
     int gg = (go * a + gc * v) >> 8;    // new green
     int bb = (bo * a + bc * v) >> 8;    // new blue
     dat[id] = (rr<<16) + (gg<<8) + bb;  // setting pixel
}
```

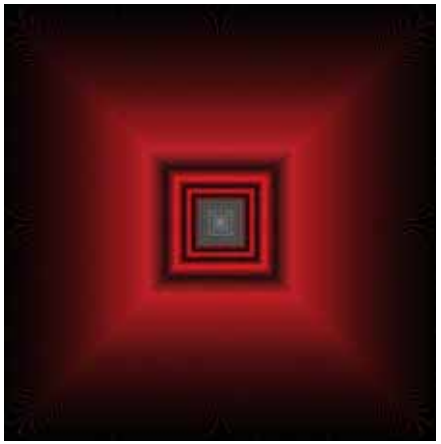
Code 3.3

Dieser Code kann aber noch, ohne Änderung des Ergebnisses, optimiert werden. Der Alphawert kann ohne die UND-Operation errechnet werden, da alle anderen Bits weggeschoben werden. Anstelle der Subtraktion kann eine XOR-Operation treten, die die letzten 8 Bit von a umkehrt. Das Shiften der Farbwerte kann überhaupt entfallen, da die Operationen auch für Bitverschobene Zahlen das korrekte Ergebnis liefern. Letztendlich können die drei Shifts die durch die Multiplikation notwendig werden auf eine einzige Shiftoperation reduziert werden und anstelle der drei Additionen ODER-Operationen eingesetzt werden, da sich die drei Farbwerte in keinem Bit überschneiden. Dafür müssen den berechneten Farbwerten mit UND-Befehlen die Nachkommabits abgeschnitten werden. Somit verbleiben 2 Shifts, 9 UND-Operationen, 3 Additionen, 6 Multiplikationen, 2 ODER-Befehle und ein XOR. Insgesamt sind das 23 zum Teil schnellere Operationen (Code 3.4).

Der Vorteil dieser korrekten Implementierung mit Transparenz liegt darin, dass kein Farbwert jemals kleiner als Null oder größer als 255 werden kann. Es kommt zu keinem Überlauf, da die Summe der berechneten Farbwerte nie 255 übersteigen kann. Verzichtet man auf diese Absicherung, lässt sich die Alphaberechnung kompakter programmieren. Dabei gibt es viele Möglichkeiten, die auf die jeweilige Anforderung abgestimmt werden müssen.

```
public void alpha(int id, int c) {
     int o = dat[id];                    // current color of pixel
     int a = c >>> 24;                   // alpha calculation
     int v = a ^ 0xff;                   // invers of alpha
     int ro = (o & 0x00ff0000) * a;      // old red
     int go = (o & 0x0000ff00) * a;      // old green
     int bo = (o & 0x000000ff) * a;      // old blue
     int rc = (c & 0x00ff0000) * v;      // color red
     int gc = (c & 0x0000ff00) * v;      // color green
     int bc = (c & 0x000000ff) * v;      // color blue
     int rr = (ro + rc) & 0xff000000;    // new red
     int gg = (go + gc) & 0x00ff0000;    // new green
     int bb = (bo + bc) & 0x0000ff00;    // new blue
     dat[id] = (rr|gg|bb) >>> 8;         // setting pixel
}
```

Code 3.4



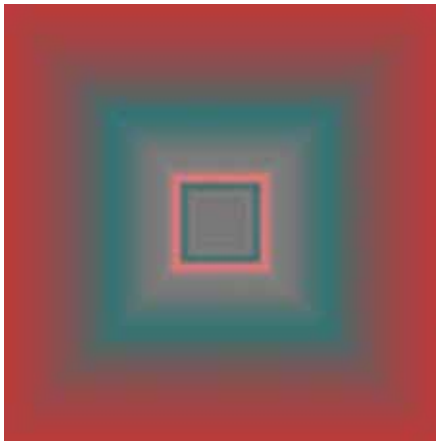


Abbildung 3.1 (I)

Hier wird zunächst eine sehr einfache und schnelle Variante vorgestellt, mit der sich ein Transparenzeffekt erzeugen lässt. Die alte und neue Farbe werden einfach miteinander addiert. Zuvor muss die neue Farbe aber entsprechend ihrer Transparenz definiert werden. Möchte man einen weißen Punkt mit Alphawert 128 setzen, so bekommt man ein nicht korrektes aber in gewissen Situationen brauchbares Ergebnis, wenn zum aktuellen Farbwert einfach der Wert `0x00808080` dazu addiert wird. In der Praxis wäre dieser Wert vermutlich zu groß, da bereits zwei solcher gesetzter Werte zu einem Überlauf führen würden. Verwendet man allerdings kleinere Werte, was einem geringeren Alphawert entspricht, kommt es erst bei sehr vielen Additionen zu Überläufen (Code 3.5).

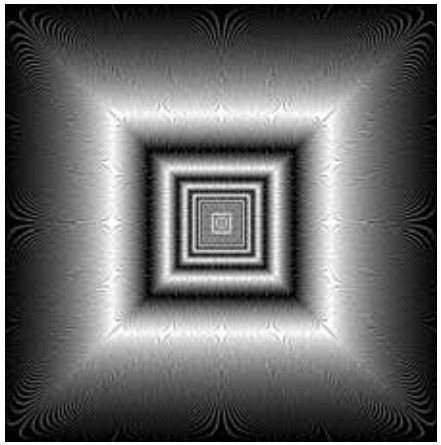
```
public void add(int id, int c) { dat[id]+= c; }
```

Code 3.5

Durch diese Implementierung entsteht zwar ein Transparenzeffekt, allerdings entspricht dieser nicht der echten Transparenz, weil die Werte nicht gemittelt werden und sich deshalb das Verhältnis zwischen den Farben anders verhält, als bei der korrekten Implementierung. Außerdem können Farbwerte nur größer werden, was zwangsläufig in Überläufen enden muss. Sollen Überläufe ausgeschlossen werden können, kann Code 3.6 herangezogen werden.

```
public void add(int id, int c) {
     int o = dat[id];
     int rr = ((o & 0x00ff0000) + (c & 0x00ff0000));
     int gg = ((o & 0x0000ff00) + (c & 0x0000ff00));
     int bb = ((o & 0x000000ff) + (c & 0x000000ff));
     if(rr > 0x00ff0000) rr = 0x00ff0000;
     if(gg > 0x0000ff00) gg = 0x0000ff00;
     if(bb > 0x000000ff) bb = 0x000000ff;
     dat[id] = (rr|gg|bb);
}
```

Code 3.6



Für das setzten eines Pixels mit der Methode add bedarf es wieder eines Lese- und eines Schreibbefehls und zusätzlich 6 UND-Verknüpfungen, 2 ODER-Verknüpfungen, 3 Additionen und 3 Abfragen auf größergleich. Da durch die Addition die Farbwerte nur steigen können benötigt man oft den Pendant - die Subtraktion - dazu. Auch diese Methode kann mit gleichem Aufwand (~O(14)) implementiert werden (Code 3.7).

```
public void sub(int id, int c) {
     int o = dat[id];
     int rr = ((o & 0x00ff0000) - (c & 0x00ff0000));
     int gg = ((o & 0x0000ff00) - (c & 0x0000ff00));
     int bb = ((o & 0x000000ff) - (c & 0x000000ff));
     if(rr < 0) rr = 0;
     if(gg < 0) gg = 0;
     if(bb < 0) bb = 0;
     dat[id] = (rr|gg|bb);
}
```

Code 3.7

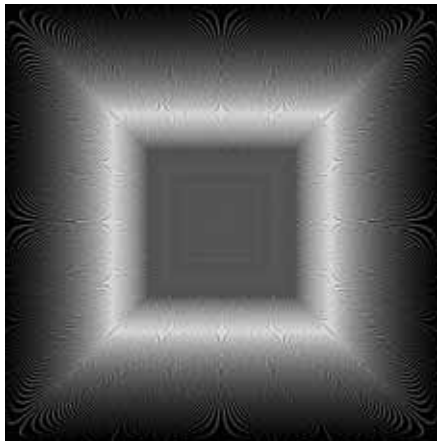
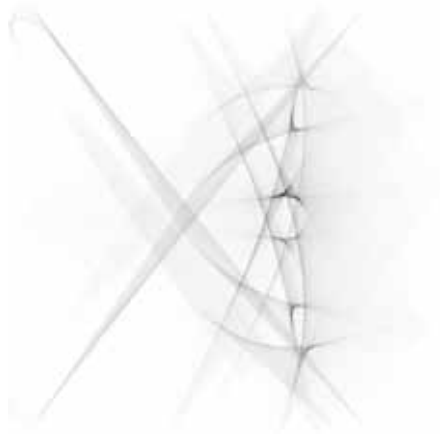


Abbildung 3.1 (II)

Da vor allem das Schreiben in den Speicher sehr viel Zeit in Anspruch nimmt, kann es sinnvoll sein, Pixelwerte nur dann zu schreiben, wenn sich der entsprechende Wert überhaupt verändert hat. Die zusätzliche Abfrage steht in keiner Relation zur benötigten Zeit für das tatsächliche Setzen des Pixels. Vor allem bei Grafiken mit wenigen Farben (z.B. Monochromgrafik) bringt diese Vorgangsweise einen gewaltigen Geschwindigkeitsvorteil. Ist aber damit zu rechnen, dass sich in den allermeisten Fällen der Wert ändern wird, ist diese Abfrage überflüssig.

Zuletzt soll erwähnt werden, dass auch mit den Binäroperationen Pixel gesetzt werden können, die oft erstaunliche Effekte hervorbringen können. Auch Multiplikationen und andere mathematische Operationen werden für Pixelmanipulationen herangezogen. So störend Überläufe im Allgemeinen sind, bewusst eingesetzt können auch diese Fehler in der Kunst ihre Verwendung finden und einen Teil des Reizes bzw. der Ästhetik ausmachen. Abbildung 3.1 zeigt einige Beispiele dazu und zum Einsatz von Binäroperationen. Dabei wurden jeweils vom Zentrum des Bildes aus Linien an alle Randpunkte des Bildes gezeichnet, wobei sich die Linien in der Nähe des Zentrums mehr und in der Nähe des Bildrandes weniger stark überschneiden.

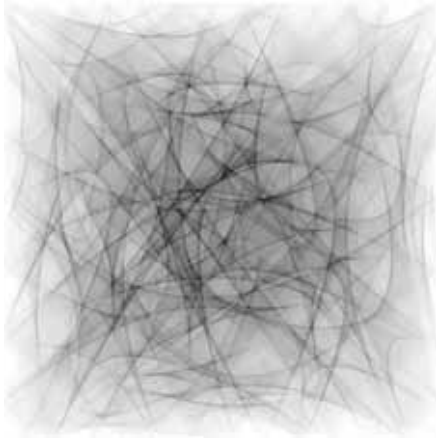
Linien



Bresenham Algorithmus

Ein wesentlicher Bestandteil bei der Erstellung von Grafiken ist das Zeichnen von Linien. Jede Programmiersprache bietet dafür Funktionen, allerdings nicht immer in ausreichender Qualität die Geschwindigkeit oder Ästhetik betreffend. Meist ist ein einfacher Algorithmus wie der Bresenham-Algorithmus ohne Antialiasing implementiert, der starke Treppeneffekte erzeugt.

Der Vorteil solcher Algorithmen liegt in der Effizienz und kann - unter gewissen Voraussetzungen - auch in der Kunst Verwendung finden:



1. Pixel-Ästhetik
2. Erzeugung von Interferenzen
3. Linien mit schwachem Kontrast
4. Waagrechte und senkrechte Linien

Will man eine Ästhetik produzieren die der Schwarzweiß- oder Pixelästhetik entspricht oder ähnelt oder diese einfach nur rezitieren, dann sollten Linien ohne Antialiasing verwendet werden. Durch die stufenweise Anordnung der einzelnen Pixel einer Linie können sich bei geschickter Anordnung der Linien Interferenzen bzw. Muster ergeben, die einen starken ästhetischen Reiz ausmachen, der allerdings ebenfalls eher der Vergangenheit zuzuordnen ist. Abbildung 4.1 zeigt eine einfache Grafik aus Linien, die solche Muster erzeugen, wobei einmal der Bresenham-Algorithmus und einmal ein Algorithmus mit Antialiasing zum Einsatz kommen. Die Interferenzen entstehen auch mit Antialiasing, da diese nur aufgrund logischer Gesetze entstehen.

Sind die Linien bezogen auf ihren Hintergrund relativ kontrastarm, kommen die Treppeneffekte weniger stark zur Geltung und können bedenkenlos eingesetzt werden. Abbildung 4.2 zeigt einige Standbilder aus einer Computeranimation, welche auf folgender Weise automatisch generiert wird (Code 4.1 *Linienmuster*, siehe *Anhang*):

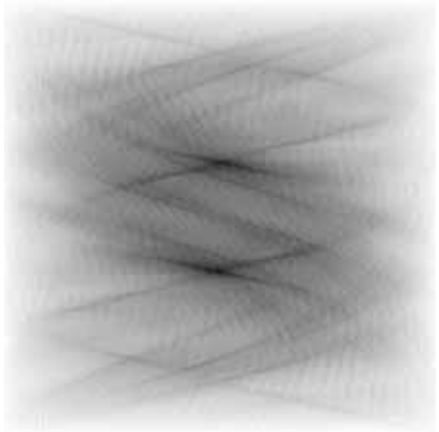


Abbildung 4.2

Zwei unsichtbare Punkte bewegen sich jeweils verschieden schnell in je eine Richtung und werden an den Bildschirmgrenzen entsprechend reflektiert. Während sie sich bewegen werden sie von fast durchsichtigen Linien verbunden, welche am Bildschirm dargestellt werden. Dabei wird im Bildspeicher für jeden Punkt der Linie der Wert leicht verändert. Eine einzelne Linie ist so nicht sichtbar. Ist ein Pixel das bereits verändert wurde erneut Punkt einer Linie, verändert sich der Wert für diesen Bildpunkt im Speicher entsprechend mehr.

Je öfter ein Punkt Teil einer Linie ist umso stärker wird er sichtbar. Daraus ergeben sich solche Grafiken. Da nach einer bestimmten Zeit alle Bildpunkte sehr oft erreicht werden, muss man - um eine komplette Färbung des Bildes zu verhindern - dafür sorgen, dass die alten Linien wieder gelöscht werden. Das erreicht man indem nach einer festgelegten Anzahl von gemalten Linien ein zweites Punktpaar, welches eine Kopie der ersten beiden darstellt, in das Bild geschickt wird und Linien in der inversen Farbe zeichnet. So werden die entsprechenden Bildpunkte wieder zurück verändert, was einem Löschen entspricht.

Bei waagrechten und senkrechten Linien können keine Treppeneffekte entstehen. Allerdings ist bei diesen Linien der Einsatz der implementierten Routinen zu hinterfragen, da solche Linien eigentlich extrem effizient ohne spezielle Algorithmen erzeugt werden können. Waagrechte Linien haben keine Steigung und können erstellt werden indem man eine bestimmte Anzahl von Pixel *hinter* einem Startpixel auf die Zeichenfarbe setzt. Senkrechte Linien haben eine

unendlich große Steigung und können erzeugt werden indem eine bestimmte Anzahl von Pixel *unter* einem Startpixel entsprechend der Zeichenfarbe gesetzt werden.

Implementierung

Java bietet zwar die Möglichkeit Linien zu zeichnen, allerdings nicht für Instanzen der Klasse Image die mit Hilfe einer MemoryImageSource kreiert wurden. Folgende Routine (Code 4.2 *Bresenham)* berücksichtigt die einfachen Fälle der horizontalen und vertikalen Linien und verzichtet aufgrund des höheren Durchsatzes auf eine gute Lesbarkeit. Durch Vertauschen der beiden Endpunkte der Linie wäre es Möglich die Anzahl der verschiedenen Fälle zu halbieren, was zu einer deutlich besseren Lesbarkeit des Codes führen würde, allerdings um den Preis der Operationen die für das Vertauchen von vier Werten notwendig sind. Aus diesem Grund sind in dieser Implementierung alle acht (bzw. zwölf mit den Waagrechten und Senkrechten bzw. dreizehn mit dem Punkt) Fälle separat ausformuliert (Tabelle 4.1).

Punkt				`x1 = x2`	`y1 = y2`
Senkrechte fallend	270°			`x1 = x2`	`y1 < y2`
Senkrechte steigend	90°			`x1 = x2`	`y1 > y2`
Waagrechte nach rechts	0°			`x1 < x2`	`y1 = y2`
Waagrechte nach links	180°			`x1 > x2`	`y1 = y2`
Schräge	315°	- 360°	`|dx| > |dy|`	`dx > 0`	`dy > 0`
Schräge	0°	- 45°	`|dx| > |dy|`	`dx > 0`	`dy < 0`
Schräge	180°	- 225°	`|dx| > |dy|`	`dx < 0`	`dy > 0`
Schräge	135°	- 180°	`|dx| > |dy|`	`dx < 0`	`dy < 0`
Schräge	270°	- 315°	`|dx| < |dy|`	`dx > 0`	`dy > 0`
Schräge	45°	- 90°	`|dx| < |dy|`	`dx > 0`	`dy < 0`
Schräge	225°	- 270°	`|dx| < |dy|`	`dx < 0`	`dy > 0`
Schräge	90°	- 135°	`|dx| < |dy|`	`dx < 0`	`dy < 0`

Tabelle 4.1

Die einzelnen Fälle sind so angeordnet, dass ein optimales Laufzeitverhalten entsteht. Der Overhead für das Zeichnen eines Punktes beträgt zwei *if*-Abfragen auf Gleichheit. Im Falle einer Linie die zufällig die Länge Null hat ist dieser Overhead minimal. Allerdings sollte diese Routine nicht zum setzen eines einzelnen Pixels verwendet werden, da in diesem Fall der Overhead völlig unnötig ist und die Methoden aus dem Kapitel *Punkte* dafür verwendet werden können.

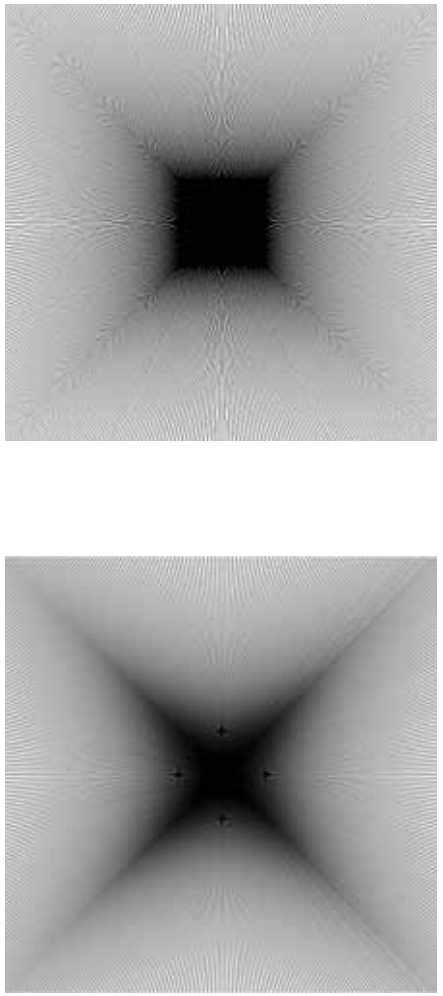
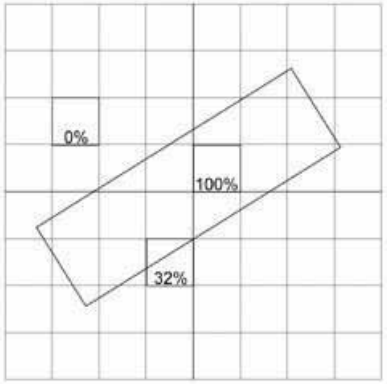


Abbildung 4.1

Antialiasing



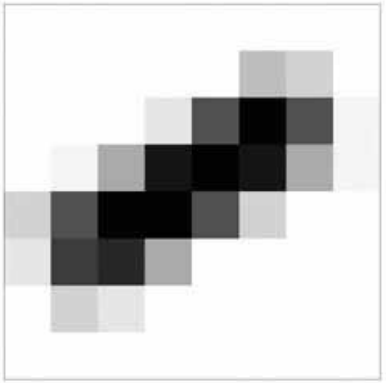


Abbildung 4.3

Der Treppeneffekt kann deutlich reduziert werden, wenn die Pixel nicht auf einen fixen Wert gesetzt werden, sondern entsprechend ihrem Anteil an der Linie. Geht man davon aus, dass jede Linie eine gewisse Breite hat, kann eine Linie als Rechteck betrachtet werden. Projiziert man dieses Rechteck auf die Pixel so gibt es Pixel die vom Rechteck vollständig überdeckt werden und andere die teilweise oder gar nicht überdeckt sind. Ist ein Pixel nicht überdeckt soll sich sein Wert nicht ändern. Ein Pixel das vollständig vom Rechteck überdeckt wird, soll seinen Wert maximal ver-ändern. Alle anderen Pixel ändern ihren Wert entsprechend ihrer Überschneidung mit dem Rechteck (siehe Abbildung 4.3).

Die folgenden Überlegungen beziehen sich auf eine Linie entsprechend Abbildung 4.4, welche links unten in Punkt S beginnt und nach rechts hin zu Punkt E ansteigt. Unter der Annahme, dass der Methode die zwei Punkte S und E und die Breite der Linie übergeben werden, ist es notwendig die Daten für das zu erstellende Rechteck aus diesen Werten zu extrahieren. Abbildung 4.4 verdeutlicht den Zusammenhang zwischen den Werten p, z und r bezogen auf die Endpunkte und die Breite der Linie. p verhält sich zu z wie dy zu dx wobei r - was der halben Breite entspricht - proportional zur Länge der Linie ist. Somit kann sowohl z als auch p berechnet werden, mit deren Hilfe sich bequem die Eckpunkte des Rechtecks bestimmen lassen (Formel 4.1).

Da das Zeichnen mit Antialiasing um ein vielfaches aufwendiger ist als ohne, fallen die Operationen für das Vertauschen von Variablen viel weniger ins Gewicht. Man kann also davon ausgehen, dass der Punkt S immer links vom Punkt E liegt (ansonsten werden die Punkte zuvor vertauscht). Unter dieser Annahme kann die Linie vom Punkt A ausgehend nach rechts bis zum Punkt C abgearbeitet werden.

$$p=\frac{dy \cdot r}{\sqrt{dx^2+dy^2}} \qquad z=\frac{dx \cdot r}{\sqrt{dx^2+dy^2}}$$

Formel 4.1

Um einen möglichst effizienten Algorithmus zu entwerfen ist es notwendig in den Codefragmenten die besonders oft ausgeführt werden mit so wenigen Operationen wie möglich auszukommen. In diesem Fall ist das die Berechnung der Überschneidung der Linie mit einem Pixel. Die Anzahl dieser Berechnungen kann man reduzieren, wenn man nur jene Pixel berücksichtigt, die Anteil an der Linie haben. Alle Pixel deren X-Koordinate kleiner sind als jene von Punkt A können nicht Anteil an der Linie haben. Ebenso jene deren X-Koordinate größer sind als die von Punkt C. Die Variable x wird zu Beginn auf den Wert der X-Koordinate des Punktes A gesetzt. Die Variable nx entspricht dem kleinstmöglichen Wert, der einer ganzen Zahl entspricht und größer als x ist, also die Grenze zum nächsten Pixel rechts vom Punkt A. Nun werden für alle Pixel, die zur Linie gehören und zwischen x und nx liegen die Anteile an der Linie berechnet und entsprechend im Bildspeicher gesetzt. Danach enthält x den Wert von nx und nx wird um eins erhöht. Dies wird so lange wiederholt solange x kleiner als die Variable ex ist, welche dem Wert der X-Koordinate des Punktes C entspricht. Abbildung 4.5 verdeutlicht dieses Vorgehen.

Nun muss festgestellt werden welche Pixel einer Pixelspalte (zwischen x und nx) zur Linie gehören. Dafür benötigt man die Werte der Y-Koordinaten der Linie an den Stellen x und nx sowohl an der oberen als auch an der unteren

Kante der Linie. Die entsprechenden Variablen lauten lo für die obere Y-Koordinate an der Stelle x und lu für den Wert der unteren Y-Koordinate. yo bekommt den Wert der oberen Y-Koordinate an der Stelle nx und yu für die untere Y-Koordinate. lo und lu werden zu Beginn auf die Y-Koordinate des Punktes A gesetzt. yo ergibt sich aus der Addition von lo mit der Steigung k der Linie multipliziert mit der Differenz von nx und x. yu ergibt sich aus der Addition von lu mit der negative Umkehrung der Steigung m der Linie ebenfalls multipliziert mit der Differenz nx - x.

Danach werden alle Pixel zwischen dem Minimum von lo und yo und dem Maximum von lu und yu auf ihren Anteil an der Linie untersucht und im Bildspeicher entsprechend gesetzt. Für die nächste Pixelspalte wird lo auf yo und lu auf yu gesetzt. yo wird um k erhöht, yu um m, allerdings nur solange, bis nx größer als ox oder ux ist. ox enthält den Wert jener X-Koordinate, an dem sich die Steigung ändert, also jene von Punkt B. ux enthält den Wert der X-Koordinate von D. Ist also nx größer als ux bedeutet das, dass x noch vor ux liegt. in diesem Fall ist die Berechnung von yu nicht trivial. Entsprechend dem Teilungsverhältnis muss lu sowohl um m als auch um k erhöht werden. Für alle folgenden Pixelspalten wird zur Berechnung von yu der Wert k zu lu addiert. Selbiges gilt für yo, lo und ox. In der letzten Pixelspalte ist nx größer als ex. In diesem Fall wird yo auf eo und yu auf eu gesetzt. Sowohl eo als auch eu entsprechen in diesem Fall der Y-Koordinate des Punktes C.

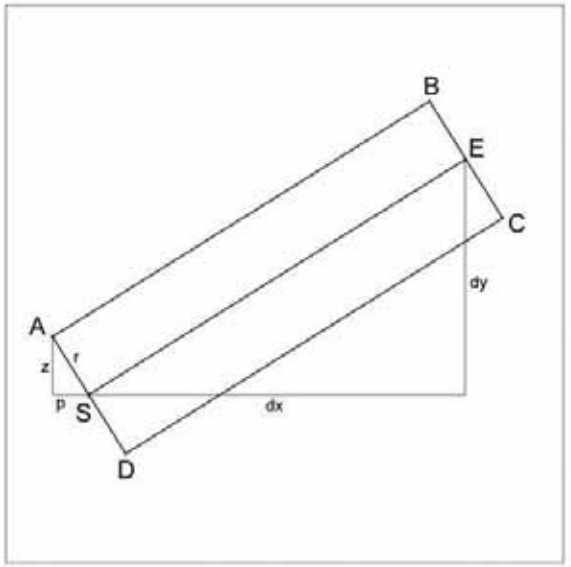


Abbildung 4.4

Implementierung

Zur Berechnung der einzelnen Pixel wird für jede Pixelspalte zunächst die Variable y auf die größte ganze Zahl, die kleiner als das Minimum von lo und yo ist, gesetzt. Die Variable ny ergibt sich aus der Erhöhung von y um eins. ey repräsentiert das Maximum von lu und yu. Der Anteil des Pixels zwischen x, nx und y, ny wird berechnet, y auf ny gesetzt und ny um eins erhöht. Dieser Vorgang wird wiederholt solange y kleiner als ey ist. Bei der Berechnung eines Pixels stehen die Werte aus Tabelle 4.2 zur Verfügung:

x	linke Begrenzung
nx	rechte Begrenzung
y	obere Begrenzung
ny	untere Begrenzung
lo	Y-Koordinate der oberen Linienkante an der Stelle x
yo	Y-Koordinate der oberen Linienkante an der Stelle nx
lu	Y-Koordinate der unteren Linienkante an der Stelle x
yu	Y-Koordinate der unteren Linienkante an der Stelle nx

Tabelle 4.2

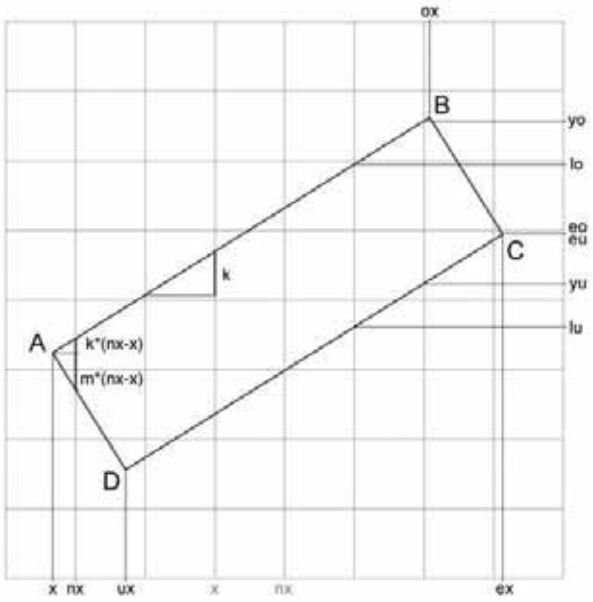


Abbildung 4.5

Um bei der Berechnung der einzelnen Pixel mit weniger Operationen auszukommen ist es von Vorteil, wenn die erste und letzte Pixelspalte separat behandelt werden, da bei diesen die Differenz zwischen nx und x kleiner als eins ist. Für alle anderen Spalten beträgt diese Differenz genau eins, wodurch sich einige Vereinfachungen ergeben. Der Linienanteil wird berechnet indem zuerst bestimmt wird wieviel der Linie sich unterhalb der oberen Linienkante im Pixel befindet. Danach wird der Anteil der sich unterhalb der unteren Linienkante im Pixel befindet subtrahiert. Das Ergebnis wird in der Variablen a gespeichert und das Pixel an der Position (x,y) entsprechend des Wertes a verändert. Dabei können die Methoden aus Kapitel *Punkte* verwendet werden. Für die erste und letzte Pixelspalte müssen die errechneten Werte von a noch mit der Differenz von nx und x bzw. ex und x multipliziert werden.

Die Berechnung des Anteiles unterhalb der oberen Pixelkante und jener unterhalb der unteren Pixelkante unterscheiden sich nicht. In beiden Fällen können neun verschiedene Fälle auftreten. Tabelle 4.3 bezieht sich auf die Berechnung des Anteiles unterhalb der oberen Kante.

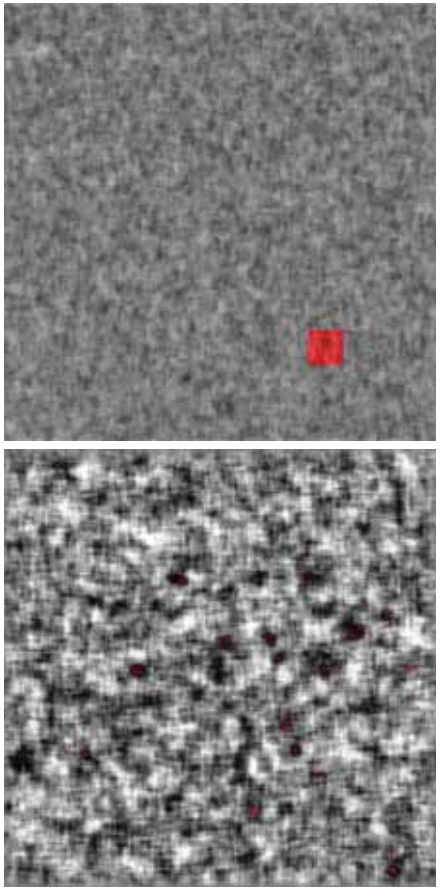


Abbildung 4.6

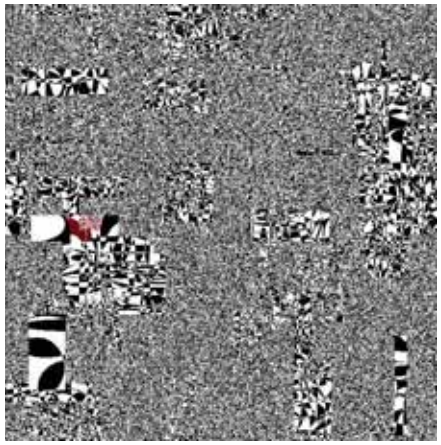
lo < y	yo < y	a = 1	O(0)	Kante befindet sich vollständig über dem Pixel
lo < y	yo > ny	a = (y + 0.5 - lo) / (yo - lo)	O(4)	Kante schneidet das Pixel zuerst an y dann an ny
lo < y	y < yo < ny	a = 1 - (yo - y)² / (2 (yo - lo))	O(6)	Kante schneidet das Pixel von oben kommend an y
lo > ny	yo < y	a = 1 - (lo - y - 0.5) / (lo - yo)	O(5)	Kante schneidet das Pixel zuerst an ny dann an y
lo > ny	yo > ny	a = 0	O(0)	Kante befindet sich vollständig unter dem Pixel
lo > ny	y < yo < ny	a = (ny - yo)² / (2 (lo - yo))	O(5)	Kante schneidet das Pixel von unten kommend an ny
y < lo < ny	yo < y	a = 1 - (lo - y)² / (2 (lo - yo))	O(6)	Kante schneidet das Pixel von innen kommend an y
y < lo < ny	yo > ny	a = (ny - lo)² / (2 (yo - lo))	O(5)	Kante schneidet das Pixel von innen kommend an ny
y < lo < ny	y < yo < ny	a = ny - (2 (lo + yo))	O(3)	Kante befindet sich innerhalb des Pixels

Tabelle 4.3

Für die obere Kante ist der häufigste Fall derjenige bei dem sich die Kante über dem Pixel befindet. Bei der unteren Kante tritt am häufigsten der Fall ein, dass sich die Kante unterhalb des Pixels befindet. Daher ist die Anordnung der einzelnen Fälle für die jeweilige Kantenberechnung unterschiedlich um einen größtmöglichen Durchsatz zu erzielen. Weiters wird auf die Verwendung von Gleitpunktzahlen verzichtet, um die Performance zusätzlich zu erhöhen. Stattdessen werden die Nachkommastellen durch Multiplikationen der Integerwerte mit dem Faktor 1024 simuliert. Diese Multiplikation bzw. Division durch 1024 lässt sich mit einem effizienten Links- bzw. Rechtsshift um 10 Bit realisieren. Die Multiplikation mit zwei entspricht einem Linksshift um 1 Bit.

Durch diese Ungenauigkeit kann es bei sehr flachen oder steilen Linien vorkommen, dass diese nicht korrekt dargestellt werden. Diese Fehler können teilweise behoben werden, wenn dx bzw. dy nicht auf Gleichheit mit Null verglichen wird, sondern überprüft wird ob dx (bzw. dy) groß genug ist um über die Distanz von dy (bzw. dx) ein korrektes Ergebnis liefern zu können. Bei sehr kurzen oder sehr dünnen Linien kommt es außerdem zu Fehlern die durch die Vernachlässigung der Spitzenwerte bei der Richtungsänderung von m nach k und umgekehrt entstehen.

Diese Artefakte beleben Bilder, die mit diesem Algorithmus entworfen werden, aber derart, dass sie nicht entfernt werden. Bisher wurde angenommen, dass die Linie von links nach rechts ansteigt. Selbige Überlegungen gelten aber auch für den anderen Fall, bei dem die Linie abfällt. Sogar die waagrechten und senkrechten Linien können durch entsprechendes Setzen der nötigen Variablen mit diesem Algorithmus (Code 4.3 *Antialiasing*) realisiert werden.



Rechtecke

Mit dem zuvor beschriebenen Algorithmus lassen sich auch Rechtecke zeichnen, wenn die Variable r entsprechend groß gesetzt wird. Benötigt man nur Rechtecke die liegen oder stehen, so können diese viel effizienter generiert werden. Folgender (Code 4.4 *Rechteck*) setzt alle Pixel die um width Pixel rechts vom Pixel id und um height Pixel unterhalb des Pixels id liegen. Abbildung 4.6 zeigt Ausschnitte aus einer Computeranimation die mit Hilfe dieses Algorithmus entstanden sind.

Ovale

Etwas komplexer ist die Erzeugung von Ovalen und Kreisen. Ein sehr effizienter aber nicht ganz korrekter Algorithmus hat einen maximalen Aufwand von O((Breite + Höhe) / 2) für die Berechnung der Werte, die für die Darstellung eines Ovals notwendig sind. Diese Implementierung liefert allerdings fehlerhafte Darstellungen bei Werten kleiner als vier der Parameter Breite und Höhe. Außerdem kann nicht garantiert werden, dass das Oval tatsächlich die angegebenen Ausmaße erreicht, oft wird es etwas breiter oder höher dargestellt.

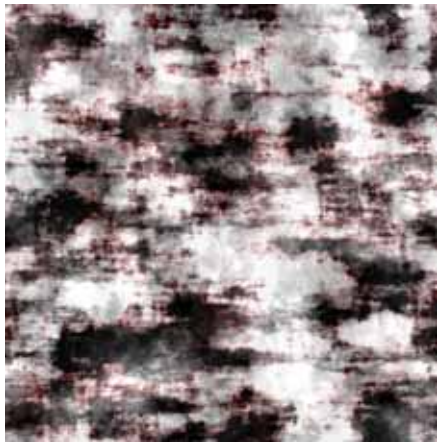


Abbildung 4.7

Um feststellen zu können, ob sich ein Pixel innerhalb oder außerhalb des Ovals befindet, wird angenommen, dass es sich beim Oval um einen Einheitskreis im Ursprung handelt und danach entsprechend skaliert. Somit ist jeder Punkt der vom Kreismittelpunkt weiter entfernt ist als eins, außerhalb des Kreises. Der Abstand vom Mittelpunkt ist genau dann größer als eins, wenn die Summe der Quadrate der Koordinaten größer als eins ist. Die Wurzel muss nicht gezogen werden, da die Wurzel aller Zahlen, die größer als eins sind, auch größer als eins ist. Dasselbe gilt für Zahlen die positiv aber kleiner als eins sind. Liegt eine Zahl zwischen 0 und 1 so liegt auch ihre Wurzel zwischen 0 und 1.

Wenn alle Pixel innerhalb des Rechtecks mit Breite w und Höhe h auf ihre Lage bezüglich des Einheitskreises überprüft werden, erhält man ein korrektes Ergebnis - auch für Parameterwerte kleiner als vier. Der Aufwand dafür betrüge aber O(w*h) bzw. O(w*h / 4) unter Berücksichtigung der Symmetrie. Dieser Aufwand kann deutlich reduziert werden, wenn die Überprüfung auf die Lage der Pixel nur für die Randpixel des Ovals anfällt. So wird zunächst das oberste Pixel des Ovals überprüft, welches sich in jedem Fall innerhalb des Ovals befindet. Nun bewegt man sich

solange nach rechts, bis sich das Pixel nicht mehr im Einheitskreis befindet. An dieser Stelle bewegt man sich nach unten und führt die Überprüfung nach rechts hin weiter fort. Unter Berücksichtigung der Symmetrie benötigt man so Breite/2 + Höhe/2 Überprüfungen. Bei jedem Sprung nach unten wird zuvor eine Linie bis zum aktuellen Punkt gemalt. Abbildung 4.7 aus einer Animation wurden mit Code 4.5 (*Oval*) erstellt.

Sound Synthese

Java Sound API

Leider ist es erst ab Version 1.3 mit Java möglich Sound zu Programmieren. Die Version 1.1 bietet nur die Möglichkeit Soundfiles abzuspielen, mit der zusätzlichen Einschränkung auf AU-Dateien mit 8 Bit. Für die Generierung von Sound in Echtzeit muss also in Applets verzichtet werden. Allerdings funktionieren Applets, wenn das Java Plugin installiert ist. Deshalb wird hier nicht sofort das Java Media Framework vorgestellt, sondern zuvor auf das im Grunde sehr ähnlich funktionierende Java Sound API. Dieses ist im Java SDK inkludiert und bietet alle Möglichkeiten um Sound abzuspielen, zu verarbeiten, zu generieren und auch aufzunehmen bzw. in Echtzeit vom Mikrophon Sound zu empfangen und diesen zu verarbeiten oder wiederzugeben.

Bei der Verarbeitung von Sound in Echtzeit spielen die Hardware und die Treiber eine entscheidende Rolle. Sound wird immer in einen Buffer geschrieben, der vom Soundsystem des Computers abgespielt wird. Die Größe des Buffers entscheidet über die Latenz, das ist die Zeit mit der Sound auf eine Eingabe reagieren kann. Ist die Latenzzeit groß, wird jede Veränderung am Sound erst verzögert wiedergegeben. Eine Latenz von mehr als 300ms eignet sich kaum mehr für den Echtzeiteinsatz. Ab einer Verzögerung von unter 100ms wird die Verzögerung unmerkbar. Gute Soundsysteme erreichen eine Latenz von unter 30ms.

In den 90er Jahren wurde von der Firma Steinberg eine Treibernorm entwickelt, die kurze Latenzzeiten garantieren soll. Diese sogenannten ASIO-Treiber existieren inzwischen für alle hochwertigen Soundkarten, sowohl für Windows- als auch für Macintoshsysteme. In das Java Sound API können diese Treiber eingebunden werden, wenn man entsprechende Software, die frei erhältlich ist, verwendet. Allerdings ist das nicht unbedingt notwendig, wenn man auf professionelle In-Out-Lösungen verzichten kann. Falls das vorhandene System allerdings nur mit großen Buffern funktioniert, kann notfalls auf ASIO-Treiber zurückgegriffen werden.

Die Größe des Soundbuffers lässt sich in Java festlegen. Allerdings ist nicht garantiert, dass das Soundsystem mit der gesetzten Buffergröße zurecht kommt. Ist der Buffer zu klein, kann es zu kurzen Aussetzern des Sounds kommen, was sich durch störende Klicks hörbar macht. Außerdem wird bei kleiner Latenz die Prozessorlast größer, da viel öfter in den Buffer geschrieben werden muss. Ist der Buffer sehr groß, wird die CPU deutlich entlastet, da der Buffer nach dem Füllen für einige Sekunden keine Daten mehr braucht und die CPU sich anderen Aufgaben widmen kann.

Die Klasse, die für die Wiedergabe von Klang verwendet wird, nennt sich *SourceDataLine*. Während die Klasse *TargetDataLine* für die Aufnahme von Klang verantwortlich ist. Die benötigte *Line* erhält man statisch von der Klasse *AudioSystem* durch die Übergabe einer *DataLine.Info*. Diese enthält zum einem das Audioformat mit dem gearbeitet werden soll und zum anderen die Größe des Buffers. Das Audioformat hängt von den in Tabelle 5.1 aufgezeigten Faktoren ab.

Begriff	Typische Werte	Beschreibung
Kanäle	Mono: 1, Stereo: 2	Anzahl der Tonspuren
Sample		Gemessene Amplitude zu einer bestimmten Zeit
Samplerate	8-48kHz, CD: 44100, DAT: 48000	Samples pro Sekunde pro Kanal
Samplegröße	8-24Bit CD: 16	Anzahl der Bit zur Codierung eines Samples
Frame		Gesamte Information zu einem bestimmten Zeitpunkt
Framegröße	4 Byte bei 16Bit stereo	Datengröße pro Frame
Framerate		Wie Samplerate bei unkomprimiertem Format
Encoding	PCM_(UN)SIGNED, ALAW, ULAW	Kodierungsformen
Vorzeichenbehaftet	Signed, Unsigned	Wertbereich entweder negativ und positiv oder nur positiv
Endian	PC: Big-Endian, MAC: Little-Endian	Reihenfolge der Byte bei 16Bit

Tabelle 5.1

Alle gängigen Soundsysteme können inzwischen alle Formate verarbeiten. Hier wird folgendes Format Verwendung finden, welches sich an der Qualität einer CD und das Dateiformat WAV anlehnt: 44100kHz, 16Bit, Stereo, PCM_SIGNED, Big-Endian. Durch Reduzierung der Samplerate, der Anzahl der Kanäle und der Bit pro Sample kann der Leistungs- und Datenaufwand deutlich reduziert werden, allerdings auch die Soundqualität. Inzwischen gibt es im Professionellen Bereich auch das Format mit 96kHz und 24Bit. Dieses Format wird intern verwendet und erst bei der Produktion in das CD-Format umgewandelt. Daraus ergibt sich der Vorteil, dass der Qualitätsverlust, der bei der Verarbeitung des Klangmaterials zwangsläufig entsteht, nicht ins Gewicht fällt, da die letzten acht Bit, in denen sich die Störgeräusche manifestieren, am Ende des Produktionszyklus abgeschnitten werden können.

Die hohe Samplerate ergibt sich aus der Verdoppelung der Samplerate des DAT-Formats mit 48kHz. Im professionellen Musikbereich hat sich zur Datenspeicherung das DAT-Band durchgesetzt. Dabei traten Probleme bei der Umwandlung in das CD-Format auf, was mit hohen Qualitätseinbußen verbunden war. Durch die Verdoppelung der Frequenz auf 96kHz ist dieses Problem beseitigt. Der zweite Vorteil der hohen Samplerate ist die Reduktion von digitalen Seiteneffekten, die entstehen, wenn Frequenzen über der halben Samplerate auftreten. Diese hohen Frequenzanteile spiegeln sich an der Obergrenze des aufzeichenbaren Spektrums und stören das korrekte Signal. Vor allem bei digitalen Effekten entstehen oft Frequenzen oberhalb des hörbaren Bereichs. Diese können entweder weggefiltert werden oder werden durch die höhere Samplerate korrekt abgebildet. Wird die Samplerate danach auf 44.1kHz reduziert, können die gesamten Anteile an hochfrequenten Störungen zuvor eliminiert werden.

AudioInputStream

Versucht man das Audiosystem mit der Klasse SourceDataLine zu programmieren treten oft nicht erklärbare Fehler auf, die sich vermeiden lassen, wenn die Daten nicht direkt erzeugt werden, sondern aus einem *AudioInputStream* gelesen werden. Dabei kann die erzeugende Klasse durchaus auch eine Unterklasse von AudioInputStream sein. Folgende Implementierung (Code 5.1 *Soundsystem*) erzeugt einen Thread, der sich um die Soundverwaltung kümmert, wobei in diesem Fall alle Samples auf Null gesetzt werden, also kein Klang hörbar ist.

Wichtige Parameter sind hierbei die Variable *frames* - mit der die Größe des internen Buffers bestimmt wird und *mul* - mit der die Größe des externen Buffers festgelegt wird. Der interne Buffer dient der Übertragung der Daten vom AudioInputStream zur SourceDataLine. Der externe ist der beschriebene, die Latenzzeit bestimmende, Systembuffer. Drei Methoden aus AudioInputStream werden überschrieben. *available* gibt die Anzahl der noch verfügbaren Bytes wieder. In diesem Fall ist das der maximale int-Wert, da die Klänge erzeugt werden und nie enden. *read* kann bei 8Bit-Mono-Signalen verwendet werden um ein einzelnes Byte auszulesen. *read* mit Parametern soll length Bytes - die ein vielfaches der Framegröße sein müssen - aus dem Stream lesen und in den Buffer data, am Index offset beginnend, geschrieben werden.

Durch die Vorkehrungen eines Fade-Ins - einem langsamen Anheben der Lautstärke - wird verhindert, dass die Initialisierung des SoundSystems hörbar wird. Die ersten Frames werden oft von Störgeräuschen begleitet, welche so verhindert werden können. Wird der Wert für samplevalue nicht auf Null gesetzt, sonder auf laufend verschiedene Werte zwischen -1 und 1 gesetzt, so entsteht ein hörbarer Klang. Zufällige Werte ergeben ein Rauschen. Periodische Signale einen Ton mit einer bestimmten Frequenz. Im Folgenden werden die Grundlagen und Techniken der Sounderzeugung beschrieben.

Wellenformen

Die künstliche Erzeugung von Klängen hat ihre Wurzeln in der Elektrotechnik. Einfache Schwingkreise erzeugen Töne mit bestimmten Frequenzen, die miteinander verkoppelt neue bisher nicht gehörte Sounds generierten. Die ersten Synthesizer bestanden aus vielen elektronischen Schaltkreisen die durch Kabel miteinander verbunden werden konnten um verschiedenste Geräusche erzeugen zu können. Dieser Vorgang war stark experimentell angelegt, wobei sich nach und nach ein Konzept durchsetzte, dass die Möglichkeiten von natürlichen Instrumenten bieten sollte und teilweise auch konnte. Dabei wurde versucht die Eigenschaften von Instrumenten elektronisch umzusetzen. Die ersten verkauften Synthesizer waren jene von Richard A. Moog (1964), die noch heute zu hohen Preisen gebraucht verkauft werden und durch ihren satten, mit digitalen Mitteln nicht erzeugbaren, einzigartigen Sound punkten.

Die Klangfarbe eines Instruments wird durch die Obertöne bestimmt, die neben der Grundschwingung hörbar sind. Eine Flöte hat kaum Obertöne, entspricht also fast einer reinen Sinusschwingung. Eine Pauke hat sehr viele Obertöne, wobei man dieser keine eindeutige Frequenz zuordnen kann. Je nach Frequenz und Amplitude der Obertöne ergibt sich ein Signal, dass unterschiedlich aussieht, wenn man es z.B. durch ein Oszilloskop visualisiert. Folgende Wellenformen spielen eine große Rolle in der Klangsynthese (Tabelle 5.2 und Abbildung 5.1).

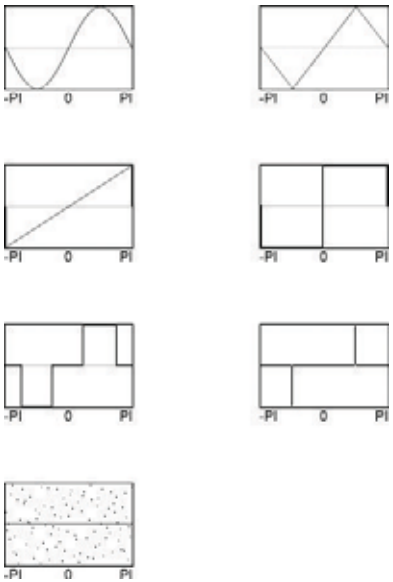


Abbildung 5.1:
Sinus, Dreieck, Sägezahn, Rechteck
Puls, Nadelimpuls, Rauschen

Wellenform	Spektrum	Klangfarbe
Sinus	nur Grundton	Flöte
Dreieck	ungerade Harmonische mit sehr stark abfallender Amplitude	Oboe
Sägezahn	alle Harmonische mit abfallender Amplitude zu je einem Drittel	Geige
Rechteck	ungerade Harmonische mit abfallender Amplitude mit je der Hälfte	Brummen
Puls	zwischen Rechteck und Nadelimpuls	Motor
Nadelimpuls (Klick)	alle Harmonische mit voller Amplitude	Zirpen
Rauschen	kein Grundton	Rauschen

Tabelle 5.2

Eine Wellenform wird vom Grundton, dessen Frequenz die Tonhöhe bestimmt (z.B. A3 = 440Hz), und seinen Obertönen, die ganzzahlige Vielfache der Basisfrequenz sind, bestimmt. Alle diese Signale, der Grundton und die Obertöne, sind reine Sinusschwingungen. Jede Wellenform lässt sich aus Sinusschwingungen aufbauen. Sind die Obertöne allerdings keine ganzzahligen Vielfachen der Grundfrequenz, so entsteht ein nichtperiodisches Signal, also ein Signal das keiner Wellenform entspricht. Bei diesen Signalen ist es schwer, den Grundton herauszuhören, sie werden deshalb den Geräuschen zugeordnet. Werden zwei Geräusche miteinander verglichen, lässt sich zwar feststellen, welches der beiden höher (größere Basisfrequenz) und welches tiefer (kleinere Basisfrequenz) ist, doch der Gesamteindruck ist zu diffus um es z.B. einem Ton, wie dem A3, zuzuordnen.

Additive und subtraktive Soundsynthese

Bei der additiven Soundsynthese werden sehr viele Sinuswellen mit unterschiedlicher Frequenz und Amplitude zusammengemischt. Daraus entsteht ein spezielles Spektrum, das sich auch verändern lässt, indem man die Frequenzen und Amplituden der einzelnen Sinusgeneratoren ändert. Dabei können viele hunderte Generatoren zum Einsatz kommen, die durch ihr Zusammenwirken Schwebungen und andere komplexe Strukturen aufbauen können, ohne an den Parametern der einzelnen Oszillatoren etwas zu ändern. Code 5.2 simuliert 50 Sinusgeneratoren die in drei Gruppen eingeteilt werden: Zehn Generatoren erzeugen Wellen im Bereich zwischen 100 und 110 Hz. Zwanzig Generatoren erzeugen Sound im Bereich von 500 bis 1000 Hz und 20 Generatoren liefern Wellen im Bereich zwischen 1000 und 5000 Hz.

```
double wps = new double[50];        // stored position of each generator (starting at 0)
double frq = new double[50];        // movement of position after 1 sample for each generator

public void init() {
     ...
     for(int i=0; i< 50) {          // for all generators
          s+= Math.sin(wps[fi]);    // sinus of current generatorposition added to samplevalue
          wps[fi]+= frq[fi];        // generatorposition moved
          fi++;                     // next generator
     }
     s/= 50.0;                      // samplevalue must be divided to be in range [-1,1]
     ...
}
```

Code 5.2

Bei der subtraktiven Soundsynthese wird zunächst weißes Rauschen erzeugt, welches alle Frequenzen enthält. Diesem Rauschen werden gezielt gewisse Spektralanteile durch Filter entzogen. In der Theorie können die Ergebnisse beider Verfahren, sowohl der additiven als auch der subtraktiven Klangmischung, von beiden Syntheseverfahren erzeugt werden. Allerdings eignet sich in der Praxis die additive Soundsynthese besser zur Erzeugung von Tönen und die subtraktive Soundsynthese besser zur Gestaltung von Geräuschen.

Elektronische Bauteile im analogen Synthesizer

Ein VCO (Voltage Controlled Oscillator) ist ein Wellengenerator, dessen Frequenz von einer Spannungsquelle gesteuert wird. Ist die Spannung hoch, schwingt auch der Oszillator mit hoher Frequenz. Damit kann die Tonhöhe z.B. mit einem Keyboard, das für jede gedrückte Taste eine andere Spannung liefert, bestimmt werden. Soll der Synthesizer mehrstimmig sein, so bedarf es mehrerer solcher VCOs. Die ersten Synthesizer waren noch monophon. Im Prinzip ist ein mehrstimmiger Synthesizer ein Konglomerat mehrerer gleichartiger monophoner Synthesizer.

Hüllkurve

ADSR

Ein VCA (Voltage Controlled Amplifier) verstärkt ein Signal entsprechend der anliegenden Spannung. Damit lässt sich die Amplitude des Signals spannungsabhängig verändern. Dieser Bauteil wird zur Generierung von Hüllkurven, ein weiterer wesentlicher Faktor bei der Simulierung von Instrumenten, benötigt. Die Hüllkurve bestimmt den Amplitudenverlauf eines Signals. Die einzelnen Instrumente unterscheiden sich nicht nur in der Klangfarbe, sondern auch in ihrer Hüllkurve signifikant voneinander. Dabei können bei den meisten Instrumenten folgende Parameter der Hüllkurve bestimmt werden: Attack, Decay, Sustain, Release.

Decay = 0

Sustain = 100%

Eine sogenannte ADSR-Kurve wird durch vier Parameter beschrieben. Zum einen die Attack-Time, die bestimmt wie lange es dauert, bis das Instrument vom ruhigen Zustand in den hörbaren Zustand übergeht. Die Amplitude ist beim Ansetzten der meisten Instrumente besonders laut. Wenn z.B. das Schwingblatt einer Klarinette sich zu bewegen beginnt, tut es das in den ersten ms besonders unkontrolliert und stark ausschlagend. Danach beruhigt es sich und liefert einen gleichbleibenden Ton, sowohl die Amplitude als auch die Frequenz betreffend. Dasselbe gilt für das Klavier, bei dem ein Hammer auf eine oder mehrere Seiten schlägt und dabei zuerst eine wilde und starke Bewegung in der Seite auslöst, die sich dann in ihrer Resonanzfrequenz stabilisiert.

Besitzt kein wirkliches Sustain

Abbildung 5.2: Klavier, Geige, Gitarre

Die Decay-Time beschreibt die Länge der Zeit, die das Instrument benötigt, um vom unkontrollierten, lautstarken Zustand in den gleichbleibenden zu gelangen. Das Sustain-Level gibt die Lautstärke an, die das Instrument in seinem gleichbleibenden Zustand besitzt. In diesem Zustand verharrt das Instrument so lange, wie z.B. die Taste am Keyboard gedrückt bleibt. Wird die Taste losgelassen, was das Ende des Tons einleiten soll, soll das Signal nicht sofort still sein, sondern langsam in die Stille übergleiten. Das Signal wird also mit einem Fade-Out beendet. Die Release-Zeit gibt an,

wie lange dieses Ausfaden dauern soll. Abbildung 5.2 zeigt die Hüllkurven verschiedener Instrumente und die Abbildung dieser auf eine ADSR-Hüllkurve.

Unter einem VCF (Voltage Controlled Filter) versteht man einen Filter (meist ein Low-Pass-Filter), dessen Cut-Off-Frequenz von der anliegenden Spannung kontrolliert wird. Die Resonanz des Filters lässt sich oft auch durch eine anliegende Spannung steuern. Gerade diese Resonanz-Filter machen den typischen Sound der analogen Synthesizer aus und lassen sich digital nicht nachahmen. Deshalb werden analoge Synthesizer nach wie vor verkauft und verwendet. Neuere Modelle arbeiten intern meist digital und lediglich die Resonanzfilter sind elektronisch verwirklicht.

Da Töne mit Instrumenten nicht immer monoton gespielt werden, sondern vielen Instrumenten - vor allem den Blasinstrumenten - Vibrationen sowohl die Tonhöhe als auch die Lautstärke betreffend entlockt werden können, bieten Synthesizer zusätzlich LFOs (Low Frequenzy Oscillators) an. Diese Bauteile arbeiten ähnlich wie die VCOs, allerdings mit sehr viel niedrigeren Frequenzen. Diese LFOs können verwendet werden um VCAs, VCOs aber auch VCFs zu steuern. Je nach Einsatz entstehen so Tremolos, Vibratos oder Klangfarbenveränderungen. Die einzelnen Bauteile können entweder fix verdrahtet sein oder sie sind frei miteinander verbindbar. Häufig sind analoge Synthesizer so aufgebaut, dass zwei VCOs (die verschiedene Wellenformen und Frequenzen haben können) von einem Keyboard gesteuert werden. Die Ausgänge der VCOs münden in einen VCF der von einer ADSR-Hüllkurve gesteuert wird. Die Ausgänge der Filter gehen in einen VCA dessen Verhalten ebenfalls von einer ADSR-Hüllkurve gesteuert wird. Das Signal kann vom VCA direkt in ein Mischpult oder einen Verstärker geleitet werden. Alle diese Bauteile werden zusätzlich von LFOs *moduliert*. Ist die Amplitude der LFOs Null, wirkt sich der LFO nicht auf den Bauteil aus, ist die Amplitude maximal, wird der Bauteil maximal von ihm beeinflusst.

Code 5.3 (*Synthesizer*)verwendet die frei erhältliche Library ALab [14]. Diese ist nicht für den Echtzeiteinsatz entworfen worden, kann aber auf Leistungsfähigen Rechnern in Realtime eingesetzt werden. Mit dieser Library kann man viele Synthesearten verwirklichen. In diesem Beispiel wird ein Ton mit Hilfe zweier Dreieck-VCOs, die leicht gegeneinander verstimmt sind und von zwei LFOs moduliert werden, erzeugt. Zwei VCAs, die ebenfalls durch zwei LFOs moduliert werden, werden zusammengemischt und von einer ADSR-Kurve gesteuert. Das Ergebnis wird dreimal hintereinander abgespielt. Hier soll noch erwähnt werden, dass das Pedant zu Voltage-Controlled auf Rechnern Digital-Controlled heißt, also DCO statt VCO usw.

Modulationsformen

Eine andere Form der Klangbearbeitung ist die Ringmodulation. Dabei werden die Signale zweier Oszillatoren durch Addition und Subtraktion zu einem neuen Signal vermischt. Das Ergebnis der Ringmodulation kann mit zwei VCOs

und einem VCA verwirklicht werden. Der VCA muss allerdings in der Lage sein auch negative Kontrollströme korrekt verarbeiten zu können. Das Signal des ersten VCOs gelangt in einen VCA, der vom zweiten VCO gesteuert wird. Kann der VCA nur positive Kontrollströme verarbeiten muss das Signal des modulierenden VCOs durch Addition mit einer Gleichspannung auf ein positives Niveau angehoben werden. Das Ergebnis entspricht dann aber nicht mehr der Ringmodulation. In diesem Fall spricht man von einer Amplitudenmodulation. Besonders interessant sind diese beiden Verfahren bei der Anwendung auf die menschliche Stimme. Dabei kommt das - entweder aufgezeichnete oder live gesprochene - Wort in den VCA und wird von einem VCO moduliert, was zu einer Verfremdung der Stimme führt. Code 5.4 führt auf ein Soundfile eine Ringmodulation mit 50Hz durch.

```
public void init() {
     ...
     AWave sig = null;
     try {
          AFile file = new AFile("stimme.wav", 0);     // soundfile declared
          sig = ALab.io.WAVE.getWave(file);            // soundfile opened as sig
     }
     catch(Exception e) { System.out.println(e); }

     AAmp rmd = new AAmp(sig, 1, 0);                   // DCA created for sig
     AWave dco = new ASin(ch, 50, 0);                  // DCO created with 50hz
     rmd.setAmpMod(dco);                               // modulator set to DCO
     snd.append(rmd, (int)rate * 10);                  // 10 seconds appended
     ...
}
```

Code 5.4

Eine der wichtigsten und erfolgreichsten Modulationsformen ist die von der Firma Yamaha entwickelte Frequenzmodulation. Der nach wie vor legendäre Yamaha DX7 setzte als erster Synthesizer diese digitale Technik ein. Dabei wird die Frequenz eines VCOs vom Signal eines anderen VCOs gesteuert. Der Yamaha DX7 bot sechs VCOs an, die fast beliebig miteinander verschaltet werden konnten. Somit ließen sich Instrumente viel realistischer Abbilden als mit analogen Synthesizern, es war aber auch möglich völlig neue Sounds aber auch Geräusche zu erzeugen. Gerade die Möglichkeit der FM auch glockenähnliche Klänge und Geräusche generieren zu können, half bei der Simulation der natürlichen Instrumente. So entstehen bei jedem Instrument zwangsläufig auch immer Geräuschanteile, vor allem in der Attack-Periode also dem Tonansatz. Man denke nur an die hohl klingende Luftströmung beim Anblasen einer Flöte oder den harten Schlag der Hämmer auf die Seiten des Klaviers. Code 5.5 erzeugt einen Klang dessen Obertöne jenen eines Glockenklangs ähneln.

```
public void init() {
     ...
     AWave fm1 = new ASin(ch,  440, 0);          // fm-unit 1
     AWave fm2 = new ASin(ch,  500, 0);          // fm-unit 2
     AWave fm3 = new ASin(ch,  440, 0);          // fm-unit 3

     fm2.setFreqMod(fm3);                        // fm2 modulated by fm3
     fm1.setFreqMod(fm2);                        // fm1 modulated by fm2
     snd.append(fm1, (int)rate * 5);             // 5 seconds appended
     ...
}
```

Code 5.5

Die Library ALab eignet sich besonders für das Auffinden der gesuchten Struktur und der gewünschten Parameter. Hat man diese gefunden ist es aus Performancegründen oft sinnvoll, die Sounderzeugung ohne die Library zu implementieren und zu optimieren. Durch die modulare und offene Struktur von ALab können zwar experimentell sehr viele Klänge überschaubar erzeugt werden, aber der Overhead, der durch die vielen dafür notwendigen Instanzen und Methoden entsteht, verursacht ein schlechtes Laufzeitverhalten. Klänge deren Berechnung komplex ist, die sich aber nicht z.B. durch Interaktivität ändern, können schon zuvor berechnet und zwischengespeichert werden. Code 5.6 (*FM*) zeigt, wie der oben angeführte Code 5.5, ohne ALab zu implementieren ist.

Diese Art der Frequenzmodulation lässt sich sehr einfach und schnell berechnen, sie entspricht aber nicht jener des DX7, da die Werte von v1, v2 und v3 sich zwischen -1 und 1 bewegen. Somit ergibt die Frequenz des modulierten Oszillators bei einem Nulldurchgang des Modulators Null. Dieser Fall soll aber nicht eintreten, vielmehr sollte das Ergebnis in diesem Fall die Basisfrequenz ergeben. Dies erreicht man, indem zu den Werten v2 und v3 jeweils Eins dazu addiert wird. Multipliziert man zuvor diese Werte mit einem Wert zwischen 0 und 1, kann man den Modulationsgrad zusätzlich beeinflussen. Code 5.7 (*FM*) zeigt wie diese komplexere aber auch interessanter klingende FM zu berechnen ist.

Die Phasenmodulation ähnelt der Frequenzmodulation sehr. Dabei beeinflusst der Modulator nicht die Frequenz sondern die Phase eines Oszillators. Zwar gibt es Situationen in denen sich diese beiden Modulationsarten deutlich hörbar voneinander Unterscheiden, in den meisten und vor allem gängigsten Anwendungen klingen sie aber fast ident. In ALab können alle Unterklassen von *AWave* (zu denen auch die Soundfile abspielende Klasse *ASampleLoop* gehört) einen Modulator für die Frequenz als auch die Phase zugewiesen bekommen. Die entsprechenden Methoden lauten *setFreqMod* und *setPhaseMod*.

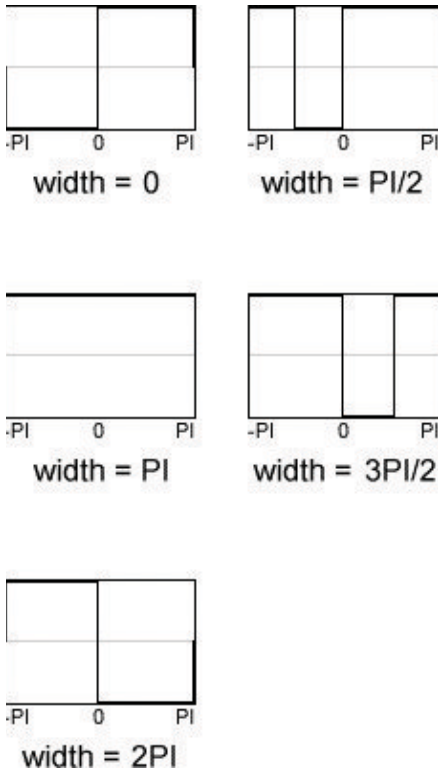


Abbildung 5.3

Eine weitere Modulationsart ist die Pulsbreitenmodulation, deren klangliche Vielfalt aber bei weitem nicht die Dimensionen der Frequenzmodulation erreicht. Eine Pulswelle befindet sich in jeder Phase entweder in einem Maximum oder auf Null. Bei der Rechteckwelle ist das Verhältnis zwischen diesen beiden Zuständen 50:50. Beim Klick oder dem Nadelimpuls beträgt das Verhältnis nahezu 0:100. Die Pulsbreite entspricht dem Zähler dieses Bruchs, bestimmt also wie lange sich die Welle im Maximum befindet. Veränderungen an diesem Parameter bewirken auch Änderungen in den Obertönen. Durch Modulation dieses Parameters - was elektronisch relativ einfach zu bewerkstelligen war und auch digital leicht realisierbar ist - kann der monoton klingenden Pulswelle mehr Lebendigkeit abverlangt werden. In ALab kann mit der Methode *setWidthMod* in der Klasse *ASquare* die Pulsbreite moduliert werden, allerdings entsprechen die Werte dabei nicht dem hier erklärten Modell. Abbildung 5.3 zeigt die Interpretation des Wertes für den Parameter *width*. Für den Wert Null beträgt das Verhältnis 50:50. Mit Werten nahe PI kann ein verkehrter (aber gleich klingender) Nadelimpuls erzeugt werden.

Sampling

Trotz der neuen Möglichkeiten der FM setzte sich letztendlich bei der Simulation von Instrumenten das Samplingverfahren durch. Dabei wird der Ton eines Instrumentes aufgenommen und dann, abhängig von der Frequenz mit der er wiedergegeben werden soll, mit anderer Geschwindigkeit abgespielt. Mit zunehmender Abspielgeschwindigkeit wird der Ton zwar immer höher, die Abspieldauer dafür aber immer kürzer. Um die Dauer eines Tones zu steuern, können sogenannte Loops erstellt werden. Das sind Bereiche in der Tonaufnahme, die, solange der Ton dauern soll, in einer Endlosschleife abgespielt werden. Das entspricht in etwa der Sustain in den zuvor besprochenen ADSR-Hüllkurven. Drückt man auf die Taste eines Keyboards wird die Attack- und Decayphase des Klangs abgespielt. Wenn das Signal die Sutstainphase erreicht wird diese wiederholt wiedergegeben bis die Taste losgelassen wird. Ab diesem Zeitpunkt wird die Releasephase des Samples wiedergegeben. Die Schwierigkeit beim Sampling ist das Auffinden geeigneter Looppunkte. Diese Punkte geben die Grenzen der Endlosschleife an, also den Punkt *an dem* zurückgesprungen werden soll und den Punkt *zu dem* gesprungen werden soll.

Es gibt zahlreiche Methoden das setzen der Loops zu vereinfachen. So kann z.B. sichergestellt werden, dass Looppunkte immer nur an Stellen gesetzt werden können, an denen das Signal einen Nulldurchgang hat. Das stellt zumindest sicher, dass keine Sprünge *im* Signal durch den *Sprung im* Signal entstehen können. Allerdings treten noch andere - viel schwerwiegendere - Probleme bei der Setzung von Loops auf. Fast alle Instrumente (mit Ausnahme der Blas- und Streichinstrumente) haben keine echte Sustain, d.h. sie werden immer leiser. Wird ein Loop in so ein Signal gesetzt ist dieses am Begin des Loops lauter als am Ende. Die Lautstärke springt also nach jedem Loopdurchgang in die Höhe, was zum einen unnatürlich klingt und zum anderen - vor allem bei sehr kurzen Loops - eine Art Ringmodulation verursacht und die Klangfarbe des Instruments verändert. Dieses Problem kann relativ einfach beseitigt werden, indem das Signal in seinem Amplitudenverlauf so verändert wird, dass es im Loopbereich einen

linearen Verlauf erhält. Dadurch entsteht zwar eine gewisse Unnatürlichkeit, die jedoch wesentlich unauffälliger ist als der Lautstärkesprung der ohne diese Korrektur entsteht.

Die Tatsache, dass ein natürliches Instrument nie in einen monoton gleichbleibenden Zustand kommt verhindert auch theoretisch die Methode des Samplings. Ein Loop mag zwar so klingen wie das simulierte Instrument, er ist aber monoton und das ist hörbar. Die Erfinder von Samplern versuchen auch diese Einschränkung durch Modulationsmöglichkeiten zu verbessern, letztendlich kann aber mit reinem Sampling über die Künstlichkeit der erzeugten Klänge nicht hinweg getäuscht werden. Konventionelle Sampler bieten z.B. die Möglichkeit das Signal währen des Loops mit LFOs zu beeinflussen, experimentellere Sampler können die Positionen der Looppunkte ändern. Durch die geringeren Kosten von Speicher ist es inzwischen auch möglich für jede gedrückte Taste mehrere Samples - abhängig z.B. von der Anschlagstärke - aufzunehmen und dementsprechend wiederzugeben. Aber auch das ändert nichts daran, dass ein wirkliches Instrument niemals ein und den selben Klang zweimal erzeugt. Jeder Ton eines Klaviers ist einzigartig und geschulte Ohren können diesen Unterschied ausmachen.

Die Library ALab bietet mit den Klassen *ASample* und *ASampleLoop* die Möglichkeit Samples abzuspielen und Loops zu erstellen. Die Klasse ASampleLoop ist eine Unterklasse von AWave und kann daher wie eine Wellenform behandelt werden, es ist also auch Frequenzmodulation damit möglich. Die Loops werden nicht direkt durch zwei Punkte bestimmt, sondern durch einen Startpunkt und die Länge des Loops in Samples, wobei beide Werte moduliert werden können. Um Amplitudenunterschiede oder andere Übergangsgeräusche beim Springen in der Schleife zu minimieren, kann ein Smoothing gesetzt werden, dass zwischen einer gewissen Anzahl von Samples jeweils am Anfang und Ende des Loops interpoliert, also eine Überblendung vollzieht. Die Anzahl dieser Samples ist ebenfalls modulierbar.

Es gibt noch andere Versuche und Methoden Instrumente zu simulieren, besonders erfolgreich gelingt dies mit den Methoden des Pysical-Modelings, wo versucht wird die physikalischen Eigenschaften der Instrumente in die Berechnung des Klangs miteinzubeziehen. Dies alles mag für Wissenschaftler interessant sein, für die Kunst spielt es insofern eine Rolle, als dass Synthesizer und Sampler - die eigentlich gebaut wurden um simulieren zu können - benutzt werden können um neue, einzigartige Instrumente und Klänge zu erstellen. Aus der sogenannten Unterhaltungs-Musik sind diese neuen Klänge inzwischen nicht mehr wegzudenken. Aber auch in der zeitgenössischen E-Musik werden elektronische Instrumente immer häufiger eingesetzt. Wo es natürlicher Instrumente bedarf, ist es sinnvoller diese auch zu verwenden.

Wavetables

Bei der Wavetablesynthese werden statt Wellengeneratoren Tabellen mit Samplewerten - also kleine Sampleloops - zur Klangerzeugung eingesetzt. Dabei können mehrere Tabellen eingesetzt werden und durch Interpolation auch

Zwischenstufen - neue Wavetables - generiert werden. Als Wavetable kommen nicht nur verschiedenste Wellenformen in Frage, sondern auch Sprachsegmente und rhythmische Patterns. Ist eine stufenlose Interpolation zwischen mehreren Wavetables möglich, kann das Verhältnis zwischen beiden Tabellen von einem Modulator beeinflusst werden. Folgendes Beispiel (Code 5.8) zeigt die Modulation zwischen einer Sägezahnwelle und einer Dreieckwelle. Durch diesen Übergang entsteht ein ähnlicher Effekt wie bei einer Tiefpassfilterung.

```
AWave saw = new ASaw(ch, 0.5, 0);                    // sawtooth-wave 0.5 hz
AWave tri = new ATri(ch, 0.5, 0);                    // triangle-wave 0.5 hz
AWave lfo = new ASin(ch, 1.0, 0);                    // lfo with 1 hz

AWave linear = new ASimpleWave(ch, 440, 0);          // linear-wave with 440 hz

ATable tab1 = null, tab2 = null;
try {
    tab1 = new ATable(saw, (int)rate, (int)rate);    // wavetable 1 with saw-samples
    tab2 = new ATable(tri, (int)rate, (int)rate);    // wavetable 2 with tri-samples
} catch(Exception e) { System.out.println(e); }

// ATablelize(AWave, table1, table2, mix)
ATablelize mod = new ATablelize(linear, tab1, tab2, 1);
mod.setMixMod(lfo);                                  // lfo modulates between tables
```

Code 5.8

Andere wichtige Syntheseverfahren sind die *Granularsynthese* und die *Resynthese*. Bei der Granularsynthese werden Klänge durch eine sehr große Anzahl von Basisklängen erzeugt. Diese Basisklänge sind einfach aufgebaute Sounds, die meist nur aus einem VCO und einer Attack-Release-Hüllkurve bestehen. Durch unterschiedliche Anordnung dieser Klänge entstehen komplex strukturierte, einzigartig klingende Sounds. Dabei sind die Parameter der Basisklänge von relativ geringer Bedeutung, entscheidend ist vielmehr, wieviele dieser Klänge zu welcher Zeit gespielt werden. Der Begriff Resynthese umfasst die Möglichkeiten, die durch die Fourieranalyse entstehen. Im Allgemeinen versteht man darunter das Zerlegen von Klängen mittels FFT in seine Frequenzanteile mit anschließender Rücktransformation. Vor der Rücktransformation können Frequenzbänder vertauscht werden, was zu erstaunlich, fremdartig klingenden Ergebnissen führen kann, wobei die ursprüngliche Struktur und Klangfarbe oft noch erkennbar bleibt (siehe Kapitel *Fourier*).

Imagedriven Music

Music Instrument Digital Interface

Animationsgesteuerter Sound bzw. Imagedriven Music gehört ebenfalls in den Bereich der Computerkunst. Dabei dienen Parameter die aus Bildern gewonnen werden als Parameter zur Steuerung von klanglichen Ereignissen. Dabei ergibt sich eine Vielzahl von Möglichkeiten, wie diese beiden Medien zusammenspielen können, wobei der Zusammenhang beider Medien für den Betrachter erkennbar oder zumindest wahrnehmbar bleiben soll.

Für die Klangerzeugung eignete sich bis vor einigen Jahren vor allem das Protokoll MIDI (Music Instrument Digital Interface), das mit wenig Aufwand von Computern verwaltet werden kann, da es nicht die Sounddaten selbst, sondern nur die Daten zur Steuerung von Sound überträgt. Die wichtigsten MIDI-Parameter sind dabei Tonhöhe und Lautstärke, die für jeden Ton angegeben werden müssen.
Das MIDI-Protokoll wurde für die serielle Datenübertragung entworfen und wird noch heute zur Ansteuerung von Synthesizern, Samplern aber auch Mischpulten und Effektgeräten verwendet. Durch die serielle Übertragung gibt es in MIDI keine gleichzeitig stattfindenden Ereignisse. Gleichzeitigkeit wird durch kurz aufeinanderfolgende Befehle simuliert. Für die meisten Anwendungen reicht diese Pseudosimultanität auch aus, allerdings nicht bei der Ansteuerung von elektronischen Schlag- und Rhythmusinstrumenten. Nur kleinste Timingunterschiede im Rhythmus verändern den Groove signifikant. Sogenannte Drummachines werden daher wenn möglich nicht mehr extern angesteuert, sondern als Softwarelösungen mit eigenen Übertragungsprotokollen verwirklicht.

In MIDI werden alle Befehle auf Bytes aufgeteilt. Dabei gibt es zwei Typen von Bytes, die Statusbytes und die Databytes. Diese Unterscheiden sich durch das erste Bit voneinander. Ist das erste Bit gesetzt, handelt es sich um ein Statusbyte, ist es nicht gesetzt um ein Databyte. Ein Databyte symbolisiert prinzipiell immer nur einen Wert für einen Parameter. Der entsprechende Parameter wird im Statusbyte deklariert. Die wichtigsten MIDI-Befehle sind die Kommandos Note-On und Note-Off. Ein Note-On veranlasst z.B. einen Synthesizer eine Note zu spielen. Ein Note-Off beendet das Spielen einer Note. Tritt ein Note-On ein aber kein entsprechender Note-Off, so wird der Ton unendlich lange weitergespielt, was auch in der Praxis immer wieder vorkommt. Dabei ist zu beachten, dass ein Note-Off unbedingt dasselbe Gerät und auch dieselbe Tonhöhe erreicht, wie das zuvor ausgesandte Note-On.

MIDI-Channel

Die meisten MIDI-tauglichen Geräte besitzen drei MIDI-Anschlüsse. Der MIDI-Out-Port sendet MIDI-Daten zu einem anderen Gerät. Der MIDI-In-Port empfängt Daten und der MIDI-Thru-Port sendet dieselben Daten an ein Gerät, die der MIDI-In-Port empfängt. So können mehrere MIDI-Geräte zusammengeschlossen werden, wobei zu beachten ist das die Kette irgendwo unterbrochen ist, da es sonst zu MIDI-Rückkoppelungen kommt und die Befehle im Kreis laufen, also immer wieder ausgeführt werden. Diese Unterbrechung wird oft im Computer vorgenommen, wo per Software das MIDI-Thru aktiviert und deaktiviert werden kann. Abbildung 6.1 zeigt eine Typische Situation für die Arbeit mit MIDI, bei der ein sogenanntes Masterkeyboard zum Einspielen der Noten verwendet wird, die von einem Computer - der als Sequenzer fungiert - aufgenommen und wiedergegeben werden können. Sowohl die vom Keyboard als auch vom Computer ausgehenden Daten werden von allen MIDI-Geräten empfangen, wenn der Computer die Daten vom MIDI-In-Port an den MIDI-Out-Port durchreicht.

Wird nun ein Note-On ausgesandt, muss jedes erreichte Instrument entscheiden können, ob dieses Note-On für ein anderes Instrument oder für sich selbst bestimmt ist. Dieses Problem wird mit Hilfe der MIDI-Channels gelöst, von denen es 16 verschiedene gibt. Jeder Synthesizer kann so eingestellt werden, dass er nur auf bestimmte MIDI-Channels reagiert. So kann z.B. festgelegt werden, dass die Glockenklänge des Yamaha DX7 auf den Kanal 7 reagieren, die athmospherischen Sounds des DX7 aber auf Kanal 8. Zusätzlich wird das Piano des AKAI-Samplers von Kanal 1 angesprochen, der Hall wird auf Kanal 15 gesteuert. Legt man nun den Bass des analogen Oberheim Synthesizers ebenfalls auf Kanal 8, so werden alle Töne die für Kanal 8 bestimmt sind sowohl vom DX7 als auch vom Oberheim gespielt, was manchmal erwünscht ist.

Abbildung 6.1

Der MIDI-Channel wird im Statusbyte codiert und zwar in den letzten 4 Bit. Beim Note-On Befehl ist zusätzlich das vierte Bit gesetzt, beim Note-Off nicht. Beide werden mit zwei nachfolgenden Bytes beendet (Tabelle 6.1). Das erste Databyte codiert die Tonhöhe in Halbtonschritten, wobei 69 das A3, also 440Hz, repräsentiert. Das zweite Valuebyte bestimmt die Anschlag- bzw. Loslaßgeschwindigkeit. Alle Keyboards liefern diese Parameter, wobei die Tonhöhe von der entsprechenden Taste und die Anschlaggeschwindigkeit (Velocity) von einem in der Mechanik der Tasten angebrachten Sensor abhängt. Da das erste Bit eines Valuebyte immer ungesetzt sein muss, können pro Byte nur 128 verschiedene Werte übertragen werden. Der Wertebereich in MIDI reicht also von 0 bis 127.

Statusbyte	Databyte 1	Databyte 2	Befehl
1001.0000 = 144	0011.1100 = 60	0110.0100 = 100	Note-On (Channel:1, Ton:C3, Vel:100)
1001.0001 = 145	0011.0000 = 48	0111.1111 = 127	Note-On (Channel:2, Ton:C2, Vel:127)
1001.1111 = 159	0011.1110 = 62	0000.0000 = 0	Note-On (Channel:16, Ton:D3, Vel:0)
1000.0000 = 128	0011.1100 = 60	0110.0100 = 100	Note-Off (Channel:1, Ton:C3, Vel:100)

Tabelle 6.1

MIDI-Befehle lassen sich in drei Gruppen einteilen: Short-Messages, Sysex-Messages und Meta-Messages. Die Short-Messages - zu denen Note-On und Note-Off gehören - sind Befehle die aus ein bis drei Bytes bestehen. Diese Befehle werden am öftesten eingesetzt und müssen deshalb besonders kurz sein, um die serielle Leitung möglichst wenig zu belasten. Es gibt zur zusätzlichen Reduzierung der benötigten Daten noch eine Vereinbarung durch die das Statusbyte entfallen kann, wenn es sich nicht ändert. Sollen z.B. zwei Töne auf demselben MIDI-Channel gespielt werden, so reicht es das Statusbyte für Note-On nur einmal zu senden, gefolgt von vier Databytes, wobei die ersten beiden die erste Note und die letzten beiden die zweite Note beschreiben. Diese Vereinbarung kommt besonders bei Messages wie Pitchbend und Modulation zum Tragen, da diese Messages sehr oft hintereinander auftreten können.

Control Change

Die Short-Messages sind wieder in Untergruppen unterteilt. Neben Note-On und Note-Off gibt es die sogenannten Control-Change-Messages. Diese Befehle dienen in erster Linie dazu, die erzeugten Sounds, während diese gespielt werden, zu modulieren bzw. zu beeinflussen. Die bekannteste Control-Change-Message nennt sich Modulation und wird von allen MIDI-Keyboards in Form des Modulationwheels - einer Drehscheibe neben der Tastatur - verwirklicht. Neben Keyboards kommen auch andere MIDI-Instrumente zum Einsatz, wenn auch wesentlich seltener. Ein MIDI-Saxophon ist z.B. ein relativ einfach zu spielendes Blasinstrument, dessen Töne durch die Griffe des Saxophons bestimmt werden. Eine wichtige Modulationsart beim MIDI-Saxophon ist das Breath-Control, das die Stärke der vom Mund erzeugten Luftströmung misst. Beim Statusbyte einer Control-Change-Message sind zusätzlich das dritte und vierte Bit gesetzt. Das erste Databyte bestimmt den Control-Type, das zweite den tatsächlichen Wert des Control-Types. Einige Controller haben zusätzlich einen zweiten Controller mit dem sich der entsprechende Parameter feiner einstellen lässt (z.B. Modulation). Tabelle 6.2 zeigt einige wichtige Control-Change-Messages. Eine vollständige Auflistung aller Controller findet sich in Tabelle 6.3 (*Controller* im *Anhang*).

Statusbyte	Databyte 1	Databyte 2	Befehl
1011.0000	0000.0001	0000.1100	Modulation MSB (Channel:1, Amount:12)
1011.0001	0010.0001	0110.0000	Modulation LSB (Channel:2, Amount:96)
1011.0010	0000.0010	0010.1000	Breath MSB (Channel:3, Amount:40)
1011.0100	0000.0111	0111.1111	Volume MSB (Channel:5, Amount:127)

Tabelle 6.2

Control-Change-Messages beziehen sich immer auf einen bestimmten MIDI-Channel und können sich nicht auf einzelne Tasten beziehen. Daher wird der Polyphone-Aftertouch, der vor allem von Master-Keyboards erzeugt wird, nicht als Control-Change-Message codiert. Aftertouch liefert den Druck der Finger auf die Tasten, wenn diese gedrückt sind. Dabei unterscheidet man zwei Arten von Aftertouch, den Channel-Aftertouch (bzw. Channel-Pressure) und den Polyphonen-Aftertouch. Polyphon-Pressure liefert wie beschrieben für jede Taste einen individuellen Wert, während Channel-Pressure nur einen Wert pro MIDI-Channel liefert. Channel-Pressure ließe sich auch als Control-Change-Message realisieren, allerdings tritt dieses Event so oft auf, dass sich der Datenfluß deutlich reduzieren lässt, wenn diese Message nicht durch den Overhead des Control-Change-Statusbytes aufgeblasen wird. Somit haben beide Aftertouch-Versionen eigene Statusbytes, wobei Channel-Pressure von einem und Polyphoner-Aftertouch von zwei Databytes gefolgt wird. Tabelle 6.4 zeigt die unterschiedliche Codierung der beiden Pressure-Messages.

Statusbyte	Databyte 1	Databyte 2	Befehl
1101.0000	0100.0010		Channel-Pressure (Channel:1, Amount:66)
1101.1111	0110.0011		Channel-Pressure (Channel:16, Amount:99)
1010.0000	0011.1100	0100.0010	Polyphon-Pressure (Channel:1, Key:60, Amount:66)
1010.1111	0011.0000	0110.0011	Polyphon-Pressure (Channel:16, Key:48, Amount:99)

Tabelle 6.4

Pitch Bend

Pichtbend ist ebenfalls ein Parameter, den jedes MIDI-Keyboard in Form des Pitch-Wheels liefert. Dieses Drehrad befindet sich meist neben dem Modulation-Wheel. Mit Hilfe des Pitchbend lässt sich die Tonhöhe leicht nach oben bzw. unten ziehen. Da das menschliche Ohr in Bezug auf die Tonhöhe sehr sensibel reagiert, reicht es nicht aus diese Tonhöhenschwankung in einen Bereich zwischen 0 und 127 einzuteilen, da dabei Tonhöhensprünge ausmachbar sind. Daher wird der Wert für das Pitchbend mit zwei Databytes beschrieben, womit sich ein Wertebereich zwischen 0 und 16383 ergibt. Da man mit Pitchbend die Frequenz eines Tones sowohl erhöhen als auch reduzieren kann, ist der Wert für keine Änderung der Tonhöhe nicht 0 sondern 8192. Der Wert 0 steht für die maximale Reduzierung der Frequenz, 16383 für die maximale Erhöhung der Frequenz. Standardmäßig beträgt das Intervall für die Erhöhung bzw. Reduzierung eine große Sekunde also zwei Halbtonschritte. Dieser Wert lässt sich aber bei den meisten Synthesizern umstellen. Preiswerte Synthesizer machen leider oft vom großen Wertebereich des Pitchbends keinen Gebrauch und verwerten nur das höherwertige Databyte und ignorieren das andere, was einem Abschneiden der letzten sieben Bit entspricht. Gerade bei diesen Synthesizern macht es Sinn das Intervall für Pitchbend auf einen Halbtonschritt zu reduzieren um die störenden Tonhöhensprünge zu minimieren.

Program Change

Zu den Short-Messages zählen auch die Program-Change-Messages, mit denen das Instrument gewählt bzw. gewechselt werden kann. Fast alle Synthesizer und Sampler bieten das Konzept der Soundbanks an. Eine Soundbank ist eine Anzahl von unterschiedlichen Instrumenten die sich gleichzeitig im Speicher das Gerätes befinden. Jedes dieser Instrumente hat eine Nummer zwischen 0 und 127. Mit Hilfe von Program-Change-Befehlen lässt sich festlegen, welches dieser Instrumente aktiv sein soll. Dabei kann für jeden MIDI-Channel separat bestimmt werden, welches Instrument gespielt werden soll. Lange Zeit waren die Nummern für die verschiedenen Instrumente nicht

genormt, d.h. es gab keine festgelegte Ordnung welche Instrumentennummer welchem Instrument entsprach. Erst durch das GM (General MIDI) wurde die Nummerierung der Instrumente festgelegt. Tabelle 6.5 (*Programs*) zeigt die Nummerierung der verschiedenen Instrumente in GM.

Beim Neukauf eines Synthesizers befindet sich im Speicher des Geräts in den meisten Fällen eine GM-Soundbank. Diese Preset-Bank ist oft auch fester Bestandteil des Geräts, lässt sich also nicht verändern. Das ist vor allem dann der Fall, wenn der Speicher des Geräts groß genug ist, um mehrer Soundbänke aufnehmen zu können. Zwischen den einzelnen Soundbänken kann dann am Gerät (oder über Control-Change-Messages) hin- und hergeschalten werden. Die Program-Change-Messages sind aber nicht in der Lage eine dieser Soundbanks auszuwählen. Diese können nur innerhalb einer Soundbank festlegen, welches Instrument auf welchen Kanal hört. Eine Program-Change-Message besteht aus einem Statusbyte mit zusätzlich gesetztem zweiten Bit und einem Databyte welches die Nummer des Instruments festlegt.

Eine Besonderheit unter den Instrumenten stellen die Schlagwerke (Drums) dar. Diese Instrumente liefern nicht wie bei den anderen Instrumenten für jede Taste dasselbe Instrument mit anderer Tonhöhe, sondern können für jede Taste ein anderes Geräusch erzeugen. So kann z.B. die Taste C1 für die Bassdrum und die Taste D1 für die Snare verwendet werden. In diesem Fall bestimmt also die Tonhöhe des Note-On-Befehls welches Instrument tatsächlich erklingt. Auch diese Anordnung war lange Zeit nicht festgelegt und wurde durch GM normiert (Tabelle 6.6 *Drums*), wobei zusätzlich festgelegt wurde, dass das Schlagwerk immer auf MIDI-Kanal 10 anspricht.

Allen bisher besprochenen MIDI-Befehlen ist gemein, dass im Statusbyte der MIDI-Channel codiert ist. Es gibt aber auch Messages die sich nicht auf einen bestimmten Kanal beziehen. Bei diesen Messages sind die ersten vier Bit immer gesetzt und die Unterscheidung wird in den letzten vier Bit codiert, also dort, wo bisher der MIDI-Channel festgelegt wurde. Folgende Tabelle 6.7 zeigt zusammenfassend alle MIDI-Messages, deren Statusbyte sich auf einen bestimmten MIDI-Channel bezieht.

Statusbyte	Databyte 1	Databyte 2	Befehl
1000.xxxx (hex: 8x)	Tonhöhe	Velocity	Note Off
1001.xxxx (hex: 9x)	Tonhöhe	Velocity	Note On
1010.xxxx (hex: Ax)	Tonhöhe	Pressure	Polyphoner Aftertouch
1011.xxxx (hex: Bx)	Kontroll-Typ	Kontroll-Wert	Control Change
1100.xxxx (hex: Cx)	Instrument		Program Change
1101.xxxx (hex: Dx)	Pressure		Channel Aftertouch
1110.xxxx (hex: Ex)	Bit 7 - 1	Bit 14 - 8	Pitch Bend
1111.xxxx (hex: Fx)			Befehle ohne MIDI-Channel

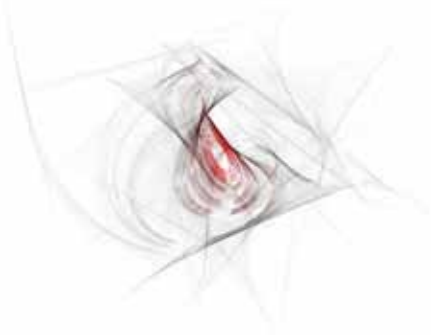
Tabelle 6.7

Erweiterte MIDI-Messages

Zu den kanallosen Messages gehört auch das, nur aus einem Byte bestehende, Active-Sensing, das ca. jede Sekunde einmal ausgeschickt wird um überprüfen zu können ob eine MIDI-Verbindung vorhanden ist oder nicht. Das ist besonders dann wichtig, wenn keine MIDI-Daten über die serielle Leitung laufen, z.B. bei längeren Pausen der MIDI-Instrumente. So kann auch in diesem Fall sichergestellt werden, dass die Leitung funktioniert und die Geräte korrekt verbunden sind. Eine System-Reset-Message veranlasst alle MIDI-Geräte zu einem Reset, die Tune-Request-Message zum Stimmen der Instrumente, was bei analogen Synthesizern wirklich immer wieder gemacht werden muss, da sich diese durch Luftdruck, Feuchtigkeit, Bertiebsdauer und vor allem Wärme leicht verstimmen. Die anderen Befehle ohne MIDI-Channel beziehen sich vor allem auf die zeitliche Synchronisation mit anderen Geräten und die Kommunikation mit Sequenzern.

Eine weiter Gruppe der MIDI-Messages sind die System-Exclusive- bzw. Sysex-Messages. Diese Befehle können eingesetzt werden um gerätespezifische Parameter zu ändern. Teilweise sind auch diese Befehle genormt. Sie beginnen immer mit dem Statusbyte `0xF0`, die ersten vier Bit sind gesetzt, die letzten vier nicht, und enden mit `0xF7`, die ersten fünf Bit sind gesetzt, die letzten drei nicht. Das zweite Byte bestimmt den Gerätehersteller (Tabelle 6.8 *Sysex*), das dritte oft das Gerät. Somit kann jedes Gerät erkennen ob der betreffende Sysex-Befehl verarbeitet werden soll, oder nicht. Die Länge einer Sysex-Message ist nicht definiert, kann aber am End-Of-Sysex-Byte (`0xF7`) erkannt werden. Empfängt ein Gerät eine Sysex-Meldung die nicht für dieses bestimmt ist, so werden alle Bytes inklusive der abschließenden End-Of-Sysex-Meldung ignoriert.

Die Gruppe der Meta-Messages bezieht sich nicht auf Geräte, sondern auf den Ablauf der MIDI-Daten, also dem Musikstück an sich. Diese Befehle sind für das MIDI-Dateiformat entwickelt worden. Ein Statusbyte, bei dem alle Bits gesetzt sind, leitet eine Meta-Message ein. Eigentlich ist das Byte 0xFF für die System-Reset-Message reserviert. Da diese Message in einem Fileformat aber keine Bedeutung hat, wurde sie zur Beschreibung einer Meta-Message herangezogen. Das zweite Byte definiert den Meta-Message-Type und wird von einer unterschiedlich großen Anzahl von Databytes gefolgt. Mit Meta-Befehlen lassen sich Tempoangaben und Cue-Points setzen, Sequence-Nummern angeben, aber auch Text und Copyrights einfügen.



Das MIDI-Format gehört zu den Formaten der Gruppe IFF (Interchange File Format) und wird durch Chunks aufgebaut. Es gibt im wesentlich zwei verschieden Arten der Datencodierung, wobei diese sich vor allem in der Art der Aufteilung der Tracks unterscheiden. Die Firma Microsoft hat zusätzlich einige Änderungen vorgenommen. Die verschiedenen Formattypen lassen sich aber leicht anhand des Fileheaders voneinander unterscheiden.

Abbildung 7.4

Interaktive MIDI-Verarbeitung

MIDI lässt sich sehr einfach mit dem Java Sound API realisieren. Dabei gibt es mehrere Möglichkeiten dem SoundSystem Töne zu entlocken. Zum einen kann ein Sequenzer aufgebaut werden, der sich in mehrere Sequenzen unterteilen lässt, die ihrerseits wieder aus Tracks bestehen. Den einzelnen Tracks können MIDI-Messages mit Zeitstempeln zugeordnet werden, die abgespielt werden, wenn der Sequenzer gestartet wird. Allerdings lässt sich damit nur umständlich Interaktivität realisieren. Soll nur der interne Synthesizer des Soundsystems angesprochen werden, können diesem MIDI-Messages an die freigegebenen Kanäle geschickt werden, die dieser dann abspielt. Am elegantesten lässt sich die MIDI-Verarbeitung über die Klasse Receiver verwirklichen. Dieser Klasse können (je nach Gerät mit oder ohne Zeitstempel) MIDI-Messages geschickt werden, die diese Klasse korrekt verarbeitet. Jedes MIDI-Device kann solch einen Receiver erzeugen. Hier wird der interne Synthesizer zur Erzeugung eines Receiver-Objektes herangezogen. Nicht alle Geräte unterstützen Zeitstempel, weswegen hier ein einfaches System implementiert wird, das sich um den korrekten zeitlichen Ablauf der MIDI-Messages kümmert.

Da das System interaktiv arbeiten soll, werden MIDI-Befehle immer sofort ausgeführt. Nur das Spielen von Tönen bereitet dabei Probleme, da je nach Länge einer Note, der Note-Off-Befehl verzögert wiedergegeben werden muss. Die Methode playNote startet einen Ton mit einem Note-On-Befehl und speichert das entsprechende Note-Off mit einem Zeitstempel in einem Array. Dieser Array wird immer wieder auf Events untersucht die bereits abgelaufen sind. Wird ein Note-Off gefunden, dessen Zeitstempel bereits veraltet ist, wird dieses ausgeführt und aus dem Array gelöscht. Code 6.1 (*Midisystem*) beschreibt die Klasse Midi, die interaktiv Noten spielen kann.

Obwohl MIDI im Musikbereich nach wie vor stark präsent ist, ist es technisch längst veraltet. Dank moderner Prozessoren ist es inzwischen möglich, Sound in Echtzeit zu generieren, zu verarbeiten und abzumischen. Imagedriven Music kann also auch ohne MIDI realisiert werden, teilweise viel bequemer, da sich die Auswirkungen der Parameter auf den Sound individueller und vor allem direkt durch Software realisieren lassen. Da in einem Szenario mit MIDI meist mehrere Geräte mitspielen, die alle fehleranfällig sind und vor jedem Einsatz in den gewünschten Zustand (Preset) gebracht werden müssen, entsteht im Einsatz mit MIDI oft ein gewaltiger Zeitaufwand, der bei reinen Softwarelösungen nicht oder nur sehr selten auftreten kann. Durch den Verzicht auf MIDI verliert man andererseits aber auch Möglichkeiten. So können z.B. analoge Synthesizer nicht mehr angesteuert werden, deren Sound mit digitalen Mitteln einfach nicht herstellbar ist.

Externe Hardwarelösungen bieten außerdem den enormen Vorteil nicht auf das beengende Mensch-Computer-Interface - Maus, Tastatur und Monitor - angewiesen zu sein. Künstlerische Arbeit lässt sich mit diesen Geräten eigentlich nicht bewerkstelligen. Die Entwicklung geeigneter Interfaces für Künstler spielt eine wesentliche Rolle bei der Akzeptanz des Computers als künstlerisches Werkzeug.

Interaktive Soundsynthese

Um interaktiv in Echtzeit Klänge abspielen zu können bedarf es einer Anpassung der Klasse SoundSystem an die neuen Herausforderungen. Zum einen müssen öffentliche Methoden existieren die das Abspielen von Klängen einleiten (triggern). Da es vorkommen kann, dass mehrere Sounds zur gleichen Zeit getriggert werden, muss auch das Abmischen von Sounds implementiert werden. Code 6.2 (*Soundsystem*) zeigt die dafür nötigen Veränderungen auf, die mit Hilfe der ALab Klasse *APart* verwirklicht wird, für die festgestellt werden kann, ob bereits alle Samples gespielt wurden. Die Implementierung ähnelt jener aus der Klasse *Midisystem* (Code 6.1).

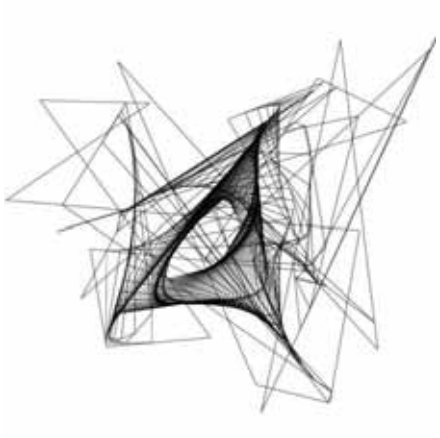
Maschinengrenzen (1992)

Die multimediale Installation "Maschinengrenzen" aus dem Jahr 1992 symbolisiert die Enge einer aus Regeln aufgebauten Welt, die Monotonie der Logik, des Verstandes. Immer wieder treten die selben Phrasen auf, zwar zeitlich voneinander verschoben, in neuen Variationen, wie der verzweifelte Versuch des Geistes aus seinen Schranken auszubrechen, doch jedes Bemühen dem Unabänderlichen durch bloßes Einnehmen anderer Perspektiven zu begegnen, scheitert, muss scheitern.

Abbildung 6.2

Code 6.3 (*Maschinengrenzen*) erzählt die Geschichte zweier Punkte, die sich in entgegengesetzter Richtung auf eine Reise begeben. Gelangen diese an die Grenzen des Bildes, drehen sie sofort um und versuchen in die andere Richtung zu fliehen, um dort nach einem Ende der Grenze zu suchen. Aber auch diese Richtung bringt nicht die erhoffte Freiheit. Deshalb biegen sie von ihrem Weg ab, probieren ob sich so ein Ausweg finden lässt. Und wieder stoßen sie an eine Grenze. Sie machen wieder kehrt und versuchen noch einmal den ersten Weg, wobei sie diesmal notfalls auch einen anderen Weg einschlagen würden. So wechseln sich beide Punkte immer und immer wieder auf ihrer Suche ab, doch keiner findet einen Ausweg. Wird eine Mauer erreicht erklingt ein lauter Ton der Ernüchterung. Der Weg zurück wird von leisen Klängen der Enttäuschung oder manchmal auch der Hoffnung begleitet. So schlagen sie immer weitere Spuren, die sie vergeblich löschen, um sich zumindest einbilden zu können, es hier noch nicht versucht zu haben.

Die Ästhetik ist bewusst sehr einfach gehalten, zeigt den unmittelbaren Zugang zur Grafik und hilft so den Algorithmus wahrnehmen zu können. Durch das einfache Setzen der Pixel und den Abständen zwischen den Punkten entsteht die Assoziation mit einer Maschine, heute wohl mit einer sehr alten Maschine. Abbildung 6.2 zeigt einen Ausschnitt aus der Animation nach etwa einer Stunde Laufzeit.



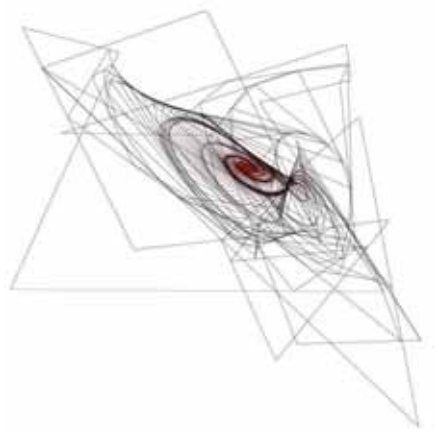


Abbildung 7.3

Doppelpufferung

In Java ist es möglich direkt in den Bildschirmspeicher zu schreiben, was aber zu störenden Fehlern beim Bildaufbau führen kann, da das Zeichnen nicht synchron zur Bildwiederholfrequenz des Monitors passiert. Das führt zu dargestellten Bildern, die nicht komplett fertiggezeichnet wurden. Wahrnehmbar werden solche Fehler durch ein mehr oder weniger starkes Flimmern bzw. Streifen die durch das Bild wandern. Diese Effekte kann man umgehen, indem das Bild zuerst im Speicher aufbereitet wird und erst dann in den Bildspeicher geschrieben wird. Diese Vorgangsweise nennt sich Doppelpufferung, wobei auch mehrfache Pufferung möglich ist.

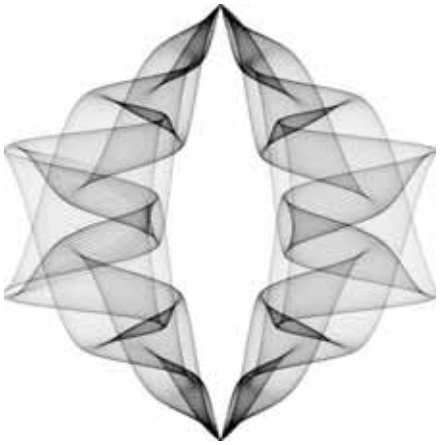
Die Geschwindigkeit einer Animation lässt sich in Java mit der Methode *sleep* der Klasse *Thread* steuern. Jede Animation muss daher einen Thread ausführen, was hier durch die Implementierung des Interfaces *Runnable* geschieht. Die zu diesem Interface gehörende Methode *run* wird ausgeführt sobald eine Instanz von Runnable durch den Methodenaufruf *start* gestartet wird. In dieser Methode muss also auch der Thread gestartet werden. Die Methode run muss sich um den Bildaufbau im Hintergrund kümmern, das fertige Bild in den Bildspeicher schreiben und danach den Thread für eine gewisse Zeit lang pausieren lassen um dann mit dem Bildaufbau des nächsten Frames zu beginnen. Code 6.4 (*Doppelpufferung*) zeigt einen einfachen Aufbau einer Klasse die in der Lage ist Animationen mit Hilfe der Doppelpufferung zu erzeugen. Der Doppelpuffer wird hier vom Applet erzeugt, kann aber auch durch eine MemoryImageSource erzeugt werden.

Interaktivität meint in der Kunst eigentlich die Reaktion eines Systems auf seine Umwelt und nicht wie hier die Steuerung eines Prozesses durch einen anderen. Technisch gesehen sind aber beide Formen ident, da auch die Aufnahme und Verwertung von Ereignissen aus der Umwelt durch einen Prozess geschehen muss, der die Ergebnisse einsetzt um einen anderen Prozess zu steuern. Ein wesentlicher Bestandteil eines interaktiven Systems ist die Parameterverwertung, die jene Werte die im Steuerungsprozess gewonnen werden in Parameter für den interaktiven Teil des Systems umwandeln muss. Verallgemeinernd kann gesagt werden, dass bei Interaktivität immer eine Instanz dafür sorgen muss, aus einer gewissen Anzahl von Eingangswerten eine andere oder gleiche Anzahl von Ausgangswerten zu erzeugen. Diese Auswertung von Eingabedaten in Ausgabedaten kann oft als Matrix dargestellt werden. Tabelle 6.9 zeigt die Parameterverwertung aus Maschinengrenzen für einen der beiden Punkte mit den Koordinaten x und y.

Input / Output	Tonhöhe	Anschlagstärke	Spieldauer
Punkt ist rot	(480 - y) / 16	50	100 ms
Umkehr	(480 - y) / 8	100	(1300 - 2x) ms
Rotation	(480 - y) / 4	100	(1300 - 2x) ms

Tabelle 6.9

Aus dieser Matrix kann abgelesen werden, dass die Tonhöhe von der Y-Koordinate des Punktes abhängt und zwar reziprok zu deren Größe. Ein Punkt der im Bild höher oben liegt erzeugt auch einen höheren Ton. Die X-Koordinate dagegen steuert die Länge des Tons, wobei Punkte die weiter rechts im Bild liegen kürzere Töne ergeben als solche die sich weiter links im Bild befinden. Die Anschlagstärke hängt dagegen nur von der Art des Ereignisses ab. Ist die Farbe des vom Punkt erreichte Pixels rot, wird ein Ton mit schwacher Velocity erzeugt. Ansonsten werden nur Töne bei Richtungswechseln erzeugt, wobei sich die Tonhöhe je nach Form der Richtungsänderung unterschiedlich ergibt.



Konstruktionsprinzipien

Alle hier vorgestellten Konstruktionsmöglichkeiten werden grafisch visualisiert, wobei erwähnt werden soll, dass sich diese auch beliebig anderwertig verwenden lassen. Prinzipiell müssen diese Prinzipien auch meist kombiniert eingesetzt werden, um wirklich komplexe und ausgereifte Ergebnisse zu erhalten. Gerade die Arbeit in der Computerkunst setzt enorme kreative Leistung voraus und oft führt nur der experimentelle Einsatz von Techniken zu guten Werken, denen ihre technisch-mathematische Herkunft nicht sofort anzumerken ist.

Schwarz-Weiß-Grafik

Um Ergebnisse mit Pixelästhetik zu erreichen, müssen folgende Einschränkungen beim Bildaufbau eingehalten werden: Es dürfen nur zwei Farben ohne Zwischenstufen verwendet werden und die Auflösung sollte relativ gering sein, wobei zweiteres durch Verdoppelung der Pixelgröße einfach zu simulieren ist (Code 7.1). Die verwendeten Farben können Schwarz und Weiß sein, es sind aber auch andere Kombinationen denkbar. Häufig war die Kombination Dunkelgrün und Schwarz oder später auch Blau und Gelb.

```
int width1 = width - 1;                  // width1 is width of image - 1
public void set(int id, int c) {
     id<<= 1;                            // position corrected
     dat[id] = c; id++;                  // next pixel id
     dat[id] = c; id+= width1;           // below pixel id
     dat[id] = c; id++;                  // next pixel id
     dat[id] = c;
}
```

Code 7.1

Zwar lassen sich auch mit den Farben Schwarz und Weiß Graustufen simulieren, indem diese in mehreren Pixeln zu Mustern zusammengefaßt werden, doch war der Einsatz solcher Muster fragwürdig, da sich die Auflösung dadurch nochmals verringerte. Muster wurden stattdessen zur Erzeugung von Strukturen eingesetzt. Farbübergänge lassen sich teilweise auch mit zwei Farben darstellen, wobei die verwendeten Ditheralgorithmen versuchen die zwei Farben, durch geschickte - möglichst zufällig wirkende - Anordnung, in nicht mehr erkennbaren Mustern einzusetzen. Abbildung 7.1 zeigt die Grautöne die mit verschieden großen Feldern erstellbar sind im Vergleich zu sogenannten Dithering-Algorithmen.

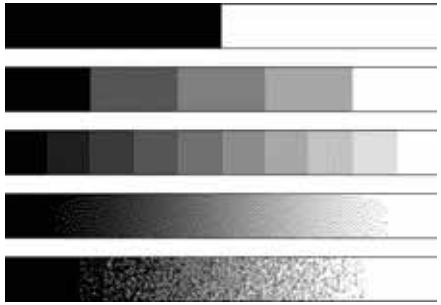


Abbildung 7.1
1x1, 2x2, 3x3, Diffusion, Störungsfilter

Liniengrafik

Grafiken die nur aus Linien bestanden hatten den Vorteil, dass diese gegebenenfalls geplottet werden konnten und danach in guter Qualität, ohne störende Pixelabstufungen, zur Verfügung standen. Diese Bilder konnten auch in Gallerien ausgestellt werden. Der Aufbau dieser Grafiken basierte auf unterschiedlichen Algorithmen, wovon einer hier vorgestellt werden soll.

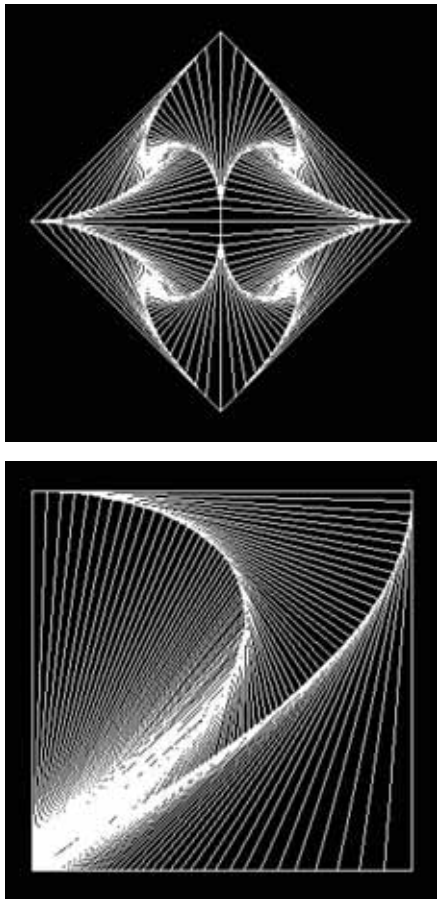


Abbildung 7.2

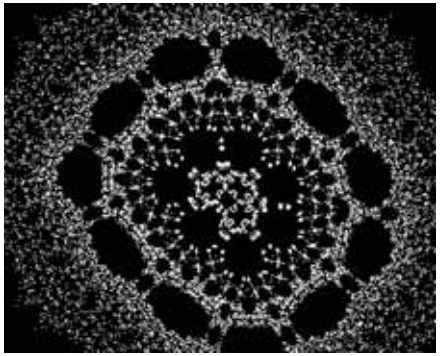


Abbildung 7.5

Ein Algorithmus der ornamentartige Grafiken in Form einer Spirale erzeugt ist in Code 7.2 (*Spirale*) wiedergegeben. Mit der Variablen n läßt sich die Anzahl der Ecken festlegen, m bestimmt die Iterationsschritte. Für jede Kante läßt sich ein Verkürzungsfaktor f festlegen, der mit der Koordinatendifferenz der jeweiligen Kantenpunkte (dx und dy) multipliziert wird. Danach werden die Ergebnisse zum ersten Kantenpunkt dazuaddiert, wodurch dieser langsam in Richtung des zweiten Punktes wandert, welcher sich wiederum zum dritten bewegt, usw. Abbildung 7.2 zeigt einige einfache Resultate dieses Algorithmus. [15]

Künstlerisch interessanter wird dieser Algorithmus, wenn die Anzahl der Ecken n und die Faktoren f größer werden. Dabei entstehen architektonisch wirkende Gebilde, die teilweise chaotisch und ungeordnet aber doch konstruiert und im Zentrum geregelt wirken. Abbildung 7.3 (siehe oben) zeigt Resultate für n > 10 und relativ große Faktoren f. Bei der letzten Variante wird nach jeder Iteration der Farbwert für Rot inkrementiert. In diesem Fall wurde die Pixelverdoppelung deaktiviert und für das Zeichnen der Linie jene Methode mit Antialiasing verwendet.

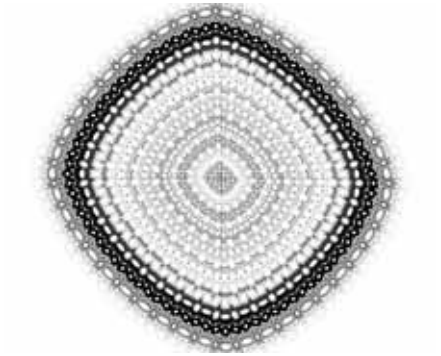


Abbildung 7.6

Sehr oft ist es auch möglich solche einfachen Algorithmen für zeitgemäße Grafik einzusetzen. Dabei sind die unregelmäßigen Linien aber störend. Aus diesem Grund werden alle Linien nur sehr schwach gezeichnet, dafür wird die Iterationstiefe m stark erhöht, während die Faktoren f Werte nahe Null zugewiesen bekommen. Abbildung 7.4 (siehe weiter oben) zeigt solche Ergebnisse, wobei auch hier wieder mit Farbe experimentiert wurde.

Um diesen Algorithmus in einer Animation verwenden zu können muss dieser zuvor optimiert werden. Zum einen ist es nicht nötig, dass alle Punkte zuerst berechnet und erst dann gezeichnet werden, zum anderen können alle Variablen vom Typ double durch den Typen int ersetzt werden, wenn die Simulation von Fließkommazahlen eingesetzt wird. Als Zeichenmethode für Linien wird wieder auf die einfache Methode ohne Antialiasing zurückgegriffen, die Methode zum Setzen der Punkte muss völlig neu geschrieben werden. Code 7.3 (*Spirale*) zeigt den optimierten Quellcode.

Die Animation basiert auf einer Verschiebung der Startpunkte und einer schrittweisen Erhöhung der Faktoren f. Die Startpunkte müssen dabei immer innerhalb des Bildes bleiben, die Verkürzungsfaktoren innerhalb festgesetzter Grenzen. Da nach jeder Iteration die Punktdaten durch neue überschrieben werden, müssen diese Werte zweimal in unterschiedlichen Arrays abgespeichert werden. Für jedes Frame müssen die aktuellen Startpunkte in das Feld für die Iterationen kopiert werden.

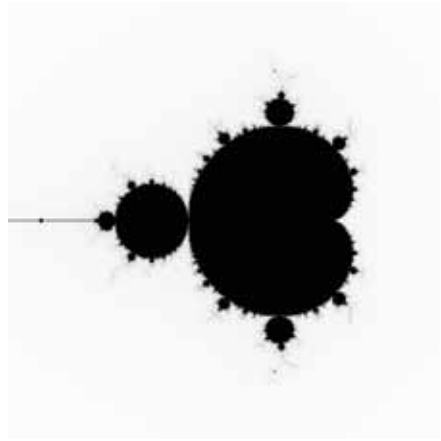
$$x_0, y_0$$
$$x_{n+1} = f(x_n, y_n)$$
$$y_{n+1} = g(x_n, y_n)$$

Formel 7.1

Iterative Systeme

Iterationsverfahren sind rückgekoppelte Prozesse die mit Hilfe mathematischer Formeln Werte berechnen, welche als Ausgansdaten für die nächste Berechnung herangezogen werden. Dabei gibt es sowohl konvergierende Verfahren, bei denen nach einer bestimmten Anzahl von Iterationen keine Veränderung an den Ergebniswerten mehr stattfindet, als

auch divergierende Verfahren, bei denen die Werte gegen Unendlich laufen. Es gibt unterschiedliche Varianten der Sichtbarmachung solcher Systeme, die einfachste Möglichkeit ist die direkt Verwendung der berechneten Werte als Koordinaten, wobei eine Funktion die X-Koordinate und eine andere die Y-Koordinate liefert. Beide sind Funktionen auf diese beiden Koordinaten, woraus sich folgende Koordinatenpaare ergeben, wobei x_0 und y_0 Startwerte für das System sind (Formel 7.1). Durch fortgesetzte Iteration entstehen so aus den Anfangswerten x_0, y_0 die Werte für x_1, y_1, x_2, y_2, usw.

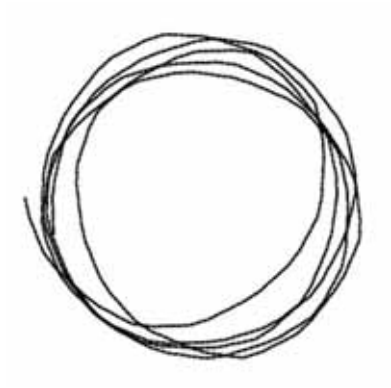


Die Schwierigkeit an solchen Verfahren liegt im Auffinden geeigneter Formelsystem und an der Wahl der Parameter (die innerhalb der Funktionen Verwendung finden), wobei geringe Änderungen an den Parametern oft völlig andere Bilder erzeugen. Iterative Systeme zeigen also chaotisches Verhalten, minimale Parameteränderungen führen zu unvorhersagbaren Veränderungen am Ergebnis. Berühmtheit hat dieses Konstruktionsprinzip durch die Mandelbrotmenge erreicht. Hier soll auf ein Formelsystem eingegangen werden, das Grafiken wie in Abbildung 7.5 erzeugt. [15]

```
int width, height;                                          // width and height of image

public double sgn(double v) {                               // signum function
    if(v < 0) return -1; else if(v > 0) return 1;
    return 0;
}

public void draw() {
    double a = 3, b = 1, c = 2;                             // parameter a, b, c
    double m = 7, x = 0, y = 0;                             // zoomfactor, x(0), y(0)
    double f,g;                                             // functions f, g
    for(int i = 0; i < 20000; i++) {                        // 20000 iterations

        f = y - sgn(x) * Math.sqrt(Math.abs(b * x + c));    // f calculated
        g = a - x;                                          // g calculated

        x = f;                                              // x(n+1) = f
        y = g;                                              // y(n+1) = g

        int xx = ((int)(x * m)) + 160;                      // trnsformed to
        int yy = ((int)(y * m)) + 128;                      // device coordinates

        set(yy * width + xx, 0x00ffffff);                   // set pixel
    }
    mis.newPixels();
}
```

Code 7.4

Abbildung 7.7



Code 7.4 zeigt die entsprechende Implementierung. Durch Deaktivierung der Pixelverdoppelung und gezielterem Setzen der einzelnen Pixel mit Graustufen ergeben sich Grafiken wie in Abbildung 7.6. Mit iterativen Systemen lassen sich keine Animationen erstellen, deren Aussehen sich langsam verändert. Werden Parameter variiert ändert sich das Bild jedesmal sehr stark, es bleiben aber grundlegende Strukturen erhalten. Die Stellen die in einem Bild hell sind, können im nächsten Bild dunkel sein und umgekehrt. In einer Animation entsteht daher ein sehr dynamisches Helligkeitsverhalten, was sich optisch als starkes Flimmern bemerkbar macht, andererseits eine Ästhetik erzeugt, die dem schnellen Schnitt von Video und Film ähnelt.

Das Apfelmännchen

1980 erhielt der amerikanische Mathematiker B. Mandelbrot bei Untersuchungen von Iterationsverfahren in der komplexen Ebene erste schemenhafte grafische Darstellungen einer Punktmenge, die heute unter dem Namen Apfelmännchen berühmt ist. Ausgangspunkt ist die Rückkoppelungsgleichung (Formel 7.2), wobei z und c komplexe Zahlen sind. Die Mandelbrotmenge ist die Menge jener Werte von c, für die diese Gleichung nicht divergiert. Jedes dieser Elemente kann als Punkt in der komplexen Ebene interpretiert werden, wodurch sich das Bild des Apfelmännchens ergibt.

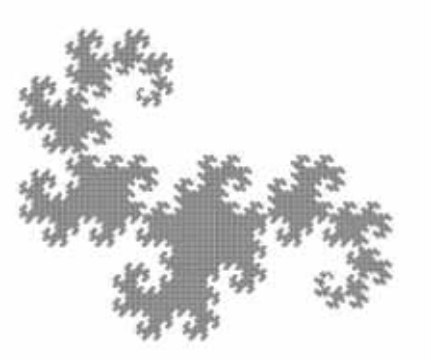


Abbildung 7.9

Um in endlicher Zeit feststellen zu können, ob die Funktion divergiert oder nicht, muss die Anzahl der Iterationen beschränkt werden, was zu einer nicht korrekten Ergebnismenge führt. Wächst die Funktion innerhalb dieser Zyklen über einen Grenzwert max hinaus, wird angenommen, dass die Funktion divergiert. Je höher dieser Grenzwert und die Anzahl an Iterationen ist, desto genauer ist das Ergebnis. Die divergierenden Punkte können in Geschwindigkeitsklassen eingeteilt und unterschiedlich eingefärbt werden, was dem Ergebnisbild einen zusätzlichen ästhetischen Reiz verleiht. Werden dazu nur die Farben Schwarz und Weiß verwendet, tritt das Apfelmännschen selbst in den Hintergrund und die Restmenge gewinnt an Attraktivität. Das Resultat sind unheimlich wirkende Strukturen, die einen starken räumlichen Eindruck hinterlassen. Abbildung 7.7 zeigt das Appfelmännchen und einen vergrößerten Ausschnitt daraus, erstellt mit Hilfe des leicht optimierten Codes 7.5 (*Mandelbrot*), wobei bei der ersten Darstellung Graustufen entsprechend der Iterationstiefe verwendet wurden. [20]

Fraktale Systeme

Das auffälligste Merkmal eines Fraktals ist die Selbstähnlichkeit, also die Tatsache, dass die Form des erzeugten Objekts aus Teilen aufgebaut wird, die selbst wiederum die Form des Objekts haben. Mathematisch betrachtet haben Fraktale eine Dimension, die keiner ganzen Zahl entspricht. So können z.B. mit der Hilbert- und Sierpinskikurve

Linien beschrieben werden, die eine Ebene vollständig ausfüllen, wenn diese Kurven unendlich weit gezeichnet werden. Da es andererseits nicht möglich sein sollte mit einer eindimensionalen Struktur eine zweidimensionale zu beschreiben, wird diesen Linien eine Dimension zwischen eins und zwei zugeordnet, wobei sich diese auch berechnen läßt.

Abbildung 7.8

Die zuvor besprochene Mandelbrotmenge gehört zu den dynamisch erzeugten Fraktalen, wobei die Selbstähnlichkeit nicht programmiert wird, sondern aufgrund der dahinter steckenden Mathematik entsteht. Durch rekursive Aufrufe von strukturbildenden Methoden, kann Selbstähnlichkeit aber auch direkt codiert werden. Als Beispiel sei hier die Kochkurve genannt. Diese besteht im Prinzip aus vier Linien, wobei die zweite um 60 Grad nach oben und die dritte um 60 Grad nach unten gedreht ist. Wird nun zum Zeichnen der Linien nicht eine Gerade, sondern wieder eine Kochkurve verwendet, die ihrerseits wieder aus Kochkurven aufgebaut ist, entsteht die sogenannte Schneeflockenkurve, die in Abbildung 7.8 in den Rekursionstiefen 1, 2, 3 und 4 abgebildet ist.

$$z_{n+1} = z_n^2 + c$$

Formel 7.2

Für derartige Fraktale eignet sich die Turtle-Geometrie bestens, welche von Abelson und diSessa beschrieben wird und das Programm Turtle von Seymour Papert benutzt. Turtle ist eine einfache Schnittstelle um Grafiken zu erstellen, die im wesentlichen aus zwei Befehlen besteht, womit ein virtueller Cursor bewegt werden kann, der dabei eine Spur am Bildschirm hinterläßt. Der Befehl *forward(length)* bewegt den Cursor um die Länge length vorwärts, mit *turn(alpha)* läßt sich der Cursor um alpha Grad drehen. Code 7.6 zeigt die Codierung der Kochkurve mit Rekursionstiefe 1.

```
forward(length);                    turn(  60);
forward(length);                    turn(-120);
forward(length);                    turn(  60);
forward(length);
```

Code 7.6

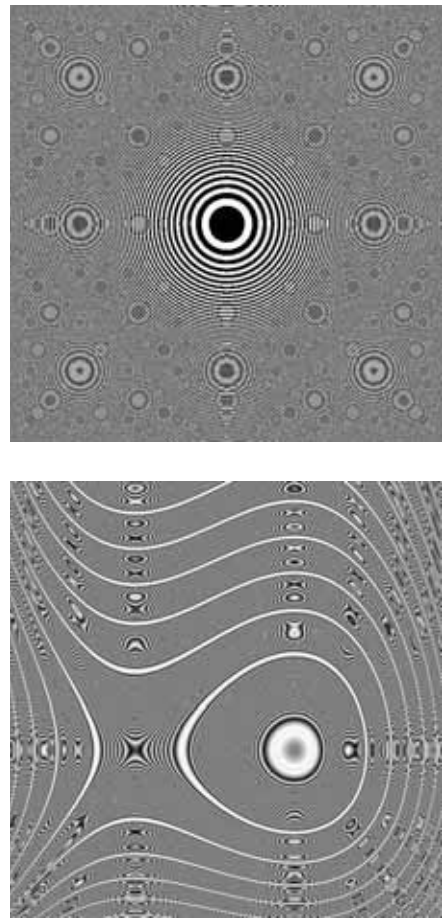
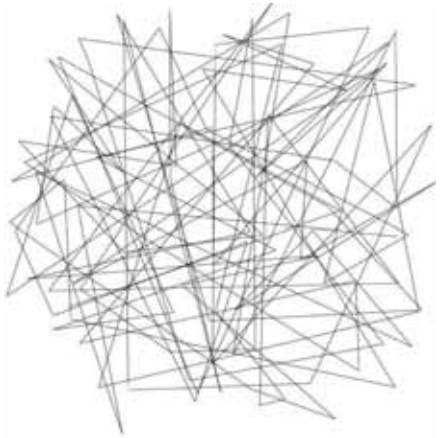


Abbildung 7.10

Für eine höhere Rekursionstiefe muss die Methode *koch(depth)* implementiert werden, die an Stelle der forward-Befehle, sich selbst mit dekremetierter depth aufruft. Hat depth den Wert Null wird forward(length) ausgeführt. Die Implementierung der beiden hier benutzten Turtle-Befehle und die Methode zum Zeichnen einer Schneeflockenkurve wird in Code 7.7 (*Turtle*) aufgezeigt, wobei die rekursive Methode *turtle* benannt ist und eine Erweiterung beinhaltet, die es ermöglicht, auch komplexere Fraktale zu erstellen. So kann hier der Drehwinkel rekursiv weitergegeben werden, was für die Erstellung von C-Kurven und Drachenkurven notwendig ist. Abbildung 7.9 zeigt verschiedene Fraktale, die sich mit diesem System erstellen lassen, wobei die dazu nötigen Parameter in Code 7.7 (*Turtle*) implementiert ist.

Obwohl Fraktale einen ausgesprochen wirkungsvollen ästhetischen Reiz haben und trotz ihrer Schönheit können diese für sich alleine nie ein Kunstwerk sein. Denkbar ist der Einsatz von Fraktalen in der Kunst aber dennoch, als kleiner



Bestandteil eines größeren Ganzen oder als visuelles Bildelement. Besonders hilfreich sind Fraktale bei der Erzeugung von Daten z.B. als Ersatz für Zufallsgeneratoren und Markovketten, da die gewonnenen Werte nicht völlig strukturlos sind und immer wieder dasselbe Ergebnis liefern. Mit Hilfe der Turtle-Geometrie lassen sich auch Kurven erzeugen, die eine starke Ähnlichkeit zu freihändig gezeichneten Kreisen aufweisen. Diese können für die Gewinnung von Koordinatenwerten benutzt werden (Siehe Code 7.7 *Turtle*). [21]

Modulo-Arithmetik

Die Modulo-Operation liefert den Rest, der bei einer ganzzahligen Division entsteht. Dabei kann das Ergebnis des Terms 'r modulo n' nie grösser oder gleich n sein. Wählt man n = 2 so besteht die Lösungsmenge für alle r maximal aus den Elementen 0 und 1. Diese Tatsache kann genutzt werden um monochrome Grafiken zu erstellen. Anstelle von r kann eine Funktion f auf x und y definiert werden, die für jedes Pixel mit den Koordinaten (x,y) (oder skalierten Werten für jedes Pixel) einen Wert ergibt, der dann durch die Modulo-Operation auf die Menge {0, 1} bzw. {Scharz, Weiß} abgebildet wird.

Das Ergebnisbild läßt sich kaum vorhersagen, da die Entscheidung, ob ein Pixel gesetzt wird oder nicht, von dem Teil der Zahl abhängt, der dessen Grössenordnung am wenigsten mitbestimmt, nämlich ausschließlich vom letzten Bit. Dass aber auch dieses Bit gewissen Ordnungen und Strukturen folgt wird offensichtlich, wenn man die Ergebnisse sieht, die dieses Vorgehen erbringt. Abbildung 7.10 zeigt ein monochromes Ergebnis und eine Variante die durch Modulo-Operationen mit n = 8 erstellt wurde, wodurch 256 verschiedene Farben in der Ergebnismenge und im Bild aufscheinen. Code 7.8 (*Modulo*) kann als Ausgangspunkt für die Erstellung solcher Grafiken dienen, wobei hier - der Übersichtlichkeit halber - keinerlei Optimierungen vorgenommen wurden. Die Modulo-Operation kann durch eine Überprüfung des letzten Bits auf 1 eingespart werden (`f&1==1`), der Typ double kann wieder durch int simuliert werden und die Multiplikation `y*y` muss nur einmal pro Pixelspalte berechnet werden.

Abbildung 7.11

Lissajousche Figuren

Eine Möglichkeit elektrische Signale zu visualisieren, die es schon vor der Erfindung der Monitore gab, bieten Oszillographen. Dabei werden zwei Signale kombiniert, wobei eines normalerweise einer Sägezahnwelle entspricht, das andere frei wählbar an das Gerät angeschlossen werden kann. Der Kathodenstrahl richtet sich nach beiden Signalen aus, wobei das gleichbleibende Signal entlang der X-Achse und das Eingangssignal entlang der Y-Achse interpretiert wird. Somit entsteht der Eindruck einer zeitlichen Aufnahme der Amplitude des Eingangssignals entlang der X-Achse. Lissajousche Figuren entstehen, wenn beide Signale durch Sinuswellen ersetzt werden, wobei diese sich

in ihrer Frequenz unterscheiden können. Je nach Teilungsverhältnis dieser beiden Frequenzen entstehen unterschiedliche Figuren. Bei der Simulation eines Oszillographen am Computer fällt sofort dessen Exaktheit auf. Da beide Signale berechnet werden, kommt es zu keinen Abweichungen in Bezug auf das Teilungsverhältnis. Die Signale selbst enthalten kein Störrauschen und so wirken die Figuren leblos und verharren völlig starr an der errechneten Position.

Um einen realistischeren Eindruck eines Oszillographen zu erreichen, kann zum einen das Teilungsverhälnis leicht verschoben werden, indem ein Frequenzwert in einer hinteren Nachkommastelle verändert wird. Die Signale selbst können durch Frequenz- oder Amplitudenmodulation mit einem anderen Sinussignal so irritiert werden, sodass sich die einzelnen Figuren in jeder Periode leicht voneinander unterscheiden. Um das Leuchten des Kathodenstrahls zu simulieren, werden auch die Nachbarpixel eines gesetzten Punktes leicht aufgehellt (Code 7.9 *Lissajousch*).

Zeichenautomaten

Ein Zeichenautomat soll in der Lage sein ohne menschliches Zutun bzw. teilweise selbständig Zeichnungen oder Bilder zu erzeugen. Als Grundlage dient solchen Automaten der Zufall der aber in solche Bahnen gelenkt wird, dass die Ergebnisse einer Zeichnung oder einem Zeichenstil ähneln. Bekanntestes Beispiel dafür ist das Programm AARON von H. Cohen. Hier soll kein fertiger Automat vorgestellt werden, sondern aufgezeigt werden, wie man vom Zufall ausgehend zu künstlerisch ansprechender Linienführung kommen kann. Der einfachste Algorithmus verbindet zufällig bestimmte Positionen im Bild durch Linien (Abbildung 7.11).

Schränkt man den Zufall so weit ein, dass dieser die Linienendpunkte nicht mehr direkt bestimmen kann, sondern lediglich die zwei Werte *dx* und *dy* für die nächste Linie, wobei sich diese Werte innerhalb der Grenzen *dx_min, dx_max* und *dy_min, dy_max* befinden müssen, so wird das Ergebnis von diesen Parametern beeinflusst, der Automat arbeitet in Abhängigkeit von vier Werten (Abbildung 7.11). Komplexere Zeichenstile ergeben sich, wenn die beiden Werte dx und dy nicht direkt bestimmt werden, sondern sich auf einen Winkel *a* und eine Linienlänge *l* beziehen. Diese werden wiederum durch zufällig erzeugte Werte - *da* für den Winkel und *dl* für die Länge - verändert, wobei sich diese innerhalb der Schranken *da_min, da_max* und *dl_min, dl_max* befinden müssen. Wird zusätzlich festgelegt, dass sich die Länge im Bereich zwischen *l_min und l_max* befinden muss, entsteht ein Algorithmus der von sechs Parametern abhängig ist. Abbildung 7.11 zeigt Ergebnisse für einen solchen Algorithmus, wobei hier zusätzlich ein siebenter Parameter die Wahrscheinlichkeit angibt, mit der sich die Richtung plötzlich um 180° ändert. Dieser Wert ist notwendig um zu verhindern, dass Linien mit kleinem da aus dem Bild laufen müssen. Ausserdem kann damit auch Krizzeln simuliert werden.

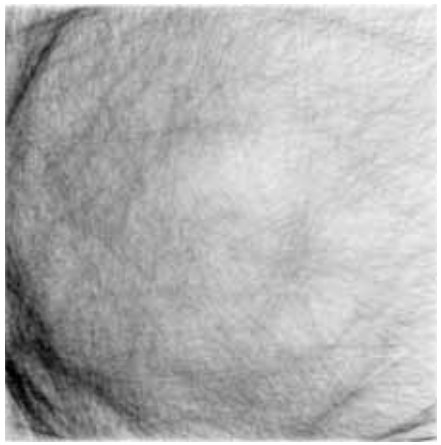


Abbildung 7.12

Zeichenautomaten müssen nicht unbedingt deckend malen. Abbildung 7.12 zeigt Beispiele die durch transparent gezogene Linien entstanden sind. Wählt man die Liniendicke relativ breit kann der Algorithmus auch Schraffuren erzeugen. Dabei wird die Wahrscheinlichkeit für die Richtungsänderung hoch angesetzt, die Winkeländerung dagegen sehr eng um Null herum. Die einzelnen Linien werden kaum deckend gezeichnet. Ein Beispiel für eine solche Schraffuren ist in Abbildung 7.12 zu sehen. Alle hier angeführeten Abbildungen wurden mit Code 7.10 (*Painter*) erstellt.

Vergleichende Bildsynthese

Bildsynthesealgorithmen sind in der Lage Bilder automatisch bzw. halbautomatisch zu generieren. Bei der vergleichenden Bildsynthese exisitiert eine Vorlage, anhand derer das Bild erzeugt wird. Dabei wird das Original allerdings nicht einfach bearbeitet oder verändert, sondern es dient lediglich zu Vergleichszwecken um feststellen zu können wie sehr das automatisch erstellte Bild sein Ziel bereits erreicht hat.

Für dieses Verfahren sind drei Bilder notwendig. Zum einen die Vorlage, zum anderen das Ergebnisbild und ein Testbild in dem eine grafische Veränderung zuerst ausprobiert wird. Wirkt sich die Veränderung positiv auf die Ähnlichkeit zum Original aus, wird die Änderung für das Ergebnisbild übernommen, tritt eine Verschlechterung ein, wird die Änderung im Testbild zurückgenommen und durch einen anderen Versuch ersetzt. Dadurch ist sichergestellt, dass das Ergebnisbild dem Vorbild immer ähnlicher wird. Nach einer zuvor bestimmten Anzahl von Änderungen wird der Algorithmus abgebrochen und das Bild kann als fertig bezeichnet werden.

Abbildung 7.13

Die Art der Bildveränderung ist dabei beliebig. Es können Zeichenautomaten zum Einsatz kommen, genauso aber auch Textelemente oder Buchstaben ins Bild gezeichnet werden. Mit Hilfe von Fotos können so Collagen erstellt werden, die von weitem betrachtet ein Bild darstellen. Diese Technik ist allerdings nur sehr bedingt für den Echtzeiteinsatz geeignet, da der häufige Vergleich mit dem Original zwangsläufig zu einem schlechten Laufzeitverhalten führt. Ausserdem kann nicht garantiert werden, dass das Ergebnis dem Original wirklich gleicht. Es steht aber fest, dass das Ergebnisbild der Vorlage mit jeder Änderung ähnlicher wird. Daher ist die Qualität des Ergebnisses auch stark von dem Wert abhängig, der angibt, nach wie vielen Änderungen der Algorithmus abbricht.

Der Vergleich des Testbildes mit dem Original kann auf unterschiedliche Weise erfolgen. Gute Ergebnisse liefert die Standardabweichung der Pixeldifferenzen. Für Farbbilder muss diese Differenz für jeden Farbkanal bestimmt werden, benötigt man nur die Helligkeit des Bildes kann pro Pixel eine Differenz berechnet werden. Um die Performance zu verbessern werden nur jene Pixel analysiert, die verändert wurden. Abbildung 7.13 zeigt solch ein generiertes Bild mit Original. Code 7.11 (*Vergleich*) kann als Ausgangspunkt für solche Automaten herangezogen werden. Um auf die Pixel des Originals zugreifen zu können, bedarf es der Klasse *PixelGrabber*, die in der Java Version 1.1.8 bereits enthalten ist.

Besonders interessante Ergebnisse liefert die vergleichende Bildsynthese bei der Erzeugung von Animationen. Dabei wird für jedes Frame aus einer bestehenden Animation ein neues Bild erzeugt und die Ergebnisbilder werden zu einer neuen Animation zusammengefügt. Durch die zusätzliche zeitliche Dimension erreicht ein solches Video eine beträchtlich deutlichere Ähnlichkeit mit dem Original.

Motion Capturing

Allen bisher beschriebenen Konstruktionsprinzipien liegen Algorithmen zur Erzeugung von Daten zu Grunde. Die vergleichende Bildsynthese orientiert sich dabei zwar an realen Vorlagen, die für Werte bestimmt, ob diese Teil der Ergebnismenge sein sollen oder nicht, die Werte selbst werden aber vom Computer generiert. Eine völlig andere Art der Datenererzeugung ist das Motioncapturing. Dabei werden Werte gemessen. Bei der Bewegungserfassung geschieht dies durch Auswerten von realen Bewegungen. Dabei können verschiedene Techniken eingesetzt werden, wobei hier eine Lösungen mittels Videokamera anhand von zwei Beispielen kurz vorgestellt werden soll.

Eine Hi-Speed-Kamera filmt sich bewegende Menschen oder Objekte die mit Markern versehen sind. Diese Marker befinden sich an relevanten Positionen des sich bewegenden Objekts aber auch an Fixpunkten auf die sich die Bewegung bezieht. Bei der Aufnahme von menschlicher Bewegung werden z.B. die Gelenke mit Markern versehen. Das Filmmaterial wird danach per Software analysiert und die Positionen der Marker für jedes Frame daraus ermittelt.

Diese Koordinaten können danach als Ausgangswerte für die weitere Verarbeitung am Computer herangezogen werden. Dabei ist es sowohl möglich die ursprüngliche Bewegung zu rekonstruieren, als auch die gewonnenen Daten in einem komplett anderen Zusammenhang einzusetzen, z.B. als Ausgangsmaterial für Animationen, als Ersatz für Zufallswerte oder zur Generierung von Parametern. Der große Vorteil, den diese Daten gegenüber künstlich erzeugten Werten bietet, ist der, dass diese natürlich sind und auch so wirken.

Die freie Java-Software Move [14] verfolgt automatisch Marker die sich über mehrere Frames hinweg bewegen und speichert die Positionen als zweidimensionale Koordinatenpaare für jedes Frame ab. Die Klasse *MarkerFile* enthält alle Methoden um auf alle Marker eines Frames zugreifen zu können. Die Klasse *Marker* selbst repräsentiert einen solchen Marker, wobei dieser die X- und Y-Koordinate als auch einen eindeutigen Namen bereitstellt. Anhand dieses Namen können die verschiedenen Marker voneinander unterschieden werden.

Der ebenfalls freie Java-Sourcecode Demove [14] reproduziert anhand dieser Daten die ursprüngliche Bewegung und visualisiert diese Bewegung mit Hilfe von Bildern, welche entsprechend einem oder mehrerer Marker skaliert, rotiert und an die korrekte Stelle eingefügt werden. Abbildung 7.14 zeigt Standbilder einer Animation die mit dieser Software erstellt wurde, wobei die einzelnen Teilbilder gemalt und danach eingescannt wurden. Genau genommen ist diese

Abbildung 7.14

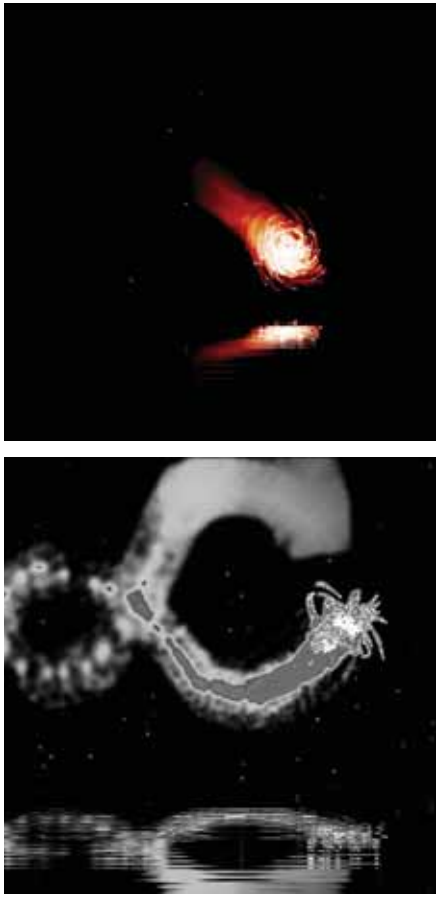


Abbildung 7.15

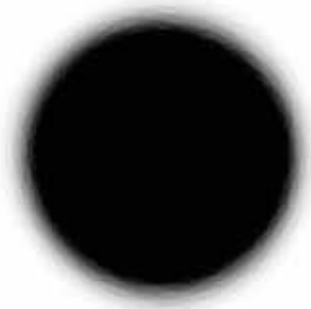
Technik nicht unmittelbarer Bestandteil der Kunstinformatik, da der Computer hier nur als Werkzeug und nicht als Medium selbst dient. Werden die so gewonnenen Daten aber im Sinne der Computerkunst weiterverwendet, öffnet gerade dieses Vorgehen der virtuellen Kunst die Türen zu anderen - nicht virtuellen - Kunstrichtungen, wie z.B. Tanz.

Motioncapturing kann auch in Echtzeit in interaktiven Systemen zum Einsatz kommen. Dafür ist in Java das Media Framework notwendig, das unter anderem Daten von einer angeschlossenen Videokamera lesen und die Frames in Echtzeit als Einzelbilder zur Verarbeitung bereitstellen kann. Aus diesen Bildern kann anhand der Farbe ein Marker sehr rasch aufgefunden werden und dessen Koordinaten zur Erzeugung von positionsabhängigen Computergrafiken verwendet werden. Abbildung 7.15 zeigt solche generierten Bilder, deren Position und Verhalten von der erfassten Bewegung abhängt.

Das Java Media Framework ist in der Lage auf Datenströme zuzugreifen, wenn das Betriebssystem diese bereitstellt. Hier soll gezeigt werden, wie über Video für Windows die Frames eines Videodatenstroms angesprochen werden können. Dafür wird eine Unterklasse von *Codec* erzeugt, wobei einige Methoden implementiert werden müssen. Zunächst bedarf es eines *Processors* welcher mit der Klasse *Manager* - unter Angabe des Videotreibers - erzeugt wird.

Danach wird nach einem verfügbaren Video-Track gesucht, das Videoformat ausgelesen und der Processor realisiert. Nach dem der Processor gestartet wurde, wird für jedes Frame die Methode *process()* aufgerufen, in der auf die Pixel der Einzelbilder zugegriffen werden kann. Die einzelnen Farben sind dabei aber nicht in einem int-Wert codiert, sondern durch drei separate Bytes. Ein solcher Codec ist in Code 7.12 (*Codec*) implementiert.

Fourier Analyse



Jedes periodische Signal kann als Überlagerung von Sinussignalen dargestellt werden, wobei sich diese Sinuswellen in ihrer Frequenz, Amplitude und Phase voneinander unterschieden. Die Fourieranalyse dient dieser Zerlegung von Signalen in ihre Sinusanteile. Da in der künstlerischen Praxis nur selten periodische Signale auftreten, wird die Fourieranalyse auf endliche Signale angewandt, wobei der gesamte Signalverlauf als eine Periode betrachtet wird. Für digitale Daten, welche auf gemessenen Werten beruhen, kommt die diskrete Fourieranalyse zum Einsatz. Dabei wird ein Signal in genau so viele Frequenzbänder unterteilt, wie konkrete Werte vorliegen. Somit kann jedes digitale Signal verlustfrei in den Frequenzraum übergeführt werden. Die inverse diskrete Fouriertransformation schiebt das Signal vom Frequenzraum in den Ortsraum zurück.

Fast Fourier Transformation

$$f(x) \sim \sum_{n=0}^{N-1} c_n \cdot e^{inx}$$

Formel 8.1
gesuchte Fourierentwicklung

Der wichtigste Algorithmus in der Kunstinformatik ist die schnelle Fouriertransformation, kurz FFT. Dieser Algorithmus ist in der Lage die diskrete Fourieranalyse um ein vielfaches schneller auszuführen, als dies durch einfache Implementierung der Fall wäre. Der Aufwand ohne Optimierung beträgt für n Werte $O(n^2)$. Die FFT reduziert diesen Aufwand auf $O(n \log(n))$. Im Folgenden soll ein optimierter FFT-Algorithmus für Java hergeleitet werden. [17]

$$c_n = \frac{1}{2\pi} \int_{-\pi}^{\pi} f(x) \cdot e^{-inx} dx$$

Formel 8.2
Fourier-Koeffizienten

Eine FFT soll für eine 2π-periodische Funktion f eine angenäherte diskrete komplexe Fourierentwicklung bestimmen können (Formel 8.1). Als Ergebnis erhält man N komplexe Koeffizienten c_n mit n = 0,1,2,...N-1, wobei N der Anzahl an vorhandenen komplexen Werten entspricht. Der Betrag eines Koeffizienten c_n beschreibt die Amplitude des n-ten Obertons, also der Sinuswelle mit der Frequenz = (Basisfrequenz * n), die Phase des Koeffizienten bestimmt die Phase des betreffenden Sinussignals (Formel 8.2). Mittels Trapezregel kann das Integral annähernd berechnet werden, wobei das Periodenintervall $[0, 2\pi]$ in N gleichlange Teilintervalle der Länge $x_{k+1} - x_k = h = 2\pi/N$ unterteilt wird (Formel 8.3). Als Stützstellen der Trapezregel (Formel 8.4) werden $x_k = 2\pi k/N$ mit k = 0,1,2,...N-1 gewählt. Wegen der Periodizität ist $f(x_0) = f(x_N)$, womit die angenäherten diskreten Fourierkoeffizienten mit (Formel 8.5) berechnet werden können, welche in Code 8.1 (*DFT*) implementiert ist. Die Werte für $f(x_k)$ entsprechen dabei den vorhandenen Daten, welche transformiert werden sollen. Ist der Parameter invers = true, wird die inverse diskrete Fouriertransformation ausgeführt, welche sich in ihrer Implementierung nur kaum von der nicht inversen unterscheidet (vergleiche Formel 8.3 und Formel 8.5).

$$f(x_k) \sim \sum_{n=0}^{N-1} c_n \cdot e^{\frac{ink2\pi}{N}}$$

Formel 8.3 (inverse DFT)
Teilintervalle der Länge h

$$\tilde{c_n} = \frac{1}{2N}\left[2 \cdot \sum_{k=1}^{N-1} f(x_k) \cdot e^{\frac{-ink2\pi}{N}} + f(x_0) + f(x_N)\right]$$

Formel 8.4
Trapezregel

Dass der Aufwand für diesen Algorithmus $O(n^2)$ ergibt, ist leicht an den geschachtelten Schleifen abzulesen. Der Aufwand pro innerem Schleifendurchgang ist für DFT und inverse DFT (kurz IDFT) der selbe und beträgt 5 Multiplikationen, 3 Additionen, 1 Subtraktion, die Berechnungen für Cosinus und Sinus, 2 Lese- und 2 Schreibzugriffe, wobei alleine für die komplexe Multiplikation 4 Multiplikationen, 1 Addition und 1 Subtraktion anfallen. Betrachtet man die Berechnungen für *wr* und *wi* stellt man fest, dass die Ergebnisse sehr oft die selben sein werden, da Sinus- und Cosinusfunktion 2π-periodisch sind.

$$\tilde{c_n} = \frac{1}{N} \sum_{k=0}^{N-1} f(x_k) \cdot e^{\frac{-ink2\pi}{N}}$$

Formel 8.5 (DFT)
Diskrete Fourier-Transformation

1965 entdeckten J.W. Cooley und J.W. Tukey wunderbare Vereinfachungen die sich ergeben, wenn N = 2 hoch p (p ist eine natürliche Zahl) gewählt wird. Unter Zuhilfenahme der Gleichung aus Formel 8.6 und der Vereinbarung w_p = w hoch p, mit p = n*k, ergibt sich für $w_N = 1$, $w_{N+1} = w$, $w_{N+2} = w*w$, ... $w_{N+N} = w_N = 1$, usw. Für N=8 ist somit $c_4 = a_0 + a_1w_4 + a_2w_8 + a_3w_{12} + a_4w_{16} + a_5w_{20} + a_6w_{24} + a_7w_{28}$. w_{16} ist aber 1, $w_{20} = w_4$, usw. womit wir zusammenfassend $c_4 = (a_0 + a_4) + w_4(a_1 + a_5) + w_8(a_2 + a_6) + w_{12}(a_3 + a_7)$ erhalten. Verallgemeinert kann die Gleichung aus Formel 8.7 mit zwei Gleichungen der halben Dimension gelöst werden, wobei Formel 8.8 alle Koeffizienten c_n mit geradem Index n = 2m und Formel 8.9 jene mit ungeradem Index n = 2m+1 berechnet. Jede dieser Gleichungen kann nun nach demselben Schema zu zwei weiteren Transformationen der halben Dimension zusammengefasst werden. Tabelle 8.1 zeigt diese Vorgangsweise für N=16. [18]

a00	x00 = (a00 + a08)	y00 = (x00 + x04)	z00 = (y00 + y02)	c00 = (z00 + z01)
a01	x01 = (a01 + a09)	y01 = (x01 + x05)	z01 = (y01 + y03)	c08 = (z00 - z01)
a02	x02 = (a02 + a10)	y02 = (x02 + x06)	z02 = (y00 - y02)	c04 = (z02 + z03)
a03	x03 = (a03 + a11)	y03 = (x03 + x07)	z03 = (y01 - y03)w4	c12 = (z02 - z03)
a04	x04 = (a04 + a12)	y04 = (x00 - x04)	z04 = (y04 + y06)	c02 = (z04 + z05)
a05	x05 = (a05 + a13)	y05 = (x01 - x05)w2	z05 = (y05 + y07)	c10 = (z04 - z05)
a06	x06 = (a06 + a14)	y06 = (x02 - x06)w4	z06 = (y04 - y06)	c06 = (z06 + z07)
a07	x07 = (a07 + a15)	y07 = (x03 - x07)w6	z07 = (y05 - y07)w4	c14 = (z06 - z07)
a08	x08 = (a00 - a08)	y08 = (x08 + x12)	z08 = (y08 + y10)	c01 = (z08 + z09)
a09	x09 = (a01 - a09)w	y09 = (x09 + x13)	z09 = (y09 + y11)	c09 = (z08 - z09)
a10	x10 = (a02 - a10)w2	y10 = (x10 + x14)	z10 = (y08 - y10)	c05 = (z10 + z11)
a11	x11 = (a03 - a11)w3	y11 = (x11 + x15)	z11 = (y09 - y11)w4	c13 = (z10 - z11)
a12	x12 = (a04 - a12)w4	y12 = (x08 - x12)	z12 = (y12 + y14)	c03 = (z12 + z13)
a13	x13 = (a05 - a13)w5	y13 = (x09 - x13)w2	z13 = (y13 + y15)	c11 = (z12 - z13)
a14	x14 = (a06 - a14)w6	y14 = (x10 - x14)w4	z14 = (y12 - y14)	c07 = (z14 + z15)
a15	x15 = (a07 - a15)w7	y15 = (x11 - x15)w6	z15 = (y13 - y15)w4	c15 = (z14 - z15)

Tabelle 8.1

$$\tilde{c}_n = \frac{1}{N}\sum_{k=0}^{N-1} a_k \cdot w^{nk}$$

$$a_k = f(x_k), \quad w = e^{\frac{-i2\pi}{N}}$$

Formel 8.6
Umformung für FFT

$$e^{i2\pi} - 1 = 0, \quad e^{-i2\pi} = 1$$

Formel 8.7
Schönste Formel der Mathematik

Die hier vorgestellte Methode hat den zusätzlichen Vorteil, dass sie keinen zusätzlichen Speicher für die Ergebniswerte benötigt, da diese die nicht transformierten Werte einfach ersetzen kann. Allerdings stehen die berechneten Koeffizienten nicht in der korrekten Reihenfolge im Array. Diese können aber sehr einfach an die richtige Position gebracht werden, da ihr Index genau der bitreversen Dualzahl des momentanen Index entspricht. Der Aufwand hat sich drastisch reduziert. Statt 256 komplexer Multiplikationen werden nur noch 17 benötigt, statt 256 komplexer Additionen nur noch 64. Die Anzahl der Winkelberechnungen vermindert sich von 256 auf ebenfalls 17 oder gar nur 7, wenn die Werte zwischengespeichert werden. Code 8.2 (*FFT rekursiv*) zeigt eine rekursive Implementierung dieses Algorithmus.

$$\tilde{c}_{2m} = \frac{1}{N} \cdot \sum_{k=0}^{N/2-1} (a_k + a_{k+N/2}) \cdot w^{2km}$$

Formel 8.8
Gerade Koeffizienten

Nun soll eine Implementierung (Code 8.3 *FFT iterativ*) vorgestellt werden, die iterativ arbeitet. Zusätzlich werden das Bitreversal Mapping und die Winkelfunktionen durch effizientere Verfahren ersetzt. Bei genauerer Betrachtung des Codes fällt auf, dass es sich hierbei nicht um die Cooley-Tukey-FFT handelt sondern um eine Danielson-Lanczos-FFT. Diese unterscheiden sich vom Aufwand her nicht, allerdings gibt es Unterschiede was die Abarbeitungsreihenfolge anbelangt. Alle weiteren Optimierungen verwenden die Cooley-Tukey-Formel, der Vollständigkeit halber wird aber auch die Danielson-Lanzos-Formel präsentiert.

$$\tilde{c}_{2m+1} = \frac{1}{N} \cdot \sum_{k=0}^{N/2-1} (a_k - a_{k+N/2}) \cdot w^{2km} \cdot w^k$$

Formel 8.9
Ungerade Koeffizienten

Vergleicht man die beiden Algorithmen auf ihr Laufzeitverhalten hin, stellt man fest, dass die rekursive FFT der iterativen nicht gewachsen ist. Der iterative Algorithmus benötigt oft nur ein Drittel der Zeit. Das trifft allerdings nur für kleinere Datensätze mit einer Anzahl von bis zu etwa 2 hoch 10 Werten zu. Ab einer Datengröße von über 2 hoch 15 Werten, bricht die iterative FFT gegenüber der rekursiven stark ein. Die rekursive Implementierung arbeitet nun

$$H(n) = C_{ne} + iC_{n0}$$

$$C_n = C_{ne} + C_{n0} \cdot e^{\frac{i2\pi n}{N}}$$

Formel 8.10
Separierung der Koeffizienten

Jean-Pierre Hébert
Untitled (1979)

fast fünfmal so schnell. Das liegt daran, dass die rekursive Lösung, durch das Aufteilen der Daten in kleinere Datensätze, den CPU-Cache besser zu Nutzen vermag. Sobald die Größe des Datensatzes die Größe des Caches übersteigt, müssen die Daten bei der iterativen FFT jedesmal neu aus dem Speicher geladen und geschrieben werden, was zu einem stark vergrößerten Aufwand führt.

Aus diesem Grund wird eine weitere Implementierung (Code 8.4 *FFT kombiniert*) vorgestellt, die eine Kombination aus beiden Algorithmen ist. Besteht ein Datensatz aus mehr Werten als Platz im Cache haben, wird zunächst rekursiv verfahren. Sobald die Datenmenge unter diesen kritischen Wert fällt, wird iterativ weitergearbeitet. Je nach Cachegröße muss dieser Schwellwert unterschiedlich groß sein, wobei in der Praxis Werte zwischen 2 hoch 10 und 2 hoch 12 besonders geeignet zu sein scheinen. Des Weiteren wird die FFT noch zusätzlich optimiert. So können alle komplexen Multiplikationen mit w hoch 0 entfallen. Ausserdem werden die letzten beiden Schritte zur entgültigen Berechnung der Koeffizienten zusammengefaßt, was einen zusätzlichen leichten Geschwindigkeitsvorteil ergibt. Das bitreverse Mapping wurde mit der Division der Koeffizoenten durch die Anzahl der Daten kombiniert.

Aufgrund mathematischer Überlegungen, kann der Algorithmus noch weiter verbessert werden. Vor der FFT auf die Daten im Array *real* und *imag* sind alle Werte in imag gleich Null. Nach der FFT ist dies nicht mehr der Fall, dafür sind die Werte in real spiegelverkehrt symmetrisch, es gilt also: real(c_n) = real(c_{N-n}) für n = 0,1,2...N/2. Ähnlich verhält es sich mit den Daten in imag. Dort gilt: imag(c_n) = -imag(c_{N-n}) für n = 0,1,2...N/2. Es sollte also möglich sein einen Algorithmus zu entwerfen, der nur halb so viel Speicherplatz benötigt wie die bisherigen Implementierungen.

Dazu wird der Array real als ein Datensatz mit N/2 komplexen Zahlen aufgefaßt, wobei abwechselnd ein realer und dann ein imaginärer Wert in das Array geschrieben wird. Wählt man alle geraden Werte für die realen Anteile und die ungeraden für die imaginären, kann der Array real unverändert übernommen werden. Nun wird dieser Datensatz transformiert. Aus dem Ergebnis müssen die korrekten Werte separiert werden, was erstaunlich einfach zu bewerkstelligen ist. Die Koeffizienten c_n können mit Hilfe der Formel 8.10 aus den Ergebniswerten H_n berechnet werden.

Werden von diesen komplexen Werten wieder abwechselnd der Real- und Imaginärwert in das Array real gespeichert, so wird das Array imag nicht mehr benötigt und der Speicherverbrauch sinkt auf die Hälfte. Allerdings gibt es ein Problem mit dem Koeffizienten an der Stelle N/2, da dieser nicht mehr in das Array paßt. Da die imaginären Anteile der Originaldaten aber alle gleich Null waren, sind die Koeffizienten c_0 und $c_{N/2}$ beide reelle Zahlen. Der Wert für $c_{N/2}$ kann also an die Stelle des Imaginäranteils von c_0 geschrieben werden.

Durch die Interpretation des Arrays real als Array mit komplexen Zahlen, reduziert sich sie die Anzahl der zu berechnenden Werte auf die Hälfte, was zu einer weiteren Steigerung der Performance führt. Ausserdem wird nur noch ein Array verwendet, was zu einer zusätzlichen Geschwindigkeitssteigerung führt. Das Laufzeitverhalten der DFT, der drei zuvor besprochenen FFTs und dieser Implementierung (Code 8.5 *FFT real*) wird in Tabelle 8.2 illustriert.

Das Gesamtvolumen der Daten umfaßt für alle Messungen 2 hoch 26 komplexe Zahlen, wobei die Hälfte davon auf die inverse Transformation fallen. Die Berechnungen wurden auf einem Rechner mit Athlon XP-2400 mit 512 MB Speicher ausgeführt.

P	N	DFT	FFT rekursiv	FFT iterativ	FFT kombiniert	FFT real
4	16	110	11.5	7.7	5.0	3.0
6	64	400	18.0	7.8	6.0	3.5
8	256	1573	24.7	8.2	6.9	3.8
10	1024	6340	31.2	9.2	8.1	4.4
12	4096	k.A.	38.2	11.0	10.2	5.0
14	16384	k.A.	48.4	35.2	18.3	6.5
16	65536	k.A.	71.0	253.2	39.1	13.8
18	262144	k.A.	81.4	304.0	47.3	17.3

Tabelle 8.2: Messergebnisse in Sekunden

Anwendung findet die eindimensionale FFT - bezogen auf Kunst - hauptsächlich in der Musik. Filterberechnungen können im Fourierraum effizienter und gezielter durchgeführt werden. Es ist möglich die Abspielgeschwindigkeit einer Aufnahme zu ändern, ohne dabei auch gleichzeitig die Tonhöhe zu beeinflussen (Timestretching). Umgekehrt ist es aber auch möglich die Tönhöhe zu ändern, wobei die Abspieldauer gleich bleibt (Pitchshifting). Auch Kombinationen beider Verfahren sind einfach zu realisieren. Experimentellere Ansätze - wie z.B. die Resynthese - versuchen durch Änderungen im Fourierraum neuartige Klänge zu generieren. So können Frequenzbänder vertauscht werden oder Phasen eines Klänges mit den Frequenzen eines anderen kombiniert werden.

Mehrdimensionale FFT

Werden mehrere gleichgroße eindimensionale Signale in den Fourierraum übertragen und diese z.B. als die diskreten Zeilen eines zweidimensionalen Signalfeldes aufgefasst, kann dieses Feld ebenfalls in den Fourierraum übertragen werden indem die jeweils zu einer Spalte gehörigen Daten der horizontalen eindimensionalen Signale als vertikale eindimensionale Signale interpretiert werden und in den Fourierraum transformiert werden. Als Ergebnis erhält man die Frequenzverläufe in jede Richtung des Feldes mit Amplitude und Phase. Somit können auch Bilder im Fourierraum bearbeitet, verändert und sogar kreiert werden (Abbildung 8.1). Häufigste Anwendung der 2D-FFT auf Bilder ist wie bei der Musik die Filterung von Frequenzen bzw. allgemeiner der Faltung, die im Fourierraum viel effizienter durchgeführt werden kann.

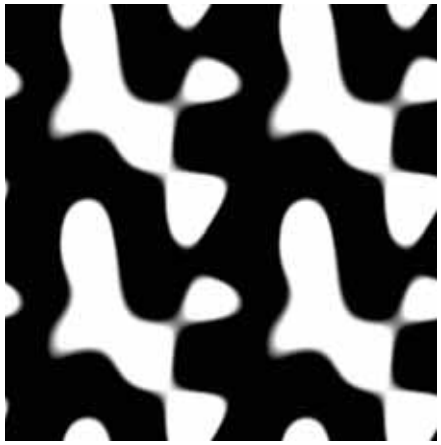
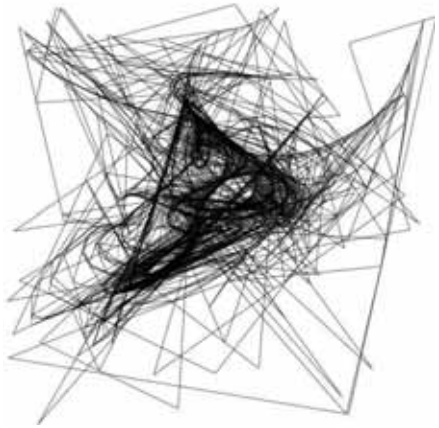


Abbildung 8.1

Geht man einen Schritt weiter und interpretiert die Fourierfelder als diskrete Zeitstempel eines sich bewegenden Bildes, so kann durch Transformation der einzelnen Positionen durch die Zeitachse auch ein Video in den Fourierraum gebracht werden. Dort werden Bewegungen bzw. Dynamiken mit Frequenz, Phase und Amplitude beschrieben. Die Einsatzmöglichkeiten sind derart vielschichtig und komplex, man denke nur an die Tatsache, dass sich dadurch Bewegungen beeinflussen lassen, dass die 3D-FFT im Videobereich kaum erforscht ist. Denkbar ist auch der Einsatz der FFT in der vierten Dimension, z.B. bei 3D-Animationen.

Je höher die Dimension der FFT ist, desto komplexer wird der Algorithmus und die Optimierung desselben. Vor allem die Reduzierung der Berechnungen auf die Hälfte durch Ausnutzen der Tatsache, dass die Werte im Ortsraum ohne Imaginäranteil vorliegen, wird bei mehr Dimensionen derart komplex, dass hier darauf verzichtet werden soll, obwohl diese Optimierung möglich wäre. Auch die Teilung der Daten durch rekursive Aufrufe gestaltet sich relativ problematisch. Andererseits ist diese bei der mehrdimensionalen FFT auch nicht unbedingt nötig, da die eindimensionalen Arrays bei Bildern und Videos nicht die Ausmaße erreichen wie z.B. bei Musikaufnahmen. Code 8.6 (*FFT mehrdimensional*) zeigt eine Implementierung einer FFT, die beliebig dimensionierte Datensätze in den Fourierraum und zurück in den Ortsraum führt. Dabei werden alle Daten in einem Array gespeichert, wobei sich wieder Real- und Imaginärwert abwechseln. Bei Bildern werden die Zeilen hintereinander in das Array geschrieben, bei Videos die einzelnen Frames. [19]

Schlußwort



Unbeachtet blieb bis jetzt die Generierung von 3D-Bildern. Durch die hohe Nachfrage an diesen Techniken, hat sich inzwischen ein eigener Industriezweig entwickelt, die sich allen Fragen der dreidimensionalen und photorealistischen Computergrafik widmet. Aufgrund dieser Tatsache mangelt es nicht an Literatur zu diesem Thema, auf welche hier also lediglich hingewiesen werden soll. Das nötige Werkzeug dazu bietet Java mit der Java 3D Library. [22]

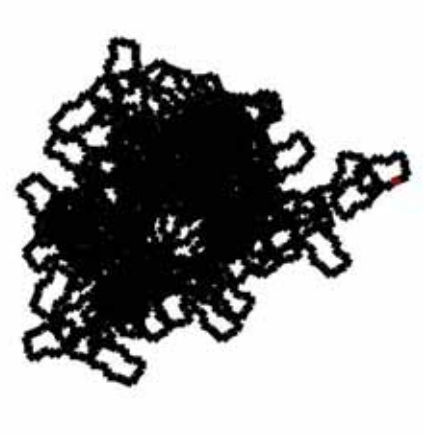
Für die Netzkunst sind weiters Informationen die Netzwerktechnik betreffend von Interesse. Java bietet bereits in der Version 1.1 viele Möglichkeiten zur Kommunikation über ein Netzwerk und im Besonderen über das Internet. Da auch zu diesem Thema reichhaltig Literatur zu finden ist, wird hier ebenfalls auf diese verwiesen. [23]

Verglichen mit anderen Kunstrichtungen, ist für die Umsetzung von Computerkunst ein enormes technisches Wissen notwendig. Während in der traditionellen Kunst und Musik das Handwerk im Vordergrund steht, eine Technik also durch körperliches Training erlernt werden muss, bedarf die Computerkunst zusätzlich eines sehr starken geistigen Trainings. Andererseits darf dieses Wissen die künstlerische Seite in keiner Weise beeinträchtigen. Die Technik muss der Kunst immer untergeordnet bleiben. Sobald die technische Seite die Überhand gewinnt, verliert das Ergebnis derartig an Wert, dass eigentlich nicht mehr von Kunst gesprochen werden kann. Anders verhält es sich mit Grafik und Design, welche die Ästhetik in den Vordergrund stellen. In diesen Bereichen darf die Technik deutlicher spürbar sein.

Andererseits darf die technische Seite auch nicht zu sehr unterbewertet werden. Auch in der traditionellen Kunst und Musik erwartet man von KünstlerInnen ein hohes Maß an technischem Können. Soll diese Tradition aufrechterhalten bleiben - und meiner Meinung nach ist das für den Wert eines Werkes von immenser Bedeutung - dann muss auch ein Computerkünstler in der Lage sein, sein Werkzeug bzw. Instrument vollkommen zu beherrschen. Erst dann ist es möglich Kunst auf einem wirklich hohen Niveau zu erschaffen. Trotzdem ist technisches Können allein viel zu wenig um an das Geheimnis Kunst heranzukommen. Die Kunst muss in ihrer Qualität in jeder Faser ihrer Erscheinung den technischen Ursprung überwinden können und ihre Wahrheit erzählen können. Die Gesetze der Gravitation dürfen in der Kunst nicht zur Geltung kommen müssen. Oder anders, deutlicher gesagt:

In der Kunst gilt die Schwerkraft nicht!

Anhang



Code

Linienmuster

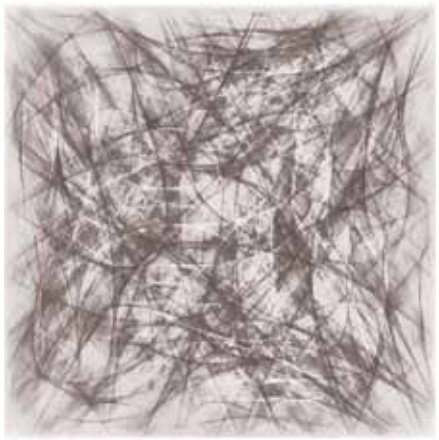
```
int width, heigth;                              // width and heigth of image
double x1,y1,x2,y2;                             // initial position of two point
double x3,y3,x4,y4;                             // copy of above points
double dx1,dy1,dx2,dy2;                         // parameter for moving-vector
double dx3,dy3,dx4,dy4;                         // copy of above parameters

int amt;                                        // amount of lines without deleting

public void draw() {

     drawLine(x1,y1,x2,y2);

     x1+= dx1; if(x1=width)  { dx1= -dx1; x1+= dx1; }
     y1+= dy1; if(y1=height) { dy1= -dy1; y1+= dy1; }
     x2+= dx2; if(x2=width)  { dx2= -dx2; x2+= dx2; }
     y2+= dy2; if(y2=height) { dy2= -dy2; y2+= dy2; }

     if(amt-- < 0) {                            // start deleting

          clearLine(x3,y3,x4,y4);

          x3+= dx3; if(x3=width)  { dx3= -dx3; x3+= dx3; }
          y3+= dy3; if(y3=height) { dy3= -dy3; y3+= dy3; }
          x4+= dx4; if(x4=width)  { dx4= -dx4; x4+= dx4; }
          y4+= dy4; if(y4=height) { dy4= -dy4; y4+= dy4; }
     }
}
```

Code 4.1

Bresenham

```
int width, height;                                  // width and height of image

public boolean bigger(int v1, int v2) {             // is Math.abs(v1) > Math.abs(v2)?
    if(v1 < 0) {
        if(v2 < 0) return v1 <  v2;
        else       return v1 < -v2;
    } else {
        if(v2 < 0) return v1 > -v2;
        else       return v1 >  v2;
    }
}

public void drawLine(int x1, int y1, int x2, int y2) {
    if(x1 == x2)
        if(y1 == y2) { set(y1 * width + x1); return; }    // point
        else {                                            // vertical line
            int i1 = y1 * width + x1;
            int i2 = y2 * width + x2;
            if(y1 < y2) while(i1 != i2) { set(i1); i1+= width; }
            else        while(i1 != i2) { set(i1); i1-= width; }
            set(i2); return;
        }
    else if(y1 == y2) {                                   // horizontal line
        int id = y1 * width;
        int i1 = id + x1;
        int i2 = id + x2;
        if(x1 < x2) while(i1 != i2) set(i1++);
        else        while(i1 != i2) set(i1--);
        set(i2); return;
    }                                                     // other lines:
    int hp, f1, f2;                                       // bresenham algorithm
    int dx = x2 - x1;
    int dy = y2 - y1;
    int i1 = y1 * width + x1;
    int i2 = y2 * width + x2;
```

```
if(bigger(dx, dy)) {                        // alpha < 45 degrees
     if(dx > 0) {
          if(dy > 0) {
               f1 = dy *  2; hp = f1 - dx; f2 = hp - dx;
               while(i1 != i2) {
                    set(i1);
                    if(hp < 0) hp+= f1; else { hp+= f2; i1+= width; }
                    i1++; }
          } else {
               f1 = dy * -2; hp = f1 - dx; f2 = hp - dx;
               while(i1 != i2) {
                    set(i1);
                    if(hp < 0) hp+= f1; else { hp+= f2; i1-= width; }
                    i1++; }
          }
     } else {
          if(dy > 0) {
               f1 = dy *  2; hp = f1 + dx; f2 = hp + dx;
               while(i1 != i2) {
                    set(i1);
                    if(hp < 0) hp+= f1; else { hp+= f2; i1+= width; }
                    i1--; }
          } else {
               f1 = dy * -2; hp = f1 + dx; f2 = hp + dx;
               while(i1 != i2) {
                    set(i1);
                    if(hp < 0) hp+= f1; else { hp+= f2; i1-= width; }
                    i1--; }
          }
     }
```

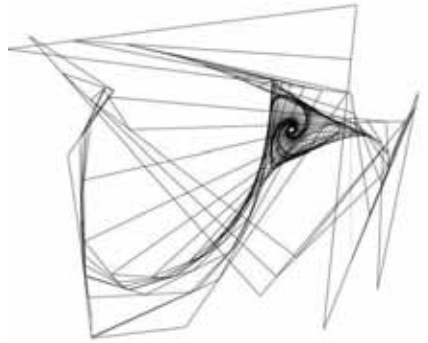


```
    } else {                                        // alpha > 45 degrees
        if(dx > 0) {
            if(dy > 0) {
                f1 = dx *  2; hp = f1 - dy; f2 = hp - dy;
                while(i1 != i2) {
                    set(i1);
                    if(hp < 0) hp+= f1; else { hp+= f2; i1++; }
                    i1+= width; }
            } else {
                f1 = dx *  2; hp = f1 + dy; f2 = hp + dy;
                while(i1 != i2) {
                    set(i1);
                    if(hp < 0) hp+= f1; else { hp+= f2; i1++; }
                    i1-= width; }
            }
        } else {
            if(dy > 0) {
                f1 = dx * -2; hp = f1 - dy; f2 = hp - dy;
                while(i1 != i2) {
                    set(i1);
                    if(hp < 0) hp+= f1; else { hp+= f2; i1--; }
                    i1+= width; }
            } else {
                f1 = dx * -2; hp = f1 + dy; f2 = hp + dy;
                while(i1 != i2) {
                    set(i1);
                    if(hp < 0) hp+= f1; else { hp+= f2; i1--; }
                    i1-= width; }
            }
        }
    }
    set(i2);
}
```

Code 4.2

Antialiasing

```
public void drawLine(double xs, double ys, double xe, double ye, double size) {
    int x,nx,ex,y,ny,ey,lo,yo,eo,lu,yu,eu,ox,ux,k,m,r,a,f,b;

    // points are changed if order is right to left
    if(xs > xe) { double t = xe; xe = xs; xs = t; t = ye; ye = ys; ys = t; }

    int x1 = (int)(xs * 1024), y1 = (int)(ys * 1024);      // float-simulation of x1, y1
    int x2 = (int)(xe * 1024), y2 = (int)(ye * 1024);      // float-simulation of x2, y2
    int idx = x2 - x1, idy = y2 - y1;                      // idx, idy calculation

    if((Math.abs(idx)<< Math.abs(idy)) {                   // vertical line
        if(y1 > y2) { int t = y2; y2 = y1; y1 = t; }       // up-down-order guaranteed
         r = (int)(size * 512);                            // r = size / 2
         x = x1 - r; ex = x1 + r;                          // first and last x-value
        lo = y1; lu = y2;                                  // y-values at x
        ox = x1; ux = x2;                                  // x-values of m-k-switch
        eo = y1; eu = y2;                                  // y-values at ex
        m  = 0;  k  = 0;
    } else if((Math.abs(idy)<< Math.abs(idx)) {            // horizontal line
         r = (int)(size * 512);                            // r = size / 2
         x = x1; ex = x2;                                  // first and last x-value
        lo = y1 - r; lu = y1 + r;                          // y-values at x
        ox = x2; ux = x1;                                  // x-values of m-k-switch
        eo = y1 - r; eu = y2 + r;                          // y-values at ex
        m  = 0;  k  = 0;
    } else if(idy < 0) {                                   // line going upwards
        double dx = xe - xs, dy = ye - ys;                 // dx, dy calculation
        double c = (512 * size) / Math.sqrt(dx * dx + dy * dy);
        int p = (int)(dy * c);                             // p calculation
        int z = (int)(dx * c);                             // z calculation
         x = x1 + p; ex = x2 - p;                          // first and last x-value
        lo = y1 - z; lu = lo;                              // y-values at x
        ox = x2 + p; ux = x1 - p;                          // x-values of m-k-switch
        eo = y2 + z; eu = y2 + z;                          // y-values at ex
        k  = (int)((dy / dx) * 1024); m = (int)((- dx / dy) * 1024);
```



```
} else {                                                    // line going downwards
    double dx = xe - xs, dy = ye - ys;                      // dx, dy calculation
    double c = (512 * size) / Math.sqrt(dx * dx + dy * dy);
    int p = (int)(dy * c);                                  // p calculation
    int z = (int)(dx * c);                                  // z calculation
     x = x1 - p; ex = x2 + p;                               // first and last x-value
    lo = y1 + z; lu = lo;                                   // y-values at x
    ox = x1 + p; ux = x2 - p;                               // x-values of m-k-switch
    eo = y2 - z; eu = y2 - z;                               // y-values at ex
    m  = (int)((dy / dx) * 1024); k = (int)((- dx / dy) * 1024);
}

nx = (x & 0xfffffc00) + 1024;                               // nx calculation
f = nx - x;                                                 //  f calculation

// first-time yo and yu calculation
if(nx > ex) { yo = eo; yu = eu; f+= ex - nx; }              // last pixel-row
else {
    if(x < ox)
        if(nx > ox)                                         // m-k-switch
            yo = lo + (((ox - x) * k + (nx - ox) * m) >> 10);
        else yo = lo + ((f * k) >> 10);                     // left of ox
    else yo = lo + ((f * m) >> 10);                         // right of ox
    if(x < ux)
        if(nx > ux)                                         // m-k-switch
            yu = lu + (((ux - x) * m + (nx - ux) * k) >> 10);
        else yu = lu + ((f * m) >> 10);                     // left of ux
    else yu = lu + ((f * k) >> 10);                         // right of ux
}
while(x < ex) {                                             // for all pixel-rows

    if(yo < lo)  y = (yo & 0xfffffc00);                     // y set to non-float
    else         y = (lo & 0xfffffc00);                     // min of yo and lo
    if(yu < lu) ey = lu; else ey = yu;                      // ey to max of yu, lu
    ny = y + 1024;                                          // ny calculation

    while(y < ey) {                                         // for line-pixels in row
```

```
    // a calculation: amount under upper lineend (add)
    if(lo < y) {
        if(yo < y) a = 1024;
        else if(yo > ny) a = ((y + 512 - lo) << 10) / (yo - lo);
        else { b = yo - y; a = 1024 - (b * b) / ((yo - lo) << 1); }
    } else if(lo > ny) {
        if(yo > ny) a = 0;
        else if(yo < y) a = 1024 - ((lo - (y + 512)) << 10) / (lo - yo);
        else { b = ny - yo; a = (b * b) / ((lo - yo) << 1); }
    } else {
        if(yo < y) { b = lo - y; a = 1024 - (b * b) / ((lo - yo) << 1); }
        else if(yo > ny) { b = ny - lo; a = (b * b) / ((yo - lo) << 1); }
        else a = ny - ((lo + yo) >> 1);
    }
    // a calculation: amount under lower lineend (sub)
    if(lu > ny) {
        if(yu > ny) ;                                       // a-= 0
        else if(yu < y) a-= 1024 - ((lu - (y + 512)) << 10) / (lu - yu);
        else { b = ny - yu; a-= (b * b) / ((lu - yu) << 1); }
    } else if(lu < y) {
        if(yu < y) a-= 1024;
        else if(yu > ny) a-= ((y + 512 - lu) << 10) / (yu - lu);
        else { b = yu - y; a-= 1024 - (b * b) / ((yu - lu) << 1); }
    } else {
        if(yu < y) { b = lu - y; a-= 1024 - (b * b) / ((lu - yu) << 1); }
        else if(yu > ny) { b = ny - lu; a-= (b * b) / ((yu - lu) << 1); }
        else a-= ny - ((lu + yu) >> 1);
    }

    if(f == 1024) a>>= 2;                                   // nx - x = 1
    else a = (a * f) >> 12;                                 // first or last column

    set(x>>10,y>>10,a);                                     // set pixel
    y = ny; ny += 1024;                                     // next y and ny
}
if(nx > ex) break;                                          // stop after last row
lo = yo; lu = yu;                                           // next lo and lu
x = nx; nx+= 1024;                                          // next x and nx
f = 1024;                                                   // f = 1
```

```
        // next yo and yu calculation
        if(nx > ex) { yo = eo; yu = eu; f+= ex - nx; }          // last pixel-row
        else {
            if(x < ox)
                if(nx > ox)                                     // m-k-switch
                    yo+= (((ox - x) * k + (nx - ox) * m) >> 10);
                else yo+= k;                                    // left of ox
            else yo+= m;                                        // right of ox
            if(x < ux)
                if(nx > ux)                                     // m-k-switch
                    yu+= (((ux - x) * m + (nx - ux) * k) >> 10);
                else yu+= m;                                    // left of ux
            else yu+= k;                                        // right of ux
        }
    }
}
```

Code 4.3

Rechteck

```
int width, heigth;                              // width and heigth of image

public void fillRect(int id, int w, int h) {    // width, heigth of rect

    int lx = id + w;                            // last id of current row
    int ly = id + h * width;                    // last id of first column
    int ci = id;                                // first id of current row

    while(id < ly) {                            // for all rows
        while(id < lx) set(id++);               // set all pixels in row
        ci+= width;                             // first id of next row
        lx+= width;                             // last id of next row
        id = ci;                                // id set to next row
    }
}
```

Code 4.4

Oval

```
int width, height;                                  // width and height of image

public void fillOval(int x, int y, int w, int h) {
    if(w < 4 || h < 4) return;                      // only defined for values above 3
    int ol,ul,or,ur,d;
    int dx = 0x8000 / w;                            // dx trimmed to [-1,1] = 2
    int dy = 0x8000 / h;                            // dy trimmed to [-1,1] = 2

    if(w <= h) {                                    // standing oval
        int w2 = w >> 1;                            // half width of oval
        int lx = x + w2;                            // left x starts at middle
        int rx = lx;                                // right x starts at middle
        int oy = y * width;                         // above y starts at top
        int uy = (y + h) * width;                   // below y starts at bottom
        int xx = 0;                                 // point starts at x = 0
        int x2 = 0;                                 // quad of x
        int yy = 0x4000;                            // point starts at y = 1
        int y2 = 0x10000000;                        // quad of y

        while(oy < uy) {                            // above y is above below y
            d = x2 + y2;                            // quad of distanz
            if(d > 0x10000000) {                    // point is outside of oval
                ol = lx + oy;                       // above-left-index
                ul = lx + uy;                       // below-left-index
                or = rx + oy;                       // above-right-index
                ur = rx + uy;                       // below-right-index
                while(ol <= or) set(ol++);          // draws above line
                while(ul <= ur) set(ul++);          // draws below line
                yy-= dy;                            // point moved to next line
                y2 = yy * yy;                       // quad of y
                oy+= width;                         // above y moves down
                uy-= width;                         // below y moves up
            } else {                                // point is inside oval
                xx+= dx;                            // point moves to next row
                x2 = xx * xx;                       // quad of x
                rx++;                               // right x moves right
                lx--;                               // left x moves left
            }
        }
```

```
        if(oy == uy) {                          // if above y is below y
            ol = lx + oy;
            or = rx + oy;
            while(ol <= or) set(ol++);          // draws only one line
        }
    } else {                                    // same as above
        int h2 = h >> 1;
        int lx = x;
        int rx = x + w;
        int oy = (y + h2) * width;
        int uy = oy;
        int yy = 0;
        int y2 = 0;
        int xx = 0x4000;
        int x2 = 0x10000000;
        while(lx <= rx) {
            d = x2 + y2;
            if(d > 0x10000000) {
                ol = lx + oy; ul = lx + uy;
                or = rx + oy; ur = rx + uy;
                while(ol <= ul) { set(ol); ol+= width; }
                while(or <= ur) { set(or); or+= width; }
                xx-= dx;
                x2 = xx * xx;
                rx--; lx++;
            } else {
                yy+= dy;
                y2 = yy * yy;
                oy-= width;
                uy+= width;
            }
        }
        if(lx == rx) {
            ol = lx + oy;
            ul = lx + uy;
            while(ol <= ul) { set(ol); ol+= width; }
        }
    }
}
```

Code 4.5

Soundsystem

```
import javax.sound.sampled.*;
import java.io.*;

public class SoundSystem extends AudioInputStream implements Runnable {
    Thread thr;
    SourceDataLine dac = null;                  // Line to digital-analog-converter

    byte[] buf;                                 // intern buffer for writing to Line
    int pos, size;                              // current position in and size of buffer
    int bit, ch;                                // bits per sample, amount of channels
    double rate;                                // samplerate
    int frames   = 11025;                       // amount of frames to store in buffer
    int mul      = 4;                           // soundsystem buffersize will be size * mul
    int empty    = 10000;                       // null-samples at beginning
    double fade  = 0, df = 0.01;                // fade-in values

    double amp;                                 // maximal amplitude for bit bits
    boolean fin;                                // Thread runs until fin = true

    private void next() {                           // calculates values for buffer
        int id = 0;                                 // index for buffer
        while(id < size) {                          // for all indices of buffer
            double samplevalue = 0;                 // sample-calculation
            short v = (short)(samplevalue * fade);  // fading and casting to 16 bit

            for(int c = 0; c < ch; c++) {           // for all channels
                if(bit==8) buf[id++] = (byte)v;     // sample set for  8 bit
                else if(bit==16) {                  // sample set for 16 bit big endian
                    buf[id++] = (byte)((v & 0xff00)>>>8);
                    buf[id++] = (byte) (v & 0x00ff);
                }
            }
            if(--empty < 0 && fade < amp) fade+= df;// fade-in after empty null-samples
        }
    }
```

```
public SoundSystem(float rate, int bit, int ch) {// init with rate, bit and channels
    // initialization of AudioInputStream
    super(new ByteArrayInputStream(new byte[0]),
        new AudioFormat(AudioFormat.Encoding.PCM_SIGNED,
        rate, bit, ch, (ch*bit)>>3, (ch*bit*rate)/8, true),
        AudioSystem.NOT_SPECIFIED);
    this.rate   = rate;
    this.bit    = bit;
    this.ch     = ch;
    amp  = (1 << (bit - 1)) - 1;                // amplitude set to (2^bit) / 2 - 1
    size = (frames * bit * ch) >> 3;            // size set

    // creation of SourceDataLine with Buffersize size * mul
    DataLine.Info info;
    info = new DataLine.Info(SourceDataLine.class, this.getFormat(), size * mul);
    try {
        dac = (SourceDataLine)AudioSystem.getLine(info);
        dac.open(this.getFormat(), size * mul); }
    catch(Exception e) { System.out.println(e); }
    buf  = new byte[size];                      // buffer created
    thr  = new Thread(this);
    dac.start();                                // start of SoundSystem
}
public void play() {                            // starts playing sound
    fin = false; next(); pos = 0; thr.start();
}
public void run() {
    byte[] dat = new byte[size];          // buffer for copying from AudioInputStream
    int read = 0;                         // amount of read Bytes

    // data from AudioInputStream read and written to Line
    while (read != -1 && !fin) {
        try { read = this.read(dat, 0, buf.length); }
        catch (Exception e) { System.out.println(e); }
        if (read >= 0) { dac.write(dat, 0, read); }
    }
    dac.drain(); dac.close();                   // playing last samples and close Line
    dac = null;  thr = null;
}
public void end() { fin = true; }               // stops playing sound
```

```
// methods overwritten from AudioInputStream:

    public int available() { return Integer.MAX_VALUE; }   // Bytes that can be read

    public int read() throws IOException {                 // read Byte if framessize is 8Bit
        if(bit==8 && ch==1) { if(pos==size) { next(); pos = 0; } return buf[pos++]; }
        else throw new IOException("read(): framesize is not 8 bit.");
    }

    // read length Bytes to data with offset
    public int read(byte[] data, int offset, int length) throws IOException {
       if(length % ((bit*ch)>>3) != 0) throw new IOException("read(): illegal length value");

        int amt = Math.min(size - pos, length);            // amount of Bytes that can be read
        System.arraycopy(buf, pos, data, offset, amt);     // copy from buffer to data
        pos+= amt; if(pos >= size) { next(); pos = 0; }    // set position in buffer
        return amt;
    }
}
```

Code 5.1

```
APart[] snd = new APart[32];                          // array with max polyphony

private void next() {                                 // calculates values for buffer
    int id = 0;                                       // index for buffer
    while(id < size) {                                // for all indices of buffer
        double s[] = new double[ch];                  // values for each channel
        for(int c = 0; c < ch; c++) s[c] = 0;         // values set to zero
```

```
        for(int i = 0; i < snd.length; i++)          // for all sounds in array
            if(snd[i] != null)                       // if there is a sound
                for(int c = 0; c < ch; c++)          // for all channels

                    // value is incremented by current sound-value
                    try { s[c]+= snd[i].nextValue(c, (int)rate); }
                    catch(Exception e) { System.out.println(e); }

        for(int c = 0; c < ch; c++) {                // for all channels
            short v = (short)(s[c] * fade);          // fading and casting to short (16bit)
            if(bit==8) buf[id++] = (byte)v;          // sample set for  8 bit
            else if(bit==16) {                       // sample set for 16 bit big endian
                buf[id++] = (byte)((v & 0xff00)>>>8);
                buf[id++] = (byte) (v & 0x00ff);
            }
        }
        if(--empty < 0 && fade < amp) fade+= df;     // fade-in after empty null-samples
    }
}

// plays note with freqency freq and amplitude amp
// method is synchronized to avoid two notes at same array index
public synchronized void playNote(double freq, double amp) {

    int id = check();                                // search for free index in array
    if(id >= 0) {                                    // if valid index is found
        AWave vco = new ATri(ch, freq, 0);           // VCO with triangle-waveform
        AAmp  vca = new AAmp(vco, amp, 0);           // VCA initialized with amp
        ARamp att = new ARamp(ch, 0, 1,  100);       // attack   created
        ARamp rel = new ARamp(ch, 1, 0, 5000);       // release  created
        AContainer arh = new AContainer(ch);         // AR-curve created
        arh.append(att, 1000);                       // attack   appended
        arh.append(rel, 5000);                       // release  appended
        vca.setAmpMod(arh);                          // VCA modulated by AR
        APart prt = new APart(vca, 6000);            // soundpart created
        snd[id] = prt;                               // stored in array
    }
}
```

```
// checks for sounds that have finished, returns a free index in array
private int check() {
     int id = -1;
     for(int i = 0; i < snd.length; i++) {            // for all indices in array
          if(snd[i] == null) { id = i; continue; }    // if no sound continue
          else if(snd[i].endReached()) {              // if sound has finished
               snd[i] = null;                         // sound deleted from array
               id = i;
          }
     }
     return id;
}

public SoundSystem(float rate, int bit, int ch, double max) {
     ...
     amp  = ((1 << (bit - 1)) - 1) * max;            // mixing-level can be set with max
     ...
}
```

Code 6.2

```
AContainer snd;                                          // AObject consisting of three sounds

public void next() {
     ...
     for(int c = 0; c < ch; c++) {                       // for all channels
          double s = 0;
          try { s = snd.nextValue(c, (int)rate); }       // samples read from snd
          catch(Exception e) { System.out.println(e); }
          short v = (short)(s * fade);                   // fading and casting to short (16bit)

          if(bit==8) buf[id++] = (byte)v;                // sample set for  8 bit
          else if(bit==16) {                             // sample set for 16 bit big endian
               buf[id++] = (byte)((v & 0xff00)>>>8);
               buf[id++] = (byte) (v & 0x00ff);
          }
     }
     ...
}

public void init() {
     ...
     // AContainer(channels)
     snd = new AContainer(ch);                           // snd initialized

     // AWave(channels, Hz, phase)
     AWave lfo1 = new ASin(ch, 5.0, 0);                  // lfo1 with 5Hz
     AWave lfo2 = new ASin(ch, 7.0, 0);                  // lfo2 with 7Hz
     AWave lfo3 = new ASin(ch, 3.0, 0);                  // lfo3 with 3Hz
     AWave lfo4 = new ASin(ch, 2.0, 0);                  // lfo4 with 2Hz

     // AAmp(AObject, amplitude, offset)
     AAmp lfa1 = new AAmp(lfo1, 0.01, 0.98);             // lfa1 with range [0.98, 1.0]
     AAmp lfa2 = new AAmp(lfo2, 0.01, 0.98);             // lfa2 with range [0.98, 1.0]
     AAmp lfa3 = new AAmp(lfo3, 0.05, 0.9);              // lfa3 with range [0.90, 1.0]
     AAmp lfa4 = new AAmp(lfo4, 0.05, 0.9);              // lfa4 with range [0.90, 1.0]
```

```
    AWave dco1 = new ATri(ch, 440, 0);              // dco1 is triangle with 440Hz
    AWave dco2 = new ATri(ch, 444, 0);              // dco2 is triangle with 444Hz
    dco1.setFreqMod(lfa1);                          // dco1 modulated by lfa1
    dco2.setFreqMod(lfa2);                          // dco2 modulated by lfa2

    AAmp dca1 = new AAmp(dco1, 0.5, 0);             // dca1 with half amplitude
    AAmp dca2 = new AAmp(dco2, 0.5, 0);             // dca2 with half amplitude
    dca1.setAmpMod(lfa3);                           // dca1 modulated by lfa3
    dca2.setAmpMod(lfa4);                           // dca2 modulated by lfa4
    AAdd str = new AAdd(dca1,dca2);                 // str is mix of both sounds

    long atime = (long)( 10 * rate / 1000);         // attack -time is  10ms
    long dtime = (long)(200 * rate / 1000);         // decay  -time is 200ms
    long stime = (long)(900 * rate / 1000);         // sustain-time is 900ms
    long rtime = (long)(500 * rate / 1000);         // release-time is 500ms

    // ARamp(channels, start-value, end-value, time)
    ARamp att = new ARamp(ch, 0.0, 1.0, atime);     // attack  created
    ARamp dec = new ARamp(ch, 1.0, 0.7, dtime);     // decay   created
    ARamp rel = new ARamp(ch, 0.7, 0.0, rtime);     // release created

    AContainer adsr = new AContainer(ch);           // adsr initialized
    adsr.append(att, atime);                        // attack added
    adsr.append(dec, dtime + stime);                // decay and sustain added
    adsr.append(rel, rtime);                        // release added

    // AValue(channels, fix value)
    AObject nul = new AValue(ch, 0.0);              // nul is silence
    AAmp ton = new AAmp(str, 1.0, 0);               // ton is sound
    ton.setAmpMod(adsr);                            // ton modulated by adsr

    snd.append(nul, nul * 20);                      // silence added
    snd.append(ton, (int)rate * 2);                 // ton added three times
    snd.append(ton, (int)rate * 2);
    snd.append(ton, (int)rate * 2);
    ...
}
```

Code 5.3

```
// fmm means frequency-modulation-modul
double v1, f1, phase1 = 0, freq1 = 440;         // value, factor, phase, frequency
double v2, f2, phase2 = 0, freq2 = 500;
double v3, f3, phase3 = 0, freq3 = 440;

public void init() {
    ...
    double PIx2 = Math.PI * 2.0;                // constant value of PI * 2
    f1 = (freq1 * PIx2) / rate;                 // factor of fmm1 calculated
    f2 = (freq2 * PIx2) / rate;                 // factor of fmm2 calculated
    f3 = (freq3 * PIx2) / rate;                 // factor of fmm3 calculated
    ...
}
public void next() {
    ...
    v1 = Math.sin(phase1);                      // value  of fmm1 calculated
    v2 = Math.sin(phase2);                      // value  of fmm2 calculated
    v3 = Math.sin(phase3);                      // value  of fmm3 calculated

    phase1+= f1 * v2;                           // phase  of fmm1 calculated
    phase2+= f2 * v3;                           // phase  of fmm2 calculated
    phase3+= f3;                                // phase  of fmm3 calculated

    short v = (short)(v1 * fade);               // fading and casting to short (16bit)
    ...
}
```

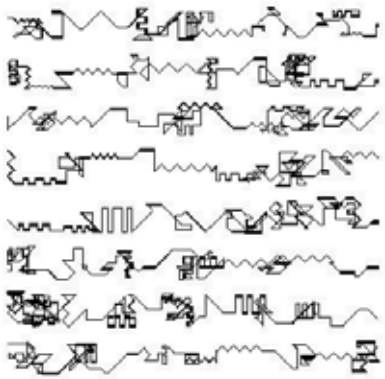
Code 5.6

```
double a2, a3;                                  // amount of modulation [0,1]
public void next() {
    ...
    phase1+= f1 * (v2 * a2 + 1.0);              // phase  of fmm1 calculated
    phase2+= f2 * (v3 * a3 + 1.0);              // phase  of fmm2 calculated
    phase3+= f3;                                // phase  of fmm3 calculated
    ...
}
```

Code 5.7

Midisystem

```
import javax.sound.midi.*;

public class Midi implements Runnable {
    Thread thr;
    boolean fin;                                        // Thread runs until fin = true

    Synthesizer syn = null;                             // intern synthesizer
    Receiver    res = null;                             // receiver of syn
    MidiEvent[] evt = null;                             // array with note-offs

    // plays note with vel and length at channel ch, which note is key
    // method is synchronized to avoid two note-offs at same index
    public synchronized void playNote(int ch, int key, int vel, int length) {

        int id = check();                               // searches for free index in array
        if(id >= 0) try {                               // if index is valid
            long time = System.currentTimeMillis();

            // note-on is created and sent
            ShortMessage on = new ShortMessage();
            on.setMessage(ShortMessage.NOTE_ON, ch, key, vel);
            res.send(on, -1);                           // -1 for no time-stamp

            // note-off is created and stored in array at index id
            ShortMessage off = new ShortMessage();
            off.setMessage(ShortMessage.NOTE_OFF, ch, key, vel);
            evt[id] = new MidiEvent(off, time + length);
        }
        catch(Exception e) { System.out.println(e); }
    }
```



Manfred Mohr
P-021/A (1969)

```
// checks for note-offs to be played, returns a free index in array
    private int check() {
        int id = -1;
        long time = System.currentTimeMillis();
        for(int i = 0; i < evt.length; i++) try {          // for all note-offs in array
            MidiEvent ev = evt[i];
            if(ev == null) { id = i; continue; }           // if no event at index continue
            if(ev.getTick() <= time) {                     // if events timestamp is over
                res.send(ev.getMessage(), -1);             // note-off sent
                evt[i] = null; id = i;                     // event deleted from array
            }
        } catch(Exception e) { System.out.println(e); }
        return id;
    }

    public Midi() {
        try {
            syn = MidiSystem.getSynthesizer();
            syn.open();                                    // synthesizer is opened
            res = syn.getReceiver();                       // receiver created from syn
            evt = new MidiEvent[syn.getMaxPolyphony()];// array size depends on poly of syn
        }
        catch(Exception e) { System.out.println(e); }
        thr = new Thread(this);
        fin = false;
        thr.start();
    }

    public void run() {
        while(!fin) { check(); pause(10); }                // note-off checking every 10ms
        res.close();
        syn.close();
        thr = null;
    }
    public void pause(int time) { try { thr.sleep(time); } catch(Exception e) {} }
    public void end() { fin = true; }                      // stops thread
}
```

Code 6.1

Maschinengrenzen

```
int width  = 640, width1  = 639;          // width  of image and width  - 1
int heigth = 480, height1 = 479;          // height of image and height - 1

int x1 = 300, y1 = 200, dx1 = 5, dy1 = 5, a1 = 0, n1 = 0;  // point 1
int x2 = 300, y2 = 200, dx2 =-5, dy2 =-5, a2 = 0, n2 = 0;  // point 2

// position=(x,y) moving-vector=(dx,dy)
// a = maximal amount of rotations of 90 degrees
// n = current amount of rotations of 90 degrees

public void draw() {
    int i1 = y1 * width + x1;                              // memory-index of point
    int i2 = y2 * width + x2;

    int c1 = dat[i1];                                      // color at index
    int c2 = dat[i2];

    if(c1 == 0x00ff0000) {                                 // if color is red
        dat[i1] = 0x00ffffff;                              // point becomes white
        snd.playNote(1, (height - y1) / 16, 50, 100);      // low note played
    }
    else if(c1 == 0x00000000) dat[i1] = 0x00ff0000;        // black becomes red
    else if(c1 == 0x00ffffff) dat[i1] = 0x00000000;        // white becomes black

    if(c2 == 0x00ff0000) {                                 // same for point 2
        dat[i2] = 0x00ffffff;
        snd.playNote(1, (height - y2) / 16, 50, 100);
    }
    else if(c2 == 0x00000000) dat[i2] = 0x00ff0000;
    else if(c2 == 0x00ffffff) dat[i2] = 0x00000000;

    x1+= dx1; y1+= dy1;                                    // point is moving
    x2+= dx2; y2+= dy2;
```

```
    if(x1 < 0 || x1 > width1) {                              // if x at border
        if(n1 == a1) { a1++; n1 = 0;                         // if max rotations reached
            dy1 = -dy1;                                      // turn around, play note
            snd.playNote(1, (hig - y1) / 8, 100, 1300 - 2 * x1);
        } else { n1++;                                       // else rotate, play note
            snd.playNote(1, (hig - y1) / 4, 100, 1300 - 2 * x1);
        }
        dx1 = -dx1;                                          // dx changing anyway
        x1+= dx1; y1+= dy1;                                  // point is moving
    }
    if(y1 < 0 || y1 > height1) {                             // same for y
        if(n1 == a1) { a1++; n1 = 0;
            dx1 = -dx1;
            snd.playNote(1, (hig - y1) / 8, 100, 1300 - 2 * x1);
        } else { n1++;
            snd.playNote(1, (hig - y1) / 4, 100, 1300 - 2 * x1);
        }
        dy1 = -dy1;
        x1+= dx1; y1+= dy1;
    }
    if(x2 < 0 || x2 > width1) {                              // same for point 2
        if(n2 == a2) { a2++; n2 = 0;
            dy2 = -dy2;
            snd.playNote(1, (hig - y2) / 8, 100, 1300 - 2 * x2);
        } else { n2++;
            snd.playNote(1, (hig - y2) / 4, 100, 1300 - 2 * x2);
        }
        dx2 = -dx2; x2+= dx2; y2+= dy2;
    }
    if(y2 < 0 || y2 > height1) {
        if(n2 == a2) { a2++; n2 = 0;
            dx2 = -dx2;
            snd.playNote(1, (hig - y2) / 8, 100, 1300 - 2 * x2);
        } else { n2++;
            snd.playNote(1, (hig - y2) / 4, 100, 1300 - 2 * x2);
        }
        dy2 = -dy2; x2+= dx2; y2+= dy2;
    }
    mis.newPixels();
}
```

Code 6.3

```
import java.applet.*;
import java.awt.*;

public class Animation extends Applet implements Runnable {
     Thread thr;                                  // the Thread
     Image img;                                   // doublebuffer
     boolean fin;                                 // Thread runs until fin = true
     int width, height;                           // width and height of image

     public void draw() {                         // painting doublebuffer
          ... }                                   // code drawing doublebuffer
     public void init() {                         // initialisation
          width  = getBounds().width;             // width  read from Applet
          height = getBounds().height;            // height read from Applet
          thr = new Thread(this);                 // Thread is initialized
          img = createImage(width, height);       // doublebuffer created
     }
     public void start() {                        // starts Runnable
          fin = false;                            // fin set to false
          thr.start(); }                          // Thread started
     public void destroy() {                      // destroys Runnable
          fin = true; }                           // fin set to true
     public void run() {
          while(!fin) {                           // while running
               draw();                            // draw doublebuffer
               repaint();                         // force call of method update()
               pause(30);                         // wait 30 ms -> 33 fps
          }
          thr = null;                             // Thread destroyed
     }
     public void update(Graphics g) {             // called by Applet through repaint()
          g.drawImage(img,0,0,this);              // draw doublebuffer to screen
     }
     public void pause(int time) {
          try { thr.sleep(time); }                // Thread pauses for time ms
          catch(Exception e) { System.out.println(e); }
     }
}
```

Code 6.4

```
int color = 0x00ffffff;                          // drawing color
int n = 3;                                       // amount of vertices
int m = 70;                                      // amount of iterations

int last = m * n;                                // amount of points
double[] x = new double[last];                   // x-coordinates of points
double[] y = new double[last];                   // y-coordinates of points
double[] f = new double[n];                      // factor for each vertex

x[0] =  20; y[0] =  20; f[0] = 0.05;             // vertices and factors set
x[1] = 300; y[1] =  20; f[1] = 0.05;
x[2] = 150; y[2] = 236; f[2] = 0.05;

x[n] = x[0];                                     // ensures closed ornament
y[n] = y[0];

int fi = 0;                                      // factor-array index

for(int i = n + 1; i < last; i++) {              // for all points
    double dx = x[i - n + 1] - x[i - n];         // dx calculated
    double dy = y[i - n + 1] - y[i - n];         // dy calculated
    x[i] = dx * f[fi] + x[i - n];                // next point calculated
    y[i] = dy * f[fi] + y[i - n];

    if(++fi == n) fi = 0;                        // increment of indices
}
for(int i = 0; i < last - 1; i++) {              // for all points
    int x1 = (int)x[i];                          // cast to int
    int y1 = (int)y[i];
    int x2 = (int)x[i+1];
    int y2 = (int)y[i+1];
    drawLine(x1, y1, x2, y2, color);             // line drawn
}
```

Code 7.2

```
int width, height, width1, heigth1;       // image properties (decremented)
int amount = width * height;
int n = 100;                              // n vertices
int m = 1000;                             // m iterations
int[] val;                                // array for iteration
int[] prs;                                // array for animation

// for black&white only
public void set(int id) {                 // set pixel with index id
    int c = dat[id];                      // color at id
    c-= 0x00040404;                       // all decremented by 4
    if(c < 0) c = 0;                      // less than 0 control
    dat[id] = c;                          // set pixel
}
// or
// for different colors
public void set(int id) {                 // set pixel with index id
    int oc = dat[id];                     // color at id
    int or = oc & 0x00ff0000;             // current red
    int og = oc & 0x0000ff00;             // current green
    int ob = oc & 0x000000ff;             // current blue
    int nr = or - 0x00030000;             // red decremented by 3
    int ng = og - 0x00000400;             // others decremented by 4
    int nb = ob - 0x00000004;
    if(nr < 0) nr = 0;                    // less than 0 control
    if(ng < 0) ng = 0;
    if(nb < 0) nb = 0;
    dat[id] = (nr|ng|nb);                 // set pixel
}

public void draw() {                      // draws next frame
    int ld,id,x1,y1,x2,y2,f,df,dx,dy;     // needed variables

    for(int i = 0; i < amount; i++)       // clear screen
        if(dat[i] != 0x00ffffff)          // clean only set pixels
            dat[i] = 0x00ffffff;
    for(int i = 0; i < (n*6); i++)        // load vertices data
        val[i] = prs[i];                  // to iterartion-array
```

```
// do iterations with val-array
for(int i = 0; i < m; i++) {                // for m iterations

    id = val.length - 6; ld = id;           // for last vertex:
    x1 = val[id++]; id++;                   // read x
    y1 = val[id++]; id++;                   // read y
    f  = val[id++]; id++;                   // read f

    id = 0;                                 // first vertex
    for(int j = 0; j < n; j++) {            // for all vertices:
        x2 = val[id++]; id++;               // read x
        y2 = val[id++]; id++;               // read y
        drawLine(x1>>10, y1>>10, x2>>10, y2>>10);
        x1+= ((x2 - x1) * f) >> 10;         // calc x for last vertex
        y1+= ((y2 - y1) * f) >> 10;         // calc y for last vertex
        f = val[id++]; id++;                // read f
        val[ld++] = x1; ld++;               // set last x
        val[ld++] = y1; ld++;               // set last y
        ld++; ld++;
        if(ld == val.length) ld = 0;        // set last to first
        x1 = x2;                            // x becomes last x
        y1 = y2;                            // y becomes last y
    }
}

// do animation with prs-array
id = 0; ld = 0;                             // first vertex
for(int i = 0; i < n; i++) {                // for all vertices:
    x1 = prs[id++];                         // read x
    dx = prs[id++];                         // read dx
    y1 = prs[id++];                         // read y
    dy = prs[id++];                         // read dy
    f  = prs[id++];                         // read f
    df = prs[id++];                         // read df
    x1+= dx;                                // values changed
    y1+= dy;
    f += df;
```

```
        // values controlled
        if(x1 < 0 || x1 > (width1 << 10)) { dx = -dx; x1+= dx; }
        if(y1 < 0 || y1 > (height1<< 10)) { dy = -dy; y1+= dy; }
        if(f < 20 || f  >  700)            { df = -df; f += df; }

        prs[ld++] = x1;                         // store x
        prs[ld++] = dx;                         // store dx
        prs[ld++] = y1;                         // store y
        prs[ld++] = dy;                         // store dy
        prs[ld++] = f;                          // store f
        prs[ld++] = df;                         // store df
    }

    mis.newPixels();
}

public void init() {
    ...
    val = new int[n*6];                         // arrays initialized
    prs = new int[n*6];

    int id = 0;                                 // preset calculation
    for(int i = 0; i < n; i++) {
        prs[id++] = (int)(Math.random() * 1024 * width);   // x
        prs[id++] = (int)(Math.random() * 1024 * 4);       // dx
        prs[id++] = (int)(Math.random() * 1024 * height);  // y
        prs[id++] = (int)(Math.random() * 1024 * 4);       // dy
        prs[id++] = (int)(Math.random() * 1024 / 2) + 20;  // f
        prs[id++] = (int)(Math.random() * 1.5) + 1;        // df
    }
}
```

Code 7.3

Mandelbrot

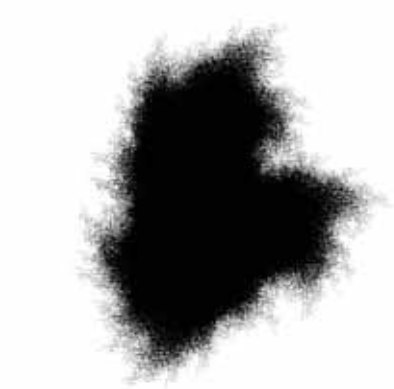
```
int width, height;                                  // width and height of image

double amin = -2;                                   // smallest real
double amax =  1;                                   // highest  real
double bmin = -1.5;                                 // smallest imag
double bmax =  1.5;                                 // highest  imag

double max = 10000000;                              // borderline
int    stp = 200;                                   // iterations
int    wht = 0x00ffffff;                            // color white
int    blk = 0x00000000;                            // color black

public void draw() {
    double w = (double)width;                       // width
    double h = (double)height;                      // height
    double adif = amax - amin;                      // real difference
    double bdif = bmax - bmin;                      // imag difference

    int v,c,id = 0;
    double a,b,ahlp,bhlp = 0;
    for(int yy = 0; yy < height; yy++) {            // for all lines
        b = bhlp / h + bmin;                        // imag calculated
        ahlp = 0;
        for(int xx = 0; xx < width; xx++) {         // for all rows
            a = ahlp / w + amin;                    // real calculated
            c = iterate(a, b);                      // iterations counted
            v = dat[id];                            // current pixel-value
            if((c & 1) == 0) {                      // if c % 2 = 0 set black
                 if(v != blk) dat[id] = blk; }
            else if(v != wht) dat[id] = wht;

            id++; ahlp+= adif;
        }
        bhlp+= bdif;
    }
    mis.newPixels();
}
```

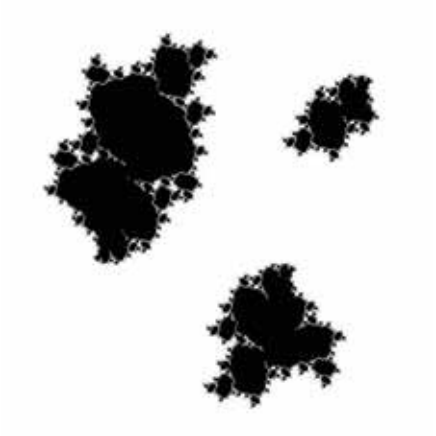
```
// does calculation for c = a + i * b
// returns amount of iterations until z > max or stp if z < max
private int iterate(double a, double b) {
     double x = a;                              // first x = a
     double y = b;                              // first y = b
     double xx,yy,xy,z;
     for(int i = 0; i < stp; i++) {             // do stp iterations
          xx = x * x;                           // x²
          yy = y * y;                           // y²
          xy = x * y;                           // xy
          x = xx - yy + a;                      // real
          y = xy + xy + b;                      // imag
          z = xx + yy;                          // z = length²
          if(z > max) return i;                 // breaks if z > max
     }
     return stp;                                // stp if z < max
}
```

Code 7.5

Turtle

```
double x = width  / 2;        // start-position
double y = heigth / 2;
double degrees = 0;           // start-angle (in degrees)
double angle;

public void forward(double length) {
     double sx = x;
     double sy = y;
     double dx = length * Math.cos(angle);
     double dy = length * Math.sin(angle);
     x+= dx;
     y-= dy;
     drawLine(sx, sy, x, y);
}
```



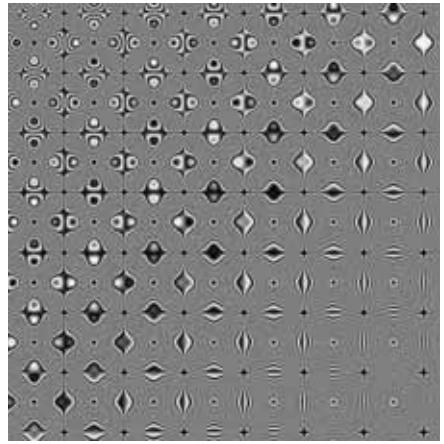


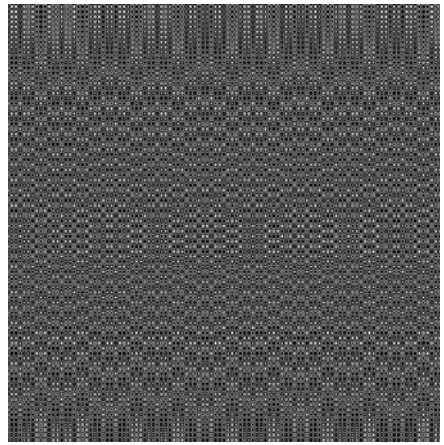
```
public void turn(double alpha) {
     degrees+= alpha;
     angle = (degrees * Math.PI) / 180.0;
}

public void turtle(int depth, double length, double rotate) {
     if(depth == 0) forward(length);
     else {
          //* draws koch-curve (rotate not used)
          turtle(depth-1, length, 0); turn(  60);
          turtle(depth-1, length, 0); turn(-120);
          turtle(depth-1, length, 0); turn(  60);
          turtle(depth-1, length, 0);
          //*/
          /* draws freehand-cirlces (rotate < 10)
          turtle(depth-1, length, 0); turn( rotate);
          turtle(depth-1, length, 0); turn(-rotate);
          turtle(depth-1, length, 0); turn(-rotate);
          //*/
          /* draws complex fractal (rotate not used)
          turtle(depth-1, length, 0); turn( 90);
          turtle(depth-1, length, 0); turn(-90);
          turtle(depth-1, length, 0); turn( 90);
          //*/
          /* draws complex 45°-fractal (rotate = 90)
          turtle(depth-1, length, rotate); turn( rotate / 2);
          turtle(depth-1, length, rotate); turn(-rotate * 2);
          turtle(depth-1, length, rotate); turn( rotate);
          //*/
          /* draws dragon-curve (rotate = 90)
          turtle(depth-1, length, 90); turn(rotate);
          turtle(depth-1, length,-90);
          //*/
     }
}
```

Code 7.7

Modulo

```
int width, heigth;                                  // width and height of image

double left  = -10;                                 // left   edge scaled to -10
double right =  10;                                 // right  edge scaled to  10
double up    = -10;                                 // upper  edge scaled to -10
double down  =  10;                                 // bottom edge scaled to  10

public void draw() {
     double x = left;
     double y = up;
     double dx = (right - left) / width;            // scaled pixel-width
     double dy = ( down -   up) / height;           // scaled pixel-heigth
     int id = 0;
     for(int j = 0; j < height; j++) {              // for all lines
          for(int i = 0; i < width; i++) {          // for all rows
               int f = (int)(x * x + y * y);        // f = x² + y²
               if(f % 2 == 1) set(id);              // if f MOD 2 is 1 set pixel
               id++;
               x+= dx;
          }
          x = left;
          y+= dy;
     }
     mis.newPixels();
}
```

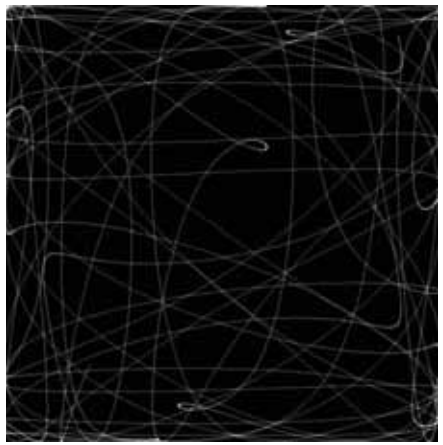




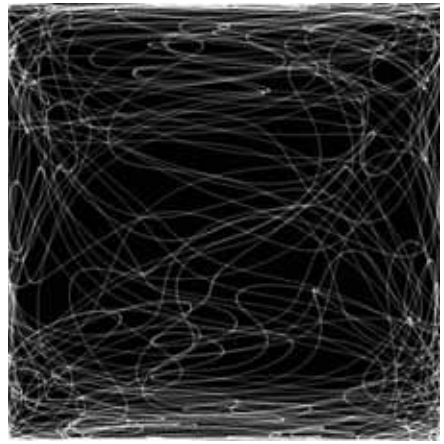
Code 7.8

Lissajousch

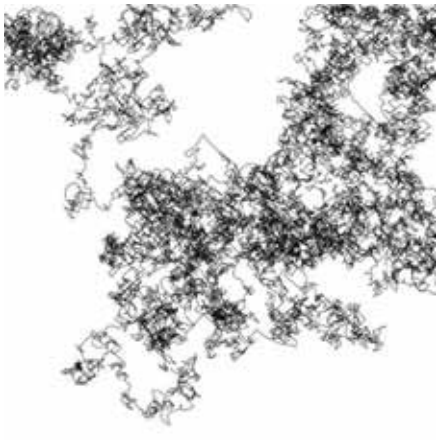
```
int width, height;                          // width and height of image
int wi2 = width  / 2 - 2;                   // half width  - 2
int hi2 = height / 2 - 2;                   // half height - 2
double pf = Math.PI / 180.0;                // pi-factor
double f1 = pf * 1.2001;                    // frequenz of signal 1
double f2 = pf * 0.6;                       // frequenz of signal 2
double f3 = pf * 2.1;                       // frequenz of modulator 1
double f4 = pf * 1.9;                       // frequenz of modulator 2
double p1 = 0;                              // phase    of signal 1
double p2 = 0;                              // phase    of signal 2
double p3 = 0;                              // phase    of modulator 1
double p4 = 0;                              // phase    of modulator 2
double v1,v2;                               // current signal values
double v3,v4;                               // last    signal values
double m1,m2;                               // current modulator values
public void set(int id) {                   // lightens pixel at id
    int c,i;

    c = dat[id] + 0x00202020;               // pixel incremented by 32
    if(c > 0x00ffffff) c = 0x00ffffff;
    dat[id] = c;
    i = id + width;                         // lower pixel
    c = dat[i] + 0x00101010;                // pixel incremented by 16
    if(c > 0x00ffffff) c = 0x00ffffff;
    dat[i] = c;
    i = id - width;                         // above pixel
    c = dat[i] + 0x00101010;                // pixel incremented by 16
    if(c > 0x00ffffff) c = 0x00ffffff;
    dat[i] = c;
    i = id + 1;                             // right pixel
    c = dat[i] + 0x00101010;                // pixel incremented by 16
    if(c > 0x00ffffff) c = 0x00ffffff;
    dat[i] = c;
    i = id - 1;                             // left pixel
    c = dat[i] + 0x00101010;                // pixel incremented by 16
    if(c > 0x00ffffff) c = 0x00ffffff;
    dat[i] = c;
}
```



```
public void draw() {
    for(int i=0; i<5000; i++) {                 // do 5000 times

        v1 = Math.sin(p1);                      // current signal 1
        v2 = Math.sin(p2);                      // current signal 2
        m1 = Math.sin(p3);                      // current modulator 1
        m2 = Math.sin(p4);                      // current modulator 2

        p1+= f1 * (m1 * 0.02 + 1.0);            // frequency-modulation
        p2+= f2 * (m2 * 0.02 + 1.0);
        p3+= f3;
        p4+= f4;

        int x1 = (int)(v1 * wi2 + wi2 + 1);     // coordinates calculated
        int y1 = (int)(v2 * hi2 + hi2 + 1);
        int x2 = (int)(v3 * wi2 + wi2 + 1);     // last signal values
        int y2 = (int)(v4 * hi2 + hi2 + 1);

        drawLine(x1,y1,x2,y2);                  // draws line

        v3 = v1; v4 = v2;                       // current becomes last
    }
    mis.newPixels();
}
```



Code 7.9



```
int width, heigth;                                      // width and height of image
double x,y;                                             // current position of line
double a,l;                                             // current angle and length
double da_min =     0 * Math.PI / 180.0;
double da_max =   360 * Math.PI / 180.0;
double dl_min =     0;
double dl_max = width;
double l_min  =     0;
double l_max  = width;
double p_chng =     0.5;                                // p for changing direction

public void draw() {
    double nx = -1, ny = -1, na = -1, nl = -1;
    while(nx < 0     || nx >= width                     // until all values are
       || ny < 0     || ny >= height                    // in valid range
       || nl < l_min || nl >  l_max) {

        double da = Math.random() * (da_max - da_min) + da_min;
        if(Math.random() < 0.5) da = -da;               // da calculation
        double dl = Math.random() * (dl_max - dl_min) + dl_min;
        if(Math.random() < 0.5) dl = -dl;               // dl calculation
        na = a + da;
        if(Math.random() < p_chng) na+= Math.PI;        // next angle
        nl = l + dl;                                    // next length

        double dx = Math.sin(na) * nl;                  // dx calculation
        double dy = Math.cos(na) * nl;                  // dy calculation
        nx = x + dx;                                    // next x
        ny = y + dy;                                    // next y
    }
    drawLine(x,y,nx,ny);                                // draws line

    x = nx;                                             // next becomes current
    y = ny;
    a = na;
    l = nl;
    mis.newPixels();
}
```

Code 7.10

Vergleich

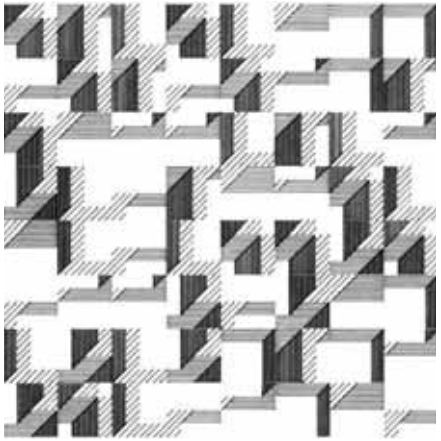
```
int width, height;                                  // width and height of images
int pixel = width * height;                         // amount of pixels

MemoryImageSource ris,tis;                          // o = original
Image             rim,tim,oim;                      // r = result
int[]             rpx,tpx,opx;                      // t = test

int amt = 0;                                        // current amount of changes
int max;                                            // maximal amount of changes

public void action(int[] px, int x, int y, int w, int h, int c) {
    // change px inside rectangle (x,y,w,h) with color c
}

public int lum(int c) {                             // luminance for color
    int r= (c & 0x00ff0000) >> 16;
    int g= (c & 0x0000ff00) >>  8;
    int b= (c & 0x000000ff);
    return Math.min(Math.min(r,g),b) + Math.max(Math.max(r,g),b);
}

public void draw() {
    int w = (int)(Math.random() * 64);              // width of change
    int h = (int)(Math.random() * 64);              // height of change
    int x = (int)(Math.random() * (width  - w));    // x-pos of change
    int y = (int)(Math.random() * (height - h));    // y-pos of change
    int r = (int)(Math.random() * 256);             // red
    int g = (int)(Math.random() * 256);             // green
    int b = (int)(Math.random() * 256);             // blue
    int c = (r<<16)|(g<<8)|b;                       // color with red, green, blue

    action(tpx,x,y,w,h,c);                          // change test-image

    int tdf = 0;                                    // test-difference
    int rdf = 0;                                    // result-difference
    int d,ol,tl,rl,id = y * width + x;
```

```
    for(int j = 0; j < h; j++) {
        for(int i = 0; i < w; i++) {
            ol = lum(opx[id]);                          // luminance of original
            tl = lum(tpx[id]);                          // luminance of test
            rl = lum(rpx[id]);                          // luminance of result
            d = ol - tl;                                // original - test
            tdf+= d * d;                                // add quad
            d = ol - rl;                                // original - result
            rdf+= d * d;                                // add quad
            id++;
        }
        id+= width-w;
    }

    if(tdf < rdf) {                                     // keep changes
        id = y * width + x;
        for(int j = 0; j < h; j++) {
            for(int i = 0; i < w; i++) {
                int val = tpx[id];
                if(rpx[id] != val) rpx[id] = val;
                id++;
            }
            id+= width - w;
        }
        amt++;                                          // increment amount of changes
    } else {                                            // drop changes
        id = y * width + x;
        for(int j = 0; j < h; j++) {
            for(int i = 0; i < w; i++) {
                int val = rpx[id];
                if(tpx[id] != val) tpx[id] = val;
                id++;
            }
            id+= width - w;
        }
    }
}
```

```
public void init() {
     rpx = new int[pixel];
     tpx = new int[pixel];
     opx = new int[pixel];

     for(int id = 0; id < pixel; id++) {              // clear test and result
          if(rpx[id] != 0) rpx[id] = 0;
          if(tpx[id] != 0) tpx[id] = 0;
     }
                                                      // get pixels of original
     PixelGrabber pg = new PixelGrabber(oim, 0, 0, width, height, opx, 0, width);
     try { pg.grabPixels(); } catch(Exception e) { System.err.println(e); }
     if ((pg.getStatus() & ImageObserver.ABORT) != 0) {
          System.err.println("image fetch aborted or errored.");
     }
}

public void run() {
     while(amt < max) {                               // amount of changes is less than max
          draw();
          ris.newPixels();
          repaint();
          pause(30);
     }
}
```

Code 7.11

Codec



Edward Zajec
ram2/9 (1969)

```
import javax.media.*;
import javax.media.control.TrackControl;
import javax.media.format.*;

public class CaptureCodec implements ControllerListener, Codec {
    int width, height, height1;                 // width and height (decremented) of frames
    int bpp;                                    // bits per pixel
    boolean flip;                               // flipped video-format
    byte[] dat;                                 // pixel-data
    Processor proc;                             // video-capture-prozessor
    Object  sync = new Object();                // for synchronisation
    boolean trns = true;                        // for synchronisation
    Format input = null, output = null;         // current video-format
    // available RGB-video-formats
    Format  inTypes[] = new Format[] { new RGBFormat() };
    Format outTypes[] = new Format[] { new RGBFormat() };

    // initialise with e.g. driver = "vfw://0" (Video for Windows)
    public CaptureCodec(String driver) throws Exception {

       // configure processor
        proc = Manager.createProcessor(new MediaLocator(driver));
        proc.addControllerListener(this);
        proc.configure();
        if(!waitForState(proc.Configured))
            throw new Exception("Processor not configured.");
        // get tracks
        proc.setContentDescriptor(null);
        TrackControl trks[] = proc.getTrackControls();
        if(trks == null)
            throw new Exception("TrackControls not obtained.");
        // get video-track (only video can be of type RGB)
        TrackControl video = null;
        for(int i = 0; i < trks.length; i++) {
            if(trks[i].getFormat() instanceof RGBFormat) {
                video = trks[i]; break; }
        }
        if(video == null)
            throw new Exception("No VideoTrack found.");
```

```
        // get video-format
        RGBFormat frmt= (RGBFormat)video.getFormat();
        width   = (int)frmt.getSize().getWidth();
        height  = (int)frmt.getSize().getHeight();
        heigth1 = height - 1;
        bpp = frmt.getBitsPerPixel();
        flip = (frmt.getFlipped() == frmt.TRUE);
        System.out.println("Video format: "+frmt);

        // realize processor
        Codec codec[] = { this };
        video.setCodecChain(codec);
        proc.prefetch();
        if (!waitForState(proc.Prefetched))
            throw new Exception("Processor not realized.");
    }

    public void captureStart() { proc.start(); }     // start capturing
    public void captureStop() {                      // stop  capturing
        proc.stop(); proc.deallocate(); proc.close(); }

    public int process(Buffer in, Buffer out) {      // process frames
        dat = (byte[])in.getData();
        ...                                          // access pixel-data here
        return BUFFER_PROCESSED_OK;
    }

    public int getIndex(int x, int y) {              // index of red-value of pixel (x,y)
        if(flip) return ((height1 - y) * width + x) * 3;
        else return (y * width + x) * 3;
    }

    // prozessor- und codec-methods
    public void open () {}
    public void reset() {}
    public void close() {}
    public void stop () { proc.stop(); proc.setMediaTime(new Time(0.0)); }
```

```
    public Format [] getSupportedInputFormats() { return inTypes; }
    public Format [] getSupportedOutputFormats(Format in) {
        if(in == null) return outTypes; else return new Format[] { in }; }

    public Format setInputFormat (Format format) { input  = format; return input; }
    public Format setOutputFormat(Format format) { output = format; return output; }

    public Object[] getControls() { return new Object[0]; }
    public Object getControl(String type) { return null; }

    boolean waitForState(int state) {                // synchronisation
        synchronized(sync) { try {
            while (proc.getState() != state && trns) sync.wait();
        } catch (Exception e) {}}
        return trns;
    }
    public void controllerUpdate(ControllerEvent evt) {
        if(evt instanceof ConfigureCompleteEvent ||
           evt instanceof RealizeCompleteEvent   ||
           evt instanceof PrefetchCompleteEvent) {
            synchronized(sync) { trns = true;  sync.notifyAll(); }
        }
        else if(evt instanceof ResourceUnavailableEvent) {
            synchronized(sync) { trns = false; sync.notifyAll(); }
        }
    //  else if(evt instanceof EndOfMediaEvent) {
            proc.stop(); proc.deallocate(); proc.close();
        }
    }
}
```

Code 7.12

```
double[] real, imag;                                    // complex values (real + i * imag)

public void dft(boolean invers) {                       // diskret fourier transformation
    int N = real.length;                                // amount of values
    double[] r = new double[N];                         // real values of result
    double[] i = new double[N];                         // imag values of result

    double h = (2.0 * Math.PI) / (double)N;             // constant h = 2*PI/N
    for(int n = 0; n < N; n++) {                        // calculate all coeffizients
        r[n] = 0; i[n] = 0;                             // result for n set to 0
        double hn = n * h;                              // constant hn = 2*PI*n/N
        for(int k = 0; k < N; k++) {                    // sum for k = 0,1,2,...N-1
            double p  = hn * k;                         // p = 2*PI*n*k/N
            double ar = real[k];                        // real value of ak
            double ai = imag[k];                        // imag value of ak
            double wr = Math.cos(p);                    // real value of w^nk
            double wi = Math.sin(p);                    // imag value of w^nk (negative)
            if(invers) {                                // invers dft
                r[n]+= ar * wr - ai * wi;
                i[n]+= ar * wi + ai * wr;
            } else {                                    // dft (wi = -wi)
                r[n]+= ai * wi + ar * wr;
                i[n]+= ai * wr - ar * wi;
            }
        }
    }
    for(int n = 0; n < N; n++) {                        // store result
        if(!invers) {                                   // dft (cn / N)
            real[n] = r[n] / (double)N;
            imag[n] = i[n] / (double)N;
        } else { real[n] = r[n]; imag[n] = i[n]; }
    }
}
```

Code 8.1

FFT rekursiv

```
int P;                                                  // N = 2^P
int N = 1 << P;                                         // amount of values
double h = (2.0 * Math.PI) / (double)N;                 // constant h = 2*PI/N
double[] real, imag;                                    // complex values (real + i * imag)

public void rfft(boolean invers) {                      // fast fourier transformation
    recFFT(invers, N, 1, 0);                            // for N values with n=1, start-index=0
}

private void recFFT(boolean invers, int amt, int n, int start) {
    double hn = (double)n * h;                          // constant hn = 2*PI*n/N
    int half = amt >> 1;                                // half of amount
    if(half > 1) {                                      // array can be splittet
        for(int k = 0; k < half; k++) {                 // k = 0,1,...N/2-1
            int n1 = k + start;                         // index 1
            int n2 = n1 + half;                         // index 2
            double p  = hn * (double)k;                 // p = 2*PI*n*k/N
            double r1 = real[n1];                       // real value at index 1
            double i1 = imag[n1];                       // imag value at index 1
            double r2 = real[n2];                       // real value at index 2
            double i2 = imag[n2];                       // imag value at index 2
            real[n1]  = r1 + r2;                        // store addition of both real
            imag[n1]  = i1 + i2;                        // store addition of both imag
            double ar = r1 - r2;                        // subtraktion of both real
            double ai = i1 - i2;                        // subtraktion of both imag
            double wr = Math.cos(p);                    // real value of w^nk
            double wi = Math.sin(p);                    // imag value of w^nk (negative)
            if(invers) {                                // invers fft
                real[n2] = ar * wr - ai * wi;           // store real of product
                imag[n2] = ar * wi + ai * wr;           // store imag of product
            } else {                                    // fft (wi = -wi)
                real[n2] = ai * wi + ar * wr;           // store real of product
                imag[n2] = ai * wr - ar * wi;           // store imag of product
            }
        }
        recFFT(invers, half, n << 1, start);            // divide chain again with 2*n
        recFFT(invers, half, n << 1, start + half);
```

```
    } else {                                        // no more splitting possible
        int n1 = start;                             // index 1
        int n2 = n1 + 1;                            // index 2
        double r1 = real[n1];                       // real value at index 1
        double i1 = imag[n1];                       // imag value at index 1
        double r2 = real[n2];                       // real value at index 2
        double i2 = imag[n2];                       // imag value at index 2
        if(!invers) {                               // fft (cn / N)
            real[n1] = (r1 + r2) / (double)N;       // store addition of both real
            imag[n1] = (i1 + i2) / (double)N;       // store addition of both imag
            real[n2] = (r1 - r2) / (double)N;       // store subtraktion of both real
            imag[n2] = (i1 - i2) / (double)N;       // store subtraktion of both imag
        } else {                                    // invers fft
            real[n1] = r1 + r2;                     // store
            imag[n1] = i1 + i2;
            real[n2] = r1 - r2;
            imag[n2] = i1 - i2;
        }
    }
    if(n == 1) for(int n1 = 0; n1 < amt; n1++) {    // bitreversal mapping
        int n2 = n1;                                // index are equal
        for(int i = 0; i < (P >> 1); i++) {         // reverse bit-order
            int l_shf = P - 1 - i;
            int l_val = 1 << l_shf;
            int r_val = 1 << i;
            int l_bit =(n1 & l_val) >> (l_shf - i);
            int r_bit = n1 & r_val;
            if(l_bit != r_bit) n2^= l_val | r_val;
        }
        if(n2 > n1) {                               // if not changed jet
            double r1 = real[n1];                   // real value at index 1
            double i1 = imag[n1];                   // imag value at index 1
            double r2 = real[n2];                   // real value at index 2
            double i2 = imag[n2];                   // imag value at index 2
            real[n1] = r2; real[n2] = r1;           // change
            imag[n1] = i2; imag[n2] = i1;
        }
    }
}
```

Code 8.2

FFT iterativ

```
void ifft(boolean invers) {
    int half = N >> 1;
    int id,amt,n1,n2 = 0;
    double a,p,wr,wi,pr,pi,tr,ti,r1,i1,r2,i2;

    for(n1 = 0; n1 < N; n1++) {                    // bitreversal mapping
        if(n2 > n1) {
            tr = real[n2]; real[n2] = real[n1]; real[n1] = tr;
            ti = imag[n2]; imag[n2] = imag[n1]; imag[n1] = ti;
        }
        id = half;
        while(id >= 1 && n2 >= id) { n2-= id; id>>= 1; }
        n2+= id;
    }
    half = 1;                                      // index differenz (half of amt)
    while(half < N) {                              // not all performed now
        amt = half << 1;                           // amount of values to calc
        p = Math.PI / (double)half;                // trigonometric recurrence
        if(!invers) p = -p;                        // fft (-p)
        a = Math.sin(0.5 * p);
        pr = -2.0 * a * a; pi = Math.sin(p);
        wr = 1.0;                                  // realof w^0
        wi = 0.0;                                  // imag of w^0
        for(id = 0; id < half; id++) {
            for(n1 = id; n1 < N; n1+= amt) {       // index 1
                n2 = n1 + half;                    // index 2
                r1 = real[n1]; i1 = imag[n1];      // Danielson-Lanczos formula
                r2 = real[n2]; i2 = imag[n2];
                tr = wr * r2 - wi * i2;
                ti = wr * i2 + wi * r2;
                real[n1] = r1 + tr; imag[n1] = i1 + ti;
                real[n2] = r1 - tr; imag[n2] = i1 - ti;
            }
            tr = wr;
            wr+= wr * pr - wi * pi;                // trigonometric recurrence
            wi+= wi * pr + tr * pi;
        }
        half = amt;
    }
```

```
    if(!invers) for(n1 = 0; n1 < N; n1++) {          // fft (cn / N)
        real[n1]/= (double)N;
        imag[n1]/= (double)N;
    }
}
```

Code 8.3

FFT kombiniert

```
public void fft(boolean invers) {
    rrecFFT(invers, N, 0);                           // call rekursive FFT
    double t;
    int id,n1,n2 = 0, half = N >> 1;

    if(!invers) {                                    // bitreversal mapping and dividing
        double Nd = (double)N;
        for(n1 = 0; n1 < N; n1++) {
            if(n2 > n1) {
                t = real[n2]; real[n2] = real[n1]; real[n1] = t / Nd;
                t = imag[n2]; imag[n2] = imag[n1]; imag[n1] = t / Nd;
            } else { real[n1]/= Nd; imag[n1]/= Nd; }
            id = half;
            while(id >= 1 && n2 >= id) { n2-= id; id>>= 1; }
            n2+= id;
        }
    }
    else for(n1 = 0; n1 < N; n1++) {                 // bitreversal mapping only
        if(n2 > n1) {
            t = real[n2]; real[n2] = real[n1]; real[n1] = t;
            t = imag[n2]; imag[n2] = imag[n1]; imag[n1] = t;
        }
        id = half;
        while(id >= 1 && n2 >= id) { n2-= id; id>>= 1; }
        n2+= id;
    }
}
```

```
private void rrecFFT(boolean invers, int amt, int start) {
    double a,p,r1,i1,r2,i2,ar,ai,wr,wi,pr,pi;
    int n1,n2,k,half = amt >> 1;

    if(half > 1024) {                                   // if above critical value
        p = Math.PI / (double)half;                     // trigonometric recurrence
        a = Math.sin(0.5 * p);
        pr = -2.0 * a * a;
        pi = Math.sin(p);
        wr = 1.0;                                       // real of w^0
        wi = 0.0;                                       // imag of w^0
        n1 = start; n2 = n1 + half;
        r1 = real[n1]; i1 = imag[n1];
        r2 = real[n2]; i2 = imag[n2];
        real[n1] = r1 + r2;                             // no need to multiply
        imag[n1] = i1 + i2;
        real[n2] = r1 - r2;
        imag[n2] = i1 - i2;
        for(k = 1; k < half; k++) {                     // for all other values
            n1++; n2++;
            a = wr;
            wr+= wr * pr - wi * pi;                     // trigonometric recurrence
            wi+= wi * pr +  a * pi;
            r1 = real[n1]; i1 = imag[n1];
            r2 = real[n2]; i2 = imag[n2];
            real[n1] = r1 + r2;
            imag[n1] = i1 + i2;
            ar = r1 - r2;
            ai = i1 - i2;
            if(invers) {
                real[n2] = ar * wr - ai * wi;
                imag[n2] = ar * wi + ai * wr;
            } else {
                real[n2] = ai * wi + ar * wr;
                imag[n2] = ai * wr - ar * wi;
            }
        }
        rrecFFT(invers, half, start);                   // divide in upper part
        rrecFFT(invers, half, start + half);            // divide in lower part
    }
```

```
else {                                          // iterative part
    int sh, end = start + amt;
    while(half > 2) {                           // for all but last two steps
        p = Math.PI / (double)half;             // trigonometric recurrence
        a = Math.sin(0.5 * p);
        pr = -2.0 * a * a;
        pi = Math.sin(p);
        wr = 1.0;                               // real of w^0
        wi = 0.0;                               // imag of w^0
        sh = start + half; n2 = sh;
        for(n1 = start; n1 < end; n1+= amt) {   // for all that need no multiply
            r1 = real[n1]; i1 = imag[n1];
            r2 = real[n2]; i2 = imag[n2];
            real[n1] = r1 + r2;
            imag[n1] = i1 + i2;
            real[n2] = r1 - r2;
            imag[n2] = i1 - i2;
            n2+= amt;
        }
        for(k = start + 1; k < sh; k++) {       // for all other values
            a = wr;
            wr+= wr * pr - wi * pi;             // trigonometric recurrence
            wi+= wi * pr +  a * pi;
            for(n1 = k; n1 < end; n1+= amt) {
                n2 = n1 + half;
                r1 = real[n1]; i1 = imag[n1];
                r2 = real[n2]; i2 = imag[n2];
                real[n1] = r1 + r2;
                imag[n1] = i1 + i2;
                ar = r1 - r2;
                ai = i1 - i2;
                if(invers) {
                    real[n2] = ar * wr - ai * wi;
                    imag[n2] = ar * wi + ai * wr;
                } else {
                    real[n2] = ai * wi + ar * wr;
                    imag[n2] = ai * wr - ar * wi;
                }
            }
        }
```

```
            amt = half;                                  // changes chunk size
            half>>= 1;
        }
        double r3,r4,i3,i4, zr1,zr2,zi1,zi2;
        n1 = start; n2 = n1;
        while(n1 < end) {                                // last two steps
            r1 = real[n1]; i1 = imag[n1++];
            r3 = real[n1]; i3 = imag[n1++];
            r2 = real[n1]; i2 = imag[n1++];
            r4 = real[n1]; i4 = imag[n1++];
            zr1 = r1 + r2; zi1 = i1 + i2;
            zr2 = r3 + r4; zi2 = i3 + i4;
            real[n2]   = zr1 + zr2;
            imag[n2++] = zi1 + zi2;
            real[n2]   = zr1 - zr2;
            imag[n2++] = zi1 - zi2;
            zr1 = r1 - r2; zi1 = i1 - i2;
            if(invers) { zr2 = i4 - i3; zi2 = r3 - r4; }
            else       { zr2 = i3 - i4; zi2 = r4 - r3; }
            real[n2]   = zr1 + zr2;
            imag[n2++] = zi1 + zi2;
            real[n2]   = zr1 - zr2;
            imag[n2++] = zi1 - zi2;
        }
    }
}
```

Code 8.4

FFT real

```
void real_fft(boolean invers) {                          // only works with real-values
    int n1 = 2, n2 = N - 1, N2 = N>>1;                   // indices and half of N
    double r1,i1,r2,i2;
    double wr,wi,pr,pi,a,p;
    p = Math.PI / (double)N2;
    if(!invers) { p = -p; sfft(false); }                 // first do fft if not invers
    a = Math.sin(0.5 * p);
    pr = -2.0 * a * a;
    pi = Math.sin(p);
    wr = pr + 1.0;
    wi = pi;
    while(n1 < N2) {                                     // for n=2,3...N/2-1 except N/4
        r1 = real[n1] + real[--n2];                      // separation of values
        if(!invers) {
            i2 = real[n2++] - real[n1++];
            r2 = real[n1  ] + real[n2  ];
        } else {
            i2 = real[n1++] - real[n2++];
            r2 =-real[n1  ] - real[n2  ];
        }

        i1 = real[n1--] - real[n2];
        real[n1++] = r1 + wr * r2 - wi * i2;
        real[n1++] = i1 + wr * i2 + wi * r2;
        real[n2--] =      wr * i2 + wi * r2 - i1;
        real[n2--] = r1 - wr * r2 + wi * i2;
        a = wr;
        wr = wr * pr - wi * pi + wr;
        wi = wi * pr +  a * pi + wi;
    }
    real[n1]*= 2.0;                                      // separation for n=N/4
    real[n2]*=-2.0;
    r1 = real[0];                                        // separation for n=0 and N/2
    r2 = real[1];
```

```
    if(!invers) {
        real[0] = 2.0 * (r1 + r2);
        real[1] = 2.0 * (r1 - r2);                    // cr(N/2) as ci(1)
    } else {
        real[0] = r1 + r2;
        real[1] = r1 - r2;                            // cr(N/2) as ci(1)
        sfft(true);                                   // do inv. fft at last (if invers)
    }
}
public void sfft(boolean invers) {
    double t;
    int id,n1 = 0, n2 = 0, N2 = N >> 1;
    recFFT(invers, N, 0);                             // call rekursive FFT

    if(!invers) {                                     // bitreversal mapping and dividing
        double Nd = (double)(N<< 1);
        while(n1 <N) {
            if(n2 > n1) {
                t = real[n2]; real[n2++] = real[n1]; real[n1++] = t / Nd;
                t = real[n2]; real[n2--] = real[n1]; real[n1++] = t / Nd;
            } else { real[n1++]/= Nd; real[n1++]/= Nd; }
            id = N2;
            while(id >= 2 && n2 >= id) { n2-= id; id>>= 1; }
            n2+= id;
        }
    }
    else while(n1 < N) {                              // bitreversal mapping only
        if(n2 > n1) {
            t = real[n2]; real[n2++] = real[n1]; real[n1++] = t;
            t = real[n2]; real[n2--] = real[n1]; real[n1++] = t;
        } else n1+=2;
        id = N2;
        while(id >= 2 && n2 >= id) { n2-= id; id>>= 1; }
        n2+= id;
    }
}
```

```
private void recsFFT(boolean invers, int amt, int start) {
    double a,p,r1,i1,r2,i2,ar,ai,wr,wi,pr,pi;
    int n1,n2,k,half = amt >> 1;

    if(half > 1024) {                                  // if above critical value
        p = Math.PI / (double)half;                    // trigonometric recurrence
        a = Math.sin(p);
        pr = -2.0 * a * a;
        pi = Math.sin(2.0 * p);
        wr = 1.0;                                      // real of w^0
        wi = 0.0;                                      // imag of w^0
        n1 = start; n2 = n1 + half;
        r1 = real[n1]; i1 = real[n1+1];
        r2 = real[n2]; i2 = real[n2+1];
        real[n1++] = r1 + r2;                          // no need to multiply
        real[n1++] = i1 + i2;
        real[n2++] = r1 - r2;
        real[n2++] = i1 - i2;
        for(k = 2; k < half; k+= 2) {                  // for all other values
            a = wr;
            wr+= wr * pr - wi * pi;                    // trigonometric recurrence
            wi+= wi * pr +  a * pi;
            r1 = real[n1]; i1 = real[n1+1];
            r2 = real[n2]; i2 = real[n2+1];
            real[n1++] = r1 + r2;
            real[n1++] = i1 + i2;
            ar = r1 - r2;
            ai = i1 - i2;

            if(invers) {
                real[n2++] = ar * wr - ai * wi;
                real[n2++] = ar * wi + ai * wr;
            } else {
                real[n2++] = ai * wi + ar * wr;
                real[n2++] = ai * wr - ar * wi;
            }
        }
        recsFFT(invers, half, start);                  // divide in upper part
        recsFFT(invers, half, start + half);           // divide in lower part
    }
```

```
else {                                          // iterative part
    int sh, end = start + amt;
    while(half > 4) {                           // for all but last two steps
        p = Math.PI / (double)half;             // trigonometric recurrence
        a = Math.sin(p);
        pr = -2.0 * a * a;
        pi = Math.sin(2.0 * p);
        wr = 1.0;                               // real of w^0
        wi = 0.0;                               // imag of w^0
        sh = start + half; n2 = sh;
        for(n1 = start; n1 < end; n1+= amt) {   // for all that need no multiply
            r1 = real[n1]; i1 = real[n1+1];
            r2 = real[n2]; i2 = real[n2+1];
            real[n1  ] = r1 + r2;
            real[n1+1] = i1 + i2;
            real[n2  ] = r1 - r2;
            real[n2+1] = i1 - i2;
            n2+= amt;
        }
        for(k = start + 2; k < sh; k+= 2) {     // for all other values
            a = wr;
            wr+= wr * pr - wi * pi;             // trigonometric recurrence
            wi+= wi * pr +  a * pi;
            for(n1 = k; n1 < end; n1+= amt) {
                n2 = n1 + half;
                r1 = real[n1]; i1 = real[n1+1];
                r2 = real[n2]; i2 = real[n2+1];
                real[n1  ] = r1 + r2;
                real[n1+1] = i1 + i2;
                ar = r1 - r2;
                ai = i1 - i2;
                if(invers) {
                    real[n2  ] = ar * wr - ai * wi;
                    real[n2+1] = ar * wi + ai * wr;
                } else {
                    real[n2  ] = ai * wi + ar * wr;
                    real[n2+1] = ai * wr - ar * wi;
                }
            }
        }
```

```
            amt = half;                                  // changes chunk size
            half>>= 1;
        }
        double r3,r4,i3,i4, zr1,zr2,zi1,zi2;
        n1 = start; n2 = n1;
        while(n1 < end) {                                // last two steps
            r1 = real[n1++]; i1 = real[n1++];
            r3 = real[n1++]; i3 = real[n1++];
            r2 = real[n1++]; i2 = real[n1++];
            r4 = real[n1++]; i4 = real[n1++];
            zr1 = r1 + r2; zi1 = i1 + i2;
            zr2 = r3 + r4; zi2 = i3 + i4;
            real[n2++] = zr1 + zr2;
            real[n2++] = zi1 + zi2;
            real[n2++] = zr1 - zr2;
            real[n2++] = zi1 - zi2;
            zr1 = r1 - r2; zi1 = i1 - i2;
            if(invers) { zr2 = i4 - i3; zi2 = r3 - r4; }
            else       { zr2 = i3 - i4; zi2 = r4 - r3; }
            real[n2++] = zr1 + zr2;
            real[n2++] = zi1 + zi2;
            real[n2++] = zr1 - zr2;
            real[n2++] = zi1 - zi2;
        }
    }
}
```

Code 8.5

FFT mehrdimensional

```
int amt;                                                    // amount of values
int[] P = {8,8,2};                                          // dimension size = (2^P[n])
double[] data;                                              // array for all complex values

public void init() {
     amt = 1;                                               // calculates amount of values
     for(int i = 0; i < P.length; i++) amt<<= P[i];
     data = new double[amt<<1];                             // array size = amt * 2
}
public void dim_fft(boolean invers) {                       // moredimensional fft and ifft
     int dim,ndim=P.length;
     int n1,n2,n3,n4,n5,np1,np2,np3,np4,np5;                // indices
     int id,k1,k2,N,npi,rem;
     double ti,tr,p,wi,pi,pr,wr,a,r1,r2,i1,i2;

     npi=1;
     for (dim = ndim-1; dim >= 0; dim--) {                  // over all dimensions
          N = 1 << P[dim];
          rem = amt / (N * npi);
          np1 = npi << 1;
          np2 = np1 * N;
          np3 = np2 * rem;
          n4 = 1;
          for(n2 = 1; n2 <= np2; n2+= np1) {                // bitreversal mapping
               if(n2 < n4) {
                    for(n1 = n2; n1 <= n2 + np1 - 2; n1+= 2) {
                         for(n3 = n1; n3 <= np3; n3+= np2) {
                              n5 = n4 + n3 - n2;
                              tr = data[n3]; data[n3--] = data[n5]; data[n5--] = tr;
                              tr = data[n3]; data[n3++] = data[n5]; data[n5++] = tr;
                         }
                    }
               }
               id = np2 >> 1;
               while(id >= np1 && n4 > id) { n4-= id; id>>= 1; }
               n4+= id;
          }
          np4 = np1;
```

```
        while(np4 < np2) {
            np5 = np4 << 1;
            p = Math.PI / (np5 / np1); if(invers) p = -p;
            a = Math.sin(p);
            pr = -2.0 * a * a; pi=Math.sin(2.0 * p);
            wr = 1.0; wi = 0.0;
            for(n1 = 1; n1 <= np1 - 1; n1+= 2) {          // multiplikation not needed
                for(n2 = n1; n2 <= np3; n2+= np5) {
                    n4 = n2 - 1; k1 = n2 + np4; k2 = k1 - 1;
                    i1 = data[n2]; r1 = data[n4];
                    i2 = data[k1]; r2 = data[k2];
                    data[k2] = r1 - r2; data[k1] = i1 - i2;
                    data[n4] = r1 + r2; data[n2] = i1 + i2;
                }
            }                                             // multiplikation needed
            for(n3 = np1 + 1; n3 <= np4; n3+= np1) {
                a = wr;
                wr = wr * pr - wi * pi + wr;
                wi = wi * pr +  a * pi + wi;
                for(n1 = n3; n1 <= n3 + np1 - 2; n1+= 2) {
                    for(n2 = n1; n2 <= np3; n2+= np5) {
                        n4 = n2 - 1; k1 = n2 + np4; k2 = k1 - 1;
                        i1 = data[n2]; r1 = data[n4];    // Danielson-Lanczos
                        i2 = data[k1]; r2 = data[k2];
                        tr = wr * r2 - wi * i2;
                        ti = wr * i2 + wi * r2;
                        data[k2] = r1 - tr; data[k1] = i1 - ti;
                        data[n4] = r1 + tr; data[n2] = i1 + ti;
                    }
                }
            }
            np4 = np5;
        }
        npi*= N;
    }
    if(!invers) {                                         // division if fft
        tr = (double)amt;
      for(n1 = 0; n1 < (amt<<1); n1++) data[n1]/= tr;
    }
}
```

MIDI-Tabellen

Controller

Regler (MSB)	Regler (LSB)	Schalter	Channel Mode Messages
0 Bank Select MSB	32 Bank Select LSB	64 Sustain pedal (Damper)	121 Reset All Controllers
1 Modulation MSB	33 Modulation LSB	65 Portamento	122 Local Control Off
2 Breath Controller MSB	34 Breath Controller LSB	66 Sustenuto	123 All Notes Off
3 nicht definiert MSB	35 nicht definiert LSB	67 Soft pedal	124 Omni Mode Off
4 Foot pedal MSB	36 Foot pedal LSB	68 nicht definiert	125 Omni Mode On
5 Portamento Time MSB	37 Portamento Time LSB	69 Hold 2	126 Mono Mode On
6 DataEntry MSB	38 DataEntry LSB	70 nicht definiert	127 Poly Mode On
7 Volume MSB	39 Volume LSB	71 Harmonic Content (XG)	
8 Balance MSB	40 Balance LSB	72 Release Time (XG)	
9 nicht definiert MSB	41 nicht definiert LSB	73 Attack Time (XG)	
10 Panpot MSB	42 Panpot LSB	74 Brightness (XG)	
11 Expression MSB	43 Expression LSB	75 - 79 nicht definiert	
12-15 nicht definiert MSB	44 - 47 nicht definiert LSB	80 General purpose 5	
16 General purpose 1 MSB	48 General purpose 1 LSB	81 General purpose 6	
17 General purpose 2 MSB	49 General purpose 2 LSB	82 General purpose 7	
18 General purpose 3 MSB	50 General purpose 3 LSB	83 General purpose 8	
19 General purpose 4 MSB	51 General purpose 4 LSB	84 - 90 nicht definiert	
20 - 31 nicht definiert MSB	52-63 nicht definiert LSB	91 External Effects Depth	
		92 Tremolo Depth	
		93 Chorus Depth	
		94 Celeste Depth	
		95 Phaser Depth	
		96 Data Increment	
		97 Data Decrement	
		98 Non registered parameter LSB	
		99 Non registered parameter MSB	
		100 Registered parameter LSB	
		101 Registered parameter MSB	
		102 - 120 nicht definiert	

Tabelle 6.3

Programs

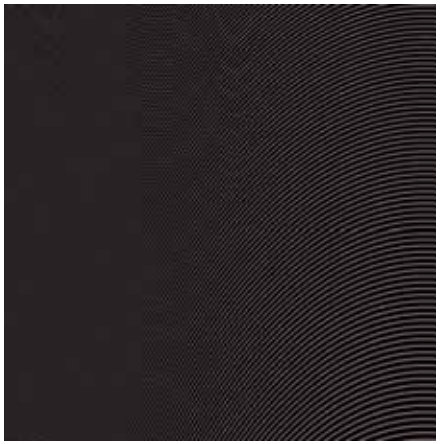
Piano	Chromatic Percussion	Organ (Orgel)	Guitar (Gitarre)
0 Acoustic Grand Piano (Flügel)	8 Celesta	16 Drawbar Organ (Hammond)	24 Acoustic Guitar (Nylon)
1 Bright Acoustic Piano (Klavier)	9 Glockenspiel	17 Percussive Organ	25 Acoustic Guitar (Steel - Stahl)
2 Electric Grand Piano	10 Music Box (Spieluhr)	18 Rock Organ	26 Electric Guitar (Jazz)
3 Honky-tonk	11 Vibraphone	19 Church Organ (Kirchenorgel)	27 Electric Guitar (clean)
4 Electric Piano 1 (Rhodes)	12 Marimba	20 Reed Organ (Drehorgel)	28 Electric Guitar (muted)
5 Electric Piano 2 (Chorus)	13 Xylophone	21 Accordion	29 Overdriven Guitar
6 Harpsichord (Cembalo)	14 Tubular Bells (Röhrenglocken)	22 Harmonica	30 Distortion Guitar (verzerrt)
7 Clavi (Clavinet)	15 Dulcimer (Hackbrett)	23 Tango Accordion (Bandeon)	31 Guitar harmonics
Bass	Strings (Streicher)	Ensemble	Brass (Blechbläser)
32 Acoustic Bass	40 Violin (Violine - Geige)	48 String Ensemble 1	56 Trumpet (Trompete)
33 Electric Bass (finger)	41 Viola (Viola - Bratsche)	49 String Ensemble 2	57 Trombone (Posaune)
34 Electric Bass (pick - gezupft)	42 Cello (Violoncello - Cello)	50 SynthString 1	58 Tuba
35 Fretless Bass (bundloser Bass)	43 Contrabass (Violone)	51 SynthString 2	59 Muted Trumpet (gedämpft)
36 Slap Bass 1	44 Tremolo Strings	52 Choir Aahs	60 French Horn
37 Slap Bass 2	45 Pizzicato Strings	53 Voice Oohs	61 Brass Section (Bläsersatz)
38 Synth Bass 1	46 Orchestral Harp (Harfe)	54 Synth Voice	62 SynthBrass 1
39 Synth Bass 2	47 Timpani (Pauke)	55 Orchestra Hit	63 SynthBrass 2
Reed (Holzbläser)	Pipe (Flöten)	Synth Lead (Solo)	Synth Pad (Flächen)
64 Soprano Sax	72 Piccolo	80 Square (Rechteck)	88 New Age
65 Alto Sax	73 Flute (Flöte)	81 Sawtooth (Sägezahn)	89 Warm
66 Tenor Sax	74 Recorder (Blockflöte)	82 Calliop	90 Polysynth
67 Baritone Sax	75 Pan Flute	83 Chiff	91 Choir
68 Oboe	76 Blown Bottle	84 Charang	92 Bowed (Streicher)
69 English Horn	77 Shakuhachi	85 Voice	93 Metallic
70 Bassoon (Fagott)	78 Whistle (Pfeifen)	86 Fifths	94 Halo
71 Clarinet	79 Ocarina	87 Bass + Lead	95 Sweep
Synth Effects	Percussion	Percussive	Sound Effects
96 Rain (Regen)	104 Sitar Ethnik	112 Tinkle Bell (Glocke)	120 Guitar Fretless Noise
97 Soundtrack	105 Banjo	113 Agogo	121 Breath Noise (Atem)
98 Crystal	106 Shamisen	114 Steel Drums	122 Seashore (Meeresbrandung)
99 Atmosphere	107 Koto	115 Woodblock	123 Bird Tweet
100 Brightness	108 Kalimba	116 Taiko Drum	124 Telephone Ring
101 Goblins	109 Bag Pipe (Dudelsack)	117 Melodic Tom	125 Helicopter
102 Echoes	110 Fiddle	118 Synth Drum	126 Applause
103 Sci-Fi (Science Fiction)	111 Shanai	119 Reverse Cymbal	127 Gun Shot (Gewehrschuss)

Tabelle 6.5

Drums

35 Acoustic Bass Drum	47 Low Mid Tom	59 Ride Cymbal 2	71 Short Whistle
36 Bass Drum 1	48 Hi Mid Tom	60 Hi Bongo	72 Long Whistle
37 Side Stick	49 Crash Cymbal	61 Low Bongo	73 Short Guiro
38 Acoustic Snare	50 High Tom	62 Mute Hi Conga	74 Long Guiro
39 Hand Clap	51 Ride Cymbal 1	63 Open Hi Conga	75 Claves
40 Electric Snare	52 Chinese Cymbal	64 Low Conga	76 Hi Wood Block
41 Low Floor Tom	53 Ride Bell	65 High Timbale	77 Low Wood Block
42 Closed Hi Hat	54 Tambourine	66 Low Timbale	78 Mute Cuica
43 High FloorTom	55 Splash Cymbal	67 High Agogo	79 Open Cuica
44 Pedal Hi Hat	56 Cowbell	68 Low Agogo	80 Mute Triangle
45 Low Tom	57 Crash Cymbal 2	69 Cabasa	81 Open Triangle
46 Open Hi Hat	58 Vibraslap	70 Maracas	

Tabelle 6.6



Sysex

Hersteller	ID (Hex.)	Hersteller	ID (Hex.)	Hersteller	ID (Hex.)	Hersteller	ID (Hex.)
360 Systems	00:00:1C	Fender	08	Lake Butler Sound	00:00:0D	Siel	21
ADA	0D	Fostex	51	Lexicon	06	Simmons	38
Adam-Smith	17	Fujitsu Elect.	4B	Lowrey	16	Solton	26
Akai	47	Future Lab	00:00:28	Marquis Music	00:00:1E	Sonus	00:00:12
AKG Acoustics	0A	Garfield Elec.	0E	Matsushita	50	Sony	4C
Alesis	00:00:0E	GEM Generalmusic	35	Meisosha	49	Soundcraft	39
Allen&Heath Brenell	00:00:1A	General MIDI	7E	Miditemp	00:20:0D	Sound Creation	00:00:0F
Apple Computer	11	GreyMatter Response	12	Mimetics	13	Soundtracs	2E
Art	1A	Gulbransen	09	Moog	04	SouthernMusic Sys.	00:00:0C
Artisyn	00:00:0A	Harmony Systems	19	Moridaira	45	Southworth	28
Audio Vertr. Strüven	2C	Harrison Systems	00:00:27	Music Logic Systems	00:00:02	Spatial Sound	00:00:18
Axxess	00:00:20	Hohner	24	Neve	2D	SSL	2B
Baldwin	1B	Hoshino Gkki	4A	New England Digital	00:00:09	Steinberg	3A
Bontempi	20	IDP	02	Nissin Onpa	4D	Steinway	09
Boss	41	Indian Valley	00:00:22	Oberheim	10	Stepp	23
Breakaways Tech.	00:00:25	Intercont. Electr.	31	Octave-Plateu	03	Stypher	00:00:06
CAE	00:00:26	IntegratedMedia Syst.	00:00:11	Opcode	00:00:16	Symetrix	00:00:01
Cannon Res. Corp.	00:00:2B	Inventronics	1D	Orban	00:00:21	Synthax	22
Casio	44	IOTA Systems	00:00:08	Othertech	00:00:04	System Product	4F
Chetah Marketing	36	IVL Technologies	00:00:0B	PAIA	00:00:03	Teac	4E
Clarity	1F	Japan Victor	48	Palm Tree Inst.	14	Temp. Acuity Prod.	00:00:13
Clavia Digital Instr.	33	Jellinghaus	27	Passac	20	Triton	00:00:23
C.T.M.	37	JEN	2A	Passport Design	05	Twister	25
Digital Music Corp.	00:00:07	JL Cooper	15	Peavey	00:00:1B	Voyce Music	0B
DOD / DigiTech	00:00:10	Kamiya Studio	46	Perfect Fretworks	00:00:15	Waldorf	3E
Drawmer	32	Kawai	40	Piano Disc	00:00:2A	Waveframe Corp.	0C
Dynacord	30	Key Concepts	1E	PPG	29	Wenger Corp.	00:00:1D
Elka	2F	K-Muse	00:00:05	Quasimidi	3F	Wersi	3B
E-mu Systems	18	KMX	00:00:19	Rane Corp.	00:00:17	Yamaha	43
Ensoniq	0F	Korg	42	Rocktron	00:00:29	Zeta Systems	00:00:1F
Eventide	1C	KTI	00:00:24	Roland	41	Zoom	52
Fairlight	14	Kurzweil	07	Sequential	01	Zyklus	34

Tabelle 6.8

Links und Literatur

Kunstmagazine

art: Das Kunstmagazin, Gruner + Jahr AG & Co KG
PARNASS: Kunstmagazin, Parnass Verlag Ges.m.b.H.
du: Zeitschrift für Kultur, du Verlag AG

Kunstgeschichte

[1] Karin Guminski: Kunst am Computer, 2002 Dietrich Reimer Verlag GmbH
[10] Heike Piehler: Die Anfänge der Computerkunst, 2002 dot-Verlag Frankfurt/Main

Kunstbände

[2] Klaus Albrecht Schröder: Victor Vasarely, 1992 Prestel-Verlag München
[4] natürlich künstlich: Artificial Life, 2001 Jovis Verlag GmbH
Kunstverein in Hamburg: Formalism. Modern Art, today, 2004 Hatje Cantz Verlag Ostfildern-Ruit

Medienkunst

ZKM | Zentrum für Kunst und Medientechnologie Karlsruhe, 1997 Prestel-Verlag München - New York
ZKM: Hardware Software Artware, 2000 Cantz Verlag Ostfildern
Ars Electronica 2003: CODE - The Language of our Time, 2003 Hatje Cantz Verlag Ostfildern

Netzkunst

du: net.art. Rebellen im Internet, November 2000, Heft Nr. 711
Tilman Baumgärtel: net.art Materialien zur Netzkunst, 1999 Verlag für moderne Kunst Nürnberg
Tilman Baumgärtel: net.art 2.0 Neue Materialien zur Netzkunst, 2001 Verlag für moderne Kunst Nürnberg

Pioniere der Computerkunst

[3] Herbert W. Franke: Phänomen Kunst, 1967 Verlag Nadolski Stuttgart
[3] Herbert W. Franke: Wege zur Computerkunst, 1995 Edition die Donau hinunter Wien - St. Peter am Wimberg
[3] Herbert W. Franke: Leonardo 2000, Kunst im Zeitalter des Computers, 1987 suhrkamp taschenbuch
[15] Karl-Heinz Werler: Programmierte Phantasie, 1991 Akademie Verlag GmbH Berlin

Mathematik

[21] John Briggs, F. David Peat: Die Entdeckung des Chaos, 1990 Carl Hanser Verlag München Wien
[20] Karl-Heinz Becker, Michael Dörfler: Dynamical systems and fractals, 1986 Friedr. Vieweg & Sohn Braunschweig
[17] Elbert Oran Brigham: FFT, 1995 R. Oldenbourg Verlag GmbH München
[18] Norbert Herrmann: Höhere Mathematik, 2004 Oldenbourg Wissenschaftsverlag GmbH München Wien

Programmierung

[13] Fritz Jobst: Programmieren in Java, 1996 Carl Hanser Verlag München Wien
[16] Horst M. Eidenberger, Roman Divotkey: Medienverarbeitung in Java, 2004 dpunkt.verlag
[19] Numerical Recipes in C: The Art of Scientific Computing, 1988-1992 Cambridge University Press

Informatik

[22] Donald Hearn, M. Pauline Baker: Computer Graphics with OpenGL, 2003 Prentice Hall
[23] Andrew S. Tanenbaum, Maarten van Steen: Distributed Systems: Principles and Paradigms, 2002 Prentice-Hall

Internet

[8] www.dam.org	Digital Art Museum Berlin
[9] www.aec.at	Ars Electronica Center Linz
www.zkm.de	Zentrum für Medientechnologie Karlsruhe
www.guggenheim.org	Guggenheimmuseum
[6] www.thing.org	The Thing
www.thing.at	The Vienna Thing
[7] www.0100101110101101.org	
[5] 404.jody.org	
[11] groupc.net	
[12] www.evolutionzone.com	
www.abstraction-now.at	
www.computerfinearts.com	
www.bitforms.com	
deluxe-arts.org.uk	
www.rhizome.org	
runme.org	
dian-network.com	
java.sun.com	Offizielle Javaseite
[24] www.cfxweb.net	Effektprogrammierung in Java
[14] michael.bliem.at.tc	ALab, Move, Demove, u.a.