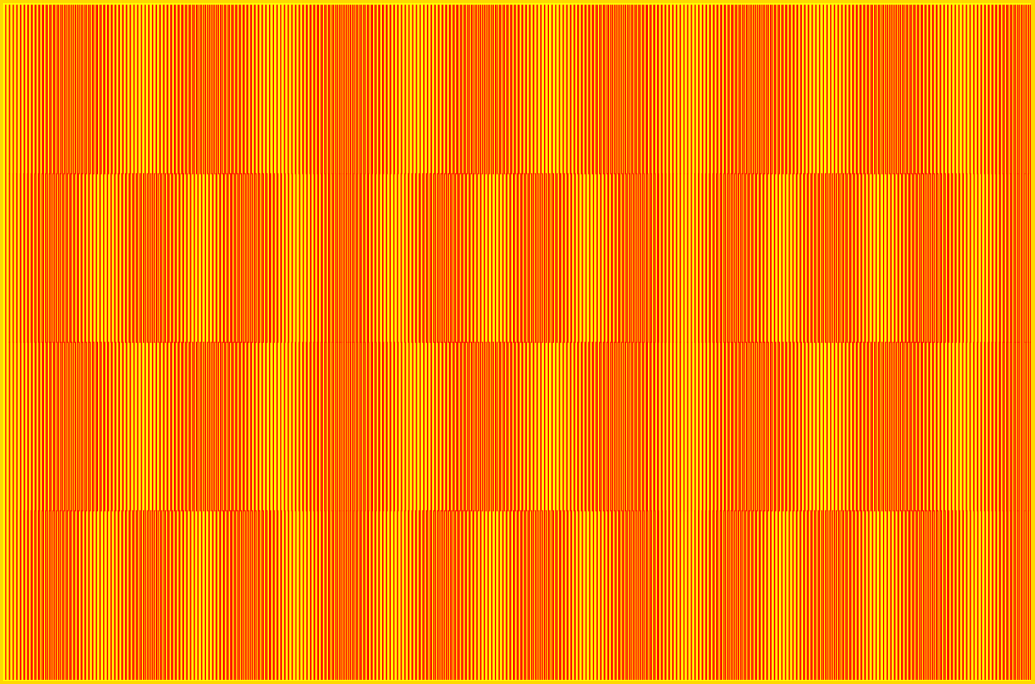
CODIERTE KUNST



Kunst Programmieren mit Snap!

Joachim Wedekind

Codierte Kunst

Kunst Programmieren mit Snap!

Joachim Wedekind

Joachim Wedekind:
Codierte Kunst. Kunst Programmieren mit Snap!

Autor: Dr. Joachim Wedekind
Eschenweg 26
72076 Tübingen

Das vorliegende Werk wurde sorgfältig erarbeitet. Dennoch übernimmt der Autor keine Haftung für die Richtigkeit von Angaben, Hinweisen und Tipps sowie für eventuelle Druckfehler.

Webseite zum Buch: http://codiertekunst.joachim-wedekind.de
Programme zum Download unter:
http://codiertekunst.joachim-wedekind.de/projekte

Titelbild: Hommage à Steller: Rhythmen 90:92:93

2018 Tübingen
© für das Gesamtwerk beim Autor
Grafische Gestaltung und Satz: Joachim Wedekind

Impressum: http://joachim-wedekind.de/impressum/

Dieses Werk von Joachim Wedekind steht unter einer Creative Commons Lizenz 4.0: Namensnennung, Nicht Kommerziell, Keine Bearbeitung.
Über diese Lizenz hinausgehende Erlaubnisse können Sie beim Autor erfragen.



INHALTSVERZEICHNIS

Vorwort 1
Einleitung 4
Teil I. Grundlagen 11
1. Computerkunst 11
1.1 Grundelemente der Computerkunst 14
1.2 Algorithmische Kunst 20
1.3 Vom Plotter zum Zeichenroboter 21
1.4 Geschichte der Computerkunst 23
2. Recoding & Remixing 24
2.1 Recoding 24
2.2 Remixing 26
2.3 Grafikelemente und Programmierkonzepte 27
3. Was ist Snap! ? 29
3.1 Ein Blick zurück: Die Anfänge von Logo 30
3.2 Konstruktionismus und Visuelle Programmierung 32
4. Snap! starten 35
4.1. Orientierung auf der Bühne 37
5. Erste Schritte in Snap! 40
5.1 Es tut sich was ... im Direktmodus 40
5.2 Eins nach dem anderen 42
5.3 Wiederholungen 45
5.4 Hommage à Steller: Rhythmen 45
5.5 Modularisierung und Erweiterbarkeit 49
5.6 Verallgemeinerung 53
5.7 Hommage à Vilder: Variationen über 9 Quadrate 55
5.8 Kontrollstrukturen 59
5.9 Hommage à Riley: Movement in Squares 61
6. Gibt's nicht? Gibt's nicht! 66
7. Alles schön bunt hier 70
7.1 Hommage à Mihich: Painting #207 75
8. Alles Zufall ... die simulierte Intuition 80
9. Der Figurenbaukasten 84
9.1 Punkte 84
9.2 Hommage à Hugonin: Binary Rhythm 85
9.3 Hommage à Müller: 64/6 88
9.4 Hommage à Struycken: Komputerstrukturen 88
9.5 Linien und Strecken 96

9.6 Hommage à Sol LeWitt: Bands of Color 98
9.7 Streckenzüge und Polygone 99
9.8 Hommage à Noll: Gaussian Quadratic 102
9.9 Krumme Linien 105
9.10 Hommage à Nees: Kreisbogen 105
9.11 Flächen 108
9.12 Exkurs: Das m*n-Raster 108
9.13 Hommage à Molnar: 25 Quadrate 109
Teil II: Recoding & Remixing Computerkunst 113
10. Vom Analogen Zum Digitalen 113
10.1 Hommage à Laposky 113
10.2 Recoding Lissajous 114
10.3 Remixing Lissajous (I): Sinus-Cosinus-Additionen 119
10.4 Remixing Lissajous (II): Anhängsel 122
11. Muster aus Figuren 127
11.1 Hommage à Aaron Marcus: Urbane Nova 128
11.2 Hommage à Komura: Optical Effect of Inequality 130
11.3 Hommage à Bartnig: 256 133
11.4 Hommage à Mohr: Scratch Code 135
11.5 Exkurs: Stempeln statt Malen 138
11.6 Hommage à Sýkora: Schwarz-Weiß-Struktur 140
11.7 Remixing Schwarz-Weiß-Strukturen 142
11.8 Hommage à Coqart: Structured Squares 145
11.9 Exkurs: Prozeduren als Daten 146
12. Hommage à Georg Nees: Schotter 149
12.1 Recoding Schotter (I) 151
12.2 Recoding Schotter (II) 153
12.3 Remixing Schotter: Formen und Farben 154
13. Ein Block ist ein Block ist ein Block 161
13.1 Exkurs: lokale vs. globale Variable 166
14. Hommage à Kolomyjec: Boxes 168
15. Hommage à Nees: Die Sprache G 177
15.1 Exkurs: Felder und Listen 182
15.2 Gestörte Gewebe 184
16. Hommage à Nake: Einfache Grafiken 189
16.1 Rechteckschraffuren 189
16.2 Labyrinth 192
16.3 Geradenscharen 195
17. Hommage à Michael Noll: Stilstudien 200

17.1 Recoding Noll, Remixing Riley: Sinusoide 200
17.2 Recoding Noll, Remixing Mondrian: Composition With Lines....202
18. Hommage à Vera Molnar: Unordnungen 209
18.1 Recoding & Remixing: Ein Prozent Unordnung....209
18.2 Recoding & Remixing: (Un)Ordnungen....212
19. Hommage à Schneeberger: SNE KAO....214
20. Hommage à Sýkora: Linien Malen....219
21. Alles in Bewegung 223
21.1 Farbinstallation: Malen ... Wischen ... Malen....224
21.2 Exkurs: Das Arbeiten mit Objekten (I) - Sprites 227
21.3 Remixing Schotter (II) - Klone 231
21.4 Hommage à CTG: Return to Square....236
21.5 Hommage à Csuri: Hummingbird....240
21.6 Hommage à Resch: Netzstrukturen 242
Teil III: Kunst Programmieren ?!....245
22. Noch mehr Computerkunst 245
22.1 ASCII-Art: Texte als Grafiken 245
22.2 Hommage à Barbadillo: Bogenbilder....248
22.3 Hommage à Bartnig: Quadratentwicklung....250
22.4 Hommage à Beckmann: DONKO Generator 251
22.5 Hommage à Beyls: Grid Based Systems 251
22.6 Hommage à Csuri & Schaffer: Feeding Time....254
22.7 Hommage à CTG: Random Windows 255
22.8 Hommage à Franke: Quadrate und Kreise....258
22.9 Hommage à Gruppo N: Dynamic Visions 259
22.10 Hommage à Korneder: Zentren....262
22.11 Hommage à Land & Cohen: Flowers 265
22.12 Hommage à Nash: Triangle 9....266
22.13 Hommage à Roubaud: Kreissegmente....267
22.14 Hommage à Schott: Zerkiemung 269
22.15 Hommage à Sonderegger: Schmetterling....272
22.16 Hommage à Steller: Tektonik....273
22.17 Hommage à Strand u.a.: Drehflächen....274
22.18 Hommage à von Graevenitz: Große horizontale Verteilung 276
23. Moiré-Muster....278
24. Kinetische Kunst 284
24.1 Hommage à Gerhard von Graevenitz....284
24.2 Hommage à van Weeghel: Dynamic Structures 286
24.3 Exkurs: Gekoppelte Objekte....288

24.4 Hommage à Talman: k25 293
25. Codierte Kunst 297
25.1 Hommage à Josef Albers: Interaction of Color 299
25.2 Hommage à Morellet: Zufall 302
25.3 Ein kleiner Exkurs: Malen mit Maus und Finger 306
25.4 Hommage à Jackson Pollock: Drip Painting 308
26. Blick über den Tellerrand 315
26.1 Mathematik und Kunst 315
26.2 Biologie und Kunst 319
26.3 Malmaschinen und Malroboter 320
26.4 Software-Werkzeuge für die Kunstproduktion 322
26.5 Exkurs: Live-Coding mit Snap! 324
27. Fazit und Ausblick 327
27.1 Medienkunst: Interaktion & Multimedia & Vernetzung 327
Literatur 330
Bildernachweise 336
Anhang A: Tipps und Tricks für Snap! 337
Anhang B: Die Befehlsblöcke von Snap! 343
Anhang C: Blockbibliothek 354
Personenregister 365
Sachregister 367
Befehlsregister 369

VORWORT

In diesem Buch werde ich zwei auf den ersten Blick unabhängige Themen miteinander verbinden. Es ist zum einen die *Computerkunst (Computerart)* und zum anderen das *Programmieren*, genauer das Programmieren mit der Computersprache *Logo* bzw. mit *Snap!*, eine ihrer Nachfolgeversionen. Meine immer intensivere Beschäftigung mit beiden Themen führte neben ersten Ausstellungsaktivitäten und einem Blog zum Thema *Digital Art* auch dazu, meine Erfahrungen und Ideen in diesem Buch *Codierte Kunst* zu dokumentieren.

Beim *Codieren* geht es ums Programmieren, bei der *Computerkunst* um einen Zweig der zeitgenössischen Kunst. Mit dem Buchtitel soll deutlich werden, dass es sich nicht um eine systematische Einführung in das Programmieren handelt, sondern um einen Ansatz, das Programmieren über das Erstellen von Bildern in der Tradition der frühen Computerkunst und anderer Kunstrichtungen zu lernen.

Voraussetzung ist ein Interesse an dieser Kunstrichtung, d.h. an überwiegend gegenstandslosen Motiven, abstrakten geometrischen Formen und Farbfiguren. Praktisch geht es um die programmtechnischen Möglichkeiten, die gezeigten und andere solcher Bildbeispiele nachzubilden. Im Mittelpunkt steht deshalb die Analyse, das Nachempfinden, das Nachprogrammieren (*Recoding*) und daran anknüpfend das Umsetzen eigener Varianten (*Remixing*).

Dazu wird die visuelle Programmierumgebung Snap! verwendet, wobei keinerlei Programmierkenntnisse vorausgesetzt werden. Mit diesem kostenfrei zugänglichen Werkzeug können Sie nach dem Durcharbeiten dieser Einführung eigene Ideen weiter verfolgen. Sie finden damit einen Einstieg in das hochaktuelle, spannende Feld der *Medienkunst*. Vielleicht werden Ihre eigenen Ergebnisse ja einmal Bestandteil einer Medienkunst-Ausstellung? Mit dieser Einführung möchte ich es allen Interessierten - gerade auch solchen ohne Vorkenntnisse und Programmiererfahrungen - ermöglichen, sehr rasch entsprechende Werke zu realisieren.

Es wird etliche LeserInnen beruhigen, dass die Projekte dabei (fast) ohne Mathematik auskommen. Logisches Denken reicht! Ich beschränke mich darauf, immer genau die informatischen Konzepte einzuführen, die für ein konkretes Projekt gebraucht werden, und Wege aufzuzeigen, wie auch komplexere, anspruchsvolle und ästhetische Grafiken und Animationen umgesetzt werden können. Diese Themenorientierung unterscheidet sich sicher von den üblichen Einführungen in das Programmieren, bei denen zumeist theorielastiger und systematischer vorgegangen wird.

Snap! ist für diesen Ansatz besonders geeignet, weil es die Erstellung anspruchsvoller Grafiken hervorragend unterstützt und durch die interaktiven Möglichkeiten und direkten Rückmeldungen ein experimentelles Vorgehen geradezu herausfordert. Ein ganz zentraler Bestandteil von Snap! ist die sogenannte *Schildkrötengrafik* (*Turtle-Graphics*, im

Deutschen oft auch als *Igelgrafik* bezeichnet). Mit ihr werden alle grafischen Komponenten der vorgestellten Projekte umgesetzt werden.

Die Snap!-Programmierumgebung ist bestens geeignet zum Selbststudium und natürlich auch für Anfängerkurse mit (sehr) jungen Adressaten. Snap! steht damit ganz in der Tradition der Programmiersprache Logo, die ursprünglich für Kinder und Schulen konzipiert wurde. Ich selbst habe bei der programmtechnischen Umsetzung der hier vorgestellten Beispiele etliche Sprachelemente von Logo besonders schätzen gelernt. Es hat mir großen Spaß gemacht, jeweils möglichst kompakte und gleichzeitig flexible Programme zu entwickeln, wodurch die dabei entstehenden Grafiken sich leicht vielfältig variieren lassen.

Im Buch werde ich die dafür notwendigen Grundkonzepte behandeln und in den Umgang mit Logo bzw. Snap! einführen. Dabei hoffe ich, ein wenig von der Faszination zu vermitteln, die ich selbst bei der Erzeugung der Grafiken empfunden habe. Da ich weder Informatiker noch Kunsthistoriker bin (sondern Unterrichtstechnologe und Mediendidaktiker), dürfen Sie keine informatisch geprägte Einführung in das Programmieren erwarten; auch bei den kunstgeschichtlichen Hintergründen muss ich auf die einschlägige Literatur verweisen (die Sie im umfangreichen Literaturverzeichnis finden). Herausgekommen ist so eine Kombination aus Bildband und Programmierleitfaden.

Danksagungen

Beim Verfassen des Buches habe ich in unterschiedlichen Phasen wertvolle Unterstützung erfahren. Mein Dank gilt dabei zuallererst Jens Mönig, dem „Chefentwickler“ von Snap!. Jens hat nicht nur meine Fragen und Anregungen zu Snap! immer ungewöhnlich rasch beantwortet bzw. umgesetzt; ohne seine ansteckende Begeisterung hätte ich dieses Buch wohl nicht in dieser Form in Angriff genommen.

Besonderer Dank geht auch an Werner Holtmann und Bernd Mühlemeier, beide schon geschätzte Wegbegleiter in früheren beruflichen Projekten. Ihre kritische Durchsicht des Manuskripts und ihre konstruktiven Anregungen haben wesentlich zur Verbesserung von Struktur und Lesbarkeit des Textes beigetragen.

Der Bremer Gruppe um Frieder Nake, namentlich Susan Grabowski, danke ich für wichtige Hinweise und Impulse zur theoretischen Auseinandersetzung mit der Computerkunst. Sie erlaubten mir eine bessere Einordnung der Computerkunst und meines Ansatzes des *Recoding & Remixing*. Rainer Schneeberger danke ich für die Überlassung des Originalcodes von SNE COMP ART.

Inge Wedekind war nicht nur - wie so oft - eine genaue Korrekturleserin, die mich vor den schlimmsten stilistischen und grammatikalischen Ausrutschern bewahrt hat, sie hat auch geduldig meine Fokussierung auf die Arbeit am Buch toleriert. Dafür danke ich ihr sehr.

Für die verbliebenen Fehler und Unzulänglichkeiten bleibe ich natürlich selbst verantwortlich.

Tübingen, März 2018 Joachim Wedekind

EINLEITUNG

Wenn ich in diesem Band die *Computerkunst* gemeinsam mit dem *Programmieren* behandle, dann geht das auf zwei Auslöser zurück, die mich im Jahr 2015 auf die Verbindung der beiden Themen brachten:

Auslöser I: Fünfzig Jahre Computerkunst. Vor etwas über fünfzig Jahren, im Februar 1965, fand an der Universität Stuttgart die weltweit erste Ausstellung zur Computerkunst statt. Gezeigt wurden computergenerierte Bilder des Ingenieurs Georg Nees. Auf dieses Jubiläum wurde ich im Vorfeld durch Hinweise in einschlägigen Zeitschriften aufmerksam und so konnte ich - wie andere auch[1] - rechtzeitig zum Jubiläumsjahr mit einer kleinen Ausstellung (dokumentiert bei digitalart) einen Beitrag leisten.

Auslöser II: Fünfzig Jahre Programmiersprache Logo. Ebenfalls ziemlich genau vor fünfzig Jahren wurde die Programmiersprache Logo vorgestellt. Durch *Apple Logo*, die erste Version für den damals sehr populären 8-Bit-Rechner Apple II, lernte ich selbst 1982 erstmals Logo näher kennen. Ende der 80er Jahre arbeitete ich dann an einem Lehrerfortbildungsprojekt mit Studienmaterialien zum Thema *"Lehren und Lernen mit dem Computer"*. Diese boten u.a. erprobte Unterrichtseinheiten mitsamt der entsprechenden Software. Eine davon war der Studienbrief *„Schildkrötengrafik - Einfaches Programmieren von Grafiken“*, den ich zusammen mit Hans Rauch verfasst habe, der seinerseits auch die Software TURTLE-LLC dafür entwickelt hatte (Rauch & Wedekind, 1989). Auf diese Erfahrungen kann ich nun Jahre später zurückgreifen.

Beide Aspekte bestimmen den Charakter dieses Buches. Teil I behandelt zu beidem die *Grundlagen*. Das Kapitel *Computerkunst* geht auf die Grundelemente dieser Kunstrichtung ein. Im Kapitel *Recoding & Remixing* wird der Aspekt des Programmierens hinzugefügt, denn das *Recoden*, das Nachprogrammieren, und das *Remixen*, das Abwandeln und Erweitern von Vorlagen, sind inzwischen (nicht unumstrittene) Bestandteile der Medienkultur geworden.

In den darauf folgenden Kapiteln werde ich Sie an den Umgang mit unserem Werkzeugkasten - der *Snap!*-Programmierumgebung - heranführen, der das Nachempfinden ausgewählter Werke der frühen Computerkunst und anderer Kunstrichtungen erlaubt, um davon ausgehend eigenständige Variationen und Weiterentwicklungen zu erstellen. Snap! ist dabei zwar ein notwendiges Werkzeug, aber dennoch nur ein Element des gesamten Entwurfs- und Gestaltungsprozesses. Die Vermittlung der Programmierkonzepte erfolgt trotzdem nicht beiläufig, sondern immer experimentell und ergebnisorientiert. Deshalb ist dies keine systematische Einführung in die Programmierung mit Snap!, sondern das Kennenlernen und Einsetzen derjenigen Sprachelemente, die uns das Erstellen eigener grafischer Exponate erlauben. So werden in diesem Rahmen etli-

[1] so das Digital Art Museum (DAM) Berlin mit AESTHETICA - 50 Jahre computergenerierte Kunst

che Sprachelemente und Möglichkeiten, die Snap! darüber hinaus bietet, nicht behandelt[2].

Ein zentraler Bestandteil von Snap! ist die *Schildkrötengrafik*, durch die Sie mit einem überschaubaren Befehlssatz zunächst einfache, aber dann auch zunehmend komplexe Grafiken programmieren können. Sie lernen dabei die Formulierung von Algorithmen und deren Codierung. Sie werden nach und nach zentrale Programmierkonzepte kennenlernen, wie

- die Wiederholung, Modularisierung und Erweiterbarkeit, Verallgemeinerung und Kontrollstrukturen (im Kapitel *Erste Schritte ...*),
- die Rekursion (im Kapitel *Ein Block ist ein Block ist ein Block ...*),
- Felder und Listen (im Kapitel *Die Sprache G*),
- das Arbeiten mit Objekten in Form von Sprites und Klonen sowie die Kommunikation zwischen Objekten (im Kapitel *Alles in Bewegung*),
- die Kopplung von Objekten (im Kapitel *Kinetische Kunst*) und
- Prozeduren als Daten (im Kapitel *Coqart: Quadratische Strukturen*).

Die hierbei gewonnenen Erfahrungen können bei Bedarf leicht auf andere moderne, höhere Programmiersprachen übertragen werden.

Mit den Komponenten von Snap! und der Schildkrötengrafik wird nach und nach ein *ästhetisches Laboratorium* aufgebaut (Nees, 1995, S. 7). Mit dem Computer steht uns nämlich eine Versuchswerkstatt zur Verfügung, die es uns erlaubt, grafisch zu experimentieren, bis es uns gelingt, gezielt - oft genug aber auch als Ergebnis eines glücklichen Zufalls - ästhetische Objekte zu erzeugen. Nees (a.a.O., S. 183) hat das Vorgehen einmal als *Gestaltfallenstellen* charakterisiert: *„Irgendwann wird man durch einen außergewöhnlichen Fang belohnt“.*

Den Einstieg bildet zumeist der Nachvollzug geeigneter „Vor“-Bilder. Für solche konzentrieren wir uns auf die Bilder der frühen Computerkunst. Genauso geeignet sind aber auch Werke aus anderen Kunstrichtungen, wie dem Minimalismus, der Op Art, dem Konstruktivismus oder der Konkreten Kunst. Bei der programmtechnischen Umsetzung lernen wir dann gleich noch die wichtigsten Prinzipien der Programmierung kennen. Das ist die Voraussetzung, um nach den Anregungen durch die Vorbilder auch eigene Vorstellungen zu entwickeln und dann ebenfalls programmtechnisch umzusetzen.

Darauf bezogen werden folgende Themen in den weiteren Kapiteln behandelt:

Die Programmierumgebung Snap!, ihre Herkunft und ihre typischen Merkmale stelle ich in *Was ist Snap!* vor.

[2] Am ehesten findet sich ein vergleichbares Vorgehen in einigen Büchern zu Scratch (einem Vorgänger von Snap!). Dort werden Programmierkonzepte meist anhand der Entwicklung interaktiver Spiele eingeführt (vgl. z.B. Breen, 2015 oder Immler, 2016), ohne eine thematische Fokusssierung wie in diesem Band.

Arbeiten können Sie mit Snap! natürlich erst, wenn Sie es lauffähig auf Ihrem Rechner verfügbar haben. Das ist besonders einfach, weil Snap! im Webbrowser läuft und deshalb keine spezielle Installation verlangt. Im Kapitel *Snap! starten* erfahren Sie alles dazu Notwendige.

Eigentlich kann man mit Snap! gleich loslegen. Es braucht nicht viele Vorkenntnisse, um die ersten Erfolgserlebnisse zu erzielen. Haben Sie Snap! erfolgreich gestartet, fangen wir damit an und steigen im Kapitel *Erste Schritte* ... direkt mit einfachen Beispielen ein. Anhand der Schildkrötengrafik werden erste grundlegende Programmierkonzepte (wie *Wiederholungen, Modularisierung, Erweiterbarkeit, Verallgemeinerungen* und *Kontrollstrukturen*) eingeführt. So lernen Sie einen ersten Ausschnitt des Befehlssatzes von Snap! kennen. Bereits bei diesen Beispielen kann ich mich an direkten Vorbildern aus der frühen Computerkunst orientieren, denn erstaunlicherweise lassen sie sich bereits mit der Kenntnis einiger weniger Befehle nachvollziehen.

Damit auch absolute Programmieranfänger diesen Einstieg erfolgreich bewältigen, habe ich ihn sehr kleinschrittig gestaltet. Wenn bereits Programmiererfahrungen vorliegen, kann dieses Kapitel sehr schnell abgearbeitet werden. In den Folgekapiteln wird diese Kleinschrittigkeit nach und nach abnehmen. Letztlich sollen mit dem Programmieren Möglichkeiten erschlossen werden, sich kreativ auszudrücken: „*With Snap!* [im Original: with Scratch, JW]*, our goal is for [...] people to become fluent with coding — not only learning the mechanics and concepts of coding, but also developing their own voice and their ability to express their ideas [and ...] to work on meaningful projects"* (Resnick & Siegel, 2015).

Da Snap! nicht für alle Erfordernisse passende Befehle bietet, andererseits aber grundsätzlich fast beliebig erweiterbar ist, werden im Kapitel *Gibt's nicht? Gibt's nicht!* einige Funktionalitäten als Spracherweiterungen erstellt. Sie werden in den Folgekapiteln mehrfach benötigt und eingesetzt werden.

Beim *Remixing* werden farbige Darstellungen in hoher Auflösung eine große Rolle spielen. Im Kapitel *Alles schön bunt hier* ... wird deshalb das Farbmodell von Snap! vorgestellt. Etliche Befehle erlauben dann den gezielten Einsatz der Farbe als Gestaltungsinstrument.

Bei der computerunterstützten Erzeugung von Bildern spielt der Zufall oft eine große Rolle. Allerdings muss er zielgerichtet genutzt, in der Regel also eingeengt werden, sonst wird nur ein chaotisches Durcheinander erzeugt. Im Kapitel *Alles Zufall ... die simulierte Intuition* sehen wir am Beispiel *„Malen durch Klecksen"*, wie er ergebnisorientiert eingefangen werden kann.

Da Punkte, Linien, krumme Linien und Flächen charakteristisch für viele Arbeiten der Computerkunst sind, wird im Kapitel *Der Figurenbaukasten: Punkte, Linien und mehr* der Umgang mit diesen Bildelementen erarbeitet. Gekoppelt mit dem Prinzip der Wiederholung und der Variation erzielen wir mit ihnen bereits erstaunlich unterschiedliche Bildeffekte.

In Teil II geht es um das *Recoding & Remixing der Computerkunst*. Das Kapitel *Vom Analogen zum Digitalen* fällt insofern aus dem Rahmen, als die Vorbilder keine digitalen Artefakte sind, sondern Ergebnisse der Arbeit mit Analogcomputern. Da diese seit Laposky (vorgestellt im Abschnitt *Hommage à Laposky)* ebenfalls zur Computerkunst gezählt werden und wir dies - unter Verwendung von Lissajous-Figuren - digital simulieren können, gelingt es auch damit durch Variation zentraler Parameter eine Fülle sehr unterschiedlicher ästhetischer Objekte zu erzeugen.

Im Kapitel *Muster aus Figuren* belegen etliche Beispiele, wie mit den nun bekannten Elementen vielfältige Muster möglich werden. Unterstützt wird das durch leistungsfähige Konzepte von Snap!, wie das Verwenden von *Prozeduren als Daten*.

Es folgen mehrere Kapitel, die sich explizit mit Werken einzelner Vertreter der Computerkunst befassen. Diese Hommagen beginnen mit Georg Nees und seinem Bild *Schotter*, gefolgt von William J. Kolomyjec mit *Boxes* als erstem *Remixing* von *Schotter* (wofür in *Ein Block ist ein Block ist ein Block ...* die *Rekursion* eigeführt wird). Das Kapitel *Die Sprache G* unterstreicht die Sonderstellung, die ich Georg Nees einräume. Es folgen Frieder Nake mit *Einfachen Grafiken*, Michael Noll mit *Stilkopien* und Vera Molnar mit *Unordnungen*.

Reiner Schneeberger hat ein Grafiksystem vorgestellt, mit dem er *einen unmittelbar praktischen Zugang zur Computergrafik und speziell zur Computerkunst* eröffnen will (Limbeck & Schneeberger, 1979). Er hat seine Bilder selbst auch *Parameterkunst* genannt. Im Kapitel *SNE COMP ART* wird ein typisches Beispiel daraus nachgebaut. Auch Zdeněk Sýkora ist mit *Linien Malen* ein eigenes Kapitel gewidmet. Ihm ist es gelungen, einen ganz eigenständigen Stil mit dynamischen, kurvigen Elementen zu entwickeln, der sich deutlich von den vielen im Raster angeordneten geometrischen Figuren unterscheidet.

Zum *Remixing* gehört auch, die Computerkunst in Bewegung zu setzen. Genau das wird im Kapitel zur Animation *Alles in Bewegung* gezeigt. Die vorgestellten Beispiele repräsentieren unterschiedliche technische Realsierungen der Animation.

In Teil III *Kunst Programmieren?!* sollen die bis dahin erarbeiteten Grafikelemente und Programmierkonzepte angewendet werden, nun aber weitgehend unter Verzicht auf deren algorithmische Umsetzung und Codierung. Dazu werden achtzehn weitere typische Beispiele der frühen Computerkunst vorgestellt.

Mit musterähnlichen Wiederholungen geometrisch-abstrakter Motive, die in verschiedenen Ebenen gegeneinander verschoben werden, lassen sich faszinierende optische Wirkungen erzeugen, auch Moiré-Effekte genannt. Sie werden gerade in der Op Art gerne verwendet. Im Kapitel *Moiré-Muster* wird gezeigt, wie überraschend einfach die Erzeugung solcher Effekte ist.

Die oben genannten Animationen bilden die Grundlage für Beispiele im Kapitel *Kinetische Kunst*.

Die Computerkunst hat enge Bezüge zu anderen wichtigen Kunstrichtungen, etwa der Konzeptkunst, der Konkreten Kunst oder der Op Art. Diese Bezüge werden im Kapitel *Codierte Kunst* an einigen Beispielen verdeutlicht werden. Dem dient das *Recoding & Remixing* einzelner Werke von Josef Albers und François Morellet sowie des *Drip Painting* von Jackson Pollock. Dabei wird sich zeigen, dass darin viele Elemente der Computerkunst in neuem Kontext auftauchen.

Im abschließenden Kapitel wagen wir einen *Blick über den Tellerrand*. Denn es gibt natürlich auch andere Projekte, die sich dem *Recoding* gewidmet haben. Und es gibt andere leistungsfähige Werkzeuge - etwa Zeichenroboter - für die Umsetzung der Konzepte generativen Gestaltens, also der Verbindung grafischer Gestaltung mit dem Programmieren.

Das Buch endet mit einem kleinen *Ausblick*, der andeuten soll, was unter Einbeziehung aktueller technischer Entwicklungen alles möglich wird. Neben der *Animation*, die in einem eigenen Kapitel schon angesprochen wurde, sind es vor allem zwei Konzepte, die das Tor zu multimedialen Installationen öffnen. Es ist dies zum einen die *Interaktion*, zum anderen die *Reaktivität* von Programmen durch die Verwendung von Sensoren. Perspektivisch öffnet sich Ihnen damit der Übergang von der frühen Computerkunst zur heutigen Medienkunst.

Alles, was sich einer Darstellung anhand von beispielhaften Programmen entzieht, aber für Ihre weitere Arbeit mit Snap! unentbehrlich ist, wurde in die Anhänge verbannt. So findet sich eine Darstellung der Snap!-Benutzeroberfläche mitsamt *Tipps und Tricks* in *Anhang A*. Eine systematische Auflistung aller Snap!-Befehle bietet *Anhang B*. Im Laufe der Kapitel werden etliche Snap!-Befehle in Erweiterung des vorgegebenen Befehlssatzes erarbeitet. *Anhang C* enthält die so entstandene Blockbibliothek.

Dieser Text enthält bestimmte, immer wiederkehrende Elemente. Für solche unterschiedlichen Textbestandteile verwende ich folgende Schriftdarstellungen:

- Hervorhebungen, Anregungen und Hinweise für weiterführende Aktivitäten: *Kursivschrift*
- Langzitate: „*Kursivschrift*“
- Programmbefehle, Programmausgaben und Dateinamen: **fette**, z.T. **farbige Schrift**
- Neben vielen textbegleitenden Illustrationen gibt es (meist ganzseitige) Bilder als Beispiele für das erfolgreiche *Recoding* oder *Remixing* von Vorbildern (Bild 1 - 101): Unterstreichung.
- Web-Adresssen: Unterstreichung (um den Text frei zu halten von den oft langen Webadressen, finden Sie diese unter http://codiertekunst.joachim-wedekind.de/inhalte/linkverzeichnis)

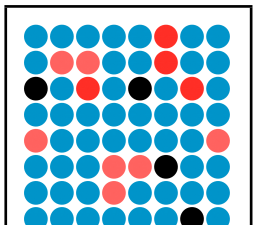


Bild 21: Hommage à Molnar

- Statt eines Glossars wird bei entsprechenden Begriffen auf Wikipedia bzw. vergleichbare (oft auch englischsprachige) Quellen verwiesen: Beispiel Wikipedia
- Elemente des Befehlssatzes von Snap! werden immer dann neu eingeführt, wenn sie für die Umsetzung eines Lösungsansatzes gebraucht werden. Sie finden ihre Beschreibung in grau unterlegten Kästen, auf die mit *Alonzo* - dem Snap!-Maskottchen - hingewiesen wird.



Hinweis: Die Snap!-Programme (Prozeduren) sind meist, aber nicht immer vollständig, bei den Beispielen abgedruckt. Für alle, die sich das Zusammenstellen der Programmblöcke ersparen wollen, können diese auch komprimiert in einer ZIP-Datei SnapProgrammeCompKunst.zip heruntergeladen, entpackt und dann mit Snap! aufgerufen und ausgeführt werden. Die Datei ist zu finden unter:
http://codiertekunst.joachim-wedekind.de/projekte

Dort findet sich auch eine Übersicht der Projekte, in der diese anklickbar sind und damit direkt in einem Fenster des Webbrowsers geöffnet werden. Damit ist dann auch der Code einseh- und verfübgbar und kann bei Bedarf eigenen Vorstellungen angepasst werden.

Die hier verwendete Programmierumgebung Snap! ist eine vollständige und leistungsfähige Implementation der Sprache Logo. Wenn Sie Spaß am Programmieren mit Snap! gefunden haben, können Sie damit weitergehende und anspruchsvollere Probleme als die von mir vorgestellten Beispiele - auch in anderen Anwendungsbereichen - in Angriff nehmen. Dabei wünsche ich Ihnen viel Freude und Erfolg.

TEIL I. GRUNDLAGEN

1. COMPUTERKUNST

Vor etwas über 50 Jahren, im Februar 1965, fand an der Universität Stuttgart die weltweit erste Ausstellung zur Computerkunst statt mit Bildern des Ingenieurs Georg Nees (die er selbst *ästhetische Objekte* nannte), organisiert von dem Philosophen Max Bense, einer damals prominenten Stuttgarter Geistesgröße. Im selben Jahr fanden in rascher Folge weitere wegweisende Ausstellungen zu dem neuen Kunstzweig statt: Im April in der New Yorker Howard Wise Gallery (mit Bildern von Noll und Julesz), im November in der Stuttgarter Szene-Buchhandlung Niedlich (mit Bildern von Nees und Nake). Im August 1968 folgte in London die Cybernetic Serendipity (Reichardt, 1968) mit dem ganzen Spektrum damaliger Multimedia-Möglichkeiten. Von 1968 an war die Computerkunst auch Bestandteil der Ausstellungen der avantgardistischen Zagreber Künstlergruppe Nove Tendencije (Rosen, 2011), die zum Forum der Vertreter experimenteller Kunst wurden.

Nach einer kurzen Blütezeit von etwa zehn Jahren verschwand die Computerkunst bereits wieder von der Bildfläche. Seit einigen Jahren ist das Interesse allerdings neu erwacht und ihre Rolle als Vorläufer und Wegbereiter der modernen Medienkunst, charakterisiert durch Multimedialität und Interaktion, ist inzwischen anerkannt.

Die Einordnung der Computerkunst in die Kunstszene des 20. Jahrhunderts ist nicht das Thema dieses Buches. Dazu kann ich auf das Buch *Computer und Kunst* von Erwin Steller (1992) verweisen, das dafür eine ausgezeichnete Grundlage darstellt.

Ich möchte als direkten Einstieg in das Thema mit Bildbeispielen beginnen (Bild 1 - 7)[3], die einen ersten Eindruck vermitteln, welche typischen Bildelemente, Darstellungsformen und Programmkomponenten die frühen Arbeiten der Pioniere der Computerkunst[4] kennzeichneten. Die zumeist schwarz-weißen Bilder bestanden aus einfachen Grundelementen, wie Linien, Linienzügen, Vielecken oder kurvigen Gebilden.

Bild 1 ist den Rechteckschraffuren von Frieder Nake aus dem Jahr 1965 nachempfunden. Grundelemente sind *Rechtecke*, die aus nebeneinander liegenden Linien gleicher Länge, den *Schraffuren*, gebildet werden. Das Programm, mit dem das Bild erzeugt wird, benötigt nur wenige Steuerelemente: Anzahl der Rechtecke (hier 20), Positionierung und Größe der Rechtecke, Dichte bzw. Abstand der Linien voneinander und die Richtung der Linien (senkrecht bzw. waagrecht), wovon die meisten zufällig festgelegt werden. Bei einer zweifarbigen Variante des Programms kommt die Wahl der Farbe eines Rechtecks hinzu.

[3] Die Bilder 1 bis 7 sind keine Reproduktionen der Originale, sondern Hommagen an die genannten Pioniere der Computerkunst, von mir recoded mit Snap!.

[4] Die hier nur mit Namen erwähnten Personen werden später in eigenen Kapiteln mit ihren Werken und der programmtechnischen Umsetzung näher vorgestellt.



Bild 1: Hommage à Nake: Rechteckschraffuren

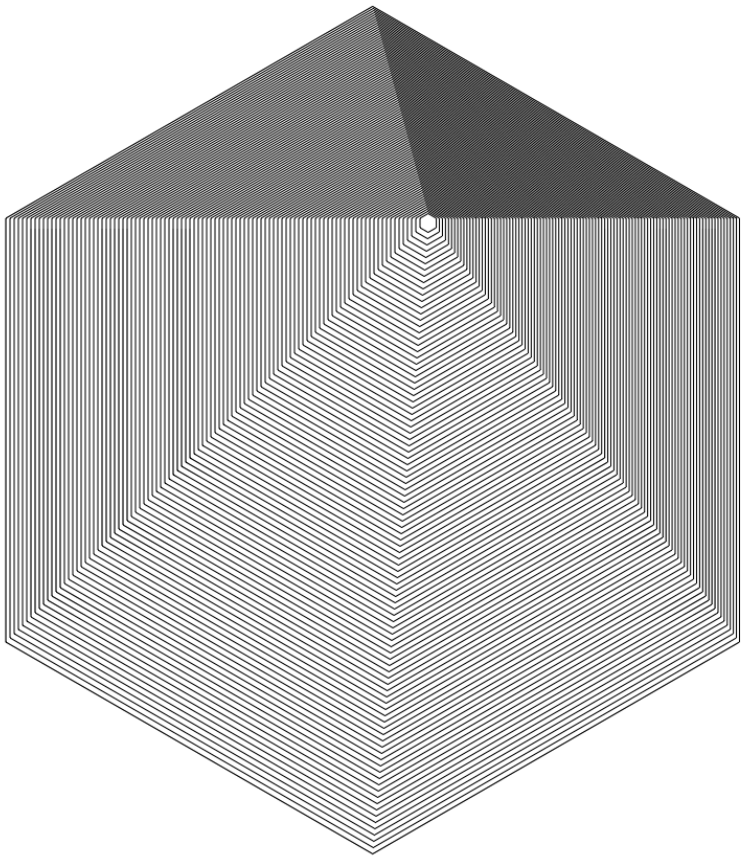


Bild 2: Hommage à Lecci: Shift 2

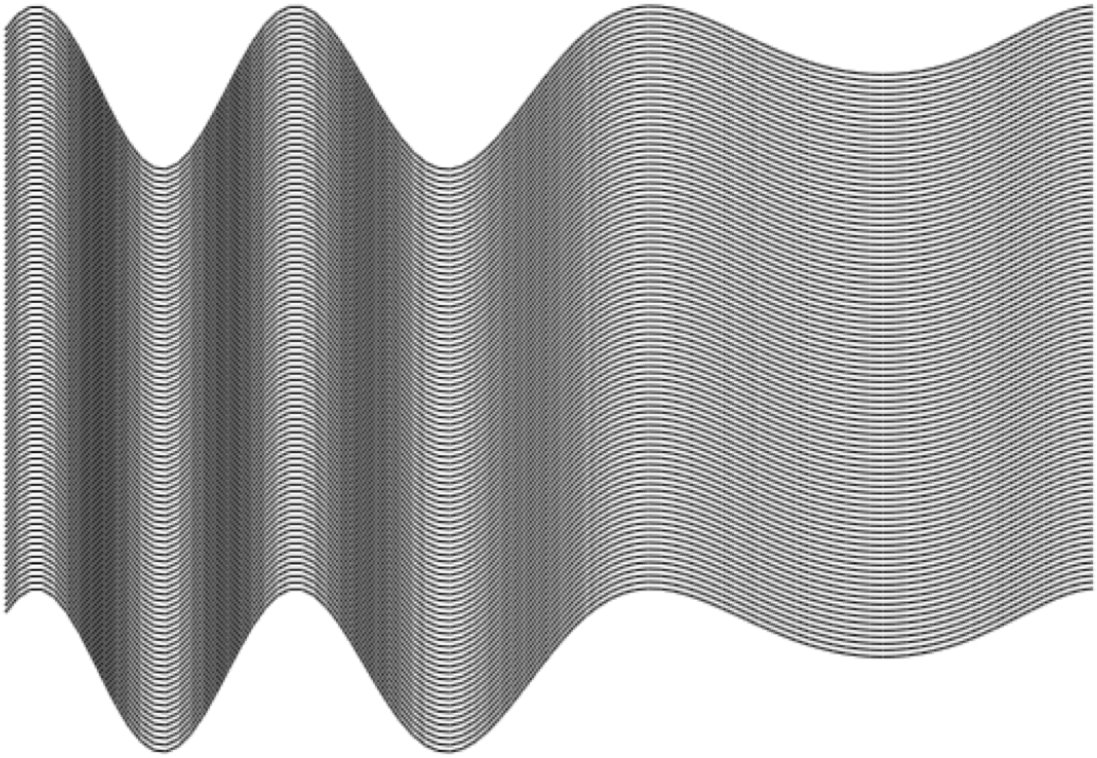


Bild 3: Hommage à Noll: Sinusoide

In Bild 2 wird eine Grundstruktur aufgegriffen, die Auro Lecci 1969 in seinem Bild Shift 2 zu einem Streifenmuster entwickelt hat. Ausgehend von einem Zentralpunkt werden Sechsecke gezeichnet, deren Mittelpunkt in gleichen Abständen voneinander versetzt und deren Seiten jeweils verlängert werden. Das entstehende Streifenmuster lässt einen räumlichen Effekt entstehen.

Bei Bild 3 mit dem vollständigen Titel Ninety Parallel Sinusoids With Linearly Increasing Period aus den frühen 60er-Jahren orientiert sich Michael Noll an dem Werk Current der Op Art-Vertreterin Bridget Riley. Die oberste Linie ist definiert über einen Sinus mit zunehmender Periode und gleichzeitig abnehmender Amplitude. Diese Linie wird dann 90 mal identisch untereinander gezeichnet.

Auch wenn diese Merkmale als typisch für die frühe Computerkunst gesehen werden können, das eigentlich entscheidende Charakteristikum ist damit noch gar nicht erfasst: Die Erzeugung dieser Bilder mit Hilfe des Computers. Einer der Pioniere, Herbert W. Franke (der Vielen eher als Autor von Science-Fiction-Romanen bekannt sein dürfte), hat die Computerkunst denn auch so definiert (Franke, 1971, S. 7 f.): „*Unter Computerkunst soll jedes ästhetische Gebilde verstanden werden, das auf Grund von logischen oder numerischen Umsetzungen gegebener Daten mit Hilfe elektronischer Automaten entstand.*“

Franke formuliert so allgemein, weil für ihn die Erzeugung von Grafiken mit Rechenanlagen mit den *Oscillons* von Ben F. Laposky beginnt[5]. Franke selbst hat ebenfalls von 1956 bis 1961 solche Analoggrafiken entwickelt, die er *Oszillogramme* nannte (Franke, 1971, S. 64, S.69).

Heute bedeutet seine Aussage weniger abstrakt formuliert, dass Computerkunst immer mit Hilfe digital gesteuerter Geräte erzeugt wird. Das betrifft einerseits die Software, also die Programme, mit denen die darzustellenden Daten erzeugt werden, andererseits die Hardware, also die Computer, auf denen die Programme laufen und die Ausgabegeräte, mit denen die Daten als wahrnehmbare Artefakte dargestellt werden, sei es z.B. als Bilder auf Monitoren oder auf Papier gebracht durch Plotter oder Drucker. Dabei ist es unerheblich, ob die Künstler die Programme selbst entwickeln, entwickeln lassen oder vorhandene Programme genutzt werden.

1.1 Grundelemente der Computerkunst

Mein Anliegen mit diesem Buch ist es, zu zeigen, wie entsprechende Programme *selbst* entwickelt werden können. Hilfreicher als die Definition von Franke scheint mir dafür, die wesentlichen Charakteristika der Bilder heraus zu destillieren. Die Bilder 1-3 zeigen bereits, was wir in vielen weiteren Beispielen wiederfinden können und was wir in eigenen Projekten aufgreifen werden:

- Einfache *Grundelemente*, wie Linien bzw. (Sinus-) Kurven (vgl. Bild 1 und Bild 3).

[5] Diesem Pionier ist ein eigener Abschnitt *Hommage à Laposky* gewidmet.

- Die *Variation* dieser Grundelemente, wie z.B. Linienlänge (Bild 1); Sechseckgröße (Bild 2).
- Die systematische *Wiederholung* der Grundelemente (Bild 1 - 3).
- *Zufallswerte* für bildbestimmende Kenngrößen (Bild 1).

So wie in Frankes Definition beschrieben, bezeichnet Computerkunst lediglich eine Vorgehensweise mit digitalen Werkzeugen, aber keine eigene Kunstrichtung. Computerkunst ist damit zwar ein eingebürgerter, aber eher unpassender Begriff; wir reden ja auch nicht von Pinselkunst oder Holzschnittkunst als eigenen Kunstrichtungen. Begonnen hat die Computerkunst mit den Experimenten einiger Ingenieure und Mathematiker, die die für wissenschaftliche bzw. technische Zwecke teuer beschafften Zeichenmaschinen (auch als Plotter bekannt) für die Erzeugung ästhetischer Objekte nutzten.

Als sie in den eingangs genannten Ausstellungen ihre Ergebnisse der Öffentlichkeit präsentierten, stießen sie in der Kunstszene und der Presse erstmal auf breite Kritik. So auch in der ersten Stuttgarter Ausstellung, von der überliefert ist, dass anwesende bildende Künstler sauer reagierten, aufgebracht und Türen knallend den Raum verließen. Max Bense gefiel das nicht und er soll ihnen nachgerufen haben: „*Meine Herren, es handelt sich hier um künstliche Kunst!*“ (zu diesem Begriff siehe auch von Herrmann, 2010). Er wollte jedenfalls eine Grenze ziehen zu der Kunst, die die Künstler selber erschaffen. Die Stuttgarter Zeitung (Skasa-Weiß, 1965) schrieb damals nämlich so: „*Ein Heinzelmann* (der Computer, JW) *macht's möglich, dass der moderne Bilderfabrikant sich nicht mehr handwerklich zu strapazieren braucht.*“

Es war aber wohl nicht nur so, wie Bense meinte, dass die Künstler sich in ihren Schöpfungsmöglichkeiten von den Maschinen bedroht fühlten, es war wohl auch die Schlichtheit vieler Grafiken und die schwierige Zuordnung von Werken zu ihren Urhebern, die manchen Zweifel an der *Kunst* in der *Computerkunst* auslösten.

Es fällt ja auf, dass die Pioniere (wie Nake, Nees, Noll, Franke oder Molnar) großteils Werke zeigten, die mit reinen Linienzügen bzw. Linienschraffuren arbeiteten; sehr beliebt war auch das Quadrat als Grundelement (z.B. bei Franke und Molnar). Die tatsächliche Urheberschaft ist dabei fast austauschbar. Dazu zwei Beispiele:

Die beiden Liniengrafiken in Bild 4 und Bild 5 zeigen große Ähnlichkeiten, obwohl sie ganz sicher unterschiedlich beschrieben und programmiert worden sind. Beide bestehen aus Linienzügen, bei denen jeweils am Endpunkt einer Linie im rechten Winkel eine weitere Linie angefügt wird, deren Länge zufällig festgelegt wird. Das Ganze spielt sich in einem Grafikfenster vorgegebener Größe ab. Bild 4 bezieht sich auf eine Bildserie *Irrwegrechtecke* von Georg Nees (Nees, 1969, 215 ff.). Bild 5 hat als Bezug das Bild Vertical-Horizontal No. 3 von Michael Noll aus dem Jahr 1962. Beide unterscheiden sich im Wesentlichen nur durch die Anzahl der Linien und die Größe des Grafikfensters.

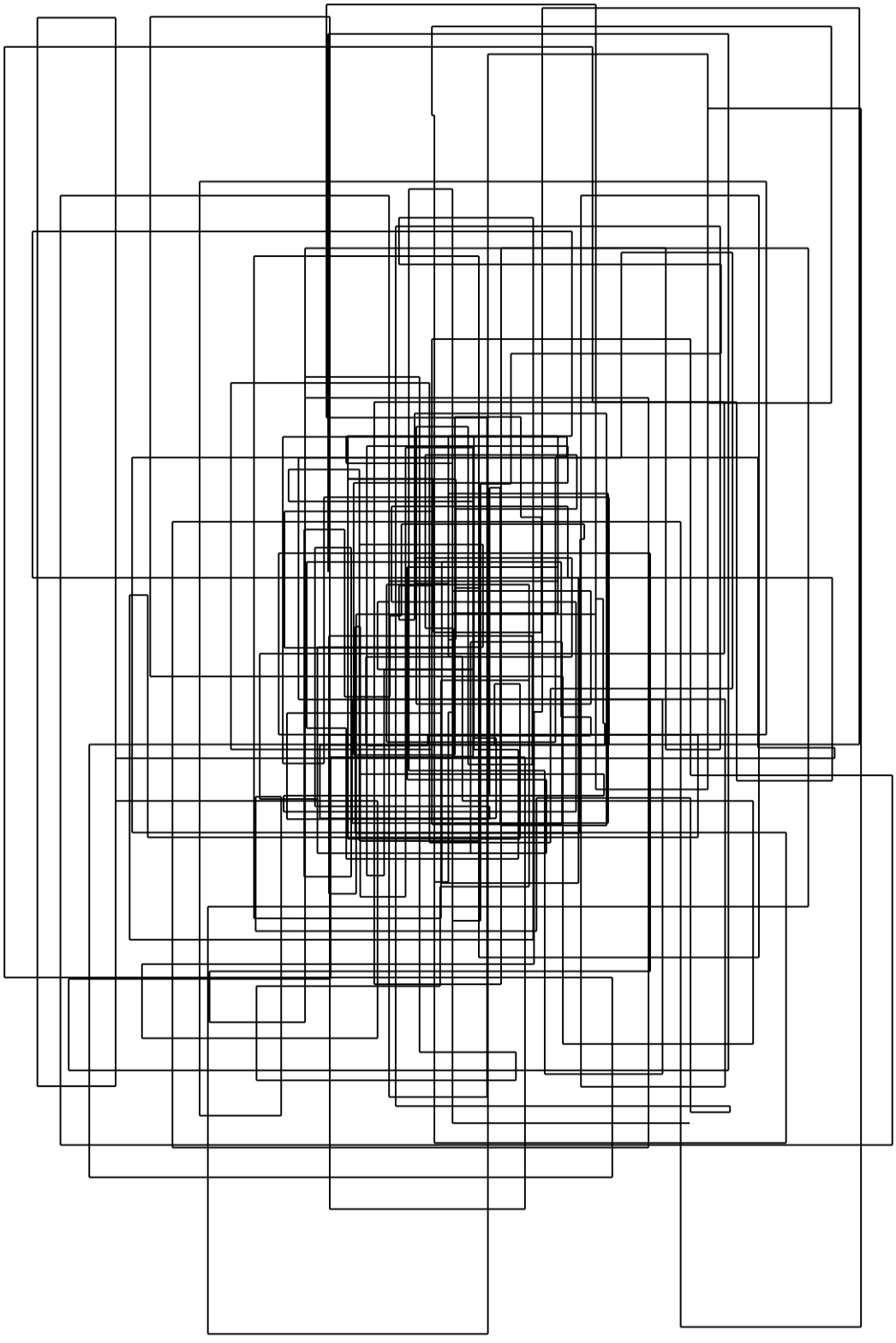


Bild 4: Hommage à Nees: achsenparalleler Irrweg

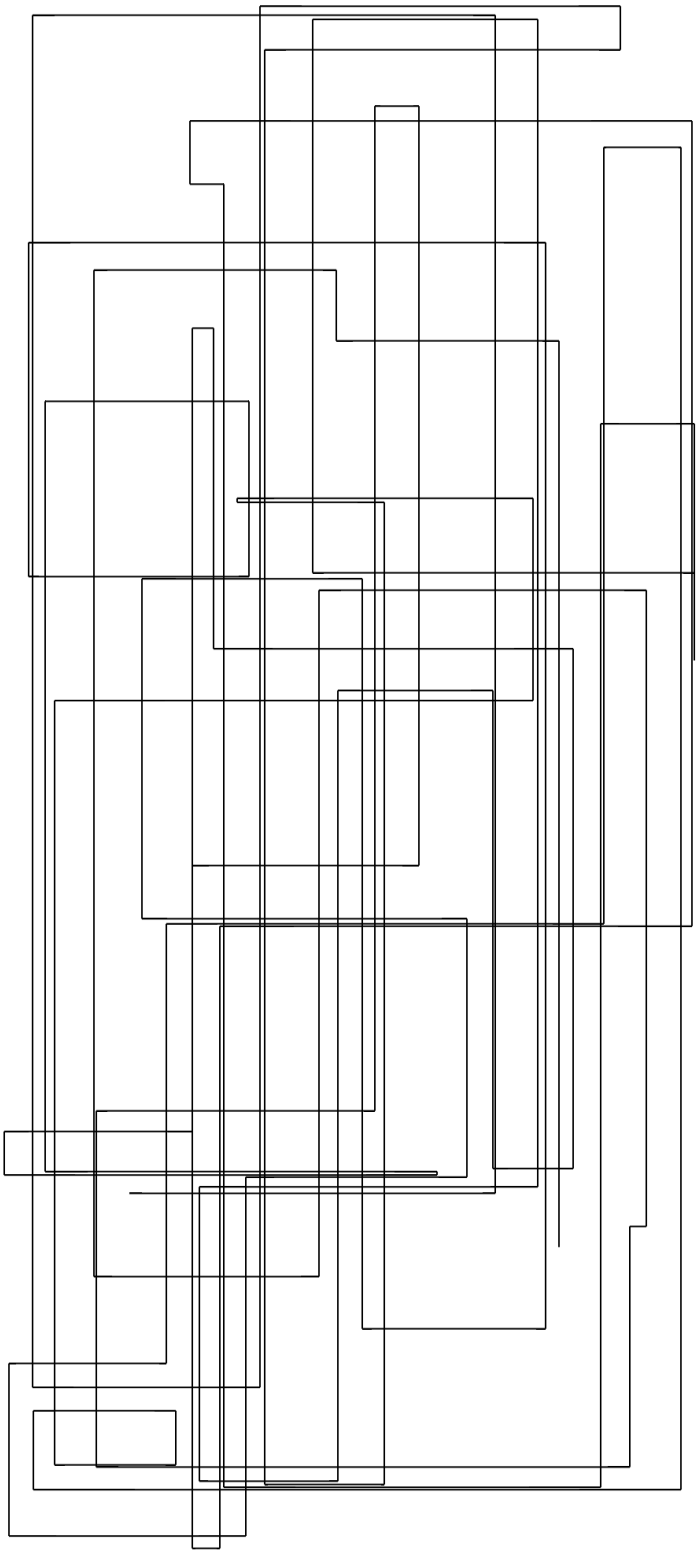


Bild 5: Hommage à Noll: Vertikal-horizontal

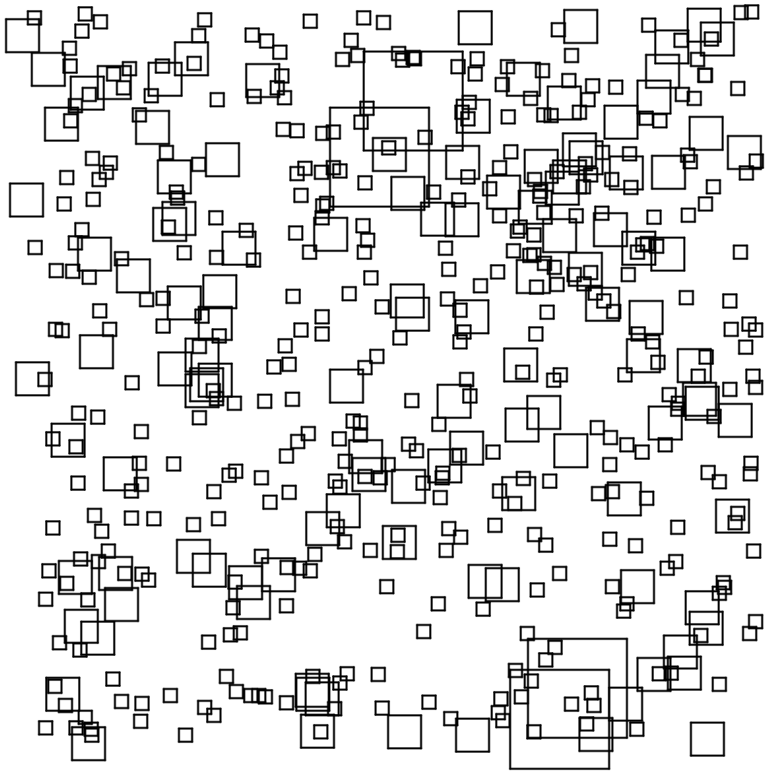


Bild 6: Hommage à Franke: Quadrate

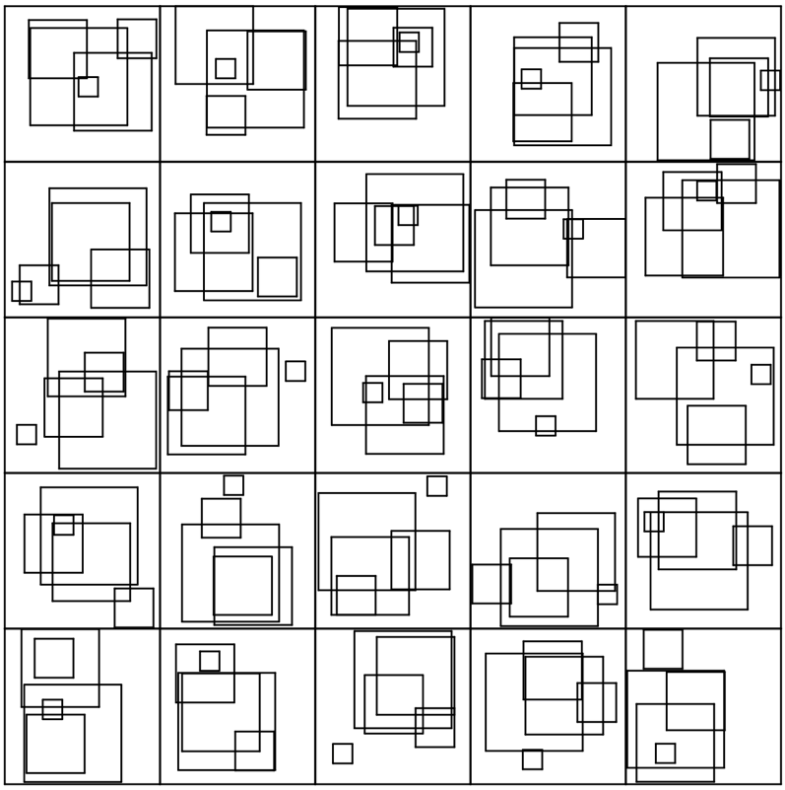


Bild 7: Hommage à Molnar: Struktur von Quadraten

Das zweite Bilderpaar Bild 6 und Bild 7 besteht ebenfalls aus gleichartigen Elementen, den Quadraten, die unterschiedlich angeordnet werden. In Bild 6, das sich auf Quadrate von Herbert W. Franke (1969) bezieht, werden Quadrate zufällig variierender Größe über das Grafikfenster (auch überlappend und ebenfalls zufällig) verteilt. In Bild 7 werden ebenfalls solche Quadrate zufällig angeordnet, aber deutlich mehr strukturiert, indem jeweils 5 Quadrate variierender zufälliger Größe in einem umschließenden Quadrat eingezeichnet werden und dies in einer 5*5-Matrix gleich großer Quadrate wiederholt wird. Dieses Bild bezieht sich auf *Struktur von Quadraten* von Vera Molnar (Lieser, 2009, S. 60).

Die Ähnlichkeit der Bilder, die nicht nur bei den hier gezeigten Beispielen zu finden ist, lag natürlich teilweise an den beschränkten technischen Möglichkeiten. Die damaligen Plotter, die für die Ausgabe der Bilder zur Verfügung standen, boten oft nur die Möglichkeit für Schwarz-Weiß-Ausgabe und nur eingeschränkt auch für farbige Darstellungen. Mit der Technik der Schrittmotoren zur Bewegung der Zeichenstifte ließen sich Linien präziser ziehen, als es der Fall bei kurvigen und flächigen Grafikelementen war. Heute finden wir solche Geräte nur noch im Museum und die durch sie gegebenen Einschränkungen sind seit langem überwunden. Es ist deshalb vielleicht auch schlicht dem technischen Fortschritt geschuldet, dass die frühe Computerkunst nach einem Jahrzehnt bereits wieder weitgehend in der Versenkung verschwand.

Leider wurden die Werke der frühen Computerkunst kaum systematisch gesammelt und so sind sie entsprechend selten in Gruppen- oder gar in Einzelausstellungen zugänglich[6]. Deshalb hat m.E. die lebhafte, theoretische Diskussion über diese Kunstform wohl eher bleibenden Einfluss genommen und behalten als die eigentlichen Werke. Das gilt für die aktuellen Formen der Digitalen Kunst bzw. der Generativen Kunst genauso wie für die Medienkunst allgemein.

So mutet der Begriff Computerkunst heute fast nostalgisch an, gewissermaßen als Reminiszenz an eine Phase, in der die Pioniere mit inzwischen völlig veralteten Maschinen in einem unbekannten Feld experimentierten. Einerseits erzeugten sie ihre ästhetischen Objekte auf eine sehr traditionelle Art; sie wurden am Ende mit Stiften auf Papier gebracht. Andererseits wurden sie völlig digital erzeugt, was aber ein gewisses technisches Wissen voraussetzte. Das betraf sowohl die Kenntnis der technischen Möglichkeiten und Grenzen, als auch die Kontrolle über den Entstehungsprozess und das Ergebnis. Damit war der Zugang zur Computerkunst für interessierte bildende Künstler durch diese technische Hürde erstmal erschwert.

Gleichzeitig wurde die Computerkunst von ihren Kritikern als ästhetisch belanglos, abstrakt und kalt bewertet (Taylor, 2014, S. 32). Dieser Zweifel an ihrem Kunstwert besteht nach wie vor, obwohl wir heute in einem Zeitalter leben, in dem die Digitalisierung einen Wandel aller Lebensbereiche - und damit eben auch der Kunst - bewirkt

[6] Die Berliner DAM-Galerie hat sich mit zahlreichen eigenen Ausstellungen darum besonders verdient gemacht.

hat. Inzwischen liegt die Erfindung des Mikroprozessors und die zunehmende Verbreitung von PCs (also den persönlich verfügbaren Computern) und der Siegeszug des Internet hinter uns. Heute können traditionelle Maltechniken simuliert werden, ermöglicht durch die Interaktivität der Systeme und durch die drastische Verbesserung der Ausgabegeräte hinsichtlich Auflösungsgenauigkeit und Farbe. Es sind nun nicht mehr allein die Ingenieure und Naturwissenschaftler, die diese Geräte nutzen, sondern die bildenden Künstler selbst eroberten die Technik für sich und ihre künstlerische Arbeit. Zu diesen Pionieren zählen Vera Molnar, Manfred Mohr oder Harald Cohen.

1.2 Algorithmische Kunst

Die überwiegend kritische Haltung gegenüber der frühen Computerkunst rührte wohl auch daher, dass - bei fehlenden Kenntnissen über und wenig Erfahrung im Umgang mit dem Computer - dessen Rolle beim Entstehungsprozess der Werke überschätzt wurde (Werler, 1991, S. 12). Dabei haben bereits die Pioniere kritisch reflektiert, was denn die Kunst an der Computerkunst sei. Vor allem haben sie sich von dem unpräzisen Begriff Computerkunst getrennt und betont, dass der Begriff *algorithmische Kunst* von Anfang an treffender gewesen wäre (etwa Nake, 2010). Damit wird betont, dass nicht alle Kunst, die mit Hilfe von Computern erzeugt wird, gemeint ist, sondern genau die Art von Kunst, deren Grundlage *Algorithmen* sind. Diese Einordnung der Computerkunst hat interessante Konsequenzen.

Ein Algorithmus[7] ist eine eindeutige Handlungsvorschrift zur Lösung eines Problems oder einer Klasse von Problemen. Ein Algorithmus besteht immer aus endlich vielen, wohldefinierten Einzelschritten. Für die konkrete Ausführung kann er in einem Computerprogramm implementiert werden. Die Umsetzung einer Idee mit dem Ziel, ein ästhetisches Objekt zu erstellen, beginnt also mit dem beschreibenden *Konzept* für das Objekt, gefolgt von dem Entwurf des *Algorithmus* sowie seiner Umsetzung in Form eines ausführbaren *Computerprogramms*. Am Ende führt dies ohne weiteres menschliches Zutun zur *Darstellung auf einem Ausgabegerät* (Plotter oder Bildschirm). Durch die Variation der Kenngrößen können nahezu beliebig viele Varianten einer Bildidee erzeugt werden. Aus den entstehenden Bildserien können gewünschte Objekte ausgewählt werden.

Daraus ergibt sich ein Kerngedanke der Computerkunst, nämlich dass das einzelne ästhetische Objekt immer nur ein „*Element einer Klasse möglicher verschiedener, aber auf bestimmte Weise gleichartiger Objekte ist*“ (vgl. dazu Klütsch, 2007, S. 16). Und: „*Um eine Klasse von Kunstwerken zu schaffen, muss der Künstler zuerst diese Klasse beschreiben und außerdem ein Verfahren angeben* (in der Computerkunst sind das die Algorithmen, JW)*, wie einzelne Instanzen (Werke) dieser Klasse realisiert werden können*“ (Klütsch, a.a.O.).

[7] Eine differenzierte Darstellung zur Begriffskombination „Algorithmus und Kunst“ hat Jochen Ziegenbalg (2005) vorgelegt.

Das steht schon deutlich im Gegensatz zum einzelnen Kunstwerk, das in der traditionellen Kunst im Zentrum steht. Nake (2014) spricht deshalb bei digital erzeugten Bildern auch von der Verdopplung der Dinge, d.h. dass jedes digitale Kunstwerk ein Sichtbares und ein Berechenbares hat. Er benennt das sichtbare digitale Kunstwerk als die analoge *Oberfläche* (surface) und den Algorithmus bzw. das Programm als die digitale *Unterfläche* (subface) (Nake, 2016). Als Betrachter haben wir in aller Regel keinen Zugriff auf die Unterfläche, die vom Künstler entwickelt wurde[8].

Für die Computerkunst finden sich durchaus Anknüpfungspunkte bei anderen Kunstrichtungen, denn für das Abarbeiten von Programmen mit verschiedenen Parametern gibt es Vorbilder. Das gilt insbesondere für die Konzeptkunst, gewissermaßen eine Verwandte der Computerkunst. So hat Sol LeWitt über mehrere Jahrzehnte seine Wall Drawings von Assistenten nach seinen Vorgaben erstellen lassen. Selbst nach seinem Tod im Jahr 2007 können und werden solche Werke angefertigt und ausgestellt.

Vergleichbares gilt für die Op Art (Optical Art) mit ihren oft seriellen Strukturen, der Exaktheit geometrischer Formen und deren Abwandlung (Transformation). Einer der bekanntesten Vertreter ist Viktor Vasarely. Er entwickelte einen quasi-industriellen Produktionsprozeß, für den er einen Ausgangs-Prototyp entwarf, der als Serigrafie, Relief oder Skulptur umgesetzt und dann seriell vervielfältigt werden konnte. Ähnliches gilt für die Konkrete Kunst. Solche Bilder sind *potentielle Computerbilder*, denn sie sind oft durch entsprechende Computerprogramme zu beschreiben und zu erzeugen.

1.3 Vom Plotter zum Zeichenroboter

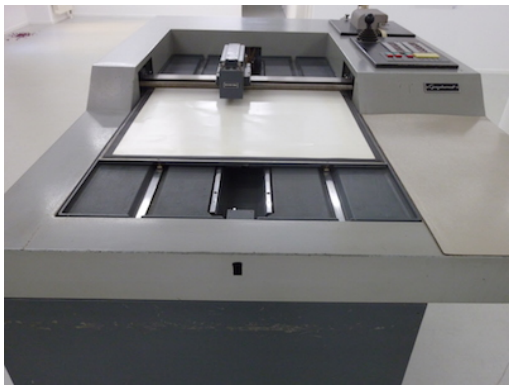
In den Anfängen der Computerkunst standen für den Ausdruck der Grafiken die ersten Plotter zur Verfügung, die damals vor allem in technischen und naturwissenschaftlichen Bereichen eingesetzt wurden. Im Grunde wurde damit eine geradezu konventionelle Bilderzeugung betrieben: Die Plotter bewegten einen Stift über die Oberfläche eines Papierbogens. Das Ergebnis sind Vektorgrafiken, keine Rastergrafiken, wie wir sie von unseren modernen Bildschirmen kennen[9]. Mit Stiftplottern können komplexe Strichzeichnungen - auch Texte - erstellt werden. Aufgrund der mechanischen Bewegung der Stifte ist die Erstellung sehr langsam. Farbige Flächen werden nur durch nahe beieinander liegende, regelmäßige Linien möglich. Lange Zeit war dies dennoch der einzige Weg vektorbasierte Kunstwerke - auch in großen Formaten - zu erzeugen.

Für das Zeichnen eigneten sich sowohl Filz- und Tuschestifte als auch Kugelschreiber, die immerhin unterschiedliche Strichbreiten erlaubten. Anfangs waren meist nur ein-

[8] Dieses Buch ist gewissermaßen ein Versuch, die Oberfläche und Unterfläche der digitalen Bilder gleichgewichtig zu behandeln.

[9] Rastergrafiken bestehen aus einzelnen Bildpunkten, deren Gesamtheit dann das Bild darstellt. Bei Vektorgrafiken werden nicht einzelne Punkte, sondern Eigenschaften der verwendeten Formen verwendet, z.B. bei einer Linie Anfangspunkt, Richtung, Länge, Dicke usw.. Vektorgrafiken lassen sich dadurch ohne Qualitätsverlust skalieren.

farbige Zeichnungen möglich, da ein Farbwechsel mit dem Unterbrechen eines laufenden Programms und dem erst dann möglichen Wechsel des Stiftes verbunden war. Später war auch ein automatischer Farbwechsel über Farbkarussells möglich. Ein bekannter Vertreter dieses Gerätetypus ist der ZUSE Graphomat Z 64, der u.a. von Georg Nees und Frieder Nake für ihre Arbeiten benutzt wurde. Heute sind Plotter weitgehend überflüssig geworden; ihr Einsatzgebiet wird heute von Laser- oder Tintenstrahldruckern entsprechender Größe übernommen.



ZUSE Graphomat Z 64, Zeichentisch



ZUSE Graphomat Z 64, Steuerelektronikschrank

Für mich wies das Erzeugen der Grafiken der frühen Computerkunst mit Plottern direkte Parallelen auf zur Erzeugung der Schildkrötengrafiken mit Logo. Es war deshalb für mich naheliegend und konsequent, Logo nun dafür zu verwenden, Beispiele der Computerkunst nachzuprogrammieren.



Yellow Turtle

Die Entwicklung von Logo fiel in etwa in den gleichen Zeitraum wie die Entstehung der Computerkunst. Die grafischen Ausgabegeräte, wie der gezeigte Graphomat, waren zu der Zeit noch sehr teuer - und auch zu sperrig und unhandlich für den Einsatz im Unterrichtskontext. Für grafische Ausgaben wurde deshalb die Unterrichtssprache Logo ein kleiner schildkrötenähnlicher Roboter entwickelt: die *Yellow Turtle* (Feurzeig, 2010). Dieses kleine Gefährt konnte sich vorwärts bewegen und drehen. Ein Schreibstift konnte angehoben und gesenkt werden, so dass durch die Bewegungen der Schildkröte auf einem darunter liegenden Papier Linien gezogen wurden.

Später wurden die Computer immer billiger und die Roboter ihrerseits zum limitierenden Kostenfaktor. Deshalb wurde die mechanische durch eine elektronische Schildkröte auf dem Bildschirm abgelöst; sie wurde zu einem kleinen Symbol mit Richtungsanzeiger.

1.4 Geschichte der Computerkunst

Die Einordnung der Computerkunst konnte in diesem Rahmen nur knapp skizziert werden. Alle, die sich damit weiter befassen möchten, kann ich auf einige fundierte Darstellungen verweisen:

- Heike Piehler (2002) zeichnet *Die Anfänge der Computerkunst* aus dem Blickwinkel der Kunsthistorikerin nach. Die systematische Darstellung der frühen Ausstellungen und charakteristischer Werke und Künstler stützt das neue Interesse an der Computerkunst.
- Christian Klütsch (2007) analysiert in *Computergrafik. Ästhetische Experimente zwischen zwei Kulturen. Die Anfänge der Computerkunst in den 1960er Jahren* die Stuttgarter Schule um Max Bense mit ihren zentralen theoretischen und künstlerischen Konzepten.
- Grant D. Taylor (2014) hat sich mit *When the Machine Made Art. The troubled History of Computer Art* auf die Suche begeben nach den Ursachen für die kritischen bis feindseligen Reaktionen auf die frühe Computerkunst und er versucht, ihr eine angemessene Einordnung und Würdigung als Vorläufer der aktuellen Medienkunst zu geben.

2. RECODING & REMIXING

Wenn Algorithmen den Werken der Computerkunst zu Grunde liegen, wäre es natürlich hilfreich gewesen, diese Algorithmen von den Algoristen[10] zu übernehmen und sie dann nur noch mit dem eigenen Programmierwerkzeug umzusetzen. Leider sind sie aber sehr selten dokumentiert. Deshalb habe ich an typischen Beispielen versucht, mit eigenen Algorithmen ähnliche Bilder zu erzeugen. Dabei war es mein Ehrgeiz, meine Umsetzung immer so kompakt und gleichzeitig so flexibel wie möglich zu formulieren. Tatsächlich reichten häufig jeweils bereits ca. 20 – 30 Codezeilen. Da ich alle relevanten Bildeigenschaften über Parameter einstellbar machte, konnte ich dann in kürzester Zeit viele Bildvarianten erzeugen – wozu die Algoristen mit Plottern sicher viele Stunden gebraucht haben.

2.1 Recoding

Mit dem Versuch, Computerkunst nachzuprogrammieren, bin ich weder der Erste noch der Einzige. Schon kurz nach Ende der ersten Blütezeit der Computerkunst gab es Bücher, die in den Nachvollzug dieser Kunstrichtung durch Nachprogrammieren einführen sollten.

Den Anfang machten Limbeck & Schneeberger (1979) mit *Computergrafik*, einem *Lehr- und Lernbuch zur Computergrafik mit pädagogisch-didaktischer Aufbereitung*. Der theoretische Teil ist heute überholt und eher von historischem Interesse. Der praktische Teil stellt SNE COMP ART vor, ein Paket grafischer Grundfunktionen für die Programmiersprache FORTRAN, das von Schneeberger - selbst als Computerkünstler aktiv - speziell für die Erfordernisse der Computerkunst entwickelt wurde. Die Codeschnipsel im Buch reichen allerdings kaum zum praktischen Nachvollzug aus.

Mark Wilson - ebenfalls Computerkünstler der ersten Stunde - nennt sein Buch (1985) *Drawing with Computers. The Artist's Guide to Computer Graphics*. Wilson gibt (mit Bezug auf die damals verfügbaren technischen Mittel) anhand konkreter Code-Beispiele mit der Programmiersprache BASIC eine Einführung, wie mit Plottern und auf Bildschirmen künstlerische Grafiken erstellt werden können.

Das dritte Buch mit einem Brückenschlag vom Computer zur Kunst stammt vom Computerkünstler Erwin Steller: *Computer und Kunst - Programmierte Gestaltung* (1992). Anhand vieler Werkbeispiele werden Fragen behandelt, die eine historische und kritische Einordnung der Computerkunst in der Kunstszene des 20. Jahrhunderts ermöglichen. Auch wenn Steller keine Code-Beispiele bereitstellt, so sind seine Analysen der vielen Bildbeispiele äußerst hilfreich für den programmtechnischen Nachvollzug.

[10] Auf Initiative von Roman Verostko formierte sich 1995 eine Reihe von Computerkünstlern zur Gruppe der *Algoristen*. Als solches bezeichnen sie alle Künstler, die bei der Erstellung künstlerischer Objekte ihre eigenen Algorithmen verwenden.

Die aktuelleren Bemühungen, die historischen Werke der Computerkunst wieder zugänglich zu machen, sind im Netz zu finden. So hat es das ReCode Project sich zur Aufgabe gemacht „*to preserve computer art by translating it into a modern programming language (Processing). Every translated work will be available to the public and contemporary artists to learn from, share, and build on*". Es konzentriert sich darauf, den Code der historischen Werke zu bewahren, ihn zugänglich und nutzbar zu machen und damit eine Lernplattform zu schaffen. Auch wenn ihre Arbeiten mit der Programmiersprache *Processing* umgesetzt werden, bieten die einsehbaren Codebeispiele eine Fülle an Anregungen.

Das Projekt recodeArt konzentriert sich in vergleichbarer Weise auf die Arbeiten von Reiner Schneeberger[11]. Auch hier erfolgt die Umsetzung mit *Processing*. Beide Projekte leisten einen Beitrag zur Konservierung digitaler Kunst.

In den genannten Büchern sind es die Computerkünstler selbst, die nicht nur den Zugang zur Computerkunst eröffnen wollen, sondern die LeserInnen auffordern, die gestalterischen Möglichkeiten praktisch zu nutzen. In vielen Fällen ist das Recoden, das Nachprogrammieren vorhandener Werke der Computerkunst, der Einstieg in die Thematik. Bei Georg Nees (1995, S. VII) findet sich eine entsprechende Ermunterung: „*Die ersten Schritte sind so leicht: Man gehe von Abbildungen aus, die einem besonders auffallen. Dann werden Neugier, genaueres Studium des Textes und dann vielleicht auch ein bereitstehender Computer den Weg in die [..] Formenwelt stufenweise bahnen.*"

Aber entstehen dann nicht lediglich Imitate, Nachbildungen, also Wiederholungen künstlerischer Werke? Im Grunde stellt sich beim *Recoden & Remixen* unter dem Aspekt des Plagiarismus die Frage neu, mit der auch die frühen Computerkünstler konfrontiert wurden, nämlich was denn Kunst an ihren Werken sei.

Schneeberger (2013) diskutiert am Beispiel des oben genannten *ReCode Project* die Grenzen zwischen dem konservatorischen Bemühen, digitale Kunst zu bewahren bzw. zugänglich zu machen, und Plagiarismus. Sein kurzer Essay zu diesem Thema sollte Pflichtlektüre sein, wenn aus den Programmierübungen im stillen Kämmerlein Produkte werden, die den Weg in den öffentlichen Raum finden sollen!

Da wie erwähnt in den meisten Fällen kein Zugriff auf die originalen Algorithmen gegeben ist, bleiben eigentlich nur die Grafiken als Vorbilder, deren Analyse die benötigten Algorithmen liefern soll. Als bewährte Vorgehensweise beim *Recoding* hat sich für mich herauskristallisiert

- die Analyse, welche Muster und Strukturen für das Bild prägend sind,
- die Zusammenstellung der verwendeten grafischen Elemente (Punkte, Linien, komplexere Gebilde),
- die Analyse, ob und welche Wiederholungsmuster erkennbar sind,

[11] Eine wichtige Komponente der Arbeiten von Schneeberger, die Routine SNE KAO, ist Gegenstand des Kapitels 19: *SNE KAO*.

- die Analyse der Bedeutung des Zufalls, also zufälliger Abweichungen von Häufigkeiten oder Formgebungen.

Das gilt ebenso, wenn ich versuche, nicht nur bei den Werken der Computerkünstler die dahinter liegenden Algorithmen zu entschlüsseln. Mein Blick auf die Werke moderner Kunstrichtungen insgesamt hat sich verändert: In den Stilrichtungen des Konstruktivismus, des Suprematismus und besonders der geometrischen Abstraktion findet sich eine Reihe immer wiederkehrender Muster und Strukturen. Manche reizen dann, sie algorithmisch nachzubilden.

2.2 Remixing

Dem Recoden folgt das Remixen, d.h. die beim Recoden erzeugten Bilder und Ausgangselemente können neu kombiniert, abgewandelt und erweitert werden und damit neue Werke entstehen: Unter *Remixing* wird das Überarbeiten und Kombinieren kreativer Artefakte verstanden, üblicherweise in der Form von Musik, Video oder anderen interaktiven Medien. Das Phänomen ist inzwischen weit verbreitet und fester Bestandteil der (Jugend) Kultur[12].

Dabei ist *„Remixing [ist] nicht nur ein modischer Stil [...] von nutzergenerierten Inhalten [...]. Vielmehr ist es eine Meta-Methode, ein viele Genres und spezifische Arbeitsweisen kennzeichnendes Verfahren, in welchem unter Verwendung bestehender kultureller Werke oder Werkfragmente neue Werke geschaffen werden. Wesentlich bei einem Remix ist sowohl Erkennbarkeit der Quellen wie auch der freie Umgang mit diesen."* (Stalder, 2009)

Remixing wird von mir bewusst in Analogie zu diesem Verständnis gesehen: Von behutsamen Veränderungen nah am Original bis zur freien Kombination bekannter Elemente in neuen Konstellationen. Dabei können bei der Computerkunst die verwendeten Artefakte zwei unterschiedlichen Ebenen entstammen:

- Es können zum einen die Programme bzw. die Algorithmen sein, mit denen die ästhetischen Objekte erzeugt wurden, aus denen dann einzelne Prozeduren bzw. Code-Schnipsel entnommen werden.
- Es können zum anderen die grafischen Ergebnisse selbst sein bzw. einzelne daraus entnommene Elemente und Strukturen (siehe das Kapitel 9: *Der Figurenbaukasten* ...).

Voraussetzung für ein solches *Remixing* ist der Zugang zu den Vorlagen. Das Internet bietet uns heute diesen Zugang quasi als eingebettetes Feature (Manovich, 2010) und damit die Voraussetzung für neue Formen der Aneignung von Kultur. Es stellt gewissermaßen eine generative Plattform mit einem nahezu unerschöplichen Zugriff auf verwertbare Artefakte dar, mit denen weiter gearbeitet werden kann. Nicht zuletzt die

[12] In einer Videodokumentation *Everything is a Remix* setzt sich Kirby Ferguson mit dem Remix-Phänomen auseinander. Anhand vieler Beispiele behandelt er die aus seiner Sicht grundlegenden Elemente der Kreativität: *Copy, Transform, Combine* (zu sehen ist der Film unter http://everythingisaremix.info/watch-the-series/).

sozialen Medien haben dem *Remixing* eine explosionsartige Zunahme beschert. In den FabLabs und Makerspaces hat es inzwischen im Rahmen der Maker-Bewegung sogar die Welt der physischen Objekte erobert (Kyriakou & Nickerson, 2014).

Auf der Website der Scratch-Community wird das Remixen von Programmier-Projekten zum Prinzip erhoben: *„Wir glauben, dass das Ansehen und Verändern interessanter Projekte ein guter Weg zum Lernen des Programmierens ist und zu guten, neuen Ideen führt. Das ist der Grund, warum der Quellcode für jedes auf der Scratch-Website veröffentlichte Projekt sichtbar ist."*

Dort wird gleichzeitig auch auf einen zentralen Aspekt verwiesen, der einer gedankenlosen Verwendung fremder Ideen oder Skripte entgegenwirken soll: *„Wir betrachten auch eine kleinere Veränderung als einen gültigen Remix, solange dem ürsprünglichen Projekt-Ersteller und anderen, die einen signifikanten Beitrag zu dem Remix geleistet haben, Anerkennung gezollt wird."*

So eingeordnet zählen denn auch für Steve Wheeler (2016) *reusing, remixing and repurposing* zu den *digital literacies*, um die die klassischen Kulturtechniken zu ergänzen sind.

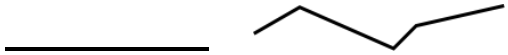
Auch für das *Remixing* der Bilder und Ausgangselemente gibt es wiederkehrende Vorgehensweisen:

- Die Bestimmung der Möglichkeiten zur Abwandlung und/oder Erweiterung des Ausgangsmaterials.
- Die Kombination der Elemente und Strukturen mehrerer Vorlagen in neuen Zusammenstellungen und Konstellationen.

Nach diesen Ansätzen, die Merkmale der frühen Computerkunst darzustellen und deren *Recoding & Remixing* einzuordnen, wird es in den folgenden Kapiteln darum gehen, die Grundlagen für die Entwicklung der notwendigen *Algorithmen* und deren *Programmierung* zu erarbeiten.

2.3 Grafikelemente und Programmierkonzepte

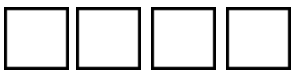
Die eigene Codierung von Computerkunst, d.h. die Erzeugung ästhetischer Objekte durch die Umsetzung entsprechender Algorithmen in Computerprogramme, ist einfacher, als es auf den ersten Blick erscheinen mag. Bei meiner Analyse der frühen Computerkunst (Bild 1 - 7 und vielen mehr) hat sich gezeigt, dass eigentlich einige wenige *grafische Grundelemente* ausreichen, um viele verschiedene Werke zu beschreiben und sie dann algorithmisch nachzuempfinden, also zu *recoden*. Sie sind in der folgenden Tabelle zusammengefasst:

Punkt und Pixel	
Linien und Linienzüge	
Linienschraffuren	

Quadrate und Vielecke	
Kreise und Ellipsen	
(Sinus-) Kurven	
Ausgabe in Schwarz/Weiß	

Deren programmtechnische Umsetzung soll deshalb am Anfang stehen. Diese typischen Grafikelemente werden im Kapitel 9: *Der Figurenbaukasten: Punkte, Linien und mehr* erarbeitet. Sie sollen dann in den unterschiedlichsten Kontexten Verwendung finden..

Ähnlich übersichtlich sind zunächst die *Programmierkonzepte* (im Kapitel 1: *Computerkunst* bereits genannt und in Bild 1 - 7 alle zu finden), die wir für die ersten Schritte in der Computerkunst benötigen:

Wiederholung der Grundelemente	
Variation der Grundelemente	
Zufallsänderungen	

Mit dem *Recoding & Remixing der Computerkunst* möchte ich das Interesse an abstrakten, ästhetischen Objekten mit der Programmierung solcher Objekte verbinden und mit dem Experimentieren zur Erzeugung eigener Grafiken. Malprogramme, wie z.B. Sketchbook oder Photoshop, bieten mit einer grundlegend anderen Konzeption dafür nur sehr eingeschränkte Möglichkeiten. Letztlich kommen wir nicht um die Verwendung einer Programmiersprache herum; anders können wir den Computer und Peripheriegeräte nicht dazu bringen, die gewünschten Ergebnisse zu erzeugen.

Die Programmiersprachen bieten dafür entsprechende Sprachelemente, also mehr oder weniger spezifizierbare Schleifenkonstrukte für Wiederholungen, die Verwendung von Variablen für die nutzergesteuerte Variation von Kennwerten der Grafiken und Zufallszahlengeneratoren für die zufällige, programmgesteuerte Variation bildbestimmender Kenngrößen. Wir werden sie in den Kapiteln 5: *Erste Schritte mit Snap*! bzw. 8: *Alles Zufall* ... kennen lernen.

3. WAS IST SNAP! ?

Es gibt sehr viele Programmiersprachen (wohl mehr als 2000!). Damit stellt sich also die Frage, welche für unsere Zwecke besonders gut geeignet ist. Ich habe mich für Snap! entschieden, einen Abkömmling von Logo. Die Gründe dafür möchte ich im Folgenden skizzieren. Mit Snap! verwenden wir eine visuelle Programmierumgebung, bei der die Sprachelemente mit grafischen Symbolen dargestellt werden und die Programmierung im Wesentlichen darin besteht, diese Symbole wie in einem Puzzle zusammen zu fügen. Durch diesen intuitiven Zugang eignet sich Snap! besonders für Programmieranfänger.

Die Entwicklung von Snap! geht auf das Hobbyprojekt BYOB (Build Your Own Blocks) des deutschen Softwareentwicklers Jens Mönig zurück, ursprünglich als Erweiterung von Scratch 1.2 gedacht, inzwischen als unabhängige Neuentwicklung in enger Zusammenarbeit mit Brian Harvey (der seinerseits das UCB Logo[13] geschrieben hatte, den Quasi-Standard der Sprache Logo). Snap! wird von der University of California at Berkeley unterstützt und in ihren Ausbildungsprojekten großflächig eingesetzt.

Snap! hat nicht nur den Vorteil des intuitiven Zugangs zum Programmieren. Es unterstützt ein experimentelles Vorgehen (vgl. Modrow, 2013, S. 5 ff.), weil alle Befehlsblöcke (auch Kacheln genannt) und Programmteile sich unmittelbar mit Mausklick ausführen lassen. Ihre Wirkung kann so unmittelbar beobachtet werden. Gleichzeitig ist Snap! sehr fehlertolerant. Wenn Fehler auftreten, führt dies nicht zum Programmabsturz, sondern der verursachende Bereich wird rot markiert - meist ohne weitere dramatische Folgen.

Dies sind natürlich auch Merkmale von anderen visuellen Programmierumgebungen wie Scratch, TurtleArt, blockly oder App Inventor. Warum also gerade Snap!?

Wer Snap! nutzen möchte, muss kein Programm installieren: Da für Snap! mit JavaScript und HTML5 eine inzwischen etablierte und zukunftsträchtige Plattform gewählt wurde, kann es in allen verbreiteten Browsern genutzt werden (bevorzugt wird bisher allerdings aus Kompatilitätsgründen Chrome). Selbst auf Tablets und Smartphones ist es ohne eine spezielle App lauffähig.

Mit Snap! können neue Befehle definiert werden in Form von Blöcken. Das ist ein wichtiges Merkmal praktisch aller modernen höheren Programmiersprachen: Befehlsfolgen lassen sich unter einem neuen Namen zusammenfassen und erweitern so die Programmiersprache. Das bietet den Vorteil kürzerer Programme, denn Codesequenzen lassen sich auslagern und können dann unter dem neuen Namen mehrfach benutzt werden. Da die Blöcke unabhängig voneinander entwickelt und getestet werden können, sinkt zudem die Fehleranfälligkeit von Programmen.

---

[13] Versionen des UCB Logo für Windows, MacOS X und Linux finden sich unter https://www.cs.berkeley.edu/~bh/logo.html, dazu sein grundlegendes 3-bändiges Werk *Computer Science Logo Style.*

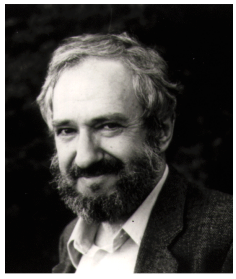
Snap! unterstützt ganz unterschiedliche Programmierstile, denn es ist sowohl prozedurale, imperative, funktionale und objektorientierte Programmierung möglich[14]. Leistungsfähige Konzepte, wie die Implementation komplexer Datenstrukturen mit Listen[15] oder First-Class-Funktionen und -Listen[16], machen Snap! zu einem mächtigen Instrument. Das bietet die Möglichkeit, Teile von Programmstrukturen zur Laufzeit zu verändern und macht Snap! zu einer „programmierbaren Programmiersprache". Ich erwähne dies, obwohl es weit über die Notwendigkeiten unserer Programmierprojekte hinausgeht. Immerhin, die Mächtigkeit des Arbeitens mit Listen werden wir uns zunutze machen (etwa im Kapitel 25.2: *Hommage à Morellet*).

Snap! bietet damit flexible Bearbeitungs- und Erweiterungsmöglichkeiten. Das hat inzwischen dazu geführt, dass es eine Reihe von Snap!-Abkömmlingen für spezielle Anwendungen gibt. So können mit BeetleBlocks Vorlagen für 3D-Drucker erstellt werden; mit turtlestitch lässt sich eine Stickmaschine programmieren und Snap4Arduini verbindet Snap! mit der elektronischen Plattform Arduino.

Eine weitere wichtige Funktion für das praktische Arbeiten ist die Sicherung der eigenen Programme und Ergebnisse. Mit Snap! gibt es mehrere Möglichkeiten, Programme bzw. Projekte zu speichern, sowohl in der „Snap-Cloud", sei es privat oder öffentlich, als auch lokal, entweder im Speicher Ihres Browsers oder auf der Festplatte. Beim Export auf die Festplatte werden die Projekte in XML-Notation abgelegt, einem reinen Textformat. Bei Bedarf könnten Sie diese Dateien deshalb sogar mit einem Texteditor bearbeiten.

3.1 Ein Blick zurück: Die Anfänge von Logo

Snap! basiert auf Logo. Für dessen Anfänge müssen wir bis 1964 zurückgehen. In diesem Jahr kam der Mathematiker Seymour Papert (1928 - 2016) an das berühmte MIT (Massachusetts Institute for Technology) in Boston. Zusammen mit Marvin Minsky (1927 - 2016) – einem der Gurus der Forschung zur Künstlichen Intelligenz (KI) – gründete er das MIT Artificial Intelligence Laboratory. Über den Kreis der KI-Fachleute hinaus ist Papert aber durch die Entwicklung von Logo bekannt geworden.



Seymour Papert

Papert hatte zuvor fünf Jahre bei Jean Piaget in Genf gearbeitet und sich dort bei dem Entwicklungspsychologen intensiv damit beschäftigt, wie Kinder lernen. Mit diesem

[14] Wer informatische Hintergründe zu den unterschiedlichen Programmierstilen sucht, findet bei Schwill (1999) eine Einführung.

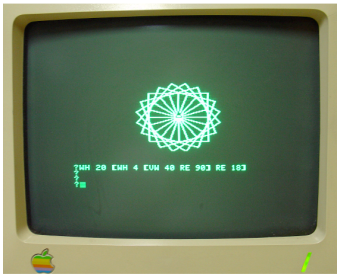
[15] Ein Konzept, das auf LISP (LISt Processor), die zweitälteste höhere Programmiersprache, zurückgeht.

[16] Dieses Merkmal geht auf die Sprache Scheme zurück. Dazu B. Harvey: *„In the glory days of the MIT Logo Lab, we used to say, "Logo is Lisp disguised as BASIC." Now, with its first class procedures, lexical scope, and first class continuations, Snap! is Scheme disguised as Scratch."*

pädagogisch-psychologischen Hintergrund wollte er nun Werkzeuge entwickeln, mit denen Kindern das „Beste der Computerwissenschaften“ an die Hand gegeben werden sollte, damit sie ihre Denk- und Lernweisen verbessern können. 1967 erschien als Ergebnis seiner Überlegungen die erste Version von Logo.

Als Sprache für Kinder, aber eben keine „Spielzeugsprache“, war Logo (ihrerseits ein Abkömmling der listenorientierten KI-Sprache LISP) modular, erweiterbar und interaktiv konzipiert. Ihre Charakterisierung mit „no treshold, no ceiling“ (keine Einstiegshürde, keine Begrenzung nach oben) soll andeuten, dass Kinder mit Logo sehr rasch problemorientiert arbeiten können, dass die Sprache aber mächtig genug ist, Lernende an komplexe Probleme heranzuführen. Als zentrale Komponente von Logo wurde ab 1970 die Turtle-Grafik (Schildkrötengrafik[17]) eingeführt, durch die mit einfachen Grundbefehlen ansprechende und komplexe Grafiken erzeugt werden können. Gerade die Schildkrötengrafik ermöglicht einen motivierenden, spielerischen Einstieg in das Programmieren. Sie ist deshalb auch in andere Sprachen übenommen worden (etwa Pascal, Java oder Python).

Die stärkere Verbreitung von Logo begann in den späten Siebzigern mit den ersten erschwinglichen Personalcomputern. Die entsprechenden Logo-Versionen gab es für die 8-bit Rechner Apple II (Apple Logo) und Texas Instrument TI 99/4 (TI Logo). Ab 1980 wurden etliche größere Schulversuche mit Logo durchgeführt, auch in der Bundesrepublik (z.B. Böcker, Fischer & Plehnert, 1986).



MIT Logo für den Apple II

Der Name Logo stand bald nicht nur für die Programmiersprache, sondern auch für prägnante (aber nicht unumstrittene) pädagogische Vorstellungen von interaktiven Lernumgebungen. So schreibt Harald Abelson (1983, S. VII), einer der Mitarbeiter Paperts:

„Die Sprachen der Logo-Familie sind gezielt so entwickelt, dass sie die Computer zu flexiblen Hilfsmitteln machen, die das Lernen, das Spielen und das Erforschen unterstützen. Wir Wissenschaftler, die wir an Logo arbeiten, lassen uns von der Vision eines pädagogischen Werkzeugs leiten, das zugleich ohne Einstiegsschwelle und ohne Begrenzung nach oben ist. Wir versuchen also, selbst sehr jungen Schülern eine selbständige und selbstbestimmte Beherrschung des Computers zu ermöglichen (...). Zugleich meinen wir, dass Logo ein Allzweckprogrammiersystem sein sollte, das beachtliche Ausdrucksfähigkeit und umfangreiche Formulierungsmöglichkeiten hat. Es ist tatsächlich so, dass wir diese beiden Ziele eher als Ergänzung denn als Widerspruch betrachten.“

[17] In deutschsprachigen Logo-Einführungen (z.B. Abelson, 1983; Hoppe & Löthe, 1984 oder Ziegenbalg, 1986) und in den deutschen Versionen der Programmiersprache Logo wird oft der Ausdruck *Igelgrafik* bzw. *Igel* (wegen der Kürze des Wortes) verwendet.

3.2 Konstruktionismus und Visuelle Programmierung

In seinem Buch „Mindstorms: Kinder, Computer und Neues Lernen“ hat Papert (1982) seine pädagogischen Überlegungen zusammengefasst. Es ist dabei weniger ein Buch über Logo, sondern über Paperts Antrieb, Kindern eine Art Mathematikland zu schaffen, in dem sie Mathematik lernen können sollen, so wie eine Sprache am besten durch Aufenthalt in dem entsprechenden Land gelernt wird. Er leitet unter Berufung auf Rousseau und Piaget ein didaktisches und bildungstheoretisches Modell ab, mit dem gesellschaftliche Anforderungen und Verpflichtungen mit den Bedürfnissen der Kinder in Einklang gebracht werden sollen - eine Lektüre, die immer noch lohnt.

Mit seinem Ansatz hat Papert die *konstruktivistische Lernphilosophie*[18] weiter entwickelt zum *Konstruktionismus*. Er geht von reformpädagogischen Ideen zum *selbstbestimmten Lernen* und zur *Partizipation* aus und verbindet sie mit lerntheoretischen Überlegungen. Im Zentrum steht das kreative Handeln im Sinne der Konstruktion von Dingen oder Artefakten. Das Produzieren der Dinge soll selbständige mentale Aktivitäten und das Verstehen unterstützen; die Produkte bieten dann Anlass für Bewertung und Diskussion. Für Papert und Kollegen bietet gerade die Computertechnik große Chancen, dass die Lernenden diese aktive Rolle bei der Produktion digitaler Medien einnehmen können[19].

Hier, beim *Recoding & Remixing* der Computerkunst, wird nun die Produktion ästhetischer Objekte verbunden mit der Erarbeitung informatischer Konzepte und der konkreten programmtechnischen Umsetzung. Diese Vorgehensweise steht m.E. ganz im Einklang mit Paperts Intentionen.

Die Programmiersprache Logo wurde im Lauf der Jahre um viele zusätzliche Funktionalitäten ergänzt (u.a. die gleichzeitige Steuerung mehrerer Schildkröten). Insgesamt gibt es eine große Zahl an Logo-Versionen, so auch speziell für Tablets (wie z.B. LogoPlus oder Logo Draw). Manche sind rein browserbasiert (wie z.B. Turtle Academy oder papert-logo.). Die Vielfalt der Versionen hat zu einer Aufsplitterung der Nutzergruppen von Logo geführt. Es gibt keinen Sprachstandard und die Versionen unterscheiden sich z.T. in Sprachumfang und Syntax erheblich. Im Grunde ist die Schildkrötengrafik der einzige verbliebene gemeinsame Nenner.

Trotz eines Werkzeugs wie Logo ist der Einstieg ins Programmieren nicht nur für Kinder meist mit hohen Hürden verbunden. Alle Anfänger haben beim Einstieg gleichzeitig viele verschiedene Dinge zu lernen: Syntax und Semantik der Befehle, Formulierung von Algorithmen, Umgang mit der Entwicklungsumgebung und dem Editor, Umgang

[18] Nicht zu verwechseln mit dem *Konstruktivismus*, der streng gegenstandslosen Stilrichtung in der Kunst seit Beginn des 20. Jahrhunderts.

[19] Diese Möglichkeiten werden gerade in der Maker-Bewegung neu ausgelotet: Die Kultur des Selbermachens (DIY: Do it Yourself) wird gezielt erweitert um Werkzeuge aus Mikroelektronik, Robotik, 3D-Druck und Lasercuttern.

mit Fehlern und mehr. Im Vordergrund sollten eingangs jedoch die Konzepte der Programmierung stehen. Professionelle Entwicklungsumgebungen verstellen oft gerade die Sicht auf das Wesentliche. Diese Hürde soll mit visuellen Programmierumgebungen niedriger werden.

Programmiersprachen sind Notationssysteme zur Beschreibung von Berechnungen und logischen Operationen in von Maschinen und Menschen lesbarer Form. Als *visuelles Programmieren* wird nun die Verwendung grafischer Darstellungen in der Softwareentwicklung bezeichnet. Visuelle Programmiersprachen zeichnen sich also dadurch aus, dass eine grafische Notation der Grundbefehle, d.h. Grundsymbole, mit denen computersprachliche Konstrukte repräsentiert werden, zur Beschreibung eines Algorithmus verwendet wird (Schiffer, 1996).

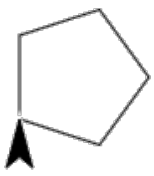
Es ist einleuchtend, dass die grafische Umsetzung komplexer Algorithmen mit vielen Symbolen schnell an eine räumliche Grenze stoßen kann (die auch Deutsch-Limit genannt wird nach dem Informatiker L. Peter Deutsch): Mehr als 50 visuelle Elemente auf dem Bildschirm sind nur schwer übersichtlich darstellbar und zu verarbeiten. Erfreulicherweise werden in der von uns verwendeten visuellen Programmierumgebung Snap! Strukturierungselemente zur Verfügung stehen, um Befehlsfolgen zusammen zu fassen und so das Platzproblem zu umgehen bzw. deutlich zu mildern.

Zur Veranschaulichung der unterschiedlichen Darstellungsformen und den daraus resultierenden Arbeitsweisen wird im folgenden Beispiel die *textuelle* der *visuellen* Programmierung gegenübergestellt. Die dargestellte Befehlsfolge bewirkt das Zeichnen eines Vielecks mit n Seiten[20]. In einer *textbasierten* Sprache (hier: ACSLogo) ergibt die Umsetzung das Programm links; dem ist rechts die Umsetzung der identischen Befehlsfolge mit einer *visuellen* Programmierumgebung (hier: Snap!) gegenübergestellt:

```
Prozedur n_eck n laenge
repeat :n
    [forward :laenge
     right 360/:n]
```



Der Aufuf der textbasierten Prozedur erfolgt durch die Eingabe von **`n_eck 5 50`** und ergibt das folgende Ergebnis:



Der Aufruf in der visuellen Programmierumgebung erfolgt durch Anklicken des entsprechenden Befehlsblocks: 

Das Ergebnis ist in beiden Fällen natürlich dasselbe.

[20] Es geht mir hier nicht um das Verstehen des Algorithmus an sich, sondern nur um den Vergleich der Darstellungsformen.

Der Vorteil beim Zusammenstellen eines Programms per „Drag & Drop“ liegt zum einen in der Vermeidung syntaktischer Fehler (im Beispiel muss nicht mehr auf Klammerungen oder den Doppelpunkt beim Variablenaufruf geachtet werden). So lassen sich falsch geschriebene oder nicht existierende Anweisungen und Operationen gar nicht erst eingeben. Die richtige Reihenfolge der Syntaxkomponenten wird durch die Form der Blöcke und deren Verbindungspunkte oft implizit vorgegeben. Die unterschiedlichen Farben und Formen für Anweisungen, numerische Eingaben, Kontrollflüsse und Ausgaben fördern zusätzlich das Verständnis[21].

Dass visuelle Programmierung besonders Einsteigern mit geringem Vorwissen den Einstieg ins Programmieren erleichtert, belegen inzwischen einige Untersuchungen (vgl. dazu Weintrop & Wilensky, 2015; Price & Barnes, 2015). Dennoch werden visuelle Programmiersprachen häufig nur als Vorstufe zur Verwendung „richtiger“ - nämlich textbasierter - Programmiersprachen akzeptiert (vgl. etwa Dorling & White, 2015). Allerdings sind Zweifel angebracht, ob die Mehrzahl der am Programmieren Interessierten für die Entwicklung wirklich großer Anwendungen die „richtigen“ Sprachen überhaupt benötigen (Modrow, 2013, S. 2). Für das Anwendungsfeld Computerkunst sehe ich jedenfalls keine zwingende Notwendigkeit, andere Werkzeuge zu verwenden.

Vor diesem Hintergrund ist es eigentlich nur konsequent, dass gerade für Logo eine Reihe visueller Programmierumgebungen entwickelt wurden und wir mit ihnen momentan ein Logo-Revival erleben, dessen Ende noch nicht abzusehen ist. Begonnen hat dies 2007 mit Scratch (entwickelt am MIT Media Lab unter Leitung von Mitchel Resnick). Daran knüpfen z.B. TurtleArt, Blockly oder App Inventor an.

Die Entwicklungsumgebung ist auf der Scratch-Homepage[22] direkt lauffähig. Diese Website ist inzwischen die Zentrale für eine weltweit wachsende Gemeinschaft von Programmieranfängern, Schülern, Studenten, Pädagogen und Hobbyisten, die sich gegenseitig motivieren und unterstützen. Durch eine leichte Übersetzbarkeit gibt es sehr viele Sprachversionen und daraus resultierend lokale Gruppen, Kurse, Beispiele und Unterrichtsmaterialien.



Für die Programmierung der Computerkunst verwende ich in diesem Band allerdings Snap!, das in den Folgekapiteln eingeführt und beschrieben wird. Dabei handelt es sich um eine interessante Weiterentwicklung von Scratch mit leistungsfähigen Erweiterungen. Nicht zuletzt dadurch hat Snap! inzwischen seinen Platz im Informatikunterricht gefunden[23].

---

[21] Mit der Zusammenstellung von Befehlsblöcken entsteht praktisch ein Strukturdiagramm, das dann sehr den Nassi-Shneiderman-Diagrammen gleicht, einer genormten Methode der strukturierten Programmierung.

[22] Die aktuelle Version Scratch 2.0 basiert komplett auf Adobe Flash.

[23] Da mit ihr auch anspruchsvolle informatische Konzepte darstellbar sind, reicht ihr Eignungsspektrum bis in den Hochschulbereich; das Curriculum *Beauty and Joy of Computing* und seine Umsetzung in den Online-Kursen BJCx belegen das eindrücklich.

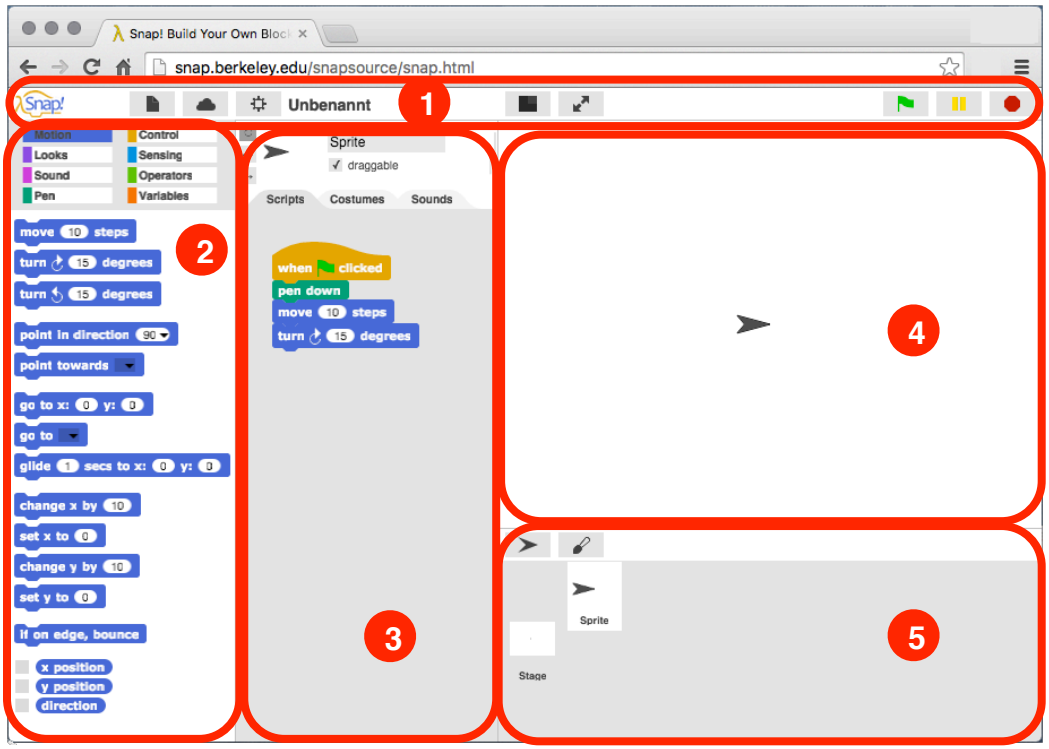
4. SNAP! STARTEN

Es sind ein paar Dinge voraus zu setzen, bevor es losgeht: Sie kennen Ihren Computer und sein Betriebssystem. Sie sind geübt im Umgang mit Tastatur und Maus. Sie wissen, wie Dateien gespeichert und geöffnet werden können. Sie haben mit einem Browser Erfahrung im WorldWideWeb gesammelt. Das sollte reichen, um zu beginnen.

Zum Start von Snap! rufen wir im Browser unserer Wahl die folgende Adresse auf:

http://snap.berkeley.edu/run

Es öffnet sich dann dieses Fenster mit fünf unterschiedlichen Komponenten der Programmierumgebung:



1 Werkzeugleiste:



Das Snap! Logo führt zu Informationen über Snap! selbst (Handbuch, Website und Quellcode).



Das Datei-Symbol führt zu Optionen des Speicherns, Ladens und Importierens von Dateien.



Das Wolken-Symbol führt zum Anmelden (bzw. Benutzerkonto einrichten).



Das Werkzeug-Symbol führt zur Wahl der Sprache (s.u.), Festlegen der Bühnengröße und ähnlichen Optionen.



Schalter zum Verkleinern und Vergrößern der Bühne

Schalter zum Umschalten auf Vollbild bzw. Zurückschalten

Start eines Programms

Pausieren bzw. Weiterlaufen eines Programms

Beenden eines Programms

2 **Befehlsgruppen**: Es gibt acht inhaltlich getrennte und farblich gekennzeichnete Befehlsgruppen, d.h. die **Palette** der darunter erscheinenden Blöcke kann anhand ihrer Farbe inhaltlich zugeordnet werden.

3 **Programmbereich:** Programmiert wird, indem man einen Block anklickt und mit gedrückter Maustaste in diesen Programmbereich zieht.

4 **Bühne:** Auf der Bühne bewegt sich die Schildkröte (in Snap! nur als Pfeil dargestellt) und hinterlässt - entsprechend dem Programmcode - ihre Spuren als Zeichnung (Näheres dazu im nächsten Abschnitt).

5 **Objektbereich:** Dort finden sich Bearbeitungsmöglichkeiten für alle Objekte (Bühne und Sprites, d.h. Grafikobjekte wie die Schildkröte).

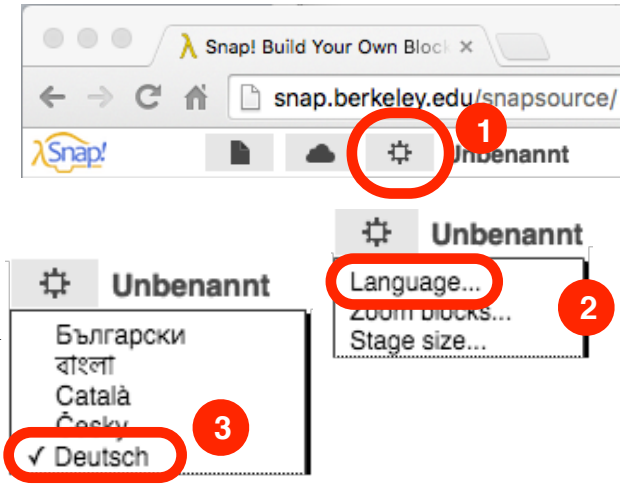
Von Snap! gibt es über 30 verschiedene Sprachversionen. Wir stellen daher als erstes die Sprache der Programmierumgebung - und damit die Befehlssyntax - auf Deutsch um[24].

Dazu wird in der Snap!-Menüzeile (*Achtung*: Nicht zu verwechseln mit der Menüzeile des Browsers bzw. des Rechnersystems!)

1 das *Werkzeugsymbol* angeklickt,

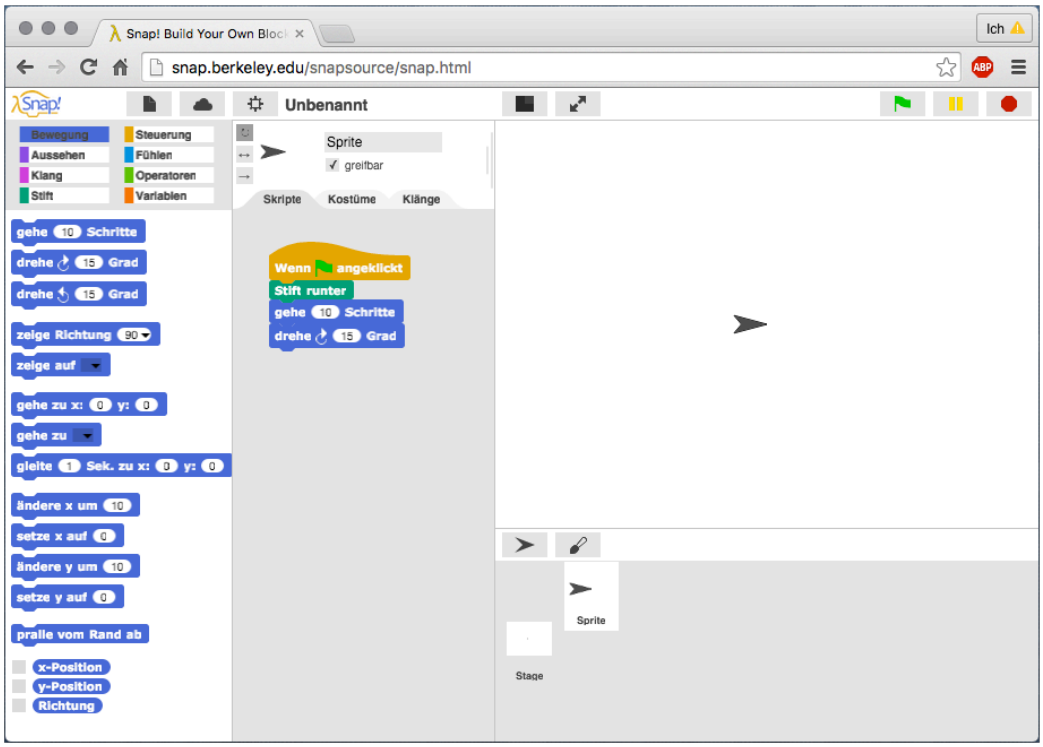
2 im aufgeklappten Menü der erste Menüpunkt *Language* angewählt

3 und im daraufhin aufklappenden Menü der Punkt *Deutsch*.



[24] Wer sich für Details der Programmierumgebung interessiert, sollte sich das Manual herunterladen (zu finden unter: SNAP! Reference Manual 4.0.10). Einen umfangreichen Online-Kurs bietet das Projekt Beauty and Joy of Computing (BJC). Dort wird kleinschrittig in die Funktionalität von Snap! eingeführt.

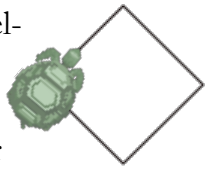
Das oben gezeigte Fenster der Programmierumgebung sieht danach aus wie folgt:



Eine Zusammenstellung wichtiger Menüoptionen von Snap! findet sich in *Anhang A*. Dort werden die Interaktionselemente von Snap! erläutert. Weiterhin werden dort etliche Tipps und Tricks vorgestellt, die den Umgang mit Snap! erleichtern.

Hinweis: *Hier und im Folgenden beziehen sich alle Angaben, die Optionen in der Werkzeugleiste, Befehle in den Blockpaletten sowie die unter der Datei-Option nachladbaren Bibliotheken, Kostüme und Klänge betreffen, auf die Snap!-Version 4.1 (Stand Januar 2018). Da Snap! aktiv weiter entwickelt wird, können sich in der aktuellen Version, mit der Sie arbeiten, Änderungen ergeben haben!*

4.1. Orientierung auf der Bühne

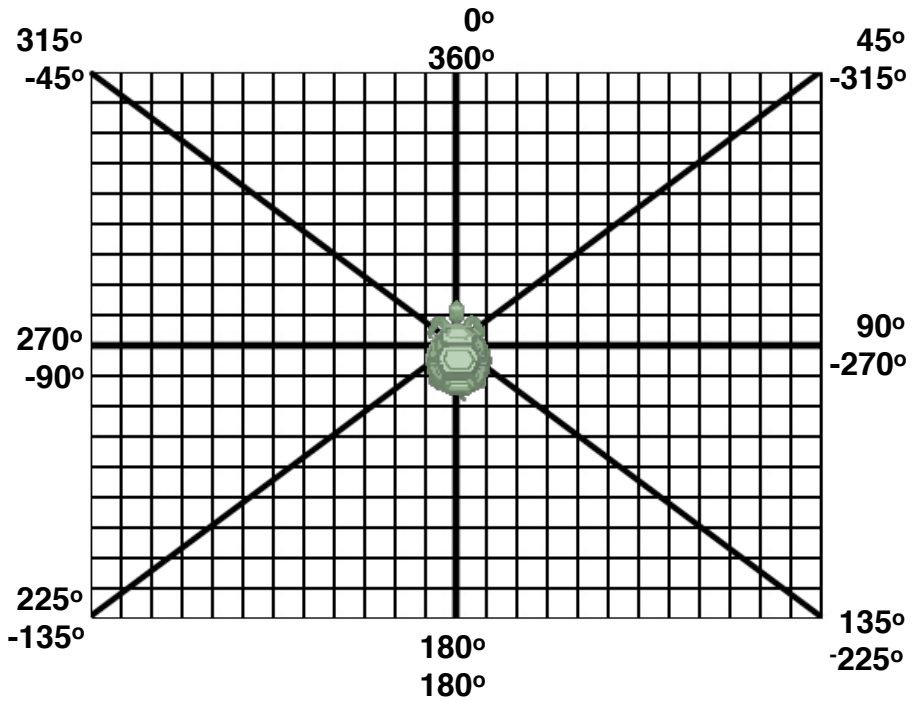


Im Zentrum dieser Einführung in die Computerkunst steht die Erstellung unterschiedlichster Computergrafiken. Die Ergebnisse unserer Programmierbemühungen sind jeweils auf der **Bühne** zu sehen, auf der sich die Schildkröte bewegt. Es ist also wichtig, den Aufbau dieser Bühne zu verstehen.

Wenn wir die Schildkröte auf der Bühne bewegen, dann entspricht die dabei entstehende *Schildkrötengrafik* einer *natürlichen Geometrie* (Hoppe, 1984, S. 24 f.). Sie vermeidet den Bezug auf ein äußeres absolutes Koordinatensystem. Sie wird mit Befehlen wie **gehe x Schritte** oder **drehe x Grad** gesteuert.

Die Schildkröte schaut „geradeaus“ und folgt stets ihrer augenblicklichen Blickrichtung. Die Blickrichtung wird in Grad gemessen. Schauen wir von „außen“, also von oben auf die Bühne mit der Schildkröte, entspricht ihre Blickrichtung nach *oben* 0 Grad

und dann im Uhrzeigersinn nach *rechts* 90 Grad, nach *unten* 180 Grad, nach *links* 270 Grad. Eine vollständige Drehung besteht aus 360 Grad. Richtungsangaben können auch entgegen dem Uhrzeigersinn angegeben werden, etwa die Blickrichtung *links* mit -90 Grad:



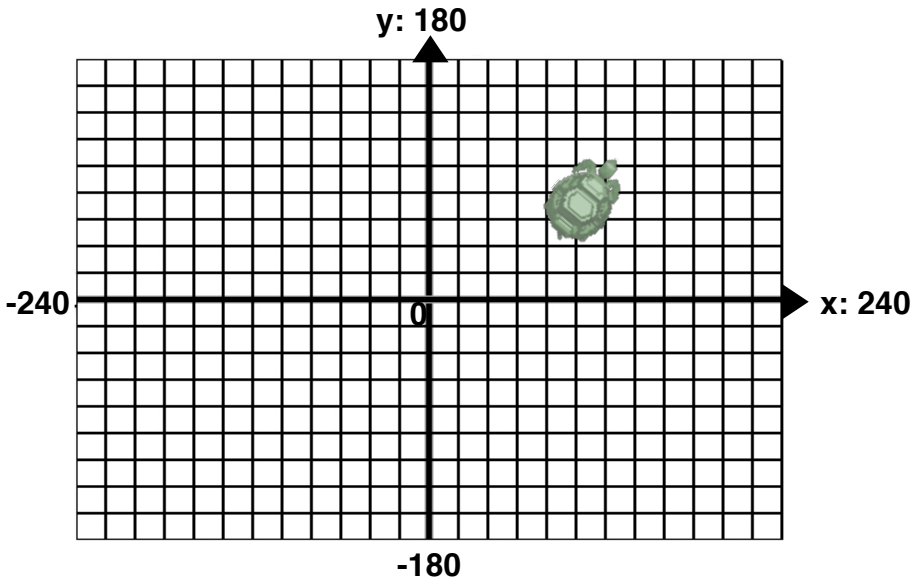
Angenommen, die aktuelle Blickrichtung der Schildkröte ist 45 Grad. Der Befehl **drehe 45 Grad** bedeutet dann nicht, dass die Schildkröte anschließend in die Richtung 45 Grad schaut, sondern der Drehungswinkel wird zur aktuellen Blickrichtung hinzu addiert. Die neue Blickrichtung beträgt demzufolge 90 Grad.

Das Verhalten der Schildkröte entspricht einem „anthropomorphen Modell“ (Papert, 1982, S. 87 ff.), denn ihre Bewegungen können motorisch oder in Gedanken nachvollzogen werden („gehe zwei Schritte nach vorne, drehe dich nach links ...“).

Die Schildkrötengrafik beschreibt damit die intrinsischen Eigenschaften der geometrischen Figuren; ein Quadrat z.B. wird dadurch immer korrekt ausgeführt, egal an welcher Stelle des Bildschirms und in welcher Blickrichtung die Schildkröte sich vor der Ausführung der Befehle befindet.

Natürlich kommt auch die Schildkrötengrafik nicht ganz ohne Bildschirmkoordinaten aus; schließlich müssen Schrittlänge und Drehrichtung in konkrete Bildschirmpunkte umgerechnet werden. Dabei befindet sich der Mittelpunkt immer im rechnerischen Mittelpunkt der Bühne, die im Ausgangszustand horizontal 480 Punkte und vertikal 360 Punkte groß ist.

Aus dem Mathematikunterricht sind wir ein ähnliches, aber doch etwas anderes Koordinatensystem gewohnt, nämlich das *kartesische Koordinatensystem*[25], mit dem sich viele geometrische Sachverhalte am besten beschreiben lassen. Auch dabei treffen sich die beiden senkrecht aufeinander stehenden Achsen, meist x-Achse und y-Achse genannt, im Koordinatenursprung. Ein Punkt wird immer durch Angabe von x- und y-Wert bestimmt. Das entspricht zwar auch den Angaben für die Position der Schildkröte, allerdings könnte in diesem System ihre Position nur durch diese Koordinatenangabe bestimmt und rechnerisch verändert werden.



Es leuchtet ein, dass dann zur Darstellung eines Punktes auf dem Bildschirm immer eine (mehr oder weniger umständliche) Umrechnung von den kartesischen Koordinaten in die Bildschirmkoordinaten erfolgen muss[26]. Bei der Schildkrötengrafik müssen wir uns darum nicht mehr kümmern. Die Schildkröte kann dies nämlich getrost „vergessen"; sie geht immer nur von ihrer aktuellen Position und Blickrichtung aus. Für die meisten der folgenden Grafikprojekte ist die Schildkrötengrafik also die intuitiv naheliegende Darstellungsform. In etlichen Ausnahmefällen ist es für uns aber auch nützlich, die Perspektive von außen bzw. von oben einzunehmen. Deshalb kommt in etlichen Projekten dann doch das kartesische Koordinatensystem zur Verwendung.

Hinweis: *In der Regel entspricht die Positionsangabe der Schildkröte dem Mittelpunkt des jeweils verwendeten Ojekts (hier also dem Schildkrötensymbol), allerdings nicht immer, vor allem wenn komplexere Ojekte verwendet werden. Bei dem in Snap! voreingestellten Pfeil entspricht dagegen die Pfeilspitze der Positionsangabe!*

---

[25] Bei mathe online findet sich eine sehr schöne Einführung in *Zeichenebene und Koordinatensystem* und seine Bedeutung für das Zeichnen geometrischer Figuren (Punkte, Strecke, Polygone, Kreise).

[26] Beide Systeme unterscheiden sich wiederum von dem sonst üblicherweise auf dem Computerbildschirm bzw. innerhalb eines Bildschirmfensters verwendeten Koordinatensystems, dessen Ursprung [0,0] links oben liegt.

5. ERSTE SCHRITTE IN SNAP!

Bevor es mit richtigen Bildern losgeht brauchen wir erstmal einige wenige Grundlagen über den Aufbau und den Umgang mit dem Programmiersystem. Die Beispiele in diesem Kapitel sollen deshalb erste Erfahrungen im Umgang mit der Schildkrötengrafik vermitteln. Gleichzeitig werden allgemeine Programmierkonzepte vorgestellt und mit konkreten Beispielen gezeigt, wie diese jeweils in Snap! umgesetzt werden können. Konkret werden dazu in diesem Kapitel eingeführt[27]:

- der Direktmodus,
- die Wiederholung,
- die Modularisierung und Erweiterbarkeit,
- die Verallgemeinerung und
- die Kontrollstrukturen.

5.1 Es tut sich was ... im Direktmodus

Unter dem Direktmodus verstehen wir die Eigenschaft von Snap!, dass einzelne Befehle oder Befehlsfolgen durch Anklicken unmittelbar zur Ausführung gebracht werden können. Die Auswirkung der Befehle bzw. Befehlsfolgen kann so unmittelbar beobachtet und analysiert werden. Da es bedeutet, dass jeder Befehl sofort in eine für den Computer ausführbare Form übersetzt und ausgeführt wird (Interpreter), ist die Ausführungsgeschwindigkeit limitiert. Soll die Geschwindigkeit erhöht werden, muss das Gesamtprogramm übersetzt und dann ausgeführt werden (Compiler)[28].



Die Funktionalität von Snap! (also die in Snap! verfügbaren Befehle), lässt sich damit experimentierend gut erschließen. Die Fülle der Befehle ist in acht Kategorien unterteilt, die in farblich unterschiedenen Paletten zugänglich sind. Einige davon benötigen wir in diesem Kapitel, andere werden wir sukzessive in späteren Kapiteln kennen lernen.



Die Auswirkung einzelner Befehle[29], die immer die Form von Puzzle-Teilen haben, werden bei **Bewegung** besonders anschaulich sichtbar, weil sie die Schildkröte (den Pfeil) auf der Bühne direkt beeinflussen (bewegen, drehen, orientieren).



[27] Snap! unterstützt weitere wichtige Konzepte wie Listen, Ereignisverarbeitung, Objektorientierung und Parallelverarbeitung, die an späterer Stelle behandelt werden.

[28] Auch Snap!-Programme können mit Snapp! compiliert werden: http://snapp.citilab.eu

[29] Zu (fast) allen Befehlen von Snap! kann durch Anklicken mit der rechten Maustaste ein Hilfe-Fenster geöffnet werden, in dem die jeweilige Funktion - oft mit kleinen Beispielen - erläutert wird (derzeit nur in Englisch!).

Tipp: *Wenn Sie die Befehlssymbole in der Palette anklicken, bewirkt das deren direkte Ausführung. Experimentieren Sie dazu mit eigenen Werten für* **gehe** *oder* **drehe**. *Die Auswirkungen können Sie kontrollieren, wenn Sie bei* **x-Position**, **y-Position** *und* **Richtung** *Häkchen setzen. Deren Werte werden dann kontinuierlich eingeblendet.*





Bisher hinterlässt die Schildkröte bei ihren Bewegungen keine Spur. In der Palette **Stift** kann das aber mit dem Befehl **Stift runter** eingeschaltet werden (Gegenstück dazu ist natürlich **Stift hoch**). Mit **wische** lassen sich alle Spuren wieder löschen (die Schildkröte verbleibt dabei an ihrer aktuellen Position und behält ihre aktuelle Blickrichtung). Mit **gehe zu x: 0 y: 0** kann sie bei Bedarf wieder zum Mittelpunkt geschickt und mit **zeige Richtung** ihre Blickrichtung neu bestimmt werden (die Voreinstellung dafür ist 90, d.h. Blickrichtung rechts).

Auch in den Paletten **Aussehen**, **Klang** und **Operatoren** kann durch Anklicken der Befehle direkt in der jeweiligen Palette deren Funktionsweise erschlossen werden.

Tipp: *Sie können die Schildkröte auch einfach mit der Maus an beliebige Stellen auf der Bühne bewegen. In der Palette* **Stift** *können Sie mit der Farbe und derStiftdicke der Spur experimentieren.*

Am besten einfach mal ausprobieren!

Befehle:



In den Paletten finden sich neben den Befehlsblöcken einige *Reporter*. Diese abgerundeten Blöcke führen keine Befehlsfolgen aus, sondern liefern einen Wert, der in anderen Blöcken verwendet werden kann.

Reporter:



Eine besondere Form (im Wortsinne verdeutlicht durch ihre hexagonale Form) der Reporter sind *Prädikate*, die immer Wahrheitswerte (**wahr** oder **falsch**) liefern.

Prädikate:



5.2 Eins nach dem anderen ...

Wir wollen die Schildkröte gezielt auf der Bühne bewegen und mit ihrer Spur unsere Grafiken erzeugen. Da macht es natürlich wenig Sinn, immer zwischen den Paletten und Befehlssymbolen hin und her zu springen und alles einzeln anzuklicken. Einfacher ist es, die Befehle in den Programmbereich zu ziehen, sie aneinander zu reihen und damit Algorithmen zusammen zu stellen, die dann „am Stück“ ausgeführt werden.

Ein *Algorithmus* ist letztlich nichts anderes als eine eindeutige Abfolge von Handlungsvorschriften zur Lösung eines *Problems*. Weil dazu aber eigentlich bereits alle Schritte zur Lösung bekannt sein müssen, sprechen übrigens die Kognitionspsychologen in diesem Fall eher von einer *Aufgabe*.

Algorithmen bestehen aus endlich vielen, wohldefinierten Einzelschritten. Sind sie in einer Programmiersprache darstellbar, können sie in einem Computerprogramm implemetiert und ausgeführt werden.

Das werden wir von **Snap!** *brauchen:*





Mit dem Befehl **gehe x Schritte** bewegt sich die Schildkröte um die Zahl von x Bildpunkten in die Richtung, in die sie aktuell zeigt. Falls der Zeichenstift abgesenkt ist, wird bei dieser Bewegung eine Strecke gezeichnet. Ansonsten wird die Schildkröte nur weiter bewegt.

Wird eine negative Zahl eingegeben, bewegt sich die Schildkröte um die entsprechende Zahl von Bildpunkten entgegen der aktuellen Blickrichtung (dies entspricht dem Befehl **Rückwärts/back**, der in anderen Logo-Versionen in der Regel zur Verfügung steht.





Mit dem Befehl **drehe rechts x Grad** wird die Schildkröte relativ zur aktuellen Richtung um den durch **x** angegebenen Winkelwert (in Grad) im Uhrzeigersinn gedreht.



Mit dem Befehl **drehe links x Grad** wird die Schildkröte relativ zur aktuellen Richtung um den durch **x** angegebenen Winkelwert (in Grad) entgegen dem Uhrzeigersinn gedreht.

Beide Drehbefehle können auch mit negativen Zahlen genutzt werden und ersetzen sich dann gegenseitig. Das kann bei berechneten Drehrichtungen hilfreich sein.

Statt Zahlenwerten können auch durch arithmetische Operationen gewonnene Werte übergeben werden, also z.B.



Wir beginnen mit einem ersten Befehl:



1. Ziehen des Befehls aus der Blockpalette in den Programmbereich.
2. Anklicken des Textfeldes mit 10, Überschreiben mit 100 als gewünschter Schrittzahl. Das Anklicken dieses Befehlsblocks bewirkt dann
3. die Ausführung des Befehls auf der Bühne.

Die Schildkröte bewegt sich in unserem Fall 100 Bildschirmpunkte (Pixel) in ihre Blickrichtung (d.h. nach Programmstart nach rechts)[30]. Die Blick- bzw. Bewegungsrichtung der Schildkröte kann dann durch den Befehl **drehe x Grad** geändert werden. Wir fügen deshalb diesen Block hinzu:



1. Wird der Block in die Nähe der Andockstelle eines anderen Blocks geschoben, leuchtet dieser weiß auf. Wird die Maus losgelassen, rastet der Block an der markierten Stelle ein.
2. Sind mehrere Blöcke aneinander gefügt, wird bei deren Anklicken
3. die so entstandene Befehlsfolge ausgeführt, hier also die Bewegung um 100 Bildpunkte und anschließende Drehung, so dass die Blickrichtung der Schildkröte nun nach unten zeigt.

Das Zeichnen eines Quadrats[31] kann also beispielsweise mit dem Befehl **gehe x Schritte** begonnen werden und es ist naheliegend, wie daraus nun ein vollständiges Quadrat zu erhalten ist. Wir fügen dazu die Befehle 1 - 8 aneinander (wie in der folgenden Abbildung).

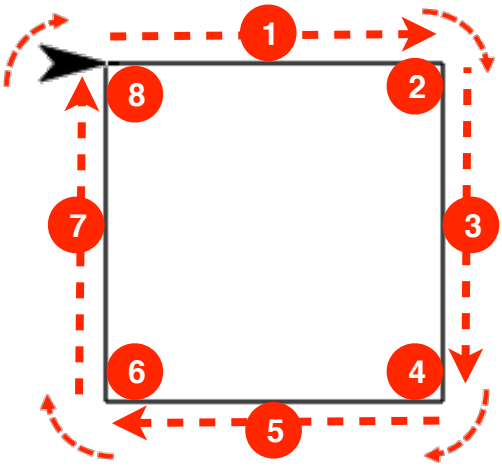
---

[30] Wie beim *Direktmodus* bereits erwähnt, ist zunächst in der Palette **Stift** der Befehl **Stift runter** anzuklicken, damit die Schildkröte eine Spur hinterlässt. Per Voreinstellung ist der Stift nämlich erstmal **Stift hoch**.

[31] In vielen Einführungen in das Programmieren wird als erstes Beispiel für eine einfache Befehlsabfolge in der jeweilig verwendeten Programmiersprache der Satz *Hello World* auf dem Bildschirm ausgegeben (siehe dazu die Sammlung in Wikipedia. Das *Quadrat* ist gewissermaßen das *Hello World* in den Sprachen der Logo-Familie!

Der letzte Befehl **drehe 90 Grad** ist eigentlich für das Zeichnen des Quadrats nicht notwendig. Es empfiehlt sich aber, ihn hinzuzufügen, da dann die Schildkröte nach dem Zeichnen des Quadrats wieder genau an ihrer Ausgangsposition mit ihrer ursprünglichen Blickrichtung steht. Das kann für das kontrollierte Positionieren weiterer Objekte hilfreich sein. Zum Starten der Befehlsfolge wird **wenn angeklickt** verwendet.





Das brauchen wir von **Snap!***:*



Wird eine Befehlsfolge mit dem Hut-Block eingeleitet **wenn angeklickt**, kann sie mit der **grünen Fahne** in der Werkzeugleiste gestartet werden. Das ist besonders dann hilfreich, wenn mehrere Befehlsfolgen gleichzeitig gestartet werden sollen.

Hut-Blöcke werden immer mit dem Wort **Wenn** eingeleitet, was deutlich macht, dass damit auf ein Ereignis - wie Tastendruck, Mausaktion, Nachrichten oder Eintreten einer Bedingung - reagiert werden soll.



Der Befehl **wische** löscht die Bühne (alle Linien und Duplikate der Schildkröte) und belässt die Schildkröte auf ihrer aktuellen Position. Hintergrundbilder auf der Bühne bleiben erhalten.



Mit dem Befehl **Stift runter** wird der Schreibstift abgesenkt und die Schildkröte zeichnet danach ihre Spur auf den Bildschirm.



Mit dem Befehl **Stift hoch** wird der Schreibstift angehoben und die Schildkröte bewegt sich danach auf dem Bildschirm ohne eine Spur zu hinterlassen.

Tipp: *Oft sollen die Befehlsfolgen mehrfach ausgeführt werden. Sind grafische Darstellungen davon betroffen, ist es meist notwendig, vorherige Grafiken zu löschen und die Schildkröte an einen Ausgangspunkt zu bewegen. Dafür sind weitere Befehle hilfreich.*

5.3 Wiederholungen

Wiederholungen sind ein immer wieder auftauchendes Strukturmerkmal in Grafiken (nicht nur) der frühen Computerkunst. Mit ihnen verrichtet der Computer gewissermaßen die für ihn so typische, redundante „Sklavenarbeit“ (Steller, 1992, S. 83), durch die die Erstellung von Bildern mit repetitiven Mustern ermöglicht und erleichtert wird.

Die Erzeugung von vielen Bildvarianten kann in unserem ästhetischen Laboratorium einfach und sehr schnell erfolgen. Der Vergleich ähnlicher Objekte mit gleicher Größe, Erzeugungstechnik und Präzision wird dadurch leicht möglich. Wir werden dieses Potential in vielen Projekten ausnutzen. In diesem Abschnitt soll zunächst gezeigt werden, was allein schon mit der Wiederholung von einzelnen Bildelementen möglich wird.



Das Zeichnen des Quadrats im letzten Abschnitt können wir durch eine Wiederholungsschleife bereits deutlich vereinfachen. Dort tauchen die Befehle **gehe 100 Schritte** bzw. **drehe 90 Grad** jeweils viermal auf. Diese längliche Abfolge kann mit Hilfe des C-förmigen Blocks **wiederhole-x-mal** verkürzt und vereinfacht werden, denn in diesen muss nur noch je einmal der **gehe**- und der **drehe**-Befehl eingefügt werden. Das Ergebnis ist wieder dasselbe Quadrat. Wir werden sehen, dass dies nicht nur eine Verkürzung, sondern auch eine Verallgemeinerung des Programmcodes bewirkt. Die folgende *Hommage à Steller: Rhythmen* (Bild 8) soll das Wiederholungsprinzip verdeutlichen und vertiefen.

5.4 Hommage à Steller: Rhythmen

Erwin Steller (geb. 1928 in München) war Lehrer für Mathematik und Physik, aber auch als Künstler tätig, der seit 1986 mit Ausstellungen seiner Computergrafiken an die Öffentlichkeit trat. Viele seiner Arbeiten sind durch seine mathematische Prägung beeinflusst: *„Mein Ziel ist es, Mathematik ästhetisch zu visualisieren. Dazu verfremde, ja zerschlage ich die Mathematik, um sie neu aufzubauen."* Von 1987 bis 1994 unterrichtete er *Computerkunst* an der Universität Karlsruhe. Sein 1992 erschienenes Buch *Computer und Kunst* ist das Ergebnis seiner Vorlesungen. Er zeichnet darin - an thematischen Schwerpunkten orientiert - ihre Entwicklung historisch und kritisch nach. Seine Vergleiche traditioneller Kunst und der Computerkunst anhand umfangreichen Bildmaterials sind aufschlußreich und lehrreich.

Die Grafik von Steller kann allein mit Wiederholungsschleifen erzeugt werden. In seinem Buch *Computer und Kunst* stellt Steller an einigen Stellen eigene Werke vor, manche

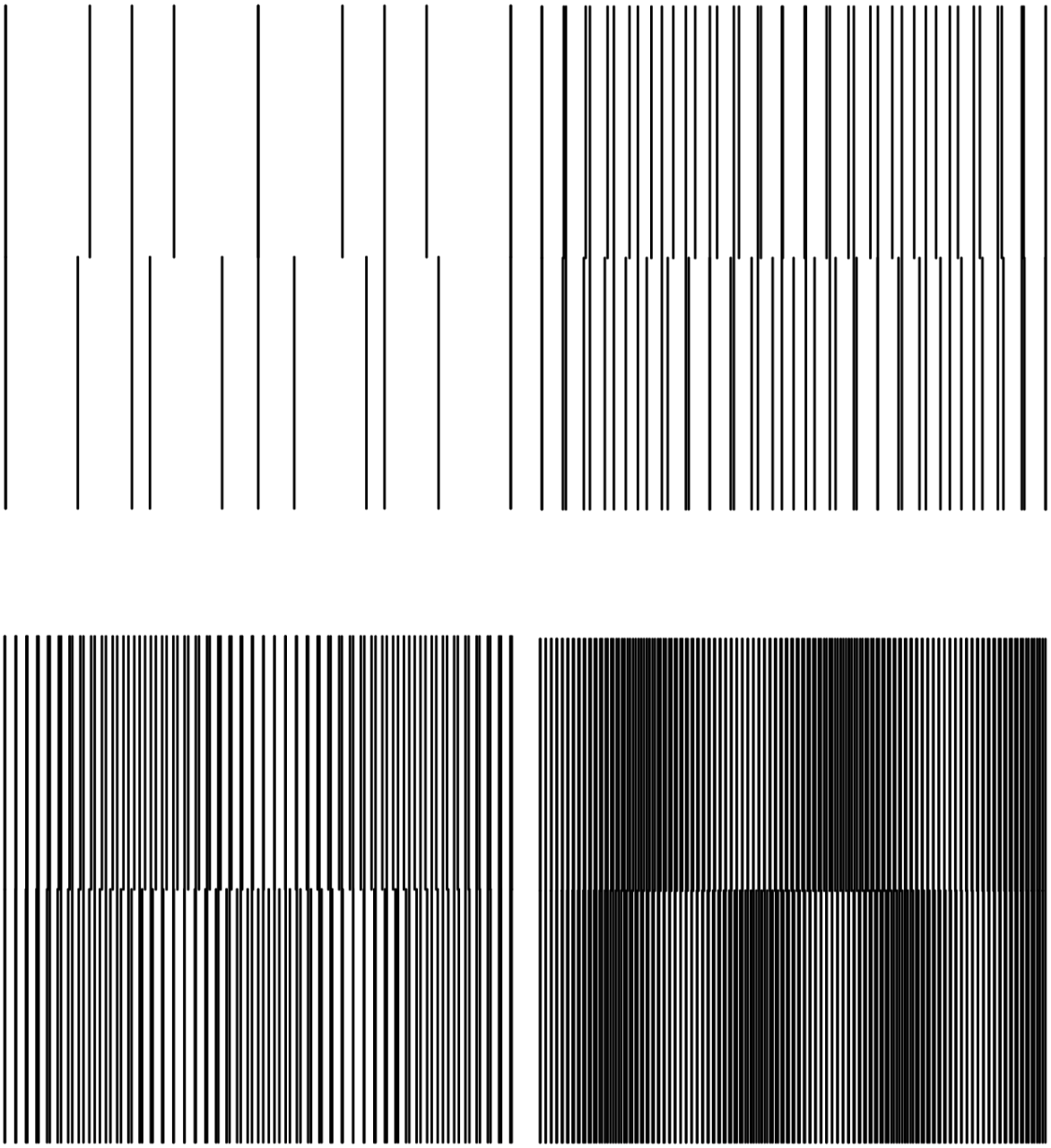


Bild 8: Hommage à Steller: Rhythmen, 4:6:7 (links oben), 20:22:23 (rechts oben), 45:47:48 (links unten), 90:92:93 (rechts unten)

mit Hinweisen auf ihre programmtechnische Umsetzung, so auch für *Rhythmen* (Steller, 1992, S. 88 ff.):

„Die Idee für die Bildserie „Rhythmen" besteht darin, zwei verschieden Perioden [...] gegeneinander auszuspielen. Die ästhetisch einfachste Version zeigt das Prinzip **(**Bild 8 links oben**).** *Sowohl die obere, als auch die untere Hälfte des Bildes sind durch fünf durchgehende Linien in vier gleiche Intervalle geteilt. Diesen vier Grundintervallen werden in der oberen Hälfte 4+2, also 6, in der unteren Hälfte 4+3, also 7 gleiche Intervalle überlagert. Die übrigen Bilder stützen sich auf die Grundintervalle 20, 45 und 90."*

Für das Bild 8 benötigen wir senkrechte Linien und zwar immer eine mehr als Intervalle (für vier Intervalle also fünf Linien, für sieben Intervalle acht usw.).



1. In einem ersten Schritt bewegen wir die Schildkröte zur Ausgangsposition; bei einer Bildbreite und Bildhöhe von 600 Punkten also mit **gehe zu x: -300 y: 300**, mit Blickrichtung nach unten (**180**).

2. Nun wird die Schildkröte in einer ersten Schleife, die durch den Befehl **wiederhole 5 mal** definiert ist (der **wiederhole**-Befehl findet sich in der Palette **Steuerung**), jeweils von der **x-Position -300** aus 600 Schritte (mit **Stift runter**) vorwärts geschickt

3. und danach (mit **Stift hoch**) an ihre neue **x-Position** 600/4 = **150** Einheiten weiter rechts.

4. Die Schildkröte wird wieder zur Ausgangsposition geschickt und in einer zweiten Schleife mit 7 Wiederholungen werden die Linien in der oberen Hälfte (daher mit halber Länge) gezeichnet. Die neue Position befindet sich nun jeweils 600/6 = **100** Einheiten weiter rechts.

5. Für die Linien in der unteren Hälfte wird die Schildkröte zur neuen Ausgangsposition mit **x: -300 y: 0** geschickt. In einer dritten Schleife mit 8 Wiederholungen werden diese Linien (erneut mit halber Länge) gezeichnet. Die neue Position befindet sich nun jeweils 600/7 = **85.7** Einheiten weiter rechts.

Die Bilder mit den Rhythmen 20:22:23, 45:47:48 bzw. 90:92:93 erhalten wir, wenn wir für die Wiederholungsschleifen jeweils die Zahl der Durchläufe auf 21:23:24, 46:48:49

bzw. 91:93:94 erhöhen und entsprechend die **x-Position** um 600/21 = 28.6, 600/23 = 26.1 usw. nach rechts verschieben.

Hinweis: *Die Zahl der Durchläufe und die Änderungen der x-Positionen sind bei dieser Befehlsfolge vorab zu errechnen und jeweils manuell einzutragen. Das ist etwas mühsam. Wir werden im Abschnitt* Verallgemeinerung *sehen, dass wir dieses Vorgehen weiter vereinfachen und die notwendigen Berechnungen direkt in den* **ändere x um***-Blöcken vornehmen können.*

Die Werte für die vier „Rhythmen" in Bild 8 sind in der folgenden Tabelle zusammengefasst. Das gewünschte Bild entsteht, wenn die Werte für die Anzahl der Durchläufe und die Verschiebungen nach rechts eingetragen werden.

Rhythmus	Durchläufe	Verschiebungen
4:6:7	5:7:8	150, 100, 85.7
20:22:23	21:23:24	28.6, 26.1, 25
45:47:48	46:48:49	13.3, 12.77, 12.5
90:92:93	91:93:94	6.7, 6.52, 6.45

Das brauchen wir von **Snap!***:*





Mit dem C-förmigen Block **wiederhole x mal** kann eine Abfolge von Befehlen zusammengefasst und eingefügt werden, die dann entsprechend der angegebenen Anzahl wiederholt wird; mit **wiederhole bis** wird sie solange wiederholt, bis eine abgefragte Bedingung zutrifft.

Die Anzahl der Wiederholungen kann auch durch eine Variable oder einen errechneten Wert angegeben werden:

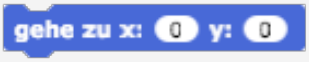




Mit dem Block **fortlaufend** wird eine Befehlsfolge solange wiederholt, bis durch ein äußeres Ereignis oder bei Abfrage eines Grenzwertes die Wiederholung abgebrochen wird.

Das brauchen wir von **Snap!**:





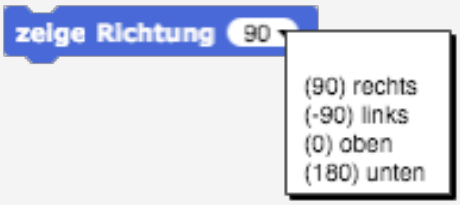
Mit dem Befehl **gehe zu x: y:** bewegt sich die Schildkröte zur Position, die durch **x** und **y** festgelegt wird, im Gegensatz zum Befehl **gehe x Schritte**, bei der ihre Bewegungen *relativ* in Abhängigkeit von der aktuellen Position und Blickrichtung erfolgt. Falls der Zeichenstift abgesenkt ist, wird bei dieser Bewegung eine Strecke gezeichnet. Ansonsten wird die Schildkröte nur weiter bewegt. Ihre Blickrichtung bleibt dabei unverändert.



Mit dem Befehl **ändere x um** wird die Schildkröte um den angegebenen Wert *relativ* in Abhängigkeit von der aktuellen Position in horizontaler Richtung verschoben.



Mit dem Befehl **ändere y um** wird die Schildkröte um den angegebenen Wert *relativ* in Abhängigkeit von der aktuellen Position in vertikaler Richtung verschoben.



Durch den Befehl **zeige Richtung** dreht sich die Schildkröte in die durch die Zahl angegebene Richtung. Die Zahl wird als Gradangabe interpretiert; bei 0° geht die Blickrichtung der Schildkröte nach oben, bei 180° nach unten.



Mit diesen Reportern wird die aktuelle **x-Position** bzw. **y-Position** der Schildkröte zurück gegeben und kann entsprechend ausgewertet werden.

5.5 Modularisierung und Erweiterbarkeit

Fast nebenbei wurde im letzten Abschnitt bei der Erzeugung der Rhythmen ein wichtiges Prinzip der Problemlösung praktiziert: Das *Zerlegen eines Problems in Teilprobleme*. Um die Rhythmen zu erzeugen, wurden die notwendigen Komponenten - also mehrere Schleifen - isoliert, die entsprechenden Algorithmen beschrieben und nacheinander ausgeführt. Diese *Modularisierung* macht die Problemlösung einfacher und den einzelnen Algorithmus überschaubar. Am Beispiel der Erzeugung des Quadrats aus dem Ab-

schnitt *Eins nach dem anderen* ... soll gezeigt werden, wie die Modularisierung in Snap! unterstützt wird.

Wie gesehen, ist das mehrfache Zeichnen gleichartiger Figuren - hier des Quadrats - ziemlich mühsam, denn anfangs mussten Sie dazu die jeweilige Befehlsfolge immer wieder neu eingeben; eine Verkürzung ermöglichte dann der **wiederhole**-Befehl. Es gibt eine weitergehende Möglichkeit zur Verkürzung, nämlich Befehlsfolgen zusammenzufassen, als neuen Befehl mit einem bestimmten Namen zu belegen und so den Befehlssatz von Snap! zu erweitern. Es ist eine wichtige Eigenschaft von Programmiersprachen, Befehlsketten zu einem neuen Befehl zusammen fassen zu können, der dann von anderen Programmteilen aufgerufen werden kann, also mehrfach verwendbar ist. Wir sprechen dann auch von einer Prozedur.

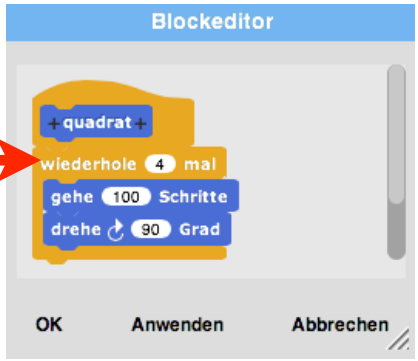
Wir wollen also die Befehlsfolge zum Zeichnen eines Quadrats zusammenfassen und unter dem Namen "quadrat" verfügbar machen. Dazu rufen Sie zunächst die Befehlsgruppe **Variablen** auf und wählen dort **Neuer Block**. Damit wird ein Dialogfenster geöffnet, in dem der Name des Blocks, die Art und die Zuordnung zu einer Blockpalette festgelegt wird.



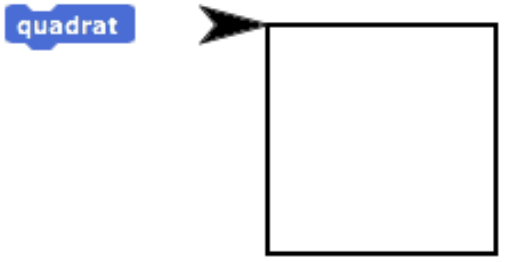
Wir nennen unseren Block **quadrat** und ordnen ihn als neuen Befehl der Palette **Bewegung** zu. Er wird dann als Erweiterung in dieser Palette mit aufgeführt werden.

Nach Drücken von **OK** öffnet sich ein weiteres Fenster mit einem sogenannten **Hut**-Block (der dann bereits den Blocknamen enthält). Wir können die Wiederholungsschleife für das Quadrat einfach dort hineinziehen und an den **Hut**-Block anhängen.



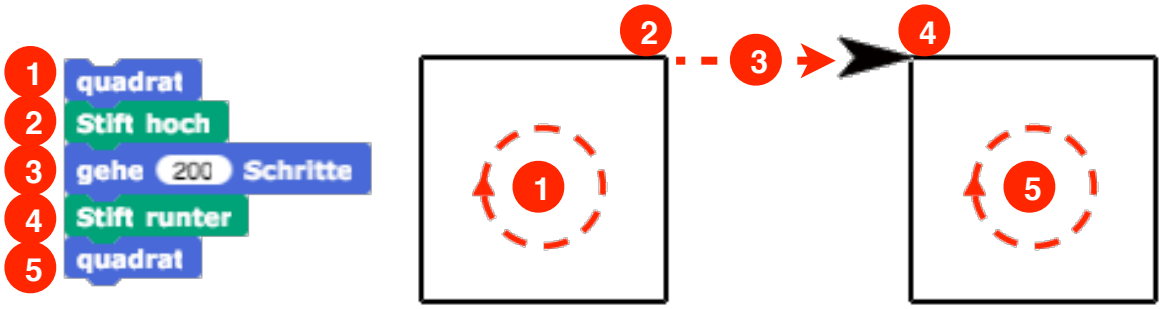


Sobald wir den Blockeditor mit **Ok** schließen[32], findet sich ab jetzt in der Befehlsgruppe **Bewegung** auch der Befehlsblock **quadrat**. Wir können ihn in den Programmbereich ziehen. Wenn wir ihn dort anklicken, wird die in **quadrat** enthaltene Befehlsfolge ausgeführt, d.h. das Quadrat gezeichnet.



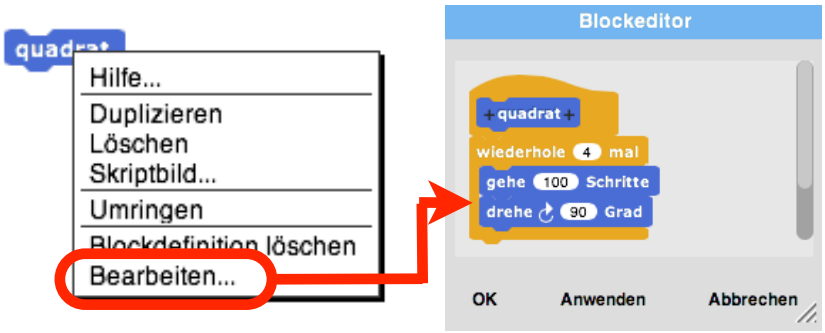
[32] Alternativ kann der Block auch durch Anklicken von **Anwenden** aktualisiert und aktiviert werden. Er bleibt dann aber geöffnet und kann so bei Bedarf leicht wieder geändert werden.

Wir können nun z.B. die Schildkröte an eine andere Stelle bewegen und dort erneut ein Quadrat zeichnen lassen. Bei der Bewegung an den neuen Startpunkt soll sie keine Spur hinterlassen.

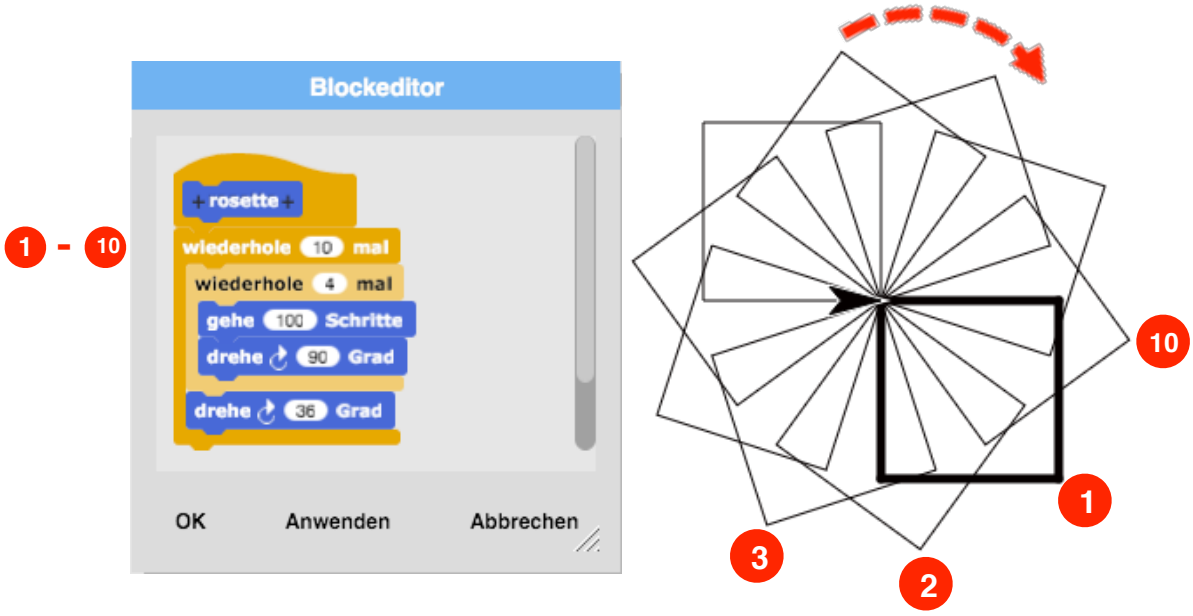


1. Zu Beginn wird das erste Quadrat gezeichnet.
2. Danach folgt der Befehl **Stift hoch**, damit die folgende Bewegung keine Spur hinterlässt.
3. Nun wird die Schildkröte mit **gehe 200 Schritte** bewegt.
4. Anschließend muss der Befehl **Stift runter** folgen, damit die nachfolgenden Bewegungen wieder eine Spur hinterlassen.
5. Ein weiteres Quadrat wird dann durch erneute Eingabe von **quadrat** gezeichnet.

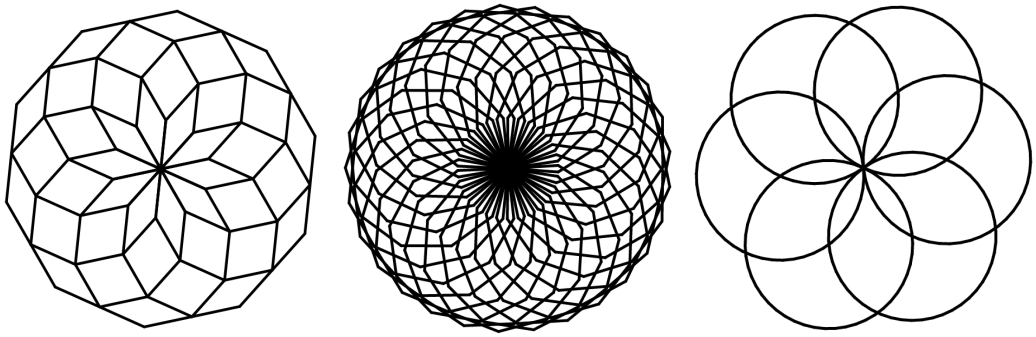
Soll die Länge der Seiten dieses Quadrats geändert werden, ist dazu der Block **quadrat** zu ändern: Mit der rechten Maustaste öffnen Sie das Kontextmenü des Blocks **quadrat**. Dort findet sich als letzter Menüpunkt die Option **Bearbeiten ...**. Im Blockeditor können Sie dann eine neue Schrittzahl - z.B. 200 - im Befehl **gehe 100 Schritte** einfügen. Wenn Sie im Programmbereich nun den Befehl **quadrat** anklicken, sollten Sie als Ergebnis ein Quadrat mit der neuen Seitenlänge erhalten.



Besonders nützlich ist, dass wir verschachtelte Wiederholungsschleifen erstellen können. So lassen sich z.B. aus dem einfachen Quadrat schnell erste Quadratmuster ableiten: Wir beginnen mit dem ersten Quadrat (in der folgenden Abbildung fett gezeichnet), drehen die Schildkröte dann nach jedem Quadrat z.B. um 36 Grad und lassen sie erneut ein Quadrat zeichnen. Wiederholen wir dies 10 mal, dreht sich die Schildkröte insgesamt um 360 Grad, also gerade einmal im Kreis. Konsequenterweise nennen wir den Block **rosette**, der mit **Neuer Block** anzulegen ist. Sie können die **wiederhole**-Schleife aus **quadrat** kopieren und einfach um die zweite **wiederhole**-Schleife ergänzen:



Anregung: *Die folgenden drei Rosetten sind mit dem Block* rosette *entstanden. Sie können Sie durch experimentelle Änderung der Zahl der inneren und äußeren Wiederholungen und entsprechende Anpassung der Drehwinkel reproduzieren.*



Hinweis: *Das gezeigte Vorgehen ist bereits ein Vorgriff auf den folgenden Kapitelabschnitt zur Verallgemeinerung, denn über die Zahl der Wiederholungsschritte können nicht nur Quadrate, sondern beliebige Vielecke erzeugt werden.*

Der Zerlegung eines Problems in Teilprobleme entspricht bei unseren Grafikprojekten oft die Zerlegung komplexer grafischer Strukturen in grundlegende Elemente, z.B. Punkte, Linien oder Vielecke. Es macht deshalb Sinn, solche immer wiederkehrenden Elemente als wiederverwendbare Module (d.h. in Snap! als Blöcke) zu definieren. Ich werde in mehreren Kapiteln darauf zurückkommen (insbesondere in Kapitel 6: *Gibt's nicht? Gibt's nicht!* und im Kapitel 9: *Figurenbaukasten*), bis uns schließlich eine Block-Bibliothek mit den grafischen Grundelementen zur Verfügung steht (eine Zusammenfassung findet sich in *Anhang C*).

5.6 Verallgemeinerung

Mit der Wiederholung von Befehlen und mit der Zusammenfassung von Befehlsfolgen in neuen Blöcken haben wir bereits zwei wichtige Hilfsmittel zur Arbeitserleichterung kennen gelernt.

Für die Malerin Vera Molnar war die Wiederholung eine große Hilfe, sowohl was die Erzeugung von einzelnen Objekten in Bildern betraf, als auch die Erzeugung von Bildvarianten (mehr zu Molnars Arbeiten im Kapitel 18: *Hommage à Vera Molnar*). Sie beschäftigte sich in den 60er-Jahren mit Bildern, die sie aus geometrischen Objekten zusammensetzte. Ab 1968 benutzte sie dafür den Computer mit folgender Begründung (Molnar, 1976, S. 35):

„*This stepwise procedure has however two important disadvantages if carried out by hand. Above all it is tedious and slow. In order to make the necessary comparisons in developing series of pictures, I must make many similar ones of the same size and with the same technique and precision. Another disadvantage is that I can make only an arbitrary choice of the modifications inside a picture that I wish to make. Since time is limited, I can consider only a few of many possible modifications.*“

Die von Molnar erwähnten Modifikationen sind besonders leicht möglich, wenn die Algorithmen nicht mit festen Werten abgearbeitet werden, sondern wenn sie durch Parametrisierung flexibilisiert werden. Das gilt insbesondere dann, wenn bestimmte Werte an mehreren Stellen gebraucht werden und deshalb bei Änderungen entsprechend oft manuell eingegeben werden müssen, was mühsam und fehleranfällig ist. Denn es geht Vera Molnar - und auch bei unseren Projekten - nicht mehr allein um die Erzeugung einzelner Objekte, sondern mit dem Computer eröffnen sich nahezu unbegrenzte Variationsmöglichkeiten.

Die Parametrisierung ist deshalb die Bestimmung und Beschreibung der variablen Elemente in einem Prozess[33]. In unseren Projekten können das Farbe, Form, Größe und Position von Objekten sein, ebenso aber auch die Anzahl von Wiederholungen, die Bestimmung von Abhängigkeiten oder Gültigkeitsbereichen und Grenzen des Prozessablaufs. Konkret sind Parameter die Werte, die beim Aufruf einzelner Befehle (also z.B. der Wert 100 beim Befehl **gehe 100 Schritte**) oder beim Aufruf von Blöcken übergeben werden.

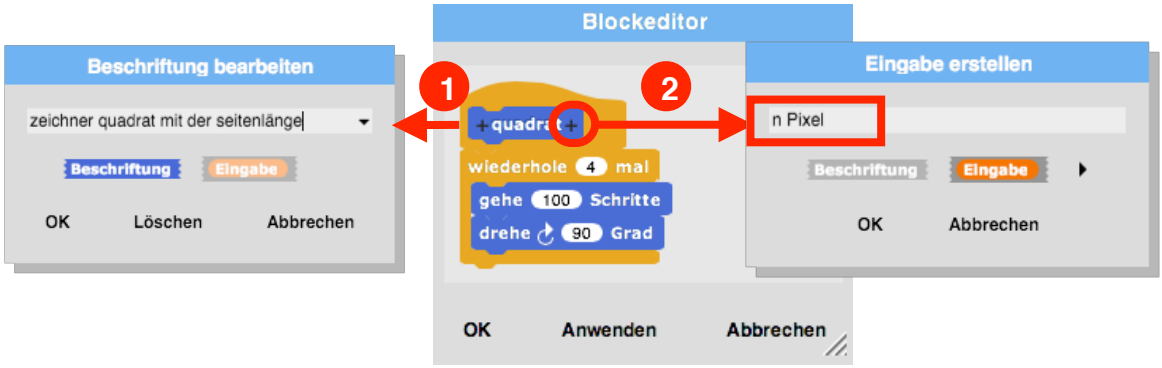
Die Nützlichkeit zeigt sich sofort, wenn wir das Quadrat-Beispiel aus dem letzten Abschnitt noch etwas weiter führen. Es sollen nun Quadrate mit jeweils unterschiedlichen Seitenlängen erstellt werden. Derzeit könnten wir das nur dadurch erreichen, indem wir innerhalb der Prozedur **quadrat** die Seitenlänge jeweils neu eingeben und dann die so veränderte Prozedur aufrufen. Wesentlich eleganter und flexibler gelingt dies, wenn wir dem Block den gewünschten Wert für die Seitenlänge übergeben.

[33] Die Begriffe Parameter und Variable werden meist gleichwertig verwendet. Variablen sind nichts anderes als Parameter, die sich während des Programmablaufs ändern können.

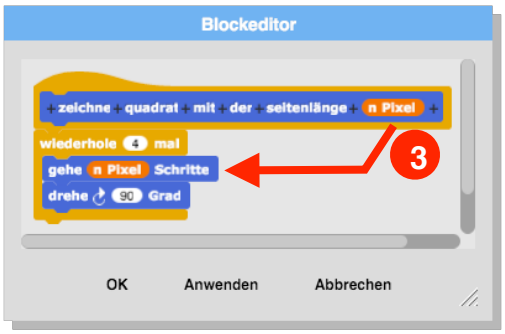
Dazu **Bearbeiten ...** wir wieder den **quadrat**-Block.

1. Im Blockeditor können wir nach Doppelklick auf **quadrat** die Beschriftung des Blocks ändern, z.B. (hier exemplarisch in sehr ausführlicher Form!) in **zeichne quadrat mit der seitenlänge**.

2. Anschließend öffnen wir mit einem Klick das +-Zeichen rechts von der Beschriftung. Dort können wir den gewünschten Namen für den Übergabewert eingeben, hier z.B. **n Pixel**.



3. Wir erhalten eine entsprechend veränderte Prozedur, deren Aufruf dann z.B. **zeichne quadrat mit der seitenlänge 30** lautet. Der Parameter **n Pixel** ist also ein Platzhalter für den Zahlenwert, den wir für die Seitenlänge des Quadrats eingeben[34]. Er wird dann einfach als Wert in den **gehe**-Befehl gezogen.



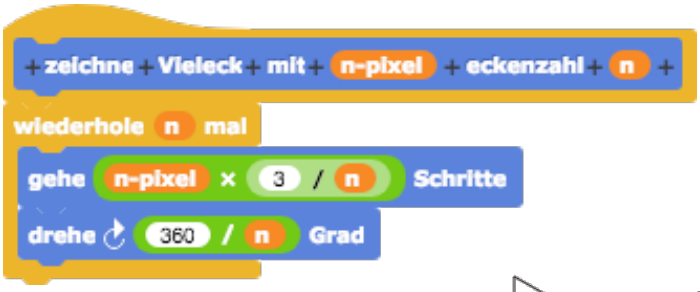
Der neue **quadrat**-Befehl (bzw. jeder andere selbstdefinierte Block) kann nun seinerseits mit allen anderen Blöcken von Snap! kombiniert werden. Die im vorigen Abschnitt gezeigte Rosette erhalten wir damit in einer kompakteren Form **rosette n Pixel**:



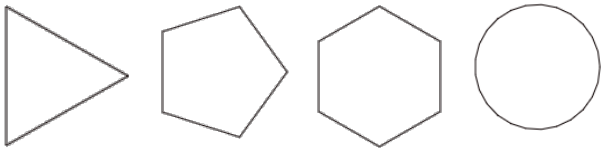
Vereinfachung und *Verallgemeinerung* ist ein durchgängiges Ziel in den folgenden Kapiteln, weil es ein systematisches Untersuchen der Eigenschaften eines Systems oder der Auswirkungen von Abbildungsvorschriften ungemein erleichtert. Die Verknüpfung von Prozeduren mit möglichst weitgehender Parametrisierung ist dafür ein zentrales Hilfsmittel.

[34] Der Parameter **n Pixel** ist nur lokal im Block **zeichne quadrat mit der seitenlänge** bekannt und kann außerhalb nicht weiter verwertet werden. Wir sprechen deshalb von einer *lokalen Variablen*. Wir werden im nächsten Abschnitt auch *globale Variablen* kennen lernen und verwenden.

Wir knöpfen uns deshalb nochmal die Prozedur **quadrat** vor. Denn mit einer leichten Veränderung können wir nicht nur unterschiedliche Quadrate, sondern gleich beliebige Vielecke (Polygone) zeichnen. Dies stellt eine Verallgemeinerung dar, die wiederum eine Vielzahl daraus abgeleiteter Figuren erlaubt. Es reicht dazu einen Parameter **n** einzuführen, mit dem die Zahl der Ecken eines Vielecks bestimmt wird:



Die Abbildung zeigt die mit dieser Prozedur erzeugten Figuren:



n = 3 *n = 5* *n = 6* *n = 25*

Hinweis: *Beachten Sie, dass bei unveränderter Seitenlänge mit steigender Anzahl der Ecken auch die Größe des Vielecks zunimmt. Sie können dem mit einem angepassten Wert von* **n-pixel** *entgegen wirken (hier:* **n-pixel x 3/n***). Dabei handelt es sich um einen Erfahrungswert, nicht um eine exakte Berechnung!*

Das brauchen wir von **Snap!***:*



Für arithmetische Operationen stehen folgende Reporter für die Grundrechenarten zur Verfügung:



Die Verknüpfungen verlangen zwei Zahlen als Eingaben und geben die Summe, die Differenz, das Produkt bzw. den Quotienten dieser Zahlen zurück. Außer Zahlen können auch Parameter oder verschachtelte Ausdrücke eingegeben werden, z.B.:



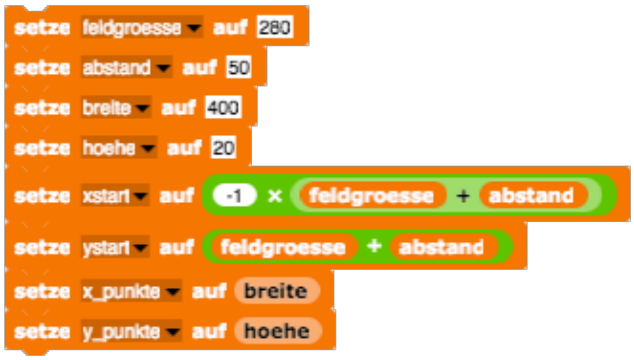
5.7 Hommage à Vilder: Variationen über 9 Quadrate

Die Möglichkeit der Modularisierung, Erweiterung und Verallgemeinerung können im *Recoding* einer Bildserie von Roger Vilder konkret genutzt werden. Vilder (geb. 1938 in Beirut, Libanon) beschäftigte sich mit mathematischen und geometrischen Konzepten und setzte sie in Bewegung um. Daraus entstanden Animationen und Skulpturen.

Seine hier vorgestellten Grafiken von 1973 waren Grundlage für eine computergenerierte Filmsequenz. In einer 3*3-Matrix ordnet Vilder geometrische Objekte an (obwohl Vilder von Quadraten spricht sind es Rechtecke, siehe die sechs Beispiele in Bild 9), die er systematisch verändert.

Die programmtechnische Umsetzung soll am einfachsten Beispiel (in Bild 9 oben links) analysiert werden. Die Grundeinheiten, die Rechtecke, werden in einem eigenen Block erstellt. Damit alle sechs Beispiele mit möglichst wenig Änderungen möglich werden, sind anfangs eine Anzahl Kenngrößen (über **Neue Variable**) festzulegen[35].

- **feldgroesse** legt die Größe der quadratischen Felder der Matrix fest.
- **abstand** legt den Abstand zwischen den Feldern der Matrix fest.
- **breite** und **hoehe** bestimmen die Größe des ersten Rechtecks innerhalb eines Feldes.
- **xstart** und **ystart** sind die Koordinaten des Startpunkts des ersten Rechtecks innerhalb des ersten Feldes.



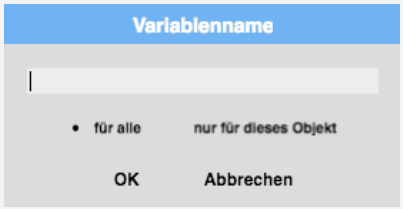
- **x_punkte** und **y_punkte** legen die Breite und Höhe eines Rechtecks fest und werden an einen Block **rechteck um xm ym breite höhe** übergeben.

Das brauchen wir von **Snap!***:*



In der Palette Variablen können mit der Option **Neue Variable** globale Variablen erzeugt werden, die dann in allen Blöcken ausgewertet und genutzt werden können. Es öffnet sich eine Dialogbox, in der sie benannt werden können:







Durch den Befehl setze Variable auf Wert wird einer Variablen ein gewünschter Wert zugewiesen. Dies kann auch ein errechneter Wert sein, z.B.:



[35] Diese Kenngrößen sind *globale Variablen*, die dann automatisch in allen Prozeduren bekannt und gültig sind.

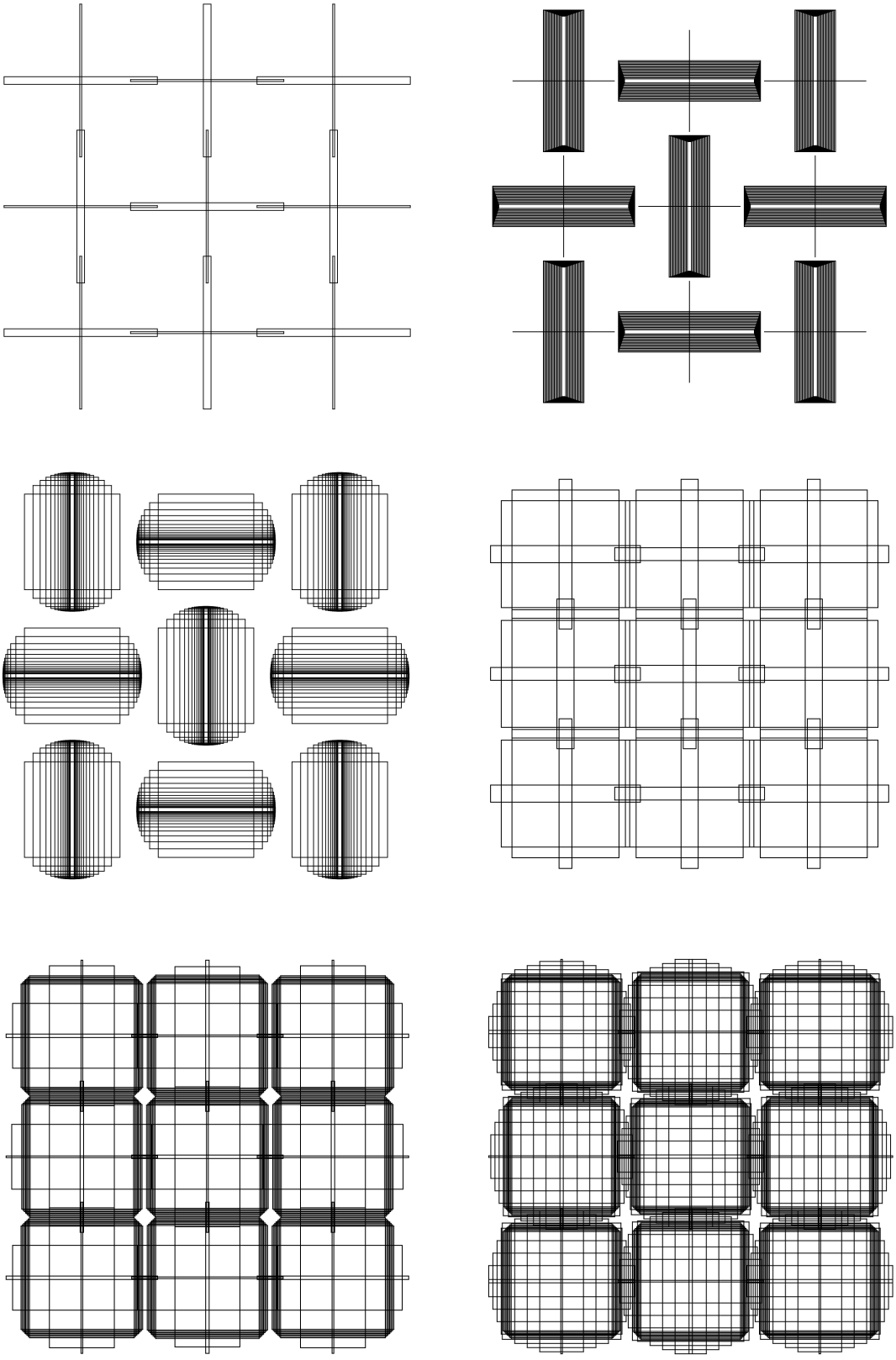


Bild 9: Hommage à Vilder: Variationen über 9 Quadrate

Für die drei Reihen bzw. Spalten benötigen wir eine Doppelschleife. Die innere Schleife enthält ihrerseits Befehlsfolgen (in jeweils weiteren innersten Schleifen) für die Erzeugung der Rechtecke in den Feldern der Matrix. Hier rechts gezeigt ist die Schleife zur Erzeugung von Bild 9 oben links mit je zwei Rechtecken.



Hinweise: *Bei den anderen Beispielen in Bild 9 werden innerste Schleifen für 16, dann 15, dann eine Doppelschleife 2-2 und dann Dreifachschleifen 2-2-8 bzw. 2-8-8 für Rechtecke unterschiedlicher* **breite** *und* **hoehe** verwendet. *Die Rechtecke wechseln innerhalb eines Feldes sowie außerdem in der Matrix von Feld zu Feld ihre Richtung um 90 Grad. Dafür ist nach jeder der innersten Schleifen die aktuelle Richtung zu verändern!*

Nachzutragen ist noch der Block **rechteck**, der in den innersten Schleifen aufgerufen wird. Wir benötigen dazu eine Prozedur, die in der Lage ist, Rechtecke in unterschiedlichen Größen zu zeichnen.



1. Gezeichnet werden soll ein Rechteck um einen Mittelpunkt (was in der Regel flexibler ist, als das Rechteck direkt von einem Eckpunkt aus zu zeichnen). Zuerst muss die Schildkröte deshalb allerdings mit gehobenem Stift von diesem Mittelpunkt an den Eckpunkt, also die linke untere Ecke des Rechtecks bewegt werden.

2. Es folgt eine Schleife, in der die Schildkröte mit abgesenktem Stift zweimal die gewählte Breite und Höhe und damit das Rechteck zeichnet.

3. Abschließend wird sie mit gehobenem Stift wieder an den Ausgangspunkt, also den Mittelpunkt des Rechtecks, zurück geführt.

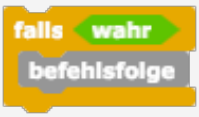
5.8 Kontrollstrukturen

In den bisherigen Beispielen erfolgte das Abarbeiten von Befehlsfolgen rein linear. Das ist aber nicht die Regel, in interaktiv gesteuerten Programmen sogar eher die Ausnahme. Zum einen kann sich während des Programmablaufs ständig der Programmstatus verändern, d.h. Variablen erhalten neue Werte, etwa wenn die Bewegung der Schildkröte neue Positionsdaten liefert und davon abhängige Reaktionen erfordert. Zum anderen können die NutzerInnen durch Tastatureingaben, Mausbewegungen oder Mausklicks das Programmverhalten beeinflussen. Wir sprechen dann von *Ereignissteuerung*.

Damit auf die Ereignisse reagiert werden kann, benötigen wir darüber die Kontrolle: Was soll passieren, wenn eine Maustaste gedrückt wird? Was soll passieren, wenn ein Zahlenwert einen vorgegebenen Grenzwert über- oder unterschreitet? Für die notwendigen Weichenstellungen stehen unterschiedliche *Kontrollstrukturen* und *Vergleichsoperatoren* bereit, durch die die Ausführung von Tätigkeiten an das Zutreffen von Bedingungen geknüpft wird.

Das brauchen wir von **Snap!**:





Mit der Kontrollstruktur **falls wahr ...** wird eine *bedingte Anweisung* beschrieben. Wenn eine formulierte Bedingung zutrifft, also den Wahrheitswert **wahr** liefert, wird die Befehlsfolge ausgeführt, ansonsten wird die Befehlsfolge ignoriert.



So wird hier ein Quadrat nur dann gezeichnet, wenn der aktuelle Wert kleiner als ein vorgegebener Grenzwert ist.



Mit der Kontrollstruktur **falls wahr ... sonst** wird eine *bedingte Verzweigung* beschrieben. Wenn eine formulierte Bedingung zutrifft, wird die erste Befehlsfolge ausgeführt, wenn nicht, wird die zweite Befehlsfolge ausgeführt.



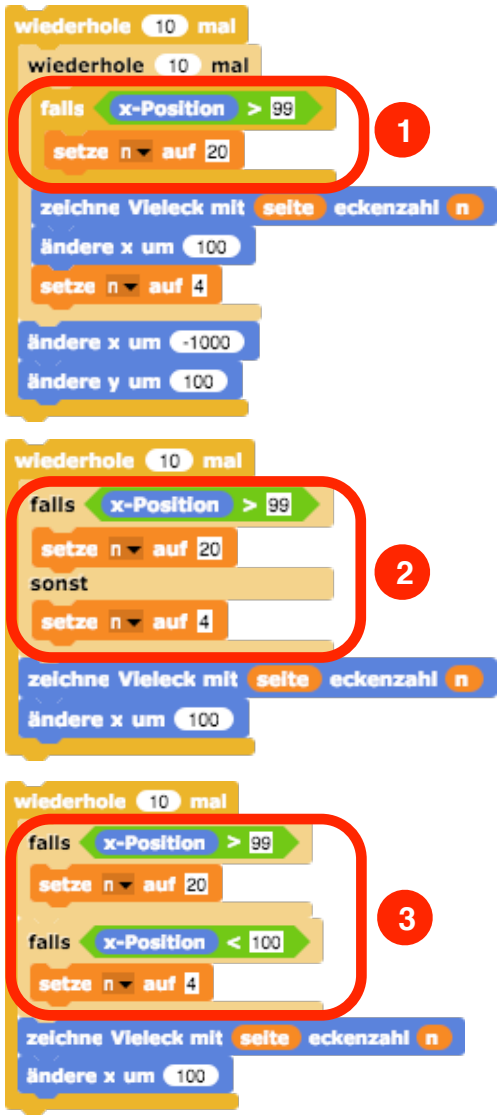
Mit den Vergleichsoperatoren **kleiner**, **gleich** und **größer** werden die Wahrheitswerte **wahr** oder **falsch** für den jeweiligen Vergleich geliefert.

Mit den logischen Verknüpfungen **und** (Konjunktion), **oder** (Disjunktion) bzw. **nicht** (Negation) können daraus komplexere logische Ausdrücke gebildet werden.





Alle Kontrollstrukturen bestehen aus einem *Test* und einer *Tätigkeit*: Liefert der Test den Wahrheitswert *wahr*, d.h. die abgefragte Bedingung trifft zu, dann wird die durch die folgende Befehlsfolge beschriebene Tätigkeit ausgeführt. Sollen beispielsweise die im letzten Abschnitt beschriebenen Vielecke je nach Ausgangsposition der Schildkröte unterschiedlich aussehen (z.B. links oder rechts der Bildmitte wie in der Abbildung unten links), kann das mit unterschiedlichen Kontrollstrukturen erreicht werden.

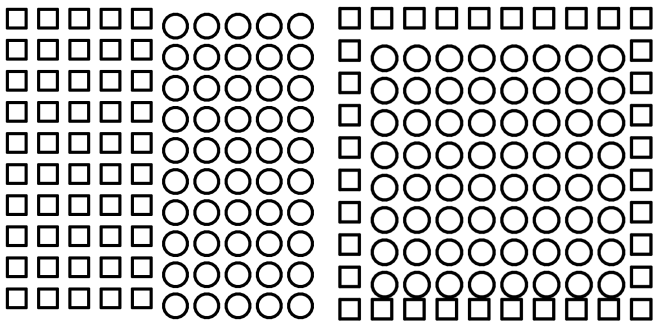


1. Wird in der *bedingten Anweisung* **falls wahr ...** festgestellt, dass die Schildkröte sich rechts der Bildmitte (plus Objektbreite) befindet (also **x-Position > 100** zutrifft), wird das **Vieleck** mit **20** Ecken gezeichnet; befindet sie sich links davon (d.h. **x-Position > 100** trifft nicht zu) dagegen mit den vorab eingestellten **4** Ecken.

2. Äquivalent ist die Feststellung der Eckenzahl mittels der *bedingten Verzweigung* **falls wahr ... sonst**. Liefert der Test den Wahrheitswert **wahr**, werden **20** Ecken gezeichnet, trifft dies nicht zu, **4** Ecken.

3. Die dritte (gleichwertige) Variante besteht aus *zwei bedingten Anweisungen* **falls wahr ...**. Die erste prüft, ob die Schildkröte sich rechts einer Position befindet (also **x-Position > 99** trifft zu), was das Zeichnen mit **20** Ecken bewirkt. Mit der zweiten wird geprüft, ob sie sich links des Wertes befindet (also **x-Position < 100** trifft zu), was das Zeichnen mit **4** Ecken bewirkt.

Anregung: *Bei der rechten Bildvariante müssen horizontal und vertikal Grenzwerte sowohl über- (links und unten) als auch unterschritten (rechts und oben) werden. Das verlangt die logische Verknüpfung mehrerer Vergleichsoperationen, also z.B.* **falls x-Position > -300** und **x-Position < 450** und **y-Position > -350** und **y-Position < 450**.



5.9 Hommage à Riley: Movement in Squares

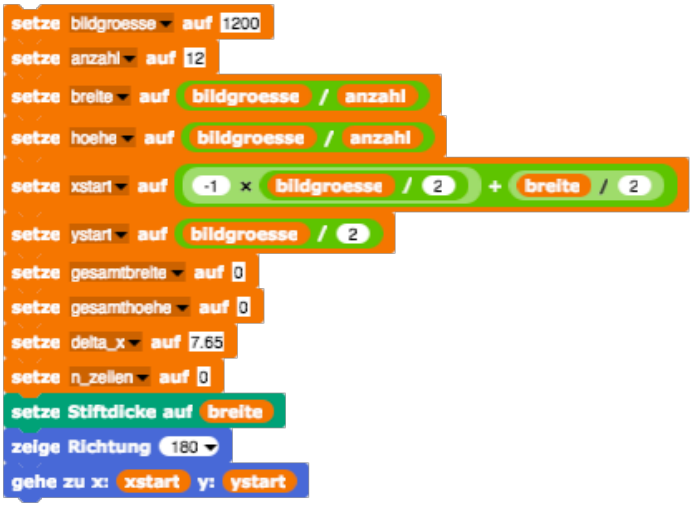
Beim abschließenden Projekt in diesem Kapitel kommen nun auch die Kontrollstrukturen zum Einsatz. Es ist eine Hommage an Bridget Rileys Bild *Movement in Squares* (Bild 10 entspricht weitgehend diesem Vorbild).

Bridget Riley (geb. 1931 in London) ist keine Vertreterin der Computerkunst, sondern als Malerin eine der führenden Vertreterinnen der Op Art. Gerade ihre frühen Arbeiten eignen sich gut für ein *Recoding*. Sie befassen sich mit geometrischen Objekten (Punkte, Linien, Quadrate) und Mustern, mit denen sie gerne optische Illusionen und Wahrnehmungstäuschungen erzeugt. Das Bild *Movement in Squares* (entstanden 1961) steht am Beginn ihrer Hinwendung zur Abstraktion, die sie selbst wie folgt beschreibt (Riley, 2009):

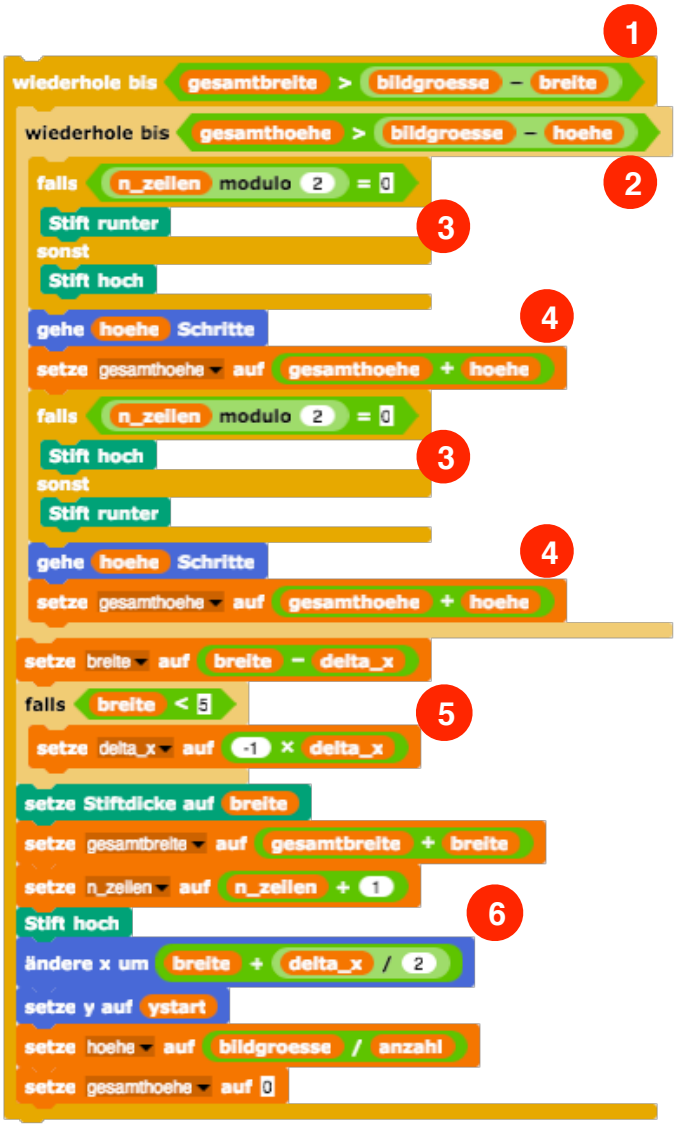
„Perhaps the time I had spent drawing allowed me to trust the eye at the end of my pencil. Movement in Squares (1961) began in this way. [..] One evening on my way to the studio, I thought of drawing a square. Everyone knows what a square looks like and how to make one in geometric terms. It is a monumental, highly conceptualised form: stable and symmetrical, equal angles, equal sides. I drew the first few squares. No discoveries there. Was there anything to be found in a square? But as I drew, things began to change. Quite suddenly something was happening down there on the paper that I had not anticipated. I continued, I went on drawing; I pushed ahead, both intuitively and consciously. The squares began to lose their original form. They were taking on a new pictorial identity. I drew the whole of Movement in Squares without a pause and then, to see more clearly what was there, I painted each alternate space black."

Das Bild beginnt links mit Quadraten. Von links nach rechts wird bei gleichbleibender Höhe deren Breite vermindert; sie werden zu Rechtecken. Kurz vor ihrem Verschwinden nimmt die Breite wieder zu. So entsteht der Eindruck einer Versenkung, für manche auch eine Bewegung. Bei der programmtechnischen Umsetzung sind anfangs wieder eine Anzahl Kenngrößen festzulegen:

- **bildgroesse** legt die Größe des (quadratischen) Gesamtbildes fest.
- Mit **anzahl** wird die Anzahl der Zeilen festgelegt.
- **breite** und **hoehe** bestimmen die Größe der Rechtecke in der linken Spalte.
- **xstart** und **ystart** sind die Koordinaten des Startpunkts des ersten Rechtecks links oben.



- **gesamtbreite** und **gesamthoehe** sind Zwischengrößen für die jeweils erreichte Breite bzw. Höhe.
- **delta_x** gibt die Abnahme der Rechteckbreite von Spalte zu Spalte an.
- **n_zeilen** ist eine Prüfgröße, ob in der ersten Zeile jeweils mit einem schwarzen oder weißen Rechteck begonnen wird.



1 In der äußeren Schleife wird geprüft, ob die zulässige **gesamtbreite** überschritten wird; ist dies der Fall, wird das Programm abgebrochen.

2 In der inneren Schleife wird erst geprüft, ob die zulässige **gesamthoehe** des Bildes überschritten wird; ist dies der Fall, wird die Schleife beendet.

3 Dann wird zweimal mit **n_zeilen** modulo 2 geprüft, ob eine gerade oder ungerade Spalte gezeichnet wird.

4 Davon abhängig wird (über **Stift runter** bzw. **Stift hoch**) zuerst ein schwarzes und dann ein weißes Rechteck gezeichnet bzw. umgekehrt. Da die **Stiftdicke** der **breite** entspricht, genügt dafür der **gehe**-Befehl.

5 Nun wird die **breite** des Objekts um **delta_x** reduziert. Ergibt die Prüfung der **breite**, dass sie eine Untergrenze unterschreitet (hier 5 Bildpunkte), wird ab jetzt die **breite** wieder um **delta_x** erhöht.

6 Nach Anpassung der **Stiftdicke**, Aktualisierung der **gesamtbreite** und der **hoehe**, Zurücksetzen der **gesamthoehe** sowie der Bewegung der Schildkröte zum nächsten Startpunkt erfolgt der nächste Schleifendurchlauf.

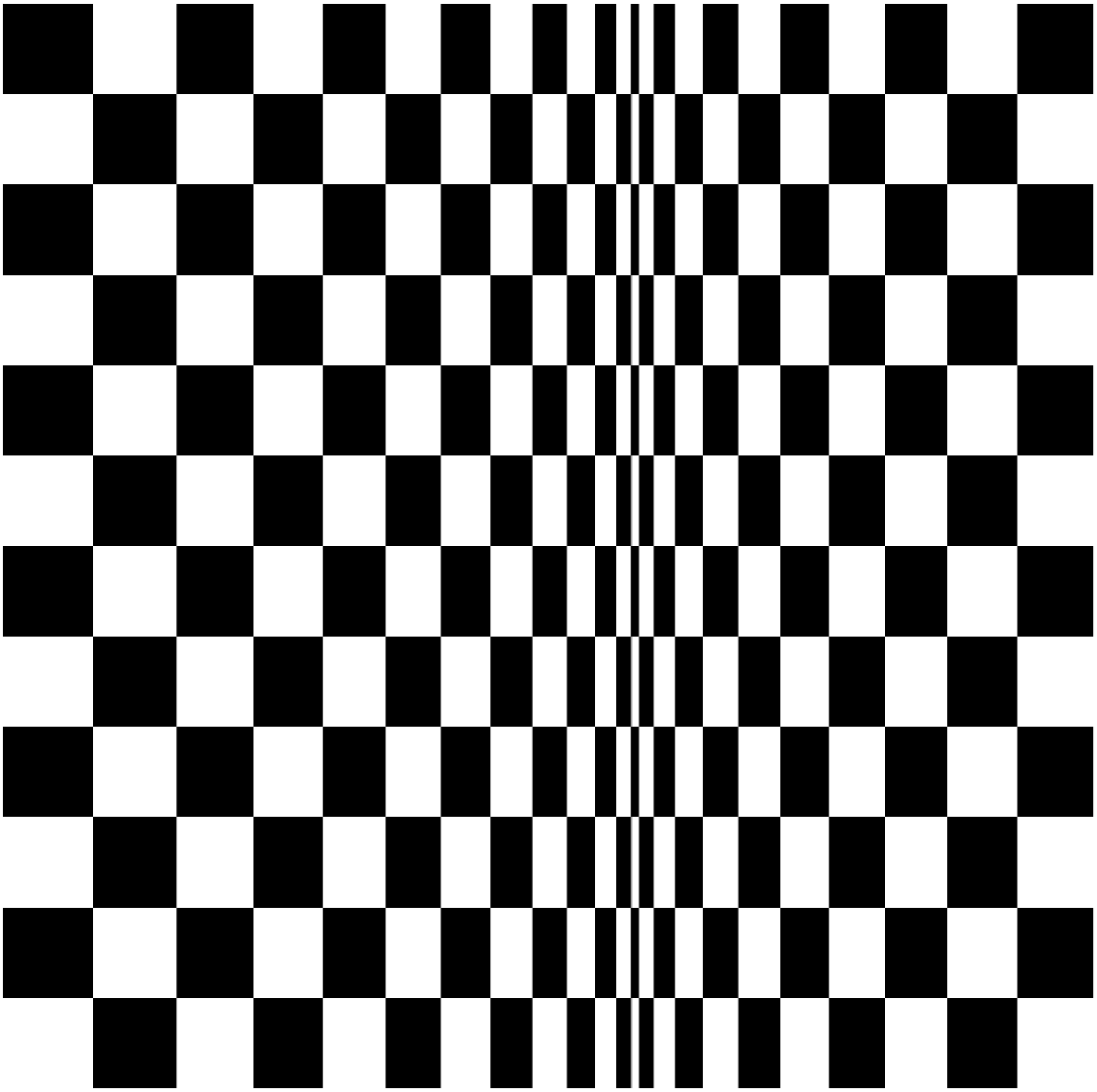
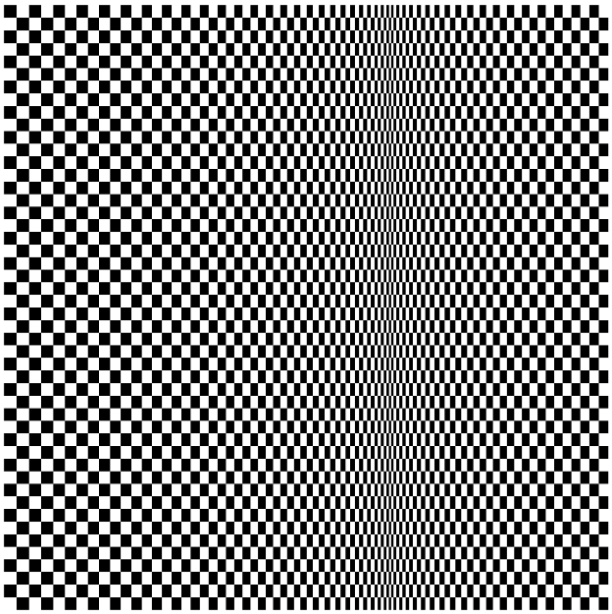


Bild 10: Hommage à Riley: Movement in Squares

5.10 Remixing Movement in Squares



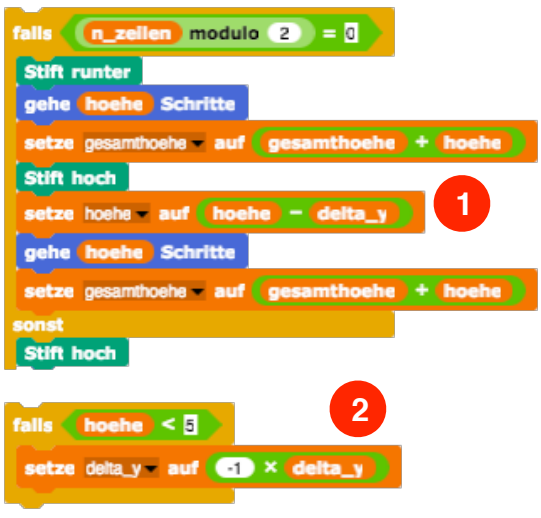
Die von Riley provozierte Irritation der Wahrnehmung kann verstärkt werden. Hier bewährt sich die Verwendung variabler Kenngrößen. Die nebenstehende Abbildung entsteht allein durch Setzen von **anzahl = 48** und **delta_x = 0.4**. Bei einer Änderung von **anzahl** ist immer darauf zu achten, dass die Zu- und Abnahme der Rechteckgröße über **delta_x** entsprechend angepasst wird.

Eine echte Erweiterung des Programms wird aber notwendig, wenn die Verdichtung der Rechtecke nicht nur senkrecht, sondern auch horizontal erfolgen soll (Bild 11).

Dafür sind weitere Kenngrößen notwendig:

- **anzahl** ist in **anzahl_spalten** und **anzahl_zeilen** aufzuteilen.
- Neben **delta_x** wird auch ein **delta_y** gebraucht.

In der inneren Schleife muss die Prüfung mit **n_zeilen** modulo 2, ob eine gerade oder ungerade Spalte gezeichnet wird, nur einmal erfolgen.



1. Hinzu kommt hier die zusätzliche Veränderung der **hoehe** um **delta_y**. Deshalb kann die Entscheidung über Schwarz und Weiß direkt mit der Ausführung gekoppelt werden.

2. Ergibt die abschließende Prüfung der **hoehe**, dass sie eine Untergrenze unterschreitet (auch hier 5 Bildpunkte), wird ab jetzt die **hoehe** wieder um **delta_y** erhöht.

Hier gilt nun noch mehr, dass bei einer Änderung von **anzahl_spalten** und **anzahl_zeilen** die Zu- und Abnahme der Rechteckgröße über **delta_x** und **delta_y** vorsichtig angepasst wird. Bei zu großen oder zu kleinen Werten geht der Effekt verloren und es können unerwartete Effekte auftreten.

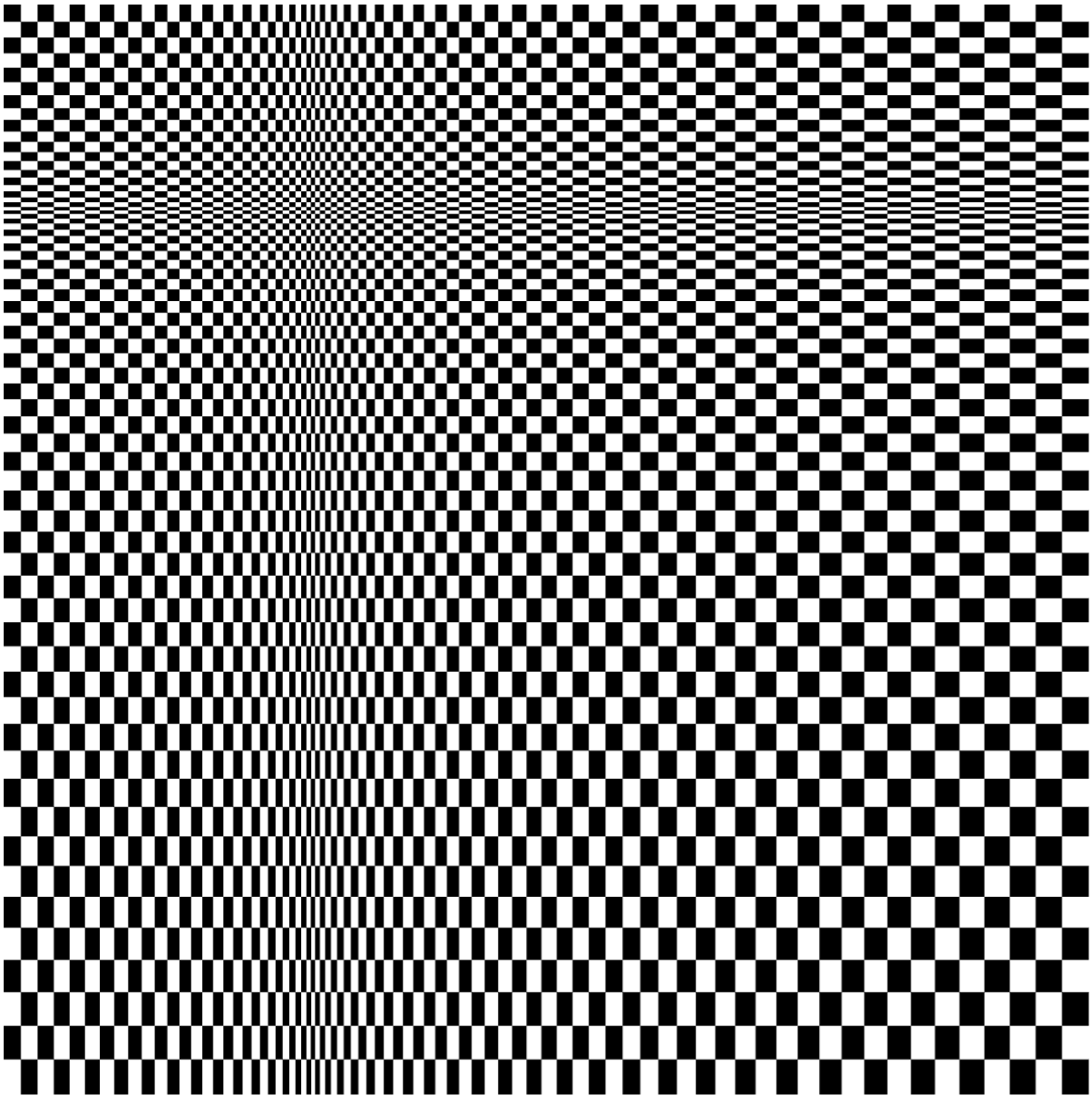


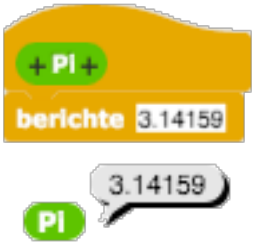
Bild 11: Remixing Movement in Squares

6. GIBT'S NICHT? GIBT'S NICHT!

Der originäre Sprachumfang von Snap! ist recht übersichtlich (etwas über 100 Befehle und Funktionen in acht Kategorien), selbst im Vergleich zu klassischen Logo-Versionen[36]. Das ist einer der Gründe für den niedrigschwelligen Einstieg ins Programmieren, denn Anfänger finden sich schnell in Snap! zurecht. Die Beispiele in den bisherigen Kapiteln sind ein Beleg dafür. Dennoch lassen sich damit durchaus anspruchsvolle Aufgaben bearbeiten. Wir haben schon gesehen, wie aus den vorhandenen Sprachelementen neue Blöcke selbst definiert werden können, die dann als neue Sprachelemente verwendet werden können. Auf diese Weise lässt sich der Sprachumfang von Snap! sukzessive erweitern.

Wenn es also Funktionen gibt, die immer wieder in Anwendungen gebraucht werden, liegt es nahe, diese als eigene Blöcke zu definieren und in einer Bibliothek abzuspeichern, die dann bei Bedarf in ein Programm nachgeladen werden kann. Es lohnt sich, alle Prozeduren, die nicht problemspezifisch sind, in diese Bibliothek aufzunehmen. Allerdings müssen Sie darauf achten, sie entsprechend allgemein und variabel anzulegen, damit sie möglichst problemlos im neuen Kontext wiederverwertbar bleiben.

Nehmen wir als ganz einfaches Beispiel die Kreiszahl π (die wir in späteren Kapiteln brauchen werden). Statt sie als Konstante immer neu zu definieren, können wir den entsprechenden Reporter einmal definieren, in unserer Bibliothek ablegen, später bei Bedarf ins jeweilige Programm einlesen und dann direkt darauf zugreifen.



Wenn wir also einen Block - hier den Reporter **Pi** - definiert haben, können wir den Block exportieren. Sie finden in der Werkzeugleiste unter dem **Datei**-Symbol die Option **Blöcke exportieren ...** mit dem eine Dialogbox geöffnet wird, die Ihre selbstdefinierten Blöcke enthält. Alle mit Häkchen markierten Blöcke werden mittels **OK** exportiert und können unter einem gewünschtem Namen in ihrem lokalen Dateisystem abgespeichert werden (oder einem anderen gewählten Speicherort). Über das **Datei**-Symbol und die Option **Importieren...** können Sie später dann darauf zugreifen.



Das Vorgehen soll an einem zweiten Beispiel vertieft werden. Dabei geht es um die kontrollierte Erzeugung von Zufallszahlen. Die Pioniere der Computerkunst mussten ihre Zufallszahlengeneratoren noch selber programmieren (vgl. z.B. Nees, 1969, S. 113 ff.; auch Limbeck & Schneeberger, 1979, 158 ff.). Heute können wir auf die eingebauten Zufallszahlengeneratoren der jeweils benutzten Programmiersprache zurück greifen. So auch in Snap! mit **Zufallszahl von bis** (Näheres dazu im Kapitel 8: *Alles Zu-*

[36] So bietet der Quasi-Standard Berkeley Logo (UCB Logo) über 300 Befehle (siehe dazu https://www.cs.berkeley.edu/~bh/logo.html)

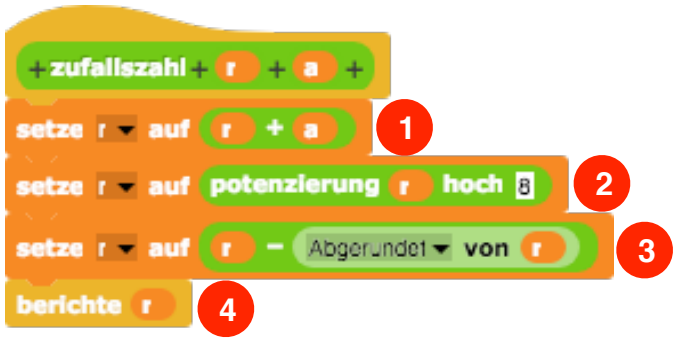
fall...). Die zu einem bestimmten Zeitpunkt erzeugten Zufallszahlenfolgen lassen sich später allerdings nicht identisch reproduzieren, da der Startwert, mit dem sie initialisiert werden, intern erzeugt wird und von uns nicht abrufbar und beeinflussbar ist. Für die Reproduktion von Grafiken mit Zufallskomponenten ist dies jedoch zwingend erforderlich. Damit wir das können, schreiben wir uns einen eigenen Generator **zufallszahl**. Wir verwenden dafür einen von der Firma Hewlett Packard in älteren Taschenrechnern implementierten Algorithmus[37]:

1. Wähle einen (beliebigen) Startwert **r** aus dem Intervall [0,1] und addiere π.
2. Berechne die 8. Potenz dieser Summe.
3. Verwende die Nachkommastellen des Ergebnisses als erste Zufallszahl.
4. Verwende dieses Ergebnis als neuen Startwert in der nächsten Berechnung einer Zufallszahl.

Für die Zufallszahl definieren wir z.B. eine Variable **zz**. Für die reproduzierbare Initialisierung des Zufallszahlengenerators wird dann **zufallszahl** mit einem Startwert für **r** sowie der Kreiszahl **Pi** übergeben, also z.B.:



Mit diesen Eingaben ergibt sich für **zz** immer der reproduzierbare Wert 0,46541066..., der seinerseits für die nächste Iteration verwendet wird.



1. Im Block **zufallszahl** selbst weisen wir der Variablen **r** die Summe aus den übergebenen Werten **r** und **a** (das hier den Wert **Pi** aufweist) zu.
2. Für die Berechnung der Potenz nutzen wir einen weiteren eigenen Block **potenzierung** (siehe *Anhang C*), mit dem wir die Summe in die 8. Potenz setzen. **potenzierung** wird mit der zu potenzierenden Basis und dem Exponent aufgerufen.
3. Abschließend werden **r** mit dem Reporter **Abgerundet** die Nachkommastellen zugewiesen und
4. als Ergebnis berichtet.

Hinweis: *Wenn die errechnete Zufallszahl* zz *wieder als Eingangswert einer weiteren Zufallszahlenberechnung benötigt wird, kann der Aufruf nicht in berechneten Ausdrücken erfolgen.*

Weil mit dem Generator die Zahlen algorithmisch erzeugt werden, sprechen wir von *Pseudozufallszahlen* (im Gegensatz zu physikalischen Zufallszahlengeneratoren, wie Wür-

[37] Dieser und andere Algorithmen sind in Lindner (2001) zu finden.

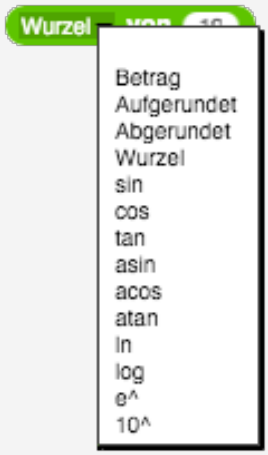
Das brauchen wir von **Snap!***:*



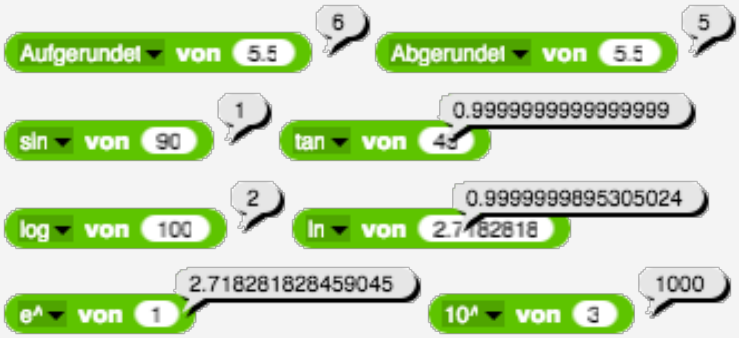


Mit **berichte** wird das Ergebnis einer Berechnung bzw. einer Befehlsabfolge zurück gegeben, z.B.:





Der Block **mathematische Funktion** führt die jeweils ausgewählte Funktion aus und liefert den Funktionswert für den übergebenen Wert, also z.B.:



Zusätzlich gibt es den Reporter **gerundet**, der im Gegensatz zu **Aufgerundet** bzw. **Abgerundet** das Ergebnis *kaufmännischen Rundens* liefert, dh. bei einer ersten Dezimalstelle zwischen 0 und 4 wird abgerundet, zwischen 5 und 9 wird aufgerundet:



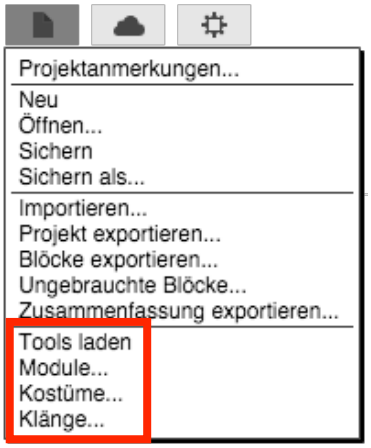
feln oder Maschinen zur Ziehung der Lottozahlen). Für wissenschaftliche Zwecke oder auch bei Verschlüsselungsverfahren müssen die erzeugten Zahlenfolgen bestimmte Gütekriterien erfüllen wie Unvorhersehbarkeit, Gleichverteilung und maximale Periodizität. Für unsere Zwecke reicht die Güte sicher aus[38].

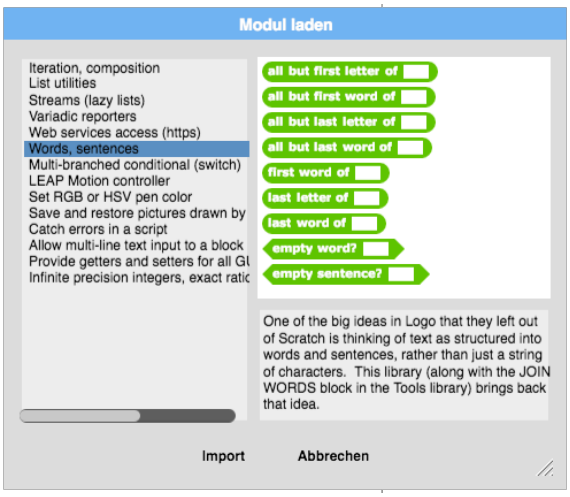
Snap! kann nicht nur mit unserer eigenen Block-Bibliothek erweitert werden. Mächtige Blöcke können unter dem **Datei**-Symbol mit den Optionen **Tools laden** bzw. **Module...** nachgeladen werden.

Mit den Tools werden Grundbefehle verfügbar, z.B. um Text auf der Bühne auszugeben. Die Blöcke sind selbst wiederum in Snap! geschrieben, vergleichbar unseren eigenen selbstgeschriebenen Blöcken. Der Code dieser Befehle ist deshalb über das Kontextmenü des jeweiligen Blocks mit der Option **Bearbeiten...** einsehbar (zugänglich durch Anklicken mit rechter Maustaste). Fortgeschrittene können daraus viel über das Programmieren mit Snap! lernen und bei Bedarf den Code auch verändern.

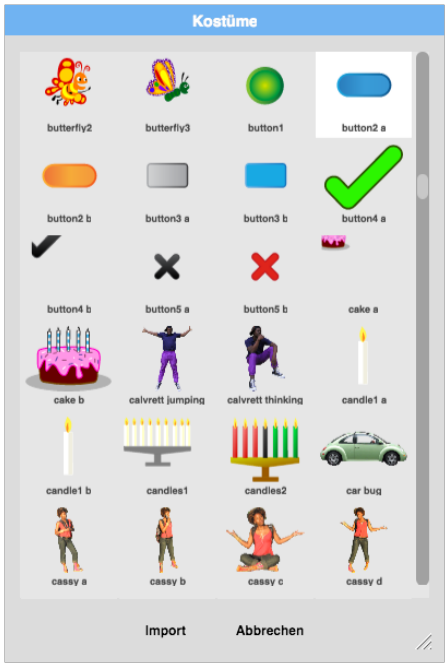
[38] In Anhang C findet sich ein Vergleich mit dem eingebauten Snap!-Zufallsgenerator, der keine wesentlichen Unterschiede zeigt.

Die Option **Module...** öffnet ein Menü **Modul laden**, mit dem der Zugang zu mehreren thematisch gegliederten Bibliotheken möglich wird. Diese betreffen u.a. Schleifenstrukturen, die Verarbeitung von Listen oder Verzweigungen.





In vergleichbarer Weise können auch **Kostüme...** - das sind unterschiedliche Sprites für die Repräsentation der Schildkröte - und **Klänge...** nachgeladen werden. Wir werden das in später nutzen.



Ein anderer Weg Snap! zu erweitern ist dadurch möglich, dass der Quellcode offen und zugänglich ist[39]. Inzwischen wurde das auch bereits mehrfach genutzt und es gibt deshalb entsprechende Erweiterungen. Etliche dienen der Steuerung externer Geräte, wie Roboter (z.B. Orbotix Sphero, Lego NXT, Finch & Hummingbird) oder programmierbare Mikrocontroller-Platinen (wie Arduino oder Raspberry Pi).

Hinzu kommen eigenständige Versionen als Ableger von Snap!, um es für spezielle Zwecke anzupassen. Mit Beetle Blocks können 3D-Modelle entworfen und für den 3D-Druck vorbereitet werden. Mit TurtleStitch können Snap!-Entwicklungen mit programmierbaren Stickmaschinen gestickt werden und so ganz handfeste Versionen der Computerkunst liefern. SQLsnap! bietet ein umfangreiches Befehlsrepertoire für den Zugriff auf SQL-Datenbanken und erlaubt damit spannende Projekte im Schulunterricht.

Wer mit Snap! bereits gearbeitet hat, findet sich in den genannten Varianten schnell zurecht.

[39] Der Quellcode findet sich auf der Plattform GitHub:
https://github.com/jmoenig/Snap--Build-Your-Own-Blocks

7. ALLES SCHÖN BUNT HIER ...

Farbe ist bei der Erzeugung ästhetischer Objekte natürlich ein zentrales Thema. Aus dem täglichen Erleben wissen wir, dass Farben Stimmungen hervorrufen können, kulturabhängige Bedeutungen tragen und als Signal- und Warnfarben Reaktionen hervorrufen sollen. Der Wiedererkennungswert von Farben hat im Marketing so große Bedeutung erlangt, dass wir heute mit manchen Farben bestimmte Unternehmen assoziieren, wie Magenta mit der Deutschen Telekom., Dunkelblau mit Nivea oder Lila mit Milka.[40]

Heute ist unser Ausgabemedium der Bildschirm, der uns ganz selbstverständlich Darstellungen in hoher Auflösung mit nahezu unerschöpflichen Farbnuancen erlaubt. Dagegen standen in den Anfängen der Computerkunst zunächst nur Plotter zur Verfügung, die oft nur einfarbige Zeichnungen erlaubten (siehe dazu den Abschnitt *Vom Plotter zur Schildkröte* im Kapitel 1: *Computerkunst*).

Beim *Remixing* ergibt sich durch die Einführung von Farben eine neue Variationsbreite, auch für bereits bekannte Bildbeispiele. So sind die folgenden Grafiken in Bild 12 aus einer Schwarz-Weiß-Grafik allein durch Hinzufügen von Farbkomponenten entstanden (dieses Beispiel wird im Kapitel 13: *Ein Block ist ein Block ist ein Block ...* noch näher vorgestellt). Die Zuordnung von Farben zu den Bildpunkten des Bildschirms wird von Snap! mit entsprechenden Befehlen unterstützt, die wir uns im Folgenden erarbeiten.

Die einfachste manuelle Farbauswahl bietet der Befehl **setze Stiftfarbe auf** ■. Diese Form der Farbauswahl ist sehr intuitiv, allerdings sind genaue Farbtöne damit nur schwer zu reproduzieren. Es gibt deshalb auch die Möglichkeit, eine Farbe absolut mit **setze Stiftfarbe auf x** oder relativ mit **ändere Stiftfarbe um x** festzulegen.

Der programmtechnische Umgang mit Farben ist nicht gerade einfach, jedenfalls dann, wenn wir sie präzise festlegen und verwenden wollen. Es gibt nämlich mehrere Farbmodelle mit unterschiedlichen Farbrepräsentationen und daraus resultierend auch weitere alternative Möglichkeiten, Farben festzulegen.

Am Bildschirm gelten die Gesetze der additiven Farbmischung. Da sie dem menschlichen Farbensehen mit drei spektral unterschiedlichen Sensoren für Rot, Grün und Blau entspricht, wird sie auch *physiologische Farbmischung* genannt. Aus den drei *Primärfarben* lassen sich mit unterschiedlichen Mischungsverhältnissen alle anderen Farben erzeugen.

Die Wertebereiche für die drei Farben (RGB-Modell) können verschieden festgelegt werden. Bei computerbasierten Darstellungen sind die Minimal- bzw. Maximalwerte

[40] Für alle, die sich mit Farbe aus interdisziplinärer Sicht auseinandersetzen möchten, kann ich das Buch *Farbe: Natur, Technik, Kunst* von Welsch & Liebmann (2003) empfehlen. Sie behandeln naturwissenschaftliche, farbpsychologische, kulturelle und technische Aspekte verständlich und anschaulich illustriert.

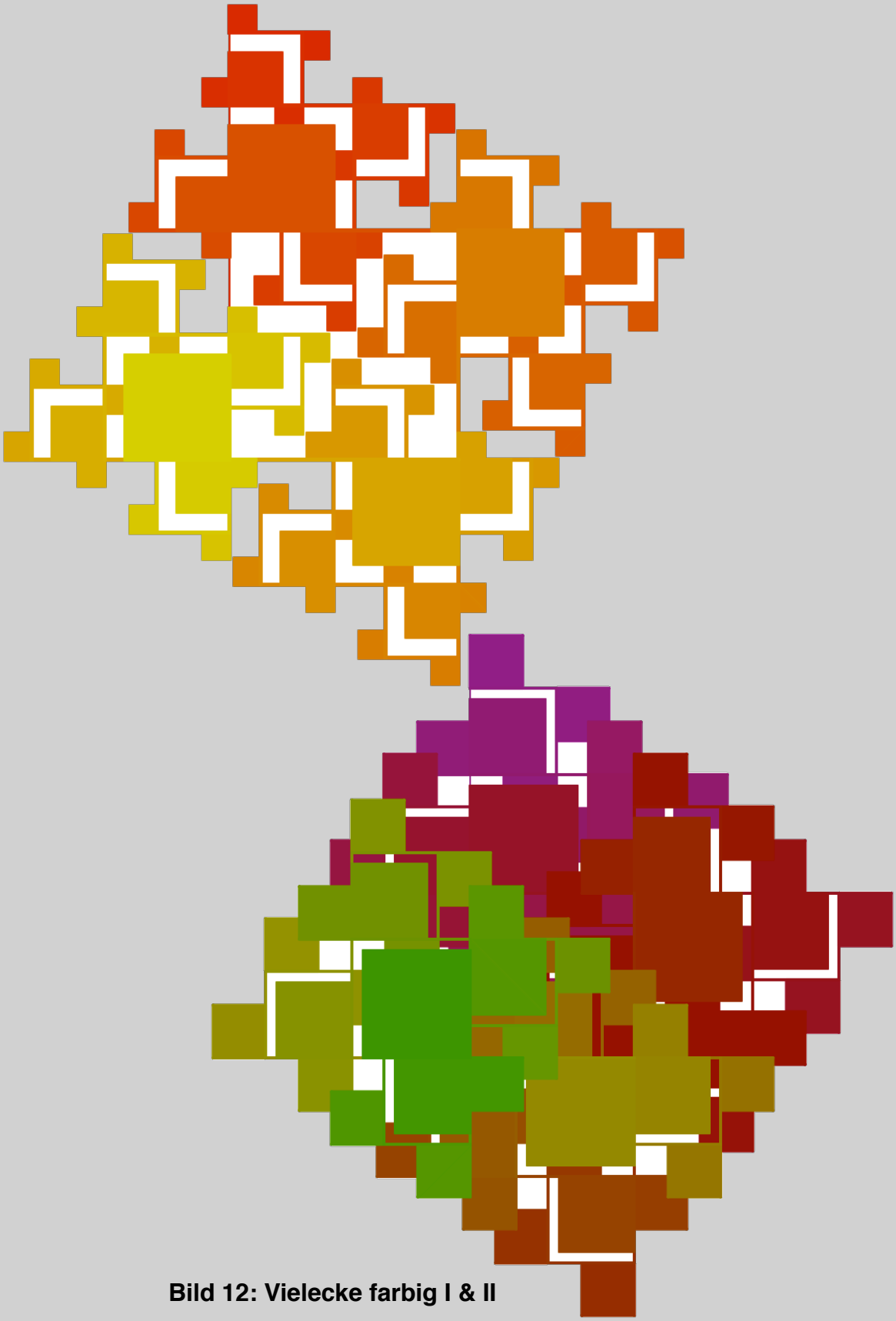


Bild 12: Vielecke farbig I & II





Mit **setze Stiftfarbe auf ...** kann aus einer Farbpalette die gewünschte Farbe ausgewählt werden. Im Farbfenster wird die Farbe unter der aktuellen Mausposition angezeigt, die dann durch Mausklick festgelegt wird. Diese Farbe wird dauerhaft beibehalten, bis sie durch eine neue Wahl geändert wird.

Hinweis: Als aktuelle Mausposition kann jede Position im Snap!-Fenster ausgewertet werden, d.h. auch in den Befehlspaletten, dem Programmbereich oder der Bühne!



Mit dem Befehl **setze Stiftfarbe auf x** wird die Stiftfarbe bestimmt. Dabei sind Werte zwischen 0 und 100 erlaubt (auch Dezimalzahlen sind möglich). Die Farbtonskala reicht von Rot (0) über Grün (35), Blau (65) und Magenta (85) bis Dunkelrot (100):





Mit dem Befehl **ändere Stiftfarbe um x** wird die aktuelle Stiftfarbe um den mit **x** angegebenen Wert verändert.

Hinweis: Die folgenden Befehle gehören zur Blockbibliothek, die nachgeladen werden kann und in *Anhang C* beschrieben wird.



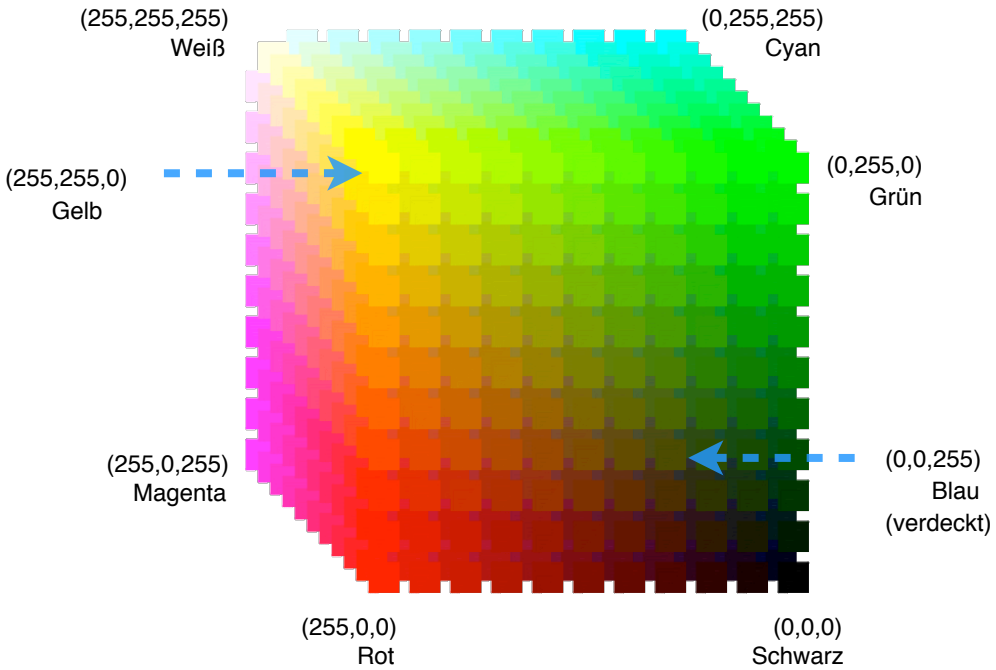
Mit dem Befehl **setze stiftfarbe auf r: g: b:** wird die Stiftfarbe nach dem RGB-Modell bestimmt. Dabei sind für r, g und b Werte zwischen 0 und 255 erlaubt. (0, 0, 0) ergibt Schwarz, (255, 255, 255) ergibt Weiß. Gleiche Werte für alle drei Farbkomponenten ergibt Graustufen.



Mit dem Befehl **setze stiftfarbe auf h: s: v:** wird die Stiftfarbe nach dem HSV-Modell bestimmt. Dabei sind für h, s und v Werte zwischen 0 und 1 erlaubt.

üblicherweise 0 bzw. 255. Das gilt auch für Snap! und so liefern die Wertekombinationen (255,0,0) Rot, (0,255,0) Grün und (0,0,255) Blau. Die additive Mischung dieser Grundfarben ergibt die *Sekundärfarben* Gelb (255,255,0), Magenta (255,0,255) und Cyan (0,255,255). Die Addition aller drei Farben zu gleichen Teilen ergibt Graustufen, z.B.

(180,180,180); bei hellster Ausprägung ergibt sich Weiß (255,255,255). Das Fehlen jeglicher Farbanteile ergibt Schwarz (0,0,0). Lokalisieren wir die Farbanteile als Koordinaten in einem dreidimensionalen Raum, erhalten wir den RGB-Farbwürfel:



Die geometrische Farbanordnung im RGB-Modell ist für die Farbwahl nicht gerade intuitiv nutzbar[41] und konkurriert insbesondere mit einem Farbmodell, das sich eher an unserer Farbwahrnehmung orientiert, dem HSV-Modell. In diesem Modell werden die Farben durch drei Eigenschaften beschrieben, dem Farbton (englisch: Hue), der Sättigung (englisch: Saturation) und der Helligkeit (englisch: Value).

Die *Farbtonskala* reicht von Rot, Gelb, Grün über Blau und Magenta bis Dunkelrot:



Die *Sättigung* einer Farbe reicht von 0% (gar keine Farbe) bis 100% (reine Farbe):



Die *Helligkeit* einer Farbe reicht von 0% (Schwarz) bis 100% (volle Helligkeit):



[41] Beim Karlsruher KIT gibt es eine Tabelle der RGB-Werte, jeweils mit den entsprechenden Farbbeispielen. Das erlaubt sowohl die Orientierung an den konkreten Farben als auch das Ermitteln der genauen RGB-Werte, die dann direkt in Snap! übernommen werden können.

Für die präzise Festlegung einer gewünschten Farbe gibt es nun zusätzlich zu den grafischen Grundbefehlen in der Blockbibliothek mit **setze stiftfarbe r: g: b:** und **setze stiftfarbe h: s: v:** zusätzliche Befehle, die sich an den etablierten Farbmodellen orientieren.





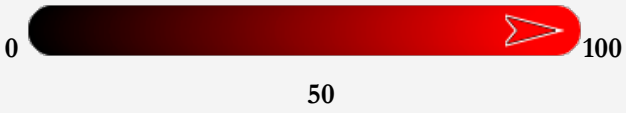
Mit dem Befehl **setze Stiftdicke auf** **dicke** wird die Stiftdicke auf die mit **dicke** festgelegte Breite (in Pixeln) festgelegt. Beim darauffolgenden Setzen von Punkten oder Ziehen von Linien wird diese Breite solange verwendet, bis sie durch eine neue Festlegung geändert wird.



Mit dem Befehl **ändere Stiftdicke um** wird die Stiftdicke um den angegebenen Wert (in Pixeln) geändert. Beim darauffolgenden Setzen von Punkten oder Ziehen von Linien wird diese Dicke solange verwendet, bis sie durch eine neue Festlegung geändert wird.



Die **Farbstärke** kann Werte zwischen 0 und 100 einnehmen. Eine Farbstärke von 0 ergibt Schwarz, eine Farbstärke von 50 (das ist auch die Voreinstellung) ergibt die gewählte Farbe, eine Farbstärke von 100 ergibt eine hellere Variante der gewählten Farbe, z.B.:





Mit **ändere Farbstärke um** kann die Farbstärke um den angegebenen Wert verändert werden.

Hinweis: Die folgenden Befehle gehören zur Blockbibliothek, die nachgeladen werden kann und in *Anhang C* beschrieben wird.



Mit dem Befehl **färbe hintergrund** wird die gesamte Bühne mit der aktuell eingestellten Farbe eingefärbt. Alle sonstigen auf der Bühne befindlichen Objekte und Linien bleiben davon unberührt.



Mit dem Befehl **setze linienende auf** wird das Ende einer Linie entweder abgerundet (**round**) oder abgeflacht (**flat**). Bei **round** wird die Abrundung halbkreisartig der Linienlänge hinzugefügt. Das kann bei großer Stiftdicke zu deutlichen Linienverlängerungen führen.

In der nachladbaren Blockbibliothek gibt es weitere Grafikbefehle, die uns in vielen der folgenden Bilder weitere Flexibilität bringen. So können Bilder vor einem gleichmäßig eingefärbten Hintergrund gezeigt werden. Dazu gibt es einen Befehl **färbe hintergrund**.

Ebenso beeinflusst das Linienende das Darstellungsergebnis: In der Voreinstellung ist ein Linienende immer abgerundet (**setze linienende auf round**), wodurch allerdings ein Objekt in Blickrichtung der Schildkröte verlängert wird (in der nebenstehenden Abbildung rechts). Mit dem Befehl **setze linienende auf flat** endet einen Linie gerade (in der Abbildung links), was im Prinzip - insbesondere bei dickeren Linien - präzisere Darstellungen erlaubt.



7.1 Hommage à Mihich: Painting #207

Den Einsatz von Farbe möchte ich anhand einer *Hommage à Vasa Mihich* (geb. 1933 in Jugoslawien) erarbeiten (vgl. Bild 13). Bekannt geworden ist Mihich eher mit seinen Skulpturen aus Acryl, die ihn als Vertreter der Konzeptkunst ausweisen. Zwischen 1998 und 2002 hat er sich aber auch intensiv mit computergenerierten Skizzen beschäftigt. In seinen Bildern geht es immer um das Zusammenspiel von Farbe und Form, womit er sich als Professor für Design an der University of California systematisch beschäftigte[42].

Für Bild 13 benötigen wir waagrechte und senkrechte Linien. Wie so etwas geht, soll an einem einfachen Linienmuster gezeigt werden. Die Grafik kann mit reinen Wiederholungsschleifen erzeugt werden. Diese sollen aber so allgemein gehalten werden, dass sie für alle farblich unterschiedlichen Liniengruppen nutzbar werden. Sie sind ein gutes Beispiel dafür, wie durch Verallgemeinerung, Modularisierung und Kontrollstrukturen eine kompakte und flexible Lösung möglich wird. Der Block wird mit entsprechenden Kenngrößen aufgerufen:

- **n** gibt die Anzahl der zu zeichnenden Linien an.
- **x** und **y** sind die Koordinaten des Startpunkts der ersten Linie.
- **d_x** und **d_y** sind die Abstände der horizontalen bzw. vertikalen Linien.
- **r** gibt die Ausrichtung der Linien an (hier horizontal bzw. vertikal).
- **l** gibt die Länge der Linien an.

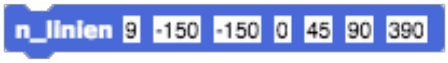
1. Zunächst wird die Schildkröte an den Startpunkt **x**, **y** bewegt mit Blickrichtung **r**.
2. Von dort wird die erste Linie gezeichnet.

[42] Auf der Webseite von Vasa Mihich (http://vasastudio.com) finden sich nicht nur zahlreiche Bilder seiner Kunstwerke, sondern auch zwei Bücher zum Download. Sie bieten - insbesondere in Band 2 - einen faszinierenden Einblick in seine Arbeitsweise und welche Rolle die Computerentwürfe dabei spielen.



Es folgen zwei Tests, ob horizontale oder vertikale Linien gezeichnet werden sollen (über Prüfung der Richtung **r**). Das ist notwendig, weil entsprechend der Startpunkt der nächsten Linie horizontal bzw. vertikal zu verschieben ist.

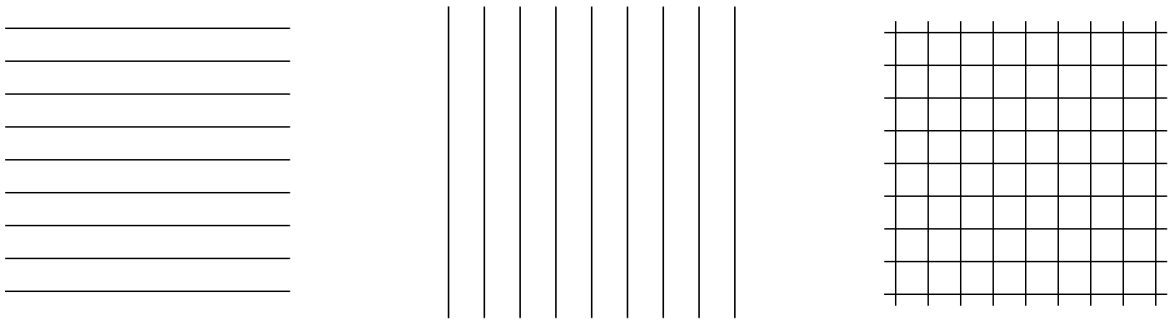
Mit dem ersten Aufruf werden 9 horizontale Linien gezeichnet, festgelegt durch **r = 90**:



Mit dem zweiten Aufruf werden 9 vertikale Linien gezeichnet, festgelegt durch **r = 0**:



Werden beide Aufrufe direkt aneindergefügt, entsteht das Gitternetz rechts.



Damit haben wir schon die Bausteine für die *Hommage à Mihich*, die allerdings durch das gewünschte Farbmuster ein wenig aufwändiger wird.

Mihich verwendete neun Farben, die er in Dreier-Gruppen einteilte. Damit die Linien nicht nur übereinander liegen, sondern den Eindruck eines Flechtwerks ergeben, reicht es daher nicht, das oben gezeigte Gitternetz mit Farben zu wiederholen, sondern es sind gesonderte Schleifen für jeweils eine Dreiergruppe nötig. Die notwendigen Kennwerte sind vorab festzulegen:

- **breite** und **hoehe** sind die Längen der horizontalen bzw. vertikalen Linien.
- **abstand** ist der Abstand zwischen den Linien einer Dreier-Gruppe.
- **n_horizontal** und **n_vertikal** ist jeweils die Anzahl der Linien in einer Farbgruppe (bei Mihich jeweils 3).
- **xstart** und **ystart** sind die Werte für die Anfangsposition der Schildkröte beim Zeichnen einer Farbgruppe.

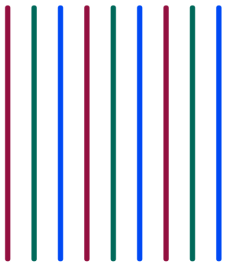
Für die Darstellung farbiger Linien ist der Block **n_linien** um das Setzen der Stiftfarbe zu erweitern:



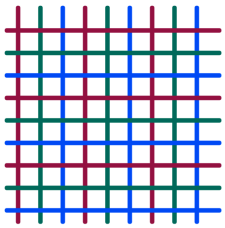
Die Komponenten des RGB-Farbmodells werden dafür an **n_linien** übergeben:



1. Mit der ersten Dreiergruppe werden nun senkrechte Linien mit bestimmtem **abstand** gezeichnet. Mihich verwendete dafür die Grundfarben Rot, Grün und Blau. Nach Positionierung der Schildkröte auf der Anfangsposition mit Blickrichtung nach oben wird in einer 3-er-Schleife jeweils eine Linie der gewählten Farbe gezogen.



2. Mit denselben Farben werden in anloger Weise darüber die horizontalen Linien gezogen.



Als Zwischenergebnis ist ein farbiges Gitternetz entstanden.

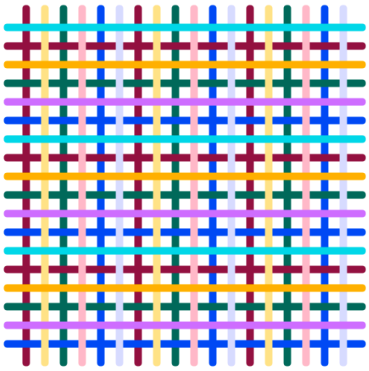
3. Mit der zweiten Dreiergruppe, also den Farben Hellblau, Hellviolett und Hellorange, werden nun Linien in die vertikalen Zwischenräume gezeichnet.

4. In die horizontalen Zwischenräume werden schließlich Linien mit der dritten Dreiergruppe gezeichnet, den Farben Türkis, Orange und Violett.

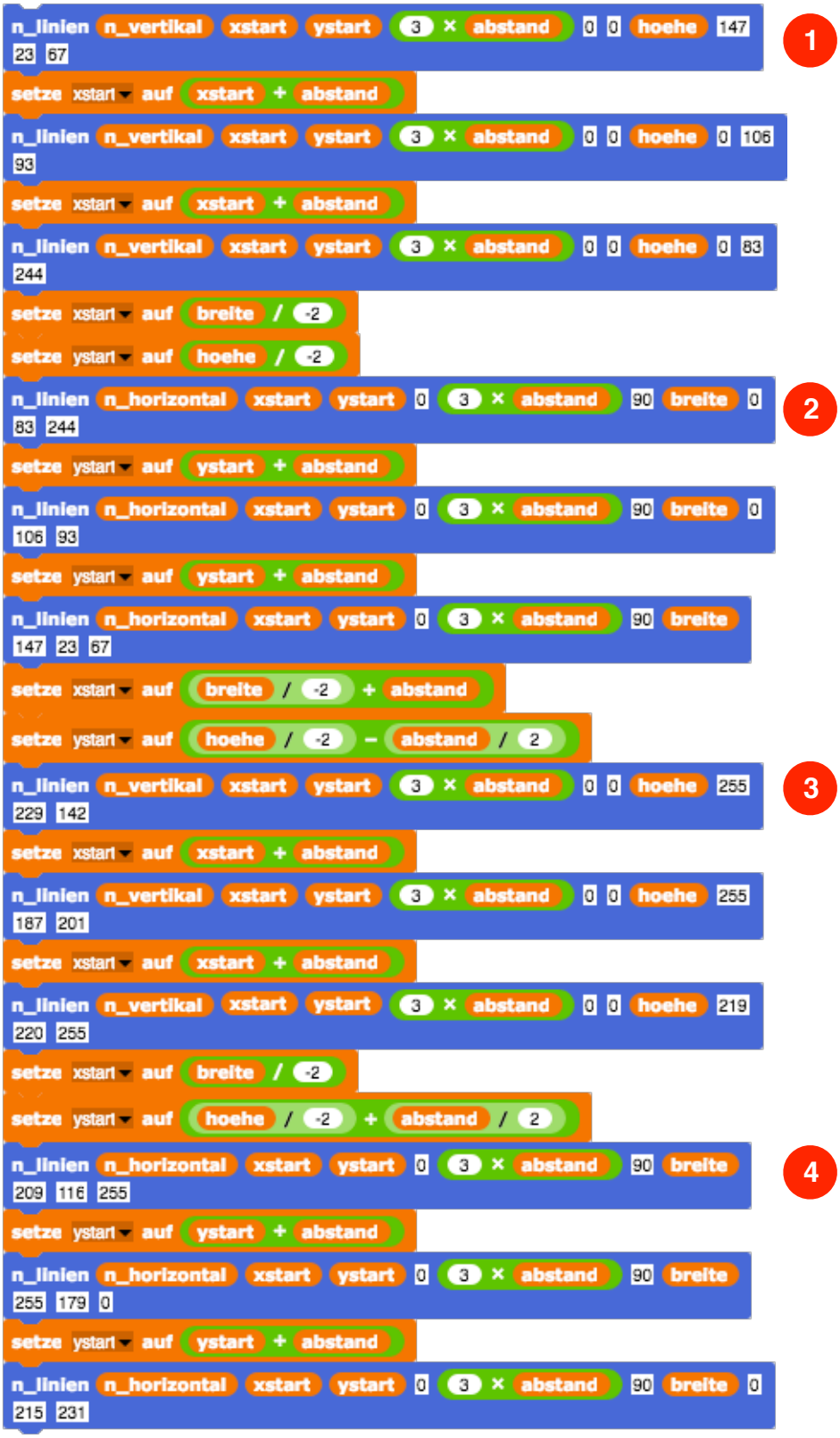


Zu beachten ist die Festlegung der horizontalen bzw. vertikalen Ausrichtung der Linien durch **r = 0** bzw. **r = 90**.

Ebenso sind nach dem Zeichnen jeder Dreiergruppe die Werte für die Anfangsposition der Schildkröte zurück zu setzen bzw. anzupassen.



Als Endergebnis ist das gewünschte Flechtwerk entstanden. Es unterscheidet sich noch deutlich von Bild 13 hinsichtlich Größe und Dicke der Linien sowie deren Abstände. Entsprechende Anpassungen können leicht über die Kennwerte vorgenommen werden.


n_linien n_vertikal xstart ystart 3 × abstand 0 0 hoehe 147
23 67
1
setze xstart auf xstart + abstand
n_linien n_vertikal xstart ystart 3 × abstand 0 0 hoehe 0 106
93
setze xstart auf xstart + abstand
n_linien n_vertikal xstart ystart 3 × abstand 0 0 hoehe 0 83
244
setze xstart auf breite / -2
setze ystart auf hoehe / -2
n_linien n_horizontal xstart ystart 0 3 × abstand 90 breite 0
83 244
2
setze ystart auf ystart + abstand
n_linien n_horizontal xstart ystart 0 3 × abstand 90 breite 0
106 93
setze ystart auf ystart + abstand
n_linien n_horizontal xstart ystart 0 3 × abstand 90 breite
147 23 67
setze xstart auf breite / -2 + abstand
setze ystart auf hoehe / -2 - abstand / 2
n_linien n_vertikal xstart ystart 3 × abstand 0 0 hoehe 255
229 142
3
setze xstart auf xstart + abstand
n_linien n_vertikal xstart ystart 3 × abstand 0 0 hoehe 255
187 201
setze xstart auf xstart + abstand
n_linien n_vertikal xstart ystart 3 × abstand 0 0 hoehe 219
220 255
setze xstart auf breite / -2
setze ystart auf hoehe / -2 + abstand / 2
n_linien n_horizontal xstart ystart 0 3 × abstand 90 breite
209 116 255
4
setze ystart auf ystart + abstand
n_linien n_horizontal xstart ystart 0 3 × abstand 90 breite
255 179 0
setze ystart auf ystart + abstand
n_linien n_horizontal xstart ystart 0 3 × abstand 90 breite 0
215 231

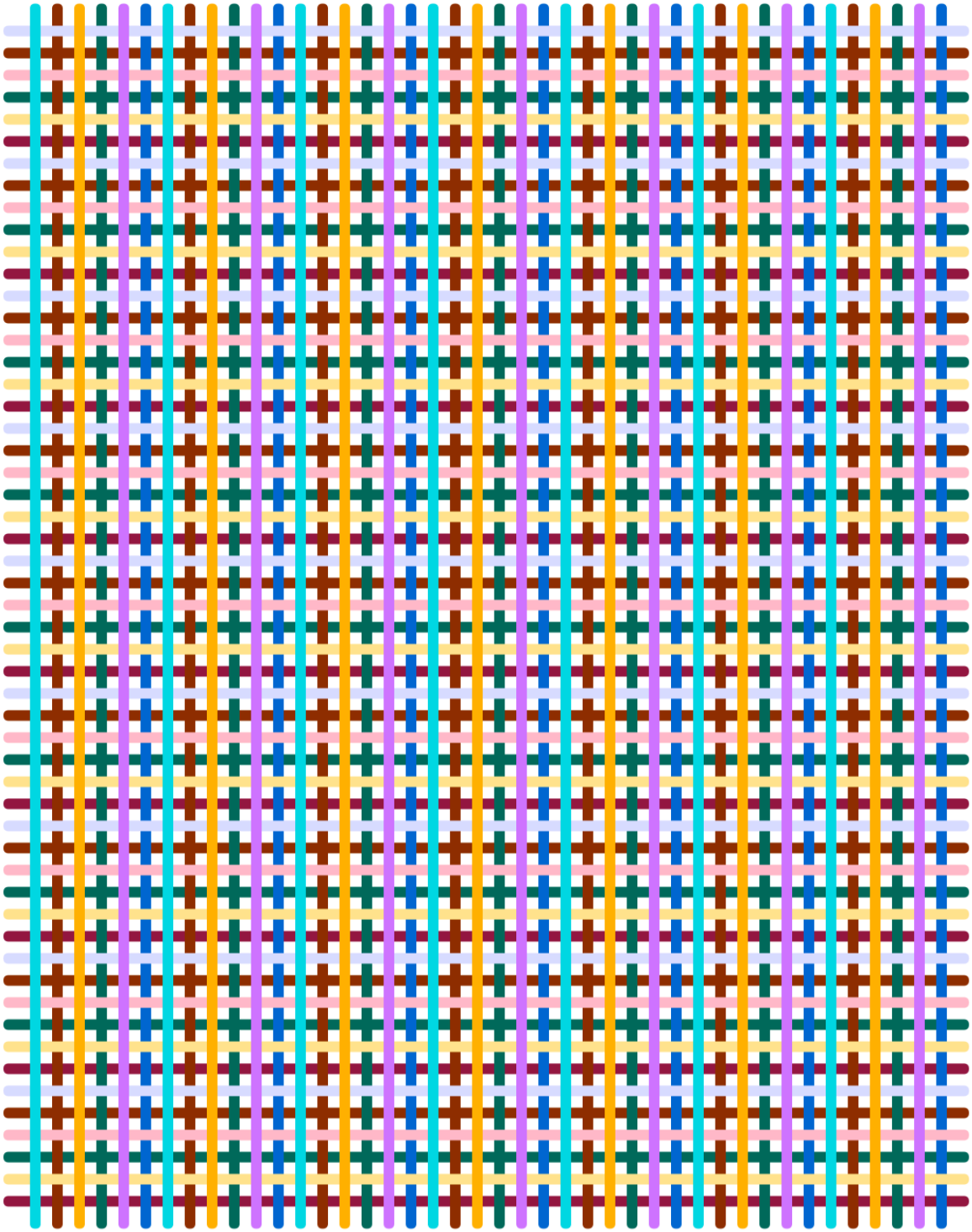


Bild 13: Hommage à Mihich: Painting #207

8. ALLES ZUFALL ... DIE SIMULIERTE INTUITION

Im Kapitel 1: *Computerkunst* hatten wir die Einschätzung von Nake u.a. kennen gelernt, dass in der Computerkunst der *Zufall* die *simulierte Intuition* sei. Das gilt im Übrigen auch für andere Kunstrichtungen. Deshalb soll in diesem Kapitel gezeigt werden, wie wir einerseits den Zufall konkret in unsere Algorithmen einbauen und durch die zufälligen Elemente die Darstellungsmöglichkeiten deutlich erweitern, andererseits ihn durch interaktive Elemente wieder bändigen.

Das Malen durch Klecksen ist eine altbekannte Technik. Unter anderem wird damit schon kleinen Kindern die Möglichkeit gegeben, sich kreativ zu betätigen[43]. Es hat aber auch Eingang gefunden in seriöse Formen der Kunstproduktion, etwa als Tropfverfahren, das von Max Ernst unter dem Namen *Oszillation* entwickelt wurde. Jackson Pollock hat - angeregt durch Max Ernst - diese Technik weiter entwickelt und ihr als *Drip Painting* international zum Durchbruch verholfen (vgl. dazu *Hommage à Pollock: Drip Painting* im Kapitel 25: *Codierte Kunst*).

Die Entstehung des als Beispiel gezeigten Klecksbilds (Bild 14) könnte wie folgt nachvollzogen werden (natürlich sind Alternativen genauso denkbar):

1. Die Ausbuchtungen eines jeden Kleckses sollen eine sternförmige Anordnung um einen Mittelpunkt haben.
2. Der einzelne Klecks hat unterschiedliche Längen und Ausrichtung.
3. Jeder Klecks hat eine eigenständige Position innerhalb des Gesamtrahmens.
4. Jeder Klecks hat eine eigene Farbe.
5. Die Kleckse haben unterschiedliche Liniendicken.

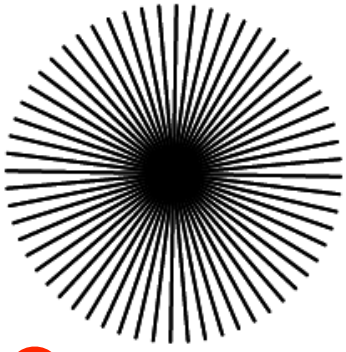
Das Bild besteht am Ende aus einer Anzahl zufällig verteilter Kleckse. Bei der Entwicklung eines Programms dafür können wir also folgende Bildelemente und Erweiterungen vorsehen:

1. Begonnen wird mit einem Einzelstern. Konkret wird in der *inneren* **wiederhole**-Schleife der Stern gezeichnet (hier mit 50 zunächst gleichlangen Linien in gleichem Winkelabstand).
2. Eine erste Variation erhalten wir dadurch, dass bei jedem Stern die Linien und der Winkelabstand zufällig in gewünschten Grenzen variiert werden[44].
3. Die *äußere* **wiederhole**-Schleife bestimmt die Zahl der Sterne (hier 30) und variiert die einzelnen Sterne durch zufällige Festlegung ...

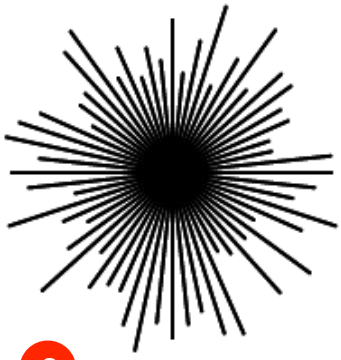
---

[43] Das „Klecksen“ hat einige Programmierer angeregt, dafür webbasierte Anwendungen anzubieten,
- z.B. Jack Rugile,
- oder interaktiv bei der Khan Akademie.

[44] Mit dem internen Zufallszahlengenerator von Snap! sind damit erzeugte Bilder immer Unikate, die sich nicht reproduzieren lassen. Sollen Bilder reproduzierbar sein, muss der Zufallszahlengenerator mit einem definierten Startwert versehen werden (vgl. **zufallszahl+startwert** im Kapitel 6: *Gibts nicht? Gibts nicht!*).



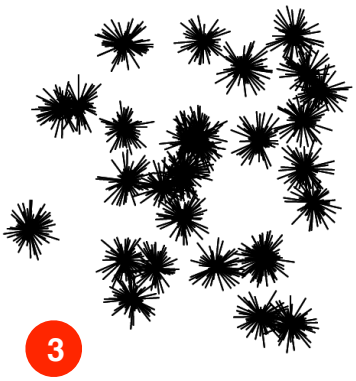
1 *Grundelement Stern aus Linien*



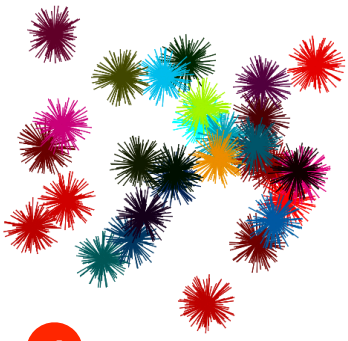
2 *unterschiedliche Linienlänge*



2 *unterschiedlicher Winkel*



3 *mehrere Sterne positionieren*



4 *unterschiedliche Farben*



5 *unterschiedliche Liniendicke*

4 des Mittelpunkts eines Sterns sowie Stiftfarbe, Farbstärke

5 und Stiftdicke.

Damit entsteht am Ende der gewünschte Gesamteindruck des Bildes. Da die Linien eines Sterns jeweils von seinem Mittelpunkt ausgehen sollen, müssen dessen Koordinaten zwischengespeichert werden. Das geschieht durch die Zuweisung der jeweiligen x- und y-Koordinaten zu den Skriptvariablen x und y.



Im Grunde besteht unser Programm nur aus wenigen bereits bekannten Befehlen, die aber allesamt durch die Übergabe zufälliger Werte eine große Variabilität erzeugen.



Bild 14: Klecksbild I

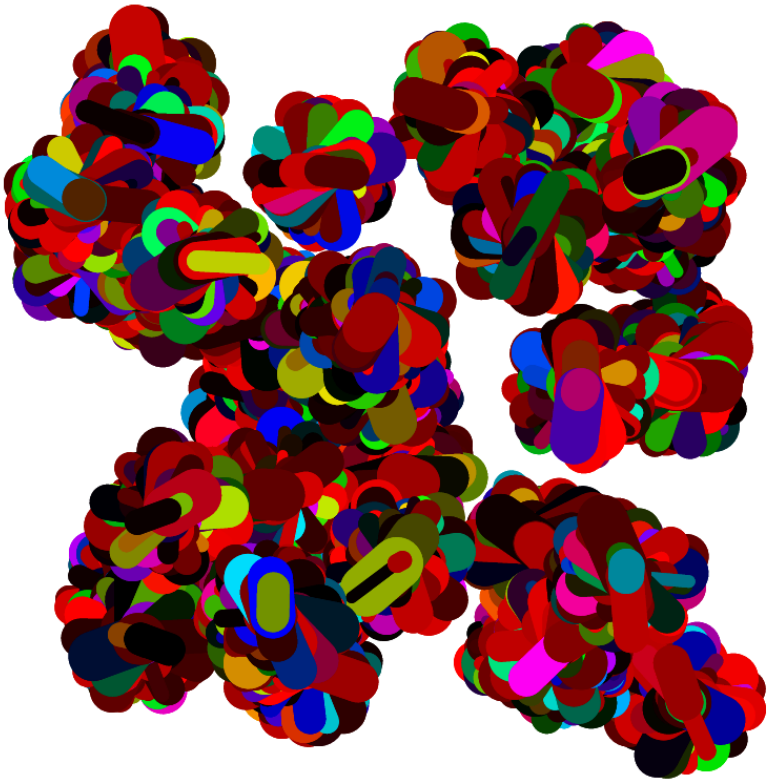


Bild 15: Klecksbild II - Farbverschiebungen

Anregung: *Das* Bild 15 *unterscheidet sich deutlich von dem vorigen* Bild 14*, wird aber erzeugt, indem lediglich die zufällige Festlegung von* Stiftdicke*,* Stiftfarbe *und* Farbstärke *an anderer Stelle erfolgt. Finden Sie heraus, wohin sie verschoben werden müssen?*





Mit dem Befehl **Skriptvariablen** werden lokale Variablen erzeugt, die nur temporär innerhalb der aktuellen Befehlsfolge oder innerhalb eines Blocks gültig sind. Voreinstellung ist eine Variable **a** (der Name kann wie üblich nach Doppelklick geändert werden). Werden weitere Variablen benötigt, lassen sich solche durch Klick auf den Rechtspfeil erzeugen.



Mit dem Befehl Zufallszahl von x bis y werden ganze Zahlen (Integer) zwischen x und y geliefert (voreingestellt ist der Bereich 1 bis 10), z.B.:



Durch die Übergabe der unteren und oberen Grenze des Zahlenbereichs lassen sich die Zufallszahlen je nach Verwendungszweck genau steuern:

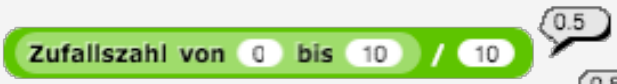


liefert entweder 0 oder 1,

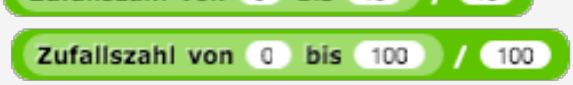


liefert ganze Zahlen im Bereich -10 bis +20.

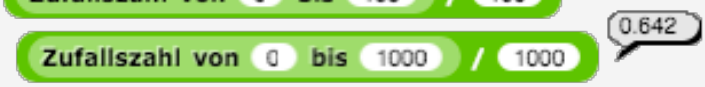
Wenn Dezimalzahlen benötigt werden, so lassen auch die sich im gewünschten Bereich mit der gewünschten Zahl an Nachkommastellen erzeugen:



mit einer Dezimalstelle,

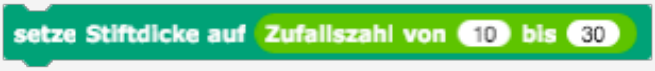


mit zwei Dezimalstellen,



mit drei Dezimalstellen.

Die Zufallszahlen können direkt als Übergabeparamter der betroffenen Befehle erzeugt werden, z.B.:



9. DER FIGURENBAUKASTEN

Der russische Maler und Grafiker Wassily Kandinsky war nicht nur einer der Wegbereiter des Expressionismus. Er war auch Kunsttheoretiker und hat mit Über das Geistige in der Kunst eine theoretische Fundierung der abstrakten Kunst versucht. In seiner Zeit als Lehrer am Bauhaus in Dessau folgte 1926 das Buch Punkt und Linie zu Fläche, ein analytisches Erarbeiten der Basiselemente der Malerei und des Zeichnens. Wir hatten ebenfalls Punkte, Linien(züge) und Schraffuren als charakteristische Elemente der frühen Computerkunst identifiziert (im Kapitel 1: *Computerkunst*). Wir erarbeiten dafür nun einen Figurenbaukasten, der Grundlage für die weiteren Kapitel bilden wird, die sich mit typischen Vertretern und Beispielen der Computerkunst befassen.

9.1 Punkte

Wir verstehen Punkte hier nicht im mathematischen Sinne. Denn mit Objekten ohne jede Ausdehnung lassen sich schwerlich ansehnliche Bilder produzieren: „*Ein Punkt hat keine Größe und ~~ist eindimensional~~* [hat die Dimension Null, JW]*, das heißt, er hat weder Breite noch Länge noch Tiefe. Er bezeichnet lediglich eine eindeutige Position. Abgesehen davon, dass er einen Ort festlegt, existiert ein Punkt also physisch eigentlich nicht*“ (Arnone, 2014, S. 30). Wir folgen also besser dem Mathematiker Oskar Perron (1962): „*Ein Punkt ist genau das, was der intelligente, aber harmlose, unverbildete Leser sich darunter vorstellt.*“ In Snap! haben wir denn auch gleich mehrere Möglichkeiten, mit Punkten zu arbeiten.

Dabei ist in Erinnerung zu rufen, dass Snap! keinen eigenen Befehl kennt, mit dem ein Bildpunkt (Pixel) direkt gesetzt werden kann. Dafür gibt es aber einen selbstgeschrie-

Hinweis: Die folgenden Befehle gehören zur Blockbibliothek, die nachgeladen werden kann und in *Anhang C* beschrieben wird.





Mit dem Befehl **Pixel bei x y** bewegt sich die Schildkröte zur Position, die durch **x** und **y** festgelegt wird. Der Zeichenstift wird abgesenkt und zeichnet eine Linie der Länge 1. Danach wird die Schildkröte an den Ausgangspunkt zurückbewegt und der Zeichenstift wieder angehoben.



Mit dem Befehl **Punkt der Ausdehnung a bei x y** bewegt sich die Schildkröte zur Position, die durch **x** und **y** festgelegt wird. Der Zeichenstift wird abgesenkt und zeichnet eine Linie der Länge **a** mittig um die Position. Die Schildkröte befindet sich danach am Ausgangspunkt, die Stiftdicke ist auf 1 zurückgesetzt und der Zeichenstift ist wieder angehoben.

benen Block **Pixel bei x y**. Damit können einzelne Punkte an beliebigen Stellen der Bühne platziert werden[45].

Allein mit der Variation der Farben und Farbstärken lassen sich komplexe Strukturen erzeugen. In den beiden Farbfeldern (Bild 16) wurden einzelne Pixel zufällig (I) bzw. systematisch (II) farblich geändert und zeilenweise nebeneinandergesetzt.

Im Beispiel Bild 16 (II) wurde durch die Bestimmung der Rot-, Grün- und Blau-Komponenten mittels der Verrechnung der aktuellen x- und y-Position die Systematik der Farbänderung festgelegt. Sie können dafür natürlich auch andere Komponenten oder mathematische Funktionen (wie z.B. den Sinus) heranziehen.

9.2 Hommage à Hugonin: Binary Rhythm

Werden dickere, quadratische Punkte gesetzt (bei Verwendung von **Punkt der Ausdehnung a bei x y** ist dazu **setze linienende auf flat** zu setzen!), entstehen Farbmuster wie in Bild 17, das sich stark an die Vorgehensweise des britischen Malers James Hugonin (geb. 1950 in Barnard Castle, Großbritannien) anlehnt. In seiner Serie *Binary Rhythm* hat er jeweils tausende farbige Rechtecke nebeneinander gesetzt. Angeregt von den pointillistischen Bildern Georges Seurats übertrug er die Technik auf seine abstrakten Bilder. Mit sorgfältig geplanten Farbkombinationen erzeugt er durch ihre Wechselwirkungen komplexe Muster[46]. Seine Bilder, die meist aus zehntausenden kleinen Farbfeldern bestehen, sind in monatelanger Arbeit in klassischer Maltechnik entstanden. Im Gegensatz zu unserem Programm bleibt dabei nichts dem Zufall überlassen: *"It's carefully orchestrated, nothing is left to chance, yet ironically, the only way we can look at the painting is to accept chance. Accept the fact that these rhythms don't all join together in a recognisable pattern. But the process of making the painting is incredibly ordered."*

Das Programmgerüst für diese Bilder ist recht simpel: Es reichen zwei geschachtelte Schleifen mit denen für die gewählte Anzahl Reihen die gewünschte Anzahl Punkte gesetzt werden.

Hugonin hat seine Farbfelder auch in rechteckiger, runder oder ovaler Form ausgemalt. Die dadurch sehr unterschiedlich wirkenden Strukturen lassen sich mit unserem Programmgerüst leicht nachempfinden, wenn statt quadratischer Punkte eben Rechtecke oder Kreise gesetzt werden.

Tipp: *Wie immer ist es ein lohnendes Unterfangen, durch sukzessives Verändern der relevanten Parameter ein Gefühl dafür zu bekommen, wie gewünschte Effekte zu erreichen sind. So bietet es sich z.B. an, den Farbraum einzuschränken und die Farbstärke zu ändern. Von nahezu monochromen Farbflächen bis zu wilden Punktmustern wird alles möglich!*

---

[45] Häufig werden allerdings Punkte bestimmter Ausdehnung größer als ein Pixel benötigt. Neben dem im Kasten beschriebenen **Punkt der Ausdehnung ...** gibt es dafür noch den Block **punktscheibe um x y radius**.

[46] In einem Interview erläutert James Hugonin seine Vorgehensweise und seinen Weg dahin.

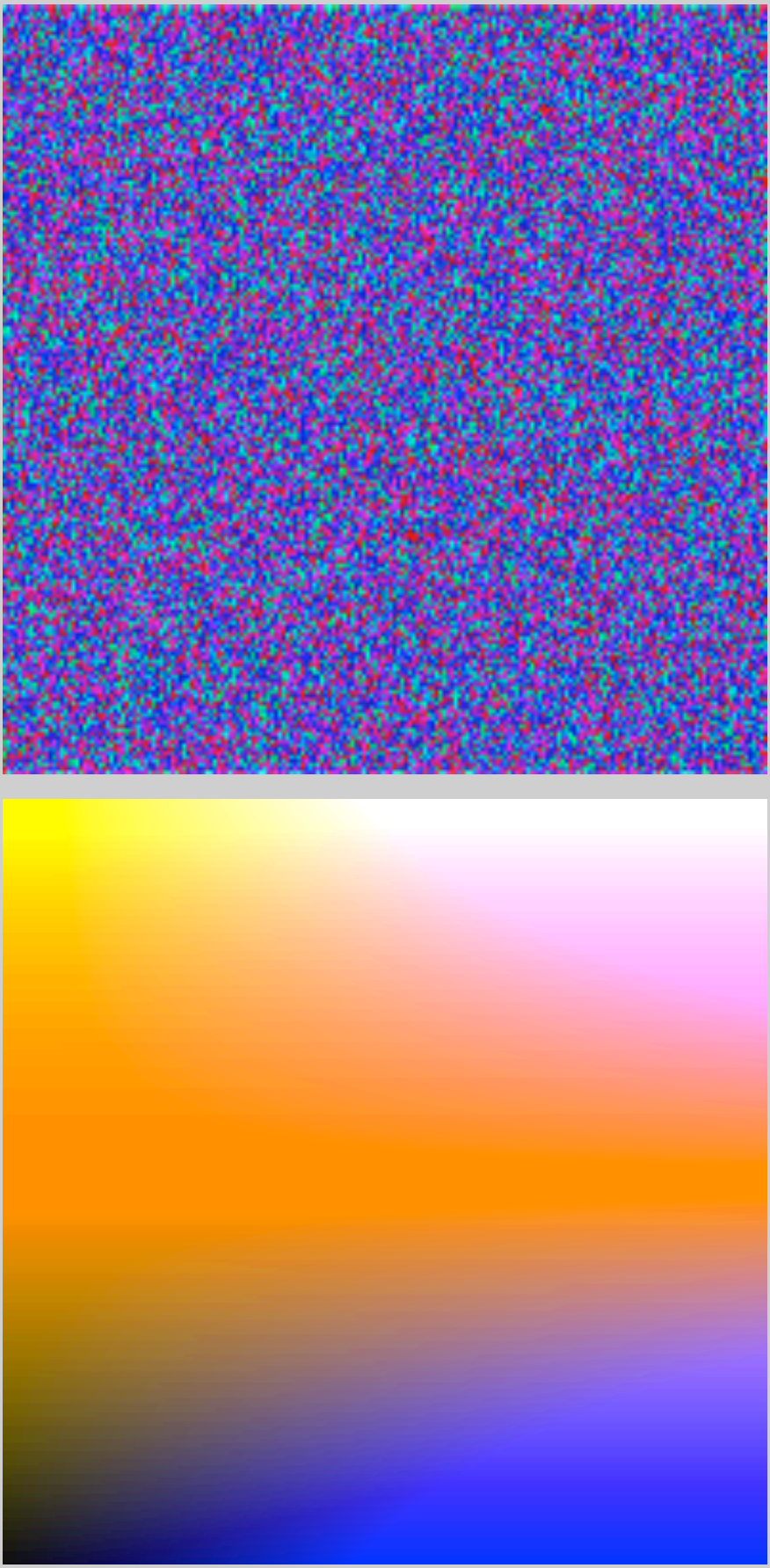


Bild 16: Farbfelder I & II

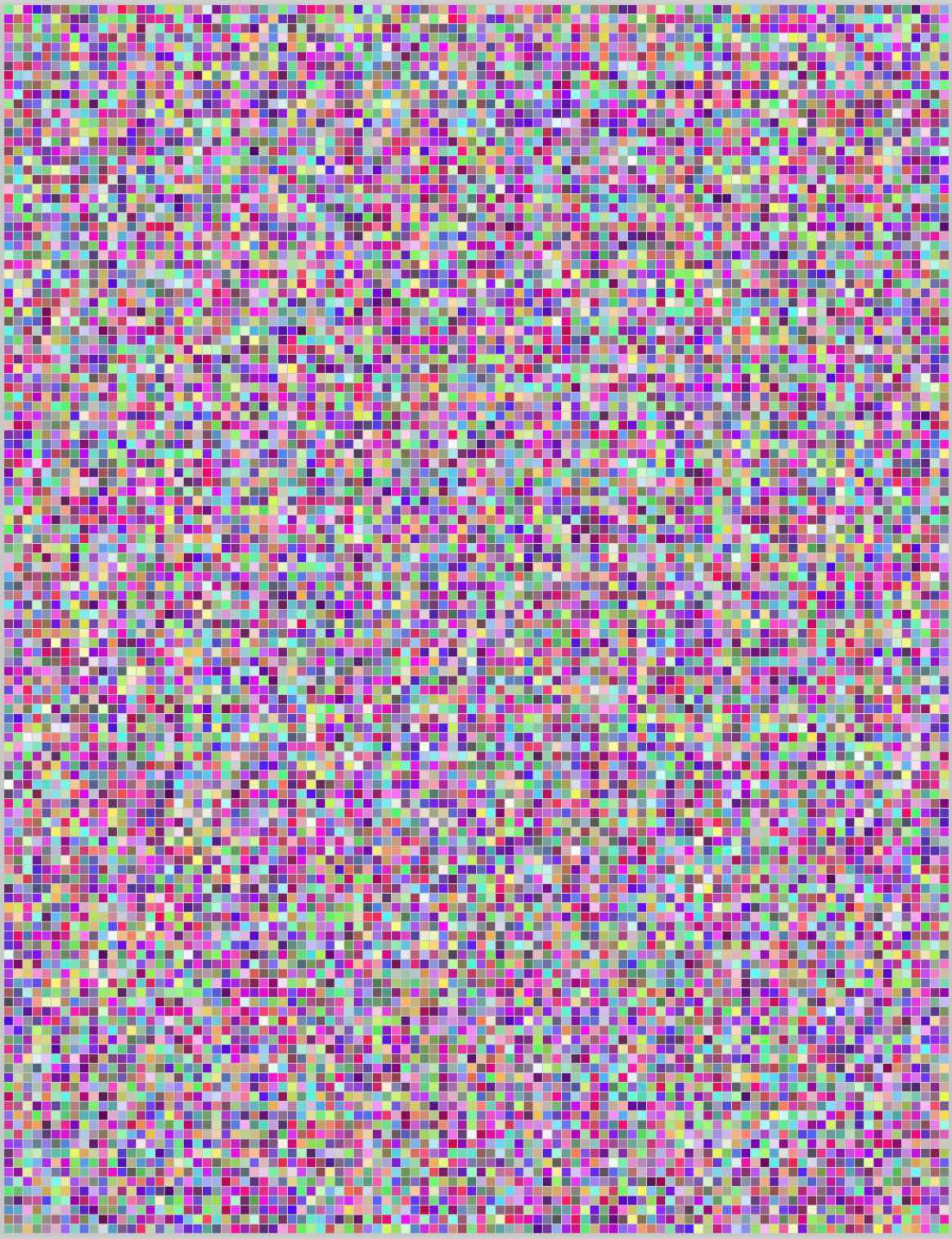


Bild 17: Hommage à Hugonin: Binary Rhythm

9.3 Hommage à Müller: 64/6

Der nächste Schritt bei der Verwendung von Punkten ist ihre Sortierung bzw. Platzierung nach vorgebenen Ordnungsschemata. Ein erstes Beispiel orientiert sich an einer Arbeit *64/6* von Gotthart Müller (gefunden in Rosen, 2011, S. 172).

Darin werden die anfangs gleichmäßig verteilten Punkte (in der Abbildung über Bild 18 links) einmal horizontal zufällig verrückt (Mitte), einmal vertikal verrückt (rechts).

Schließlich erfolgt dies gleichzeitig horizontal als auch vertikal entsprechend der Vorlage von Müller (Bild 18).

Ganz ähnlich gehen Bohnacker et al. (2009, S. 211 f.) vor. *„An der Grenze zur Unordnung ist die Spannung am höchsten“*, so beschreiben sie ihre Experimente (a.a.O., S. 210). Auch sie geben ein Raster vor, in dem sie schwarze Kreise hinter einem konstanten Gitter aus weißen Kreisen anordnen. Die Umsetzung folgt dem gleichen Schema wie das vorige Beispiel, also sowohl eine horizontale als auch vertikale Verrückung der in diesem Fall schwarzen Kreise. Mit gößer werdendem Zufallseinfluss beginnt sich das Raster dann aufzulösen.

Ausgegangen wird wieder von gleichmäßig verteilten Punkten. Dann werden die schwarzen Hintergrundkreise zunächst horizontal, dann vertikal und schließlich sowohl horizontal als auch vertikal zufällig verückt (Bild 19).

9.4 Hommage à Struycken: Komputerstrukturen

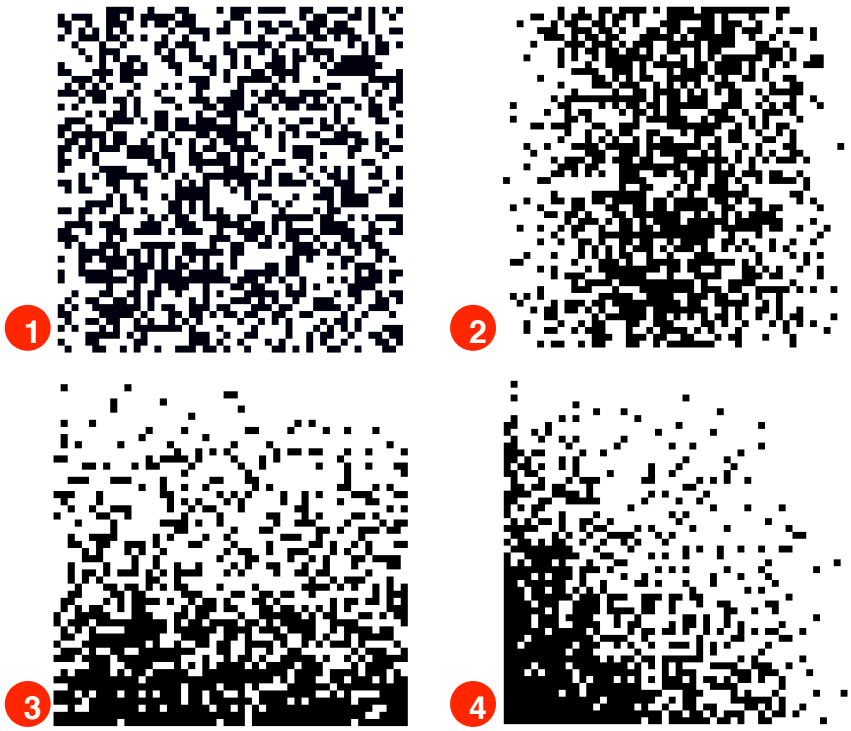
Mit dem letzten Beispiel zum Thema Punkte greife ich eine Vorlage von Peter Struycken auf. Der Niederländer Struycken (geb. 1939 in Den Haag, Niederlande) hat sich in seinem Werk intensiv und methodisch mit dem Verhältnis von Formen und Farben auseinandergesetzt, seit 1969 mit dem Computer als wichtigem Hilfsmittel. Besonders bekannt geworden ist sein Porträt von Königin Beatrix, die er 1981 für eine Briefmarkenserie der PostNL als Pixelgrafik („digitaler Pointillismus“) in bunten Punkten darstellte.

Für seine Bildserie *Komputerstrukturen* hat er ein Algol-Programm OSTRC von Stan Tempelaars[47] verwandt, mit dem er flexibel Punktmuster erzeugen und untersuchen konnte. Er verteilte dabei schwarze, viereckige Punkte zufällig in einem Rasterfeld. Die Abweichungen von reinen Gleichverteilungen erreichte er durch den Einsatz von *gewichtetem* Zufall. Beispielsweise kann die Eintrittswahrscheinlichkeit eines Ereignisses linear von 1 auf 0 abfallen (linearer Abfall) oder zwischen 0 und 1 normal verteilt sein (Glockentyp).

Das Ausgangsmaterial bei Struycken sind (vgl. Abbildung nächste Seite)

[47] Auf der Webseite Peter Struycken - OSTRC (http://www.iwriteiam.nl/PS_OSTRC.html) können originale Daten von Struycken in ein JavaScript-Programm kopiert, ausgeführt und die Ergebnisse dargestellt werden.

1. zunächst gleichverteilte schwarze Punkte in einem 50*50-Raster.
2. Varianten davon erhält er durch gewichteten Zufall, hier als Beispiel eine Dreiecksfunktion,
3. und hier mit einem linearen Abfall von unten nach oben.
4. Ein solcher gewichteter Zufall kann natürlich für die Reihen und Spalten unabhängig voneinander gewählt und kombiniert werden, hier der lineare Abfall von unten nach oben mit einem linearen Abfall von links nach rechts.



Die Arbeit mit Punkten, die bis hier geschilderten Blöcke und die Verwendung der Zufallsverteilungen sollen zusammenfassend noch einmal illustriert werden an dem Programm *Komputerstrukturen* von Peter Struycken, in dem diese Komponenten Verwendung finden. Bild 20 zeigt eine so erweiterte Variante der *Komputerstrukturen* mit farbigen Punkten in einem 60*80-Raster. Es kombiniert den Glockentyp[48] mit einem linearen Abfall von unten nach oben.

1. Es wird die reproduzierbare Erzeugung von Zufallszahlen verwendet (vgl. Kapitel 6: *Gibt's nicht? Gibt's nicht!*).
2. In einer Schleife werden 2500 Punkte gesetzt.
3. Für die x-Positionierung wird eine **zufallszahl gerade** mit linearem Abfall von links nach rechts verwendet.

[48] Die Blöcke für die Erzeugung von Zufallszahlen mit entsprechenden Gewichtungen (Dreiecksform **zufallsform dachform**, linearer Abfall bzw. Zunahme **zufallszahl gerade**, Glockentyp **zufallszahl glockenform** und Stufe **zufallsform stufe**) werden in Anhang C vorgestellt und in ihrer Funktionsweise beschrieben.

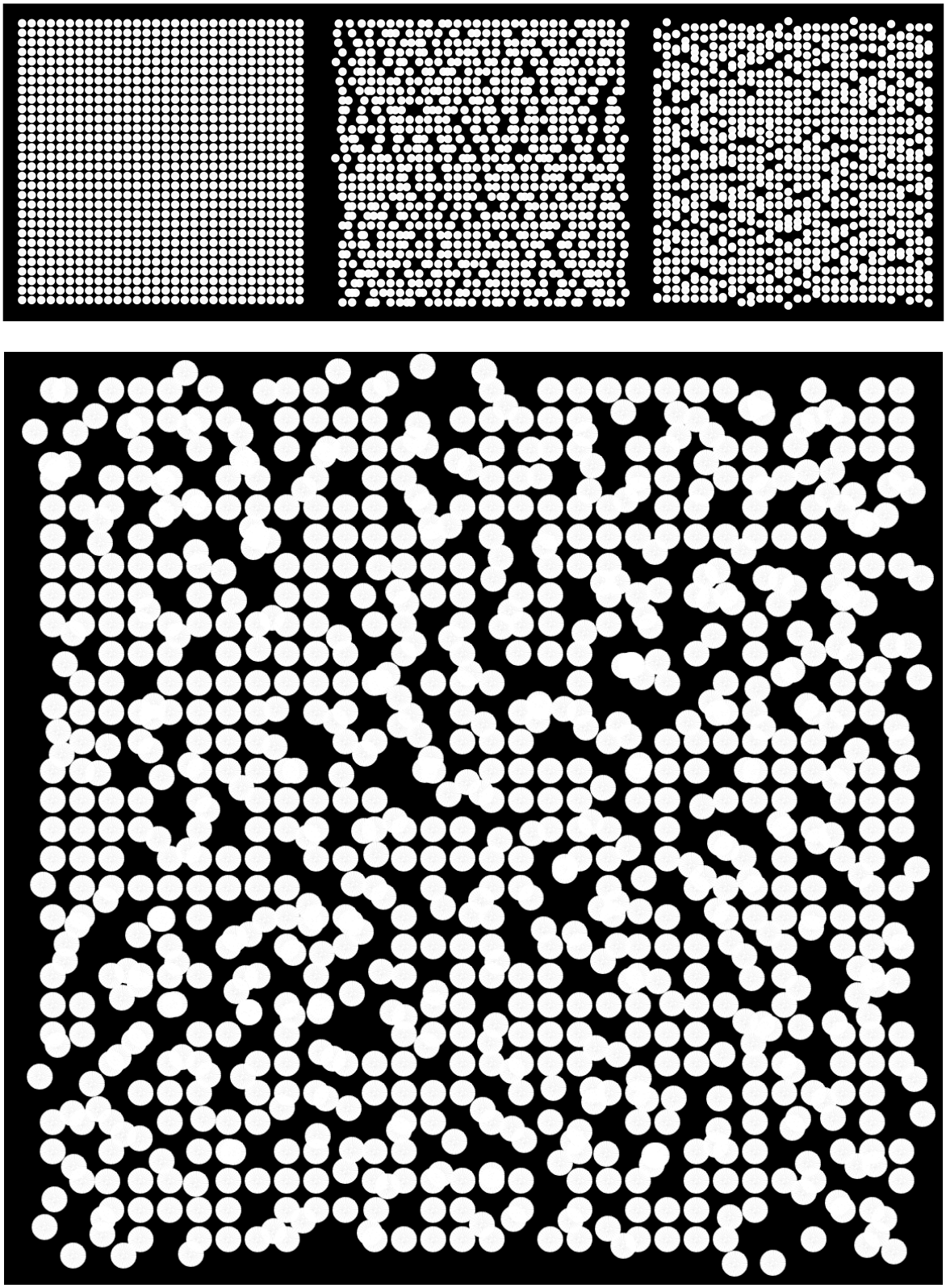


Bild 18: Hommage à Müller: 64/6

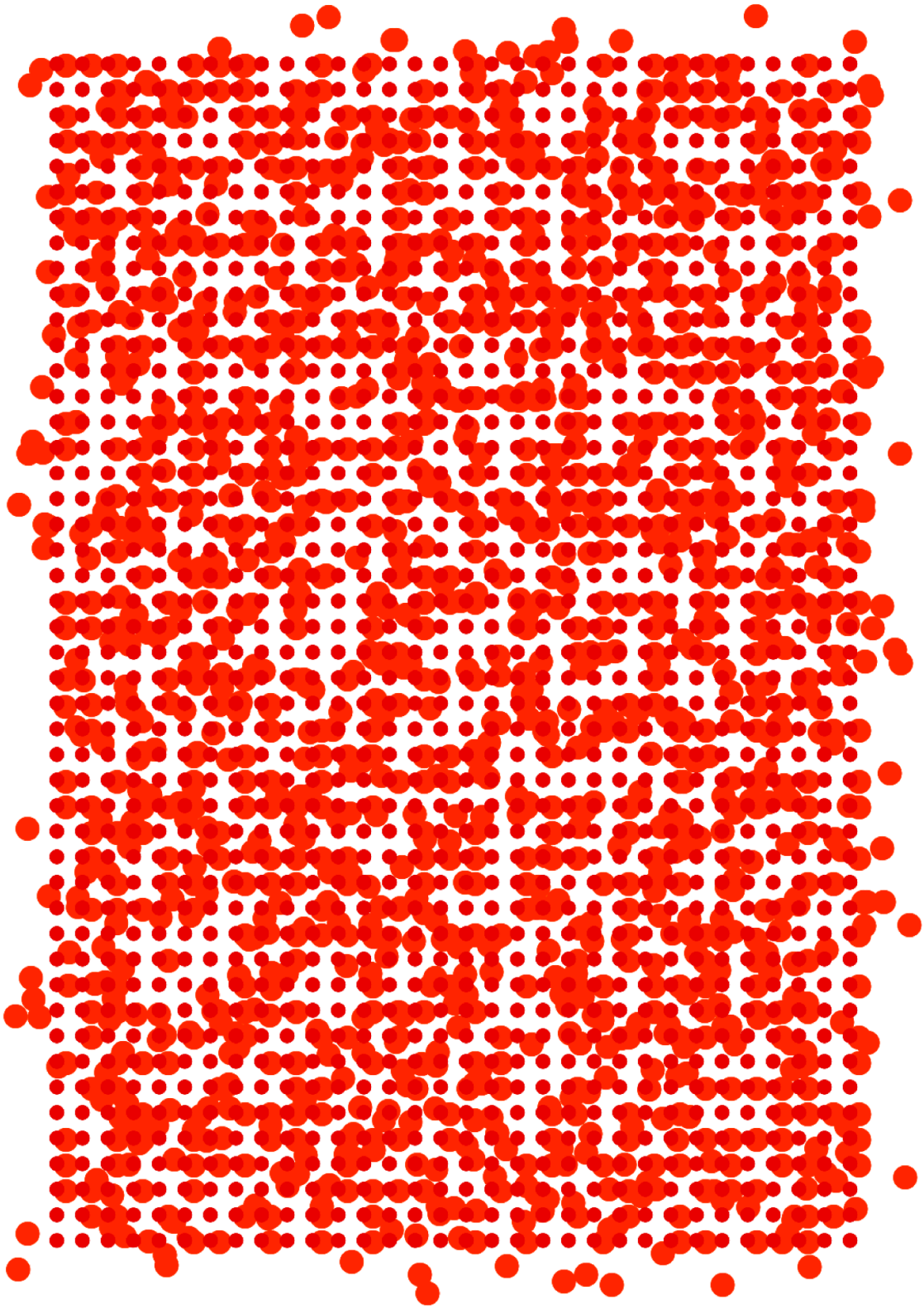


Bild 19: Rasterverschiebung

4 Für die y-Positionierung wird eine **zufallszahl glockenform** erzeugt, d.h. es werden normalverteilte Zahlen verwendet.

5 Mit einer zufällig aus einem vorgebenen Farbfenster gewählten Farbe wird dann ein Punkt gewünschter Größe gesetzt.



Wenn wir größere, „dickere“ Punkte (also **dicke** > 1) benötigen, zeigen sich schnell die Grenzen des Blocks **Punkt**. Entweder erhalten wir quadratische Punkte (nebenstehend links; bei **setze linienende auf flat**) oder abgerundete Linien (rechts; bei **setze linienende auf round**), weil durch die Abrundung das Objekt in Blickrichtung der Schildkröte verlängert wird. Um für alle unterschiedlichen Anforderungen gewappnet zu sein, gibt es deshalb zusätzliche Varianten für tatsächlich runde, kreisförmige Punkte: **punktkreis um x y radius** für einen farblosen umrandeten Kreis (nebenstehend links), sowie **punktscheibe um x y radius** für einen farbig gefüllten Kreis (rechts), jeweils zentriert um **x y** als vorgegebenem Mittelpunkt (Einzelheiten dazu in *Anhang C*).





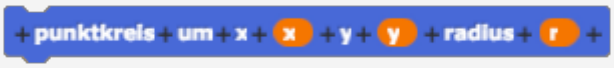
Anregung: *Sind Punkte allein nicht langweilig? Keineswegs! Punkte sind zwar die ersten Bausteine des umfangreicheren Figurenbaukastens, aber es ist erstaunlich, was sich allein damit - unter Einbeziehung von Wiederholungsschleifen, Farbe und Zufallskomponenten - bereits erzeugen lässt. Die Beispiele sollen dies andeuten und Sie anregen, weiter damit zu experimentieren.*

So bezieht sich Bild 21 auf Vera Molnar (vgl. das Kapitel 18: *Hommage à Vera Molnar*), deren frühe Arbeiten (zunächst noch ohne Verwendung von Computern) mit konstruktiven Kompositionsregeln wie Symmetrie und Wiederholung entstanden sind. Ihre Vorlage mit 64 Punkten in quadratischer Anordnung wurde von mir auf 96 in rechteckiger Anordnung erweitert.

Das brauchen wir von **Snap!**:



Hinweis: Die folgenden Befehle gehören zur Blockbibliothek, die nachgeladen werden kann und in *Anhang C* beschrieben wird.





Mit dem Befehl **punktkreis um x y radius** zeichnet die Schildkröte einen Kreis um den durch **x** und **y** festgelegten Punkt mit dem angegebenen **radius**. Die Schildkröte befindet sich nach Zeichnen des Kreises wieder am Ausgangspunkt mit ihrer ursprünglichen Richtung. Der Kreis wird mit einem Vieleck angenähert mit 36 Ecken. Je größer der Radius, desto gröber wird der Kreisumriss.



Der Befehl **punktscheibe um x y radius** ist identisch mit dem vorigen Befehl; allerdings wird hier der Kreis mit der aktuellen Stiftfarbe gefüllt (und wegen des scheibenartigen Aussehens hier **punktscheibe** genannt). Die Schildkröte befindet sich nach Zeichnen des Kreises wieder am Ausgangspunkt mit ihrer ursprünglichen Richtung.





Der Befehl **punktscheibe ...** verwendet intern den Befehl **male aus**. Damit wird die Fläche von der aktuellen Schildkrötenposition aus bis zur nächsten Linienbegrenzung mit der aktuellen Farbe ausgefüllt.



Hinweis: Wenn Linien oder Flächen den Bereich von **punktscheibe ...** schneiden, endet die Farbfüllung bereits dort. Um auch in solchen Fällen eine vollständige **punktscheibe ...** zu erhalten, gibt es zusätzlich den Befehl **punktscheibe-rekursivgefuellt um x y radius**. Bei diesem Befehl werden Kreise mit schrumpfendem **radius** um **x y** gezeichnet und so die vollständige Farbfüllung sichergestellt. Er liefert dasselbe Ergebnis, braucht aber etwas mehr Zeit.



Bild 20: Hommage à Struycken: Komputerstrukturen farbig im 60*80-Raster

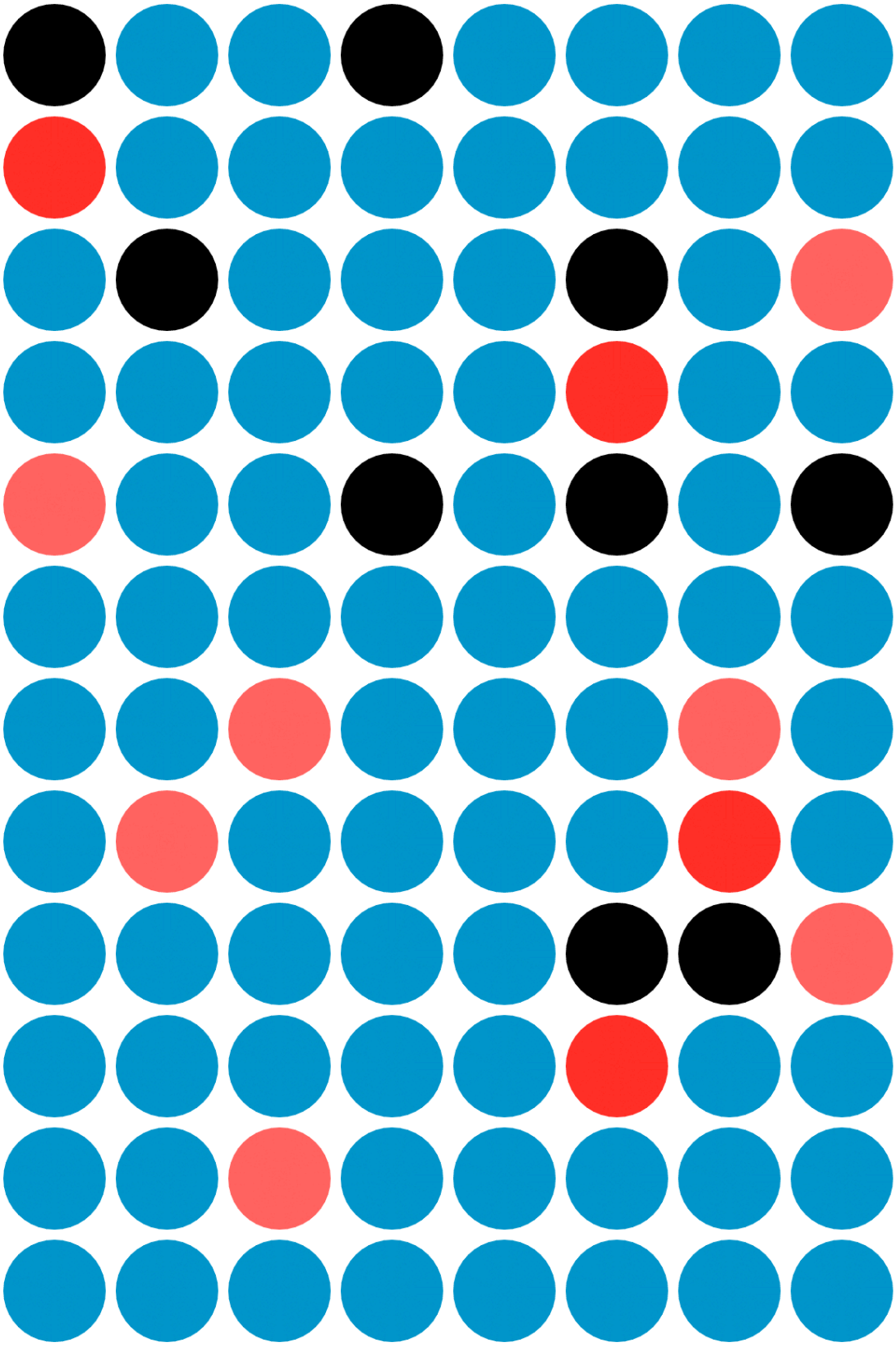


Bild 21: Hommage à Molnar: 96 Punkte

9.5 Linien und Strecken

In der Geometrie wird unter einer *Linie* die *Aneinanderreihung unendlich vieler Punkte* verstanden. Linien können dabei gerade, aber auch gekrümmt, offen oder geschlossen sein. Wir beschränken uns zunächst auf *Strecken*, d.h. gerade Linien mit einem Anfangs- und Endpunkt. Das entspricht dann der Bewegung der Schildkröte mit abgesenktem Stift, wodurch diese Strecken sichtbar gemacht werden. Für die unterschiedlichen Anwendungsfälle brauchen wir mehrere Blöcke, je nachdem ob eine Strecke von der aktuellen Position bzw. von einem Punkt A zu einem Punkt B führen soll, oder ob eine Strecke bestimmter Länge und bestimmter Richtung von der aktuellen Position bzw. von A aus gezogen werden soll.

Hinweis: Die folgenden Befehle gehören zur Blockbibliothek, die nachgeladen werden kann und in *Anhang C* beschrieben wird.



Mit dem Befehl **strecke nach xp yp** bewegt sich die Schildkröte von ihrer aktuellen Position zu der Position, die durch **xp** und **yp** festgelegt wird. Dabei ist der Zeichenstift abgesenkt zum Zeichnen der Linie. Danach wird der Zeichenstift angehoben.



Mit dem Befehl **strecke von A nach B** wird die Schildkröte von ihrer aktuellen Position mit angehobenem Zeichenstift zu der Position **A** bewegt. Dort wird der Zeichenstift abgesenkt und die Schildkröte zu der Position **B** bewegt, wodurch die Linie gezeichnet wird. Danach wird der Zeichenstift wieder angehoben.

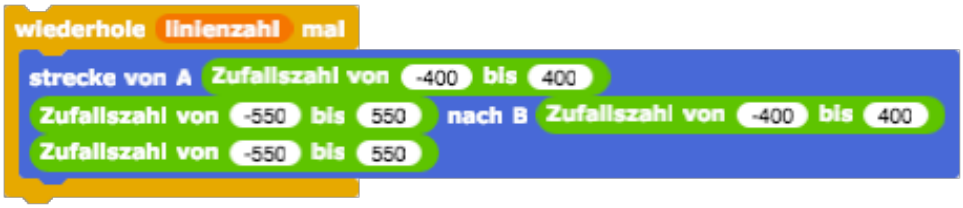


Mit dem Befehl **strecke laenge richtung** bewegt sich die Schildkröte mit abgesenktem Zeichenstift von ihrer aktuellen Position in der vorgegebenen Richtung **r** um die Strecke **l**. Dabei ist der Zeichenstift abgesenkt zum Zeichnen der Linie. Danach wird der Zeichenstift wieder angehoben.

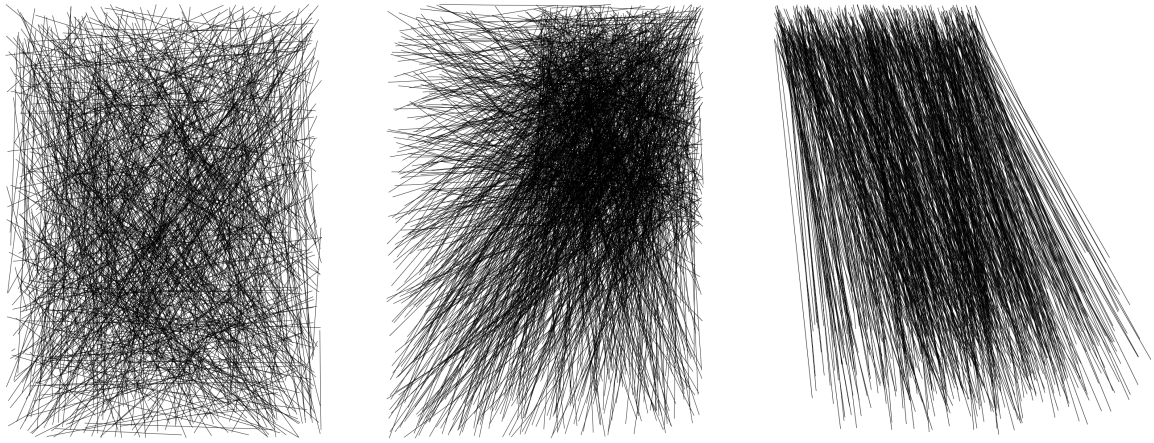


Mit dem Befehl **strecke von A laenge richtung** wird die Schildkröte von ihrer aktuellen Position mit angehobenem Zeichenstift zu der Position **A** bewegt. Dort wird der Zeichenstift abgesenkt und in der vorgegebenen Richtung **r** um die Strecke **l** bewegt. Danach wird der Zeichenstift wieder angehoben.

Abstrakte Liniengrafiken wurden von mehreren Vertretern der frühen Computerkunst entwickelt. Georg Nees hat daran systematisch die Redundanzen in zufallsgesteuerten Bildern untersucht (Nees, 1969, S. 226 u. 249). Der Nachvollzug seiner Bilder ist denkbar einfach:

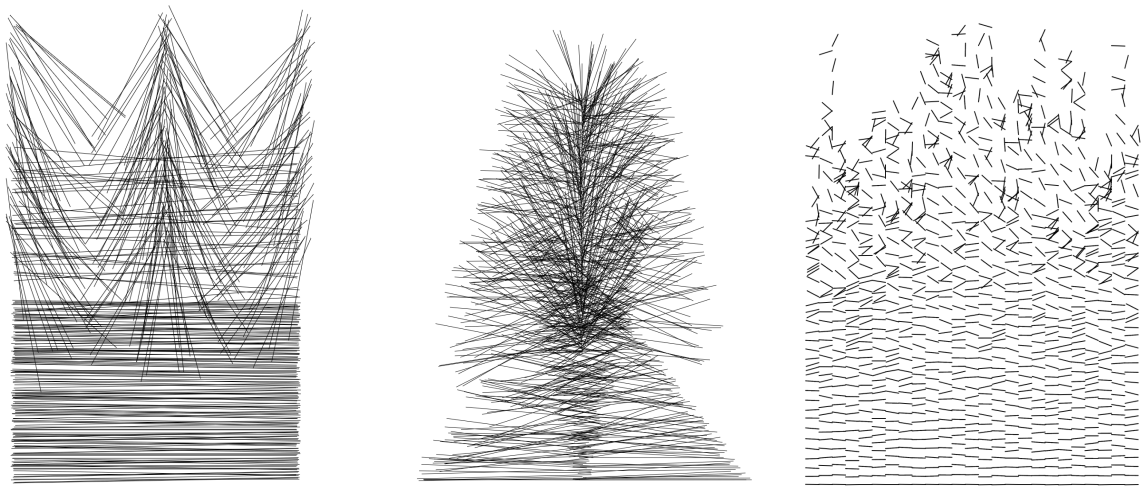


In einer Schleife mit der gewählten **linienzahl** wird jeweils von einem zufällig bestimmten Punkt **A** eine Strecke zu einem zufällig bestimmten Punkt **B** gezogen (folgende Abbildung links). Es liegt nahe, dafür den Block **strecke von A nach B** zu verwenden (das Grafikfenster reicht im Beispiel horizontal von -400 bis 400, vertikal von -550 bis 550). Mit der Eingrenzung des Ausgangspunktes A erreichen wir eine Verlagerung des Bildzentrums (Abbildung Mitte). Eine „besenartige" Struktur wird erreicht, wenn die Strecken von einem eingeschränkten Bereich der Ausgangspunkte in eine Vorzugsrichtung gezogen werden (Abbildung rechts). Dafür ist der Block **strecke von A laenge richtung** dann besser geeignet.



Das Beispiel zeigt, wie durch den gelenkten Zufall (hier für die Lage der Ausgangspunkte bzw. die Richtung der Linien) aus der rein chaotischen Linienanordnung gewollte Strukturen herausgearbeitet werden können. Nach diesem Prinzip hat die Malerin Sylvia Roubaud Arbeiten für eine Ausstellung der Firma Messerschmidt-Bölkow-Blohm MBB Computer Graphics (1972) angefertigt. Eine ganze Serie ist mit *Random superposition and explosion of straight lines* (ähnlich den folgenden Abbildungen links und Mitte) betitelt, eine weitere *Explosion of ordered structures* (ähnlich der folgenden Abbildung rechts).

Tipp: *Für die Grafiken wurden jeweils mehrere Schleifen aneinandergehängt. Das Aussehen wird wesentlich beeinflusst vom Setzen der jeweiligen Ausgangspunkte und der Richtung der Linien in jeder Schleife. Im Beispiel links sind es gesonderte Schleifen für die dichten Querlinien als Basis, darüber für die lichteren Linien, sowie vier für die nach unten gerichteten Fächer links, mittig und rechts. Im mittleren Beispiel sind es zwei für die zwei Basiskegel und eine für den darüberliegenden „buschigen" Teil. Im rechten Beispiel sind es einfach zwei geschachtelte Schleifen für die Spalten und Zeilen, wobei die zufälligen Abweichungen der einzelnen Linien von der Horizontalen nach oben hin zunimmt.*



9.6 Hommage à Sol LeWitt: Bands of Color

Wenn wir uns mit Linien befassen, kommen wir um Sol LeWitt (1928 - 2007) nicht herum. Dieser amerikanische Künstler hat sich systematisch mit geometrischen Formen und dabei auch speziell mit Linien befasst. LeWitt gilt als Vertreter der Minimal Art und Mitbegründer der Konzeptkunst (Conceptual Art). In unserem Kontext besonders interessant sind seine *Wall Drawings*. Er hat diese Werke nicht selbst ausgeführt, sondern er fomulierte genaue Anweisungen (das Konzept), die jedermann umsetzen konnte. Seine Arbeiten sind so logisch aufgebaut, das sie jederzeit wiederholbar sind. Von diesen Anweisungen an die Ausführenden vor Ort ist es eigentlich nur noch ein kleiner Schritt zu unseren Anweisungen an den Zeichenroboter auf der Bühne!

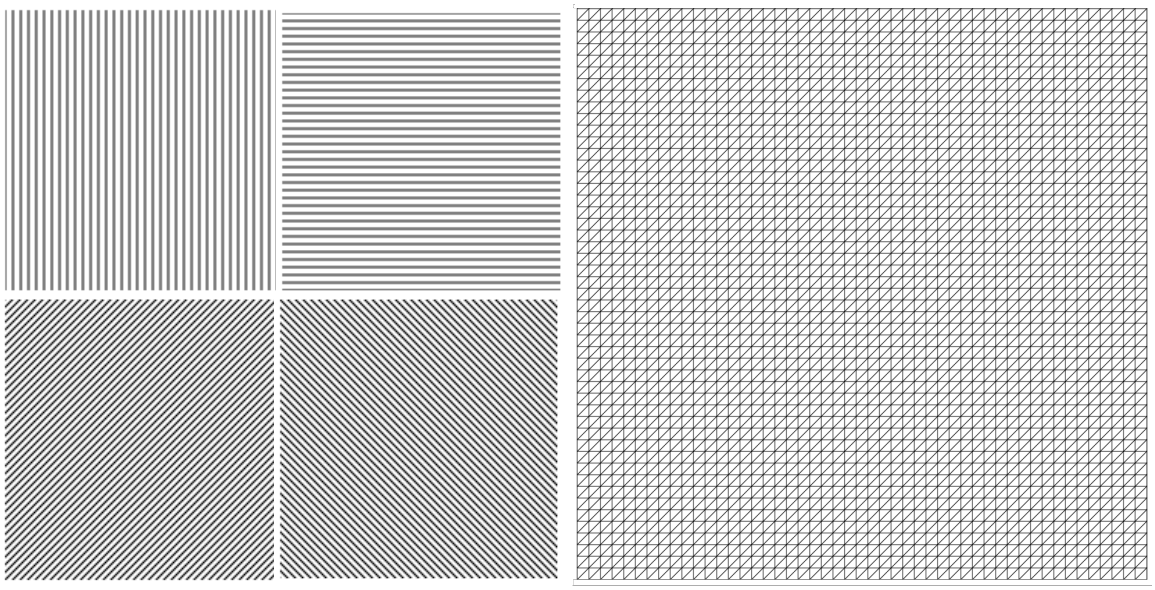
In mehreren seiner Kunstbücher konzentrierte sich LeWitt auf Linien, Winkel und Farben. Bei Linien hat er mehrfach systematische Kombinationen horizontaler, vertikaler und diagonaler Linien gezeichnet[49] und zu Texturen verdichtet. Ein markantes Beispiel ist sein Büchlein *Four basic kinds of straight lines : 1. Vertical. 2. Horizontal. 3. Diagonal l. to r. 4. Diagonal r. to l. and their combinations*. Der Titel enthält eigentlich schon alle Anweisungen für die Bilder in dem Buch. Er beginnt mit der schlichten Aneinanderreihung der vier Linientypen 1, 2, 3 und 4 (wie in der folgenden Abbildung links) und bildet dann daraus systematisch alle denkbaren Kombinationen, also 12, 13, 14, 23, 24, 34, 123, 124, 134, 234, 1234. Das folgende Beispiel rechts zeigt die Kombination 123.

Tipp: *Die Ausgangsmuster der vier Linientypen sind leicht in Wiederholungsschleifen darstellbar (vgl. etwa die* Hommage à Mihich *im Abschnitt* Wiederholungen*); ihre Kombination ergibt dann die Überlagerungen.*

Hinweis: *Da bei den Diagonallinien die Berechnung der Endpunkte etwas mühsam ist, können stattdessen einfach von den jeweiligen Ausgangspunkten lange Linien gezogen werden, die über den gewählten Bildrahmen hinausreichen. Das quadratische Bild erhalten Sie dann, wenn Sie den ge-*

[49] LeWitt: No Title: http://www.tate.org.uk/art/artworks/lewitt-no-title-p01069

wünschten Ausschnitt mit einem Quadrat umrahmen, dessen breite weiße (oder andersfarbige) Seiten die „überschüssigen" Linienteile überdecken.



Dieses Ausgangsmaterial liefert eine Fülle interessanter Strukturen, wenn Liniendicke, Farbe der Linien und die Art der Überlagerung variiert bzw. kombiniert werden. Eine Möglichkeit zeigt Bild 22, das ebenfalls einer seiner *Wall Dawings* nachempfunden ist.

All das sind Beispiele, die zeigen, dass schon ganz einfache Linienkombinationen zu interessanten Objekten führen können.

9.7 Streckenzüge und Polygone

In den letzten Beispielen wurden jeweils einzelne Strecken gezeichnet. Deren Länge, Richtung und Abstände bestimmten die sich ergebende grafische Struktur. Werden nun mehrere Strecken direkt aneinander gefügt, so sprechen wir von *Streckenzügen* (oder Polygonzügen). Endet der Streckenzug an seinem Anfangspunkt, spricht man von einem *geschlossenen Polygonzug* (oder auch Vieleck). Damit ergeben sich erweiterte Gestaltungsmöglichkeiten, die von den Pionieren der Computerkunst ausgiebig genutzt wurden. Bild 4 und Bild 5 sind entsprechende Beispiele. Für Bild 4 - von Nees *achsenparalleler Irrweg* genannt - soll an dieser Stelle der Algorithmus und seine Umsetzung nachgetragen werden.

1. Vor dem Zeichnen des Streckenzugs sind einige Kennwerte zu definieren wie die Breite und Höhe des Bildes (**bild_breite**, **bild_hoehe**), die Begrenzungen in Koordinaten (**xlinks**, **xrechts**, **yunten**, **yoben**) und die Anzahl der Linien (**linienanzahl**).
2. Das eigentliche Zeichnen vollzieht sich in einem eigenen Block **polygonzug**.
3. Darin wird zunächst ein zufälliger Anfangspunkt der Liniengrafik festgelegt (**xanf** und **yanf**) und die Schildkröte dort platziert.

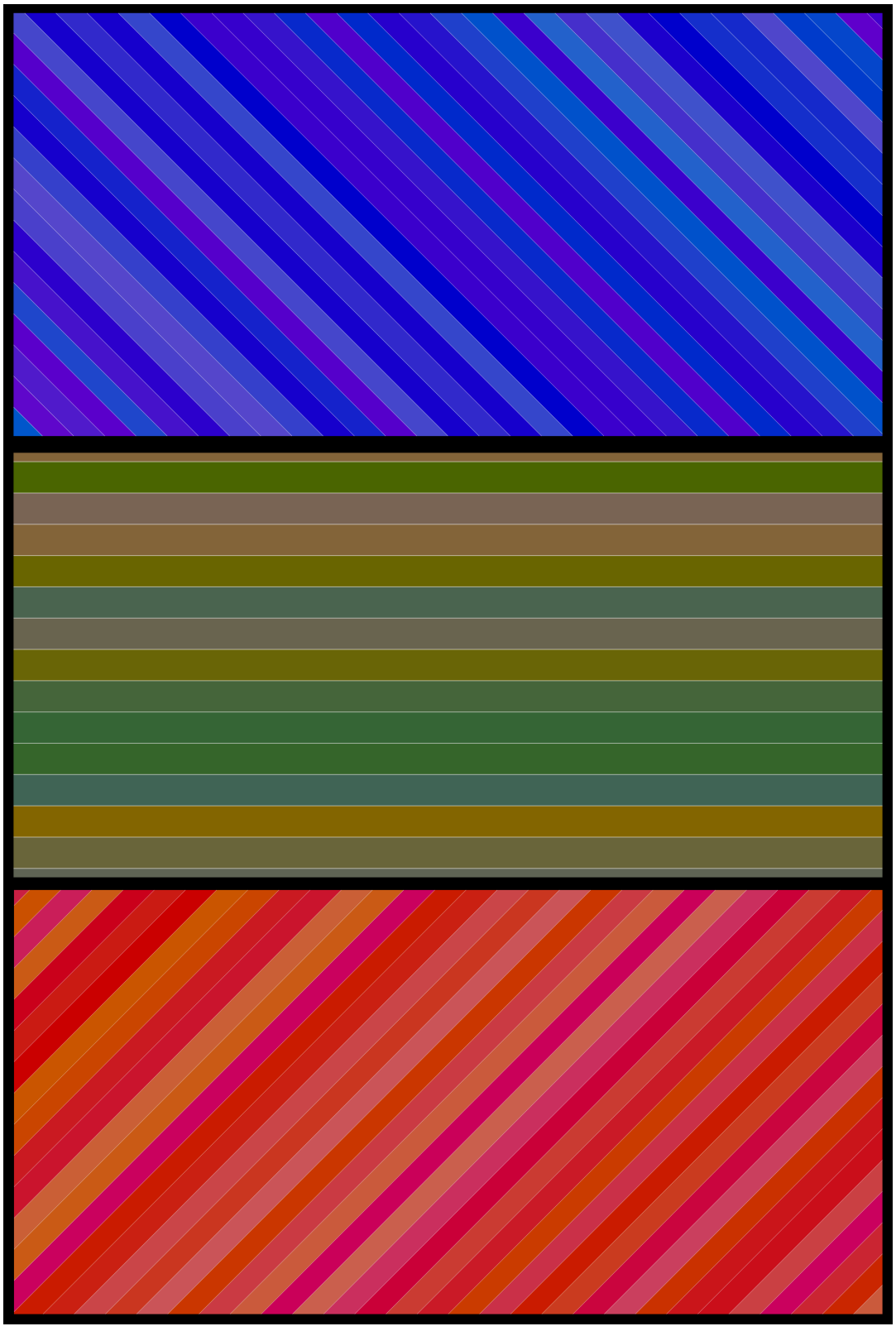
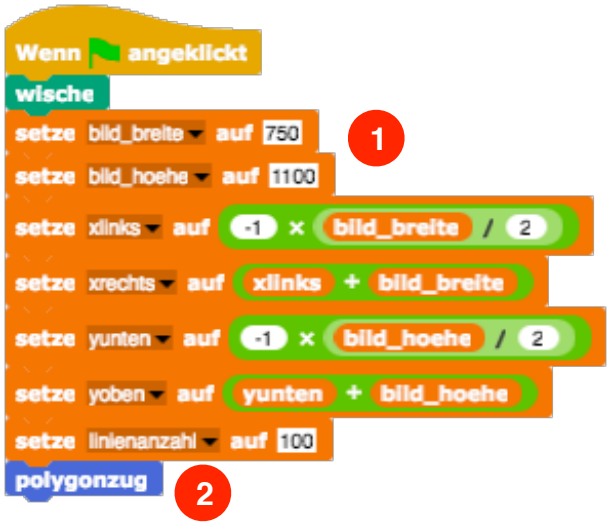
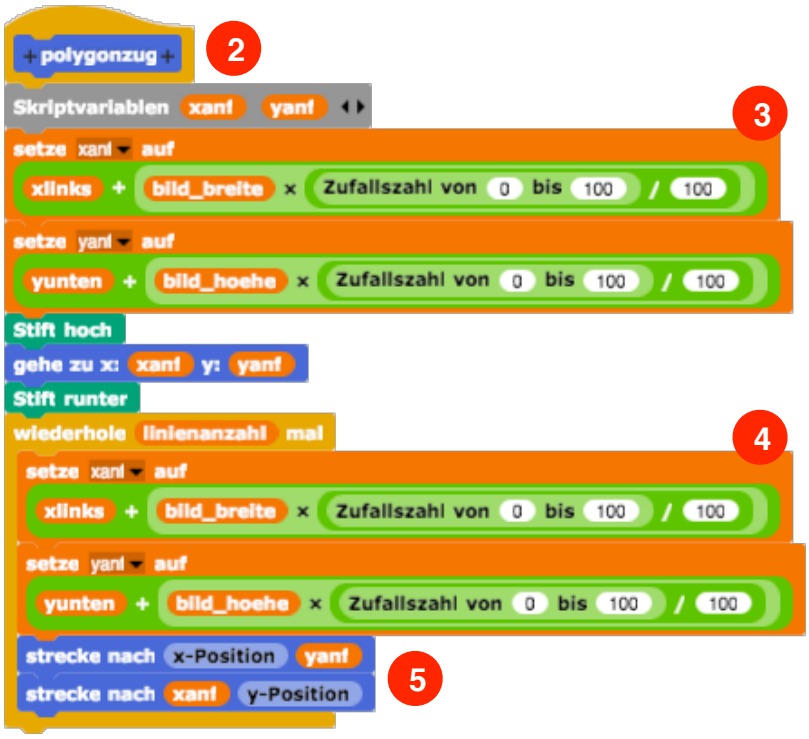


Bild 22: Hommage à LeWitt: Bands of Color

4 Anschließend wird ein neuer zufälliger Endpunkt bestimmt und die Schildkröte in zwei Schritten dorthin bewegt.

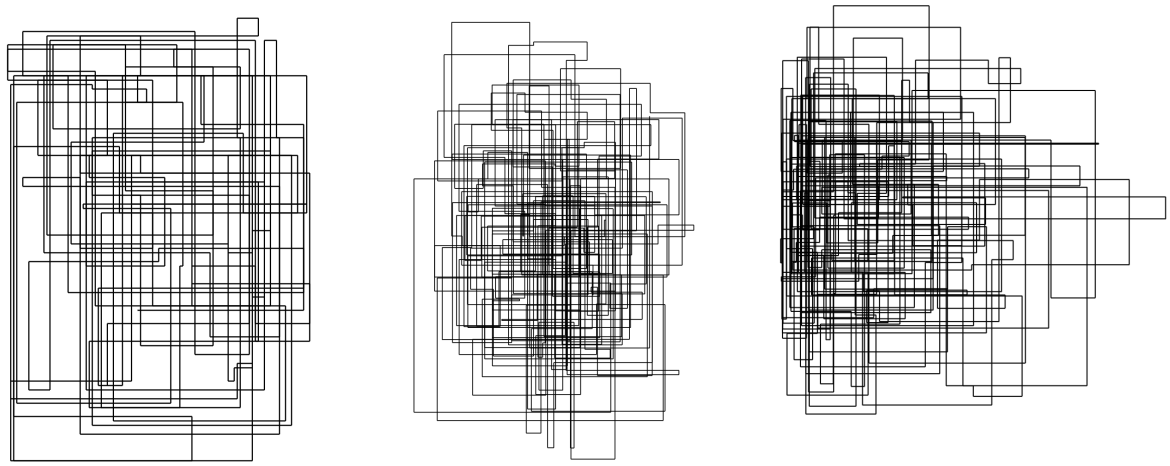
5 Im ersten Schritt geht sie von der aktuellen Position senkrecht zu einem Zwischenpunkt (mit **x-Position** und **yanf**) und von dort horizontal zum Endpunkt (mit **xanf** und **y-Position**). Dieser Zeichenvorgang wird gemäß **linienanzahl** wiederholt.





Bei Verwendung des Zufallszahlengenerators von Snap! und dessen gleichverteilten Zufallszahlen erhalten wir ein zufälliges, aber über die Fläche etwa gleichverteiltes Liniengitter wie in der folgenden Abbildung links. Gegenüber Bild 4 zeigt sich eine Abweichung, weil dort eine Verdichtung der Linien im mittleren Bildbereich deutlich erkennbar ist.

Wenn wir diese Verdichtung auch reproduzieren wollen, brauchen wir einen gewichteten Zufall, wie wir ihn bei den *Komputerstrukturen* von Struycken im Abschnitt *Punkte* bereits kennen gelernt und verwendet haben. Mit der Funktion **zufallszahl dachform** sowohl für **xanf** und **yanf** erhalten wir die gewünschte Verdichtung in der Mitte (Abbildung unten Mitte).



Die Kombination unterschiedlicher Gewichtungen verändert das Aussehen deutlich. Mit der Funktion **zufallszahl gerade** für **xanf** und der Funktion **zufallszahl glockentyp** für **yanf** ergibt sich die Grafik in der Abbildung rechts.

9.8 Hommage à Noll: Gaussian Quadratic

Mit diesem Gerüst ist das *Recoding* eines weiteren Klassikers der frühen Computerkunst sehr leicht zu bewerkstelligen. Bei seinem Bild *Gaussian-Quadratic* (1963) verwendete Michael Noll (von dem auch Bild 5 im Kapitel 1: *Computerkunst* stammt) ebenfalls gewichteten Zufall, denn rein zufällige zweidimensionale Bilder fand er „wenig interessant“ (Noll, 1967).

Bei *Gaussian-Quadratic* benutzte er eine Gauß-Verteilung für die Ermittlung der x-Werte und eine quadratische Erhöhung der y-Werte. In der Konsequenz bedeutet dies eine Konzentration der Punkte auf die Mitte in der Horizontalen und eine kontinuierliche Abstandsvergrößerung der zu verbindenden Punkte in der Vertikalen.

Für die Umsetzung können wir die Kennwerte wieder vorweg festlegen (unter Verwendung des Vorspanns aus dem vorigen Beispiel *achsenparalleler Irrweg*). Der Block **polygonzug** ist dafür neu zu definieren.

1. Dort wird als Erstes **r** definiert, auf das die Zufallszahlengeneratoren zugreifen werden.
2. In der Schleife wird **xanf** mittels **zufallszahl glockentyp** in der gewünschten Verteilung erzeugt.
3. Für **yanf** gilt nach Noll die Formel zaehler*zaehler + faktor*zaehler.
4. Für **faktor** hat Noll 5 vorgegeben.
5. In der **wiederhole**-Schleife wird dann 100 mal eine Linie zu dem ermittelten Punkt mit **xanf** und **yanf** gezogen.

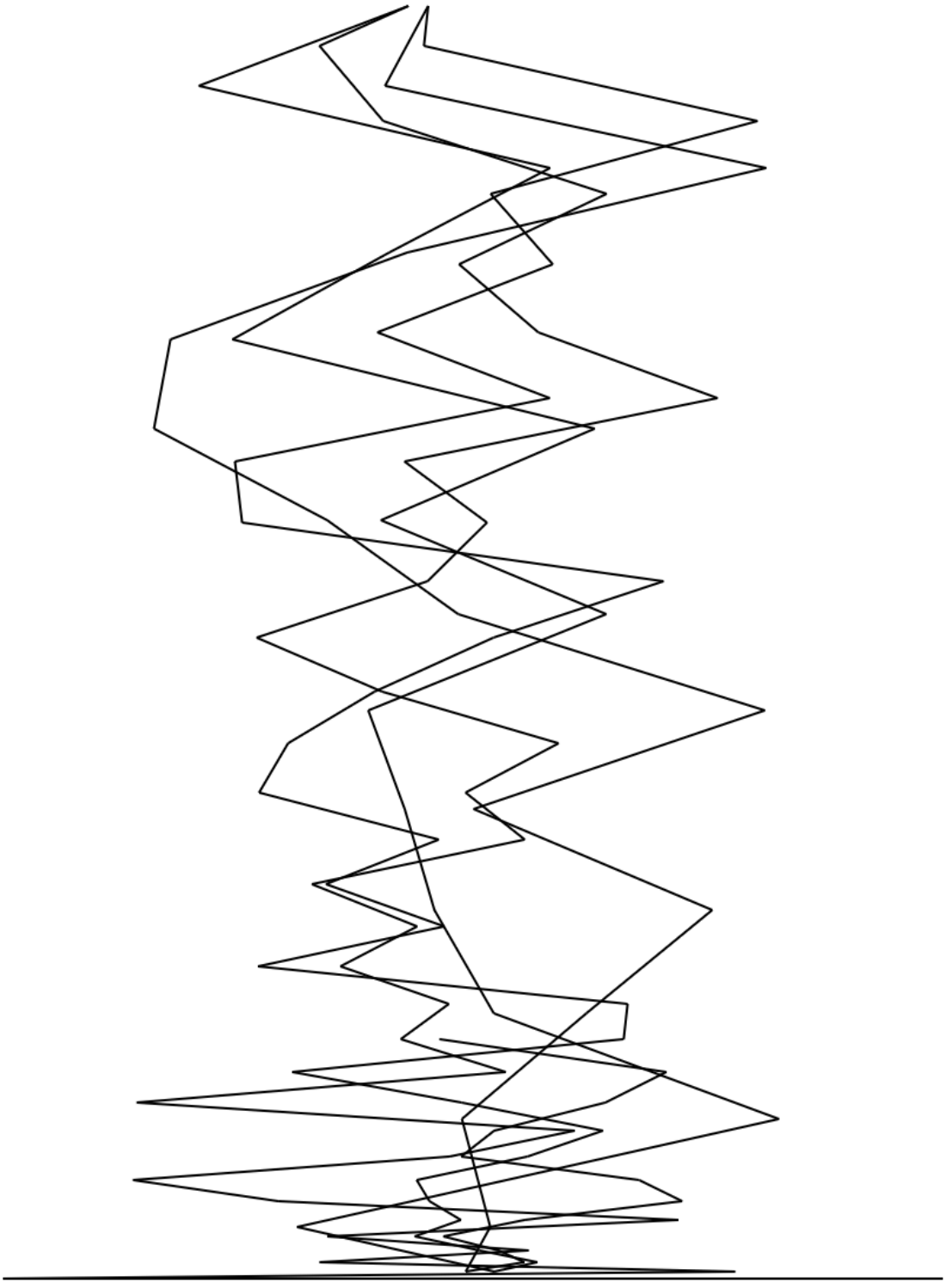
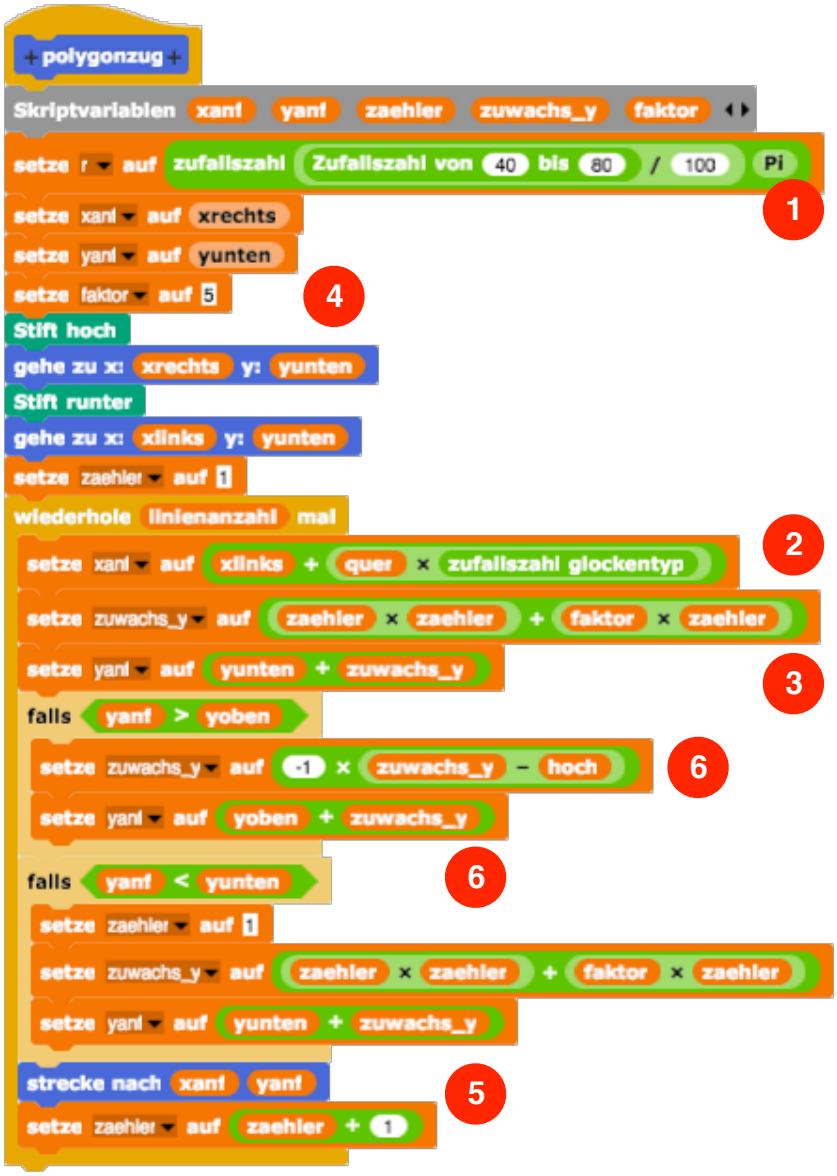


Bild 23: Hommage à Noll: Gaussian-Quadratic

6 Damit die Linien nicht außerhalb der mit **yoben** bzw. **yunten** gesetzten Grenzen hinaus reichen, wird bei Überschreiten der Grenzen der vertikale Zuwachs jeweils umgedreht.



Mit **linienanzahl** = 100 und **faktor** = 5 wird eine Liniengrafik erzeugt, die dem Original von Noll sehr nahe kommt (Bild 23). Sie können beim *Remixing*, hier durch systematisches Ändern dieser beiden Kenngrößen, herausfinden, wie sich das Erscheinungsbild ändert.

9.9 Krumme Linien

Unter den Bildern der frühen Computerkunst finden sich relativ wenige mit krummen Linien, Kreisen oder Kreisbögen. Deren Erzeugung setzt ein paar mathematische Kenntnisse voraus. Wir können dafür aber bereits auf Blöcke aus der Block-Bibliothek zurück greifen, mit denen Kreise (**kreis um x y radius**), Kreisbögen (**kreisbogen von x y radius winkel**) oder Rundbögen (**rundbogen um x y radius winkel**) gezeichnet werden.

Hinweis: Die folgenden Befehle gehören zur Blockbibliothek, die nachgeladen werden kann und in *Anhang C* beschrieben wird.



Mit dem Befehl **kreis um x y radius** zeichnet die Schildkröte einen Kreis um den durch **x** und **y** festgelegten Mittelpunkt mit dem angegebenen **radius**. Die Schildkröte befindet sich nach Zeichnen des Kreises wieder am Ausgangspunkt mit ihrer ursprünglichen Richtung. Der Kreis wird intern von einem Vieleck mit 36 Ecken angenähert. Je größer der Radius, desto gröber wird der Kreisumriss.



Der Befehl **kreisbogen von x y radius winkel** zeichnet einen Kreisbogen (also einen Kreisausschnitt) von dem mit **x** und **y** gegebenen Mittelpunkt mit Radius **r** und dem Winkel **w** in Uhrzeigerrichtung. Die Schildkröte befindet sich nach Zeichnen des Kreisbogens wieder am Ausgangspunkt mit ihrer ursprünglichen Richtung.



Der Befehl **rundbogen um x y radius winkel** zeichnet einen Kreisbogen über dem mit **x** und **y** gegebenen Mittelpunkt mit Radius **r** und dem Winkel **w** in Uhrzeigerrichtung. Er unterscheidet sich vom Kreisbogen also dadurch, dass die Schildkröte vor und nach dem Zeichnen zum Mittelpunkt des gezeichneten Kreisbogens zeigt.

9.10 Hommage à Nees: Kreisbogen

Vorbild für das folgende Projekt ist das Bild *Kreisbogen* (Nees, 1969, S. 136 f.). Statt des Programms für Bild 24 möchte ich die verschiedenen Schritte skizzieren, die zu dem gewünschten Ergebnis führen. Zur Veranschaulichung habe ich dabei Umwege in Kauf genommen.

1. Zu Beginn zeichnen wir in einer Wiederholungsschleife geschlossene Kreise mit **kreis um x y radius**. Deren Mittelpunkte liegen alle auf einer Linie; dafür kann die x-Achse gewählt werden. Der Mittelpunkt wird für jeden Kreis innerhalb eines vorgegebenen Schwankungsbereichs **delta** nach rechts verschoben. Der neue Kreisradius entspricht jeweils der aktuellen **x-Position**.

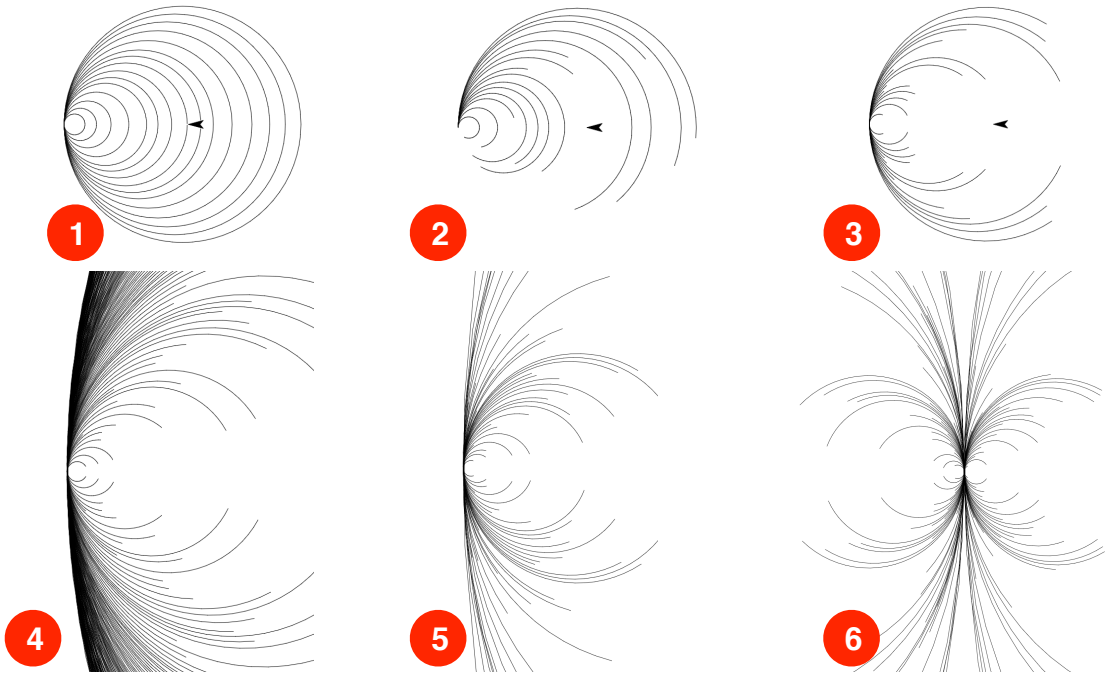
2. Im zweiten Schritt ersetzen wir die Kreise durch Kreisbögen. Die Verwendung von **kreisbogen von x y radius winkel** ist aber noch nicht geeignet, die gewünschte Achsensysmmetrie zu erreichen.

3. Die dafür notwendige Umrechnung wird vom Block **rundbogen um x y radius winkel** mitgeliefert.

4. Wird die Zahl der Rundbögen lediglich weiter linear erhöht (um die Annäherung der Linien an die y-Achse zu erreichen), zeigt sich eine unschöne Verdichtung.

5. Diese kann vermieden werden, wenn ab einem Schwellenwert für die **x-Position** der Zuwachs **delta** deutlich erhöht wird.

6. Mit diesen Einstellungen kann nun diese Vorgehensweise - gespiegelt an der y-Achse - wiederholt werden. Erst damit entsteht das gewünschte Gesamtbild. In dieser zweiten Wiederholungsschleife sind **radius** und **delta** entsprechend anzupassen.



Anregung: *Das Projekt Kreisbogen bietet sich an, mit unterschiedlichen Linien- und Hintergrundfarben Varianten zu erzeugen. Dickere Strichstärken und/oder die zufällige Variation der Kreisbogenfarbe innerhalb der Wiederholungsschleife erzeugen Bilder, die einen ganz anderen bildlichen Eindruck liefern.*

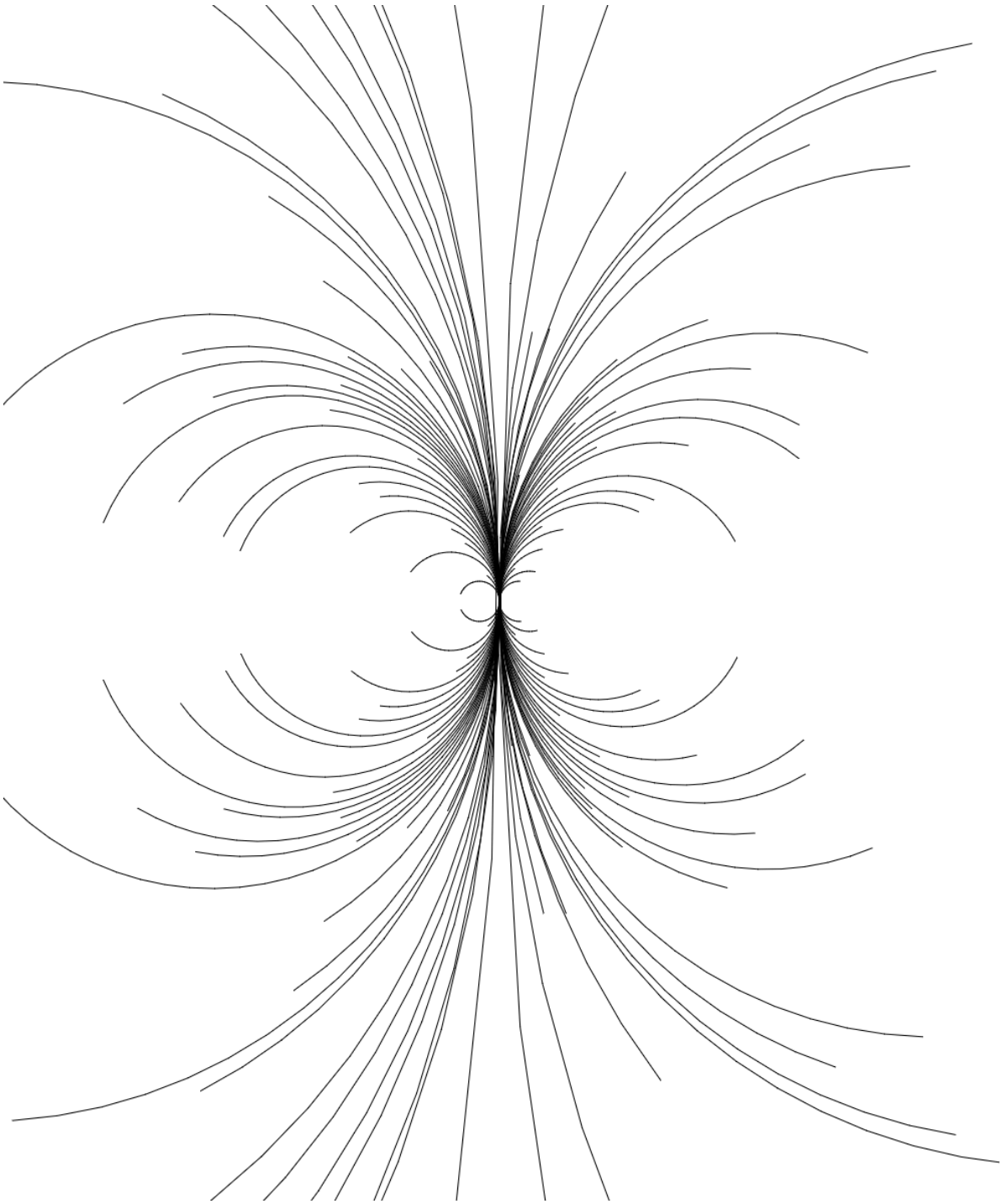


Bild 24: Hommage à Nees: Kreisbogen

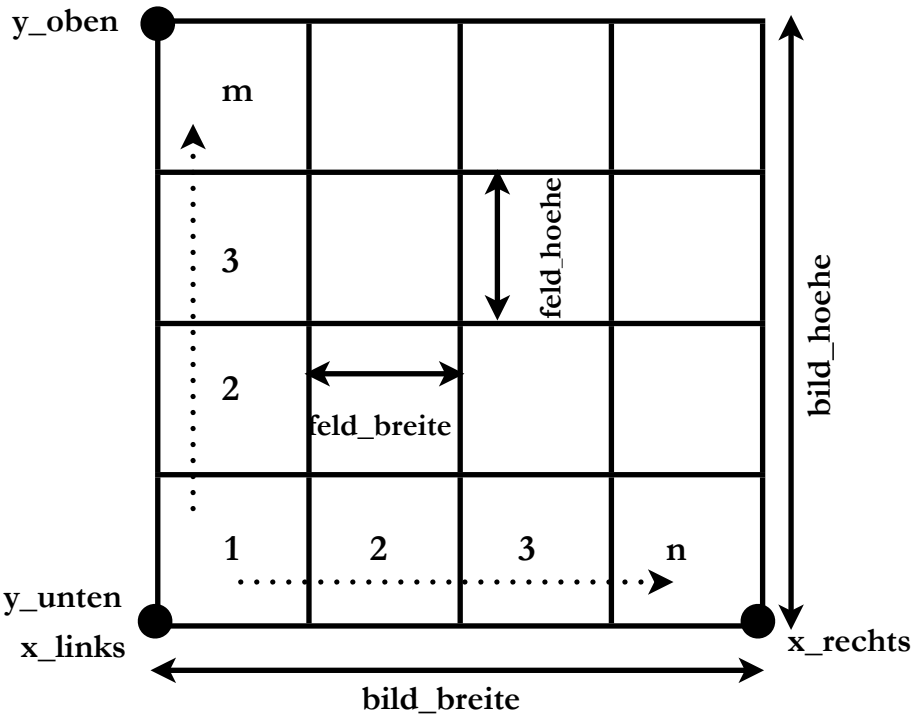
9.11 Flächen

Genau genommen arbeiteten wir bei den in den Abschnitten *Punkte* und *Linien und Strecken* vorgestellten Bildelementen bereits mit Flächen. Wir nehmen sie nur dann als Punkte bzw. Linien wahr, wenn sie relativ klein (Punkt) bzw. dünn (Linie) sind. Nehmen sie an Größe bzw. Dicke zu, haben wir eher den Eindruck von Flächen. Bei Bild 21 und Bild 22 war das bereits feststellbar. Im Extremfall bilden sie aufgrund ihrer Ausdehnung die Grundfläche des Gesamtbildes (in Snap! also der Bühne). Für die Bildgestaltung gewinnen dann weitere Aspekte Bedeutung, wie die Aufteilung der Grundfläche, die Proportionen der Flächenelemente und ihre Anordnung zueinander.

Flächige Elemente sind in der frühen Computerkunst eher selten zu finden; sie sind aber sehr bedeutsam, wenn wir in das *Recoding* und *Remixing* auch Beispiele der Konkreten Kunst oder der Op Art einbeziehen wie im Teil III: *Kunst Programmieren ?!*. Das Beispiel in diesem Abschnitt für die Bildgestaltung mit flächigen Elementen ist das Bild 25 nach 25 Quadrate von Vera Molnar. Zuvor bauen wir dafür eine Arbeitserleichterung.

9.12 Exkurs: Das m*n-Raster

Ziel ist es, ein Raster jeweils gleich großer Felder in n Spalten und m Reihen festzulegen. In die Felder können dann beliebige Figuren gezeichnet werden, die auf die Größe des Feldes normierbar sind. Notwendig ist dafür die Festlegung einer Reihe von Kenngrößen, durch die das Raster und die Felder definiert werden. Damit können dann viele Bilder mit sich wiederholenden Mustern leicht nach dem immer gleichen Schema erstellt werden:

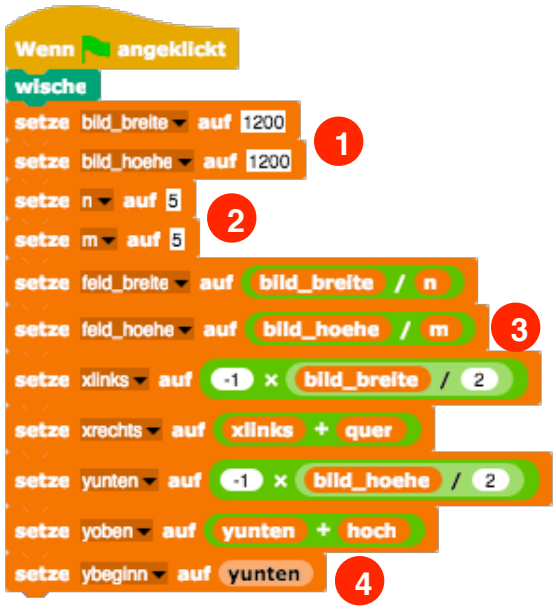


bild_breite: Gesamtbreite des Bildes in Pixeln
bild_hoehe: Gesamthöhe des Bildes in Pixeln
n: Anzahl der Spalten
m: Anzahl der Reihen
feld_breite: Breite eines einzelnen Feldes in Pixeln
feld_hoehe: Höhe eines einzelnen Feldes in Pixeln
xlinks: horizontaler Startpunkt in Bildschirmkoordinaten
xrechts: horizontaler Endpunkt in Bildschirmkoordinaten
yunten: vertikaler Startpunkt in Bildschirmkoordinaten
yoben: vertikaler Endpunkt in Bildschirmkoordinaten

Dieses m*n-Raster kommt in den folgenden Projekten gleich mehrfach zur Anwendung.

9.13 Hommage à Molnar: 25 Quadrate

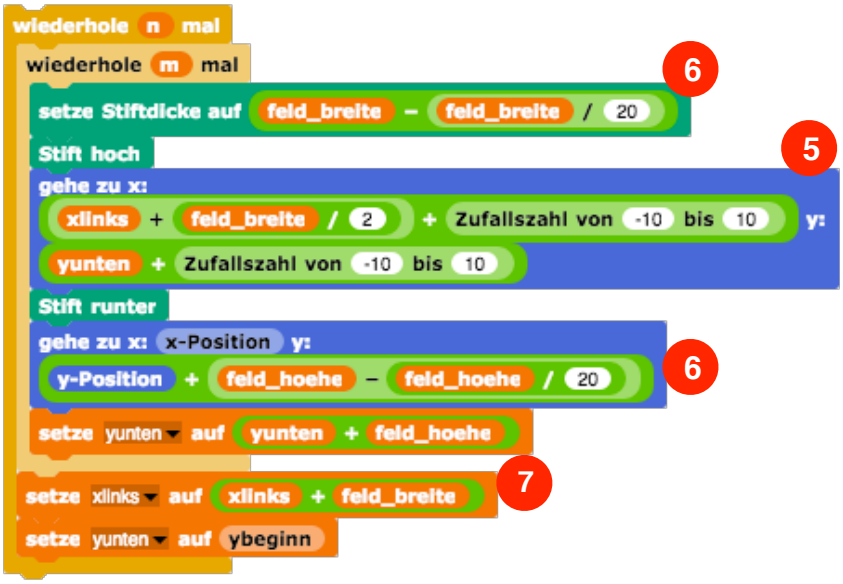
Für die Umsetzung der *25 Quadrate* nach Molnar verwenden wir nun das m*n-Raster im Vorspann:



1. Die Bildgröße wird auf 1200 Bildpunkte festgelegt.
2. Für die 25 Felder werden je fünf Reihen und Spalten benötigt.
3. Daraus werden die Breite und Höhe der Felder sowie die Start- und Endpunkte errechnet.
4. In diesem Fall wird noch eine Merkgröße **ybeginn** festgelegt, damit bei jeder Spalte wieder an der Ausgangshöhe begonnen wird.

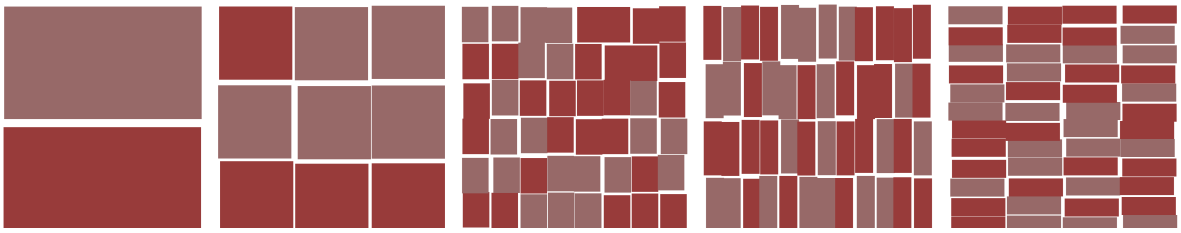
Der Vorteil ist, dass nun innerhalb eines jeden Feldes beliebige Figuren gezeichnet werden können, in unserem Beispiel also die Quadrate, durchgeführt in einer Doppelschleife für **n** Spalten und **m** Reihen:

5. Dazu wird die Schildkröte mittig an den unteren Rand des Felds gesetzt (hier mit einer Zufallsabweichung)
6. und eine Linie gezogen mit einer Dicke von annähernd der Quadratseite.
7. Am Ende beider Schleifen werden jeweils die Startwerte angepasst bzw. zurück gesetzt.



Die mit dem m*n-Raster gewonnene Flexibiliät soll am aktuellen Beispiel noch einmal verdeutlicht werden. Denn allein die Variation der Kenngrößen 1 und 2 lässt nun die Erzeugung ganz unterschiedlicher Bilder und Bildeindrücke zu.

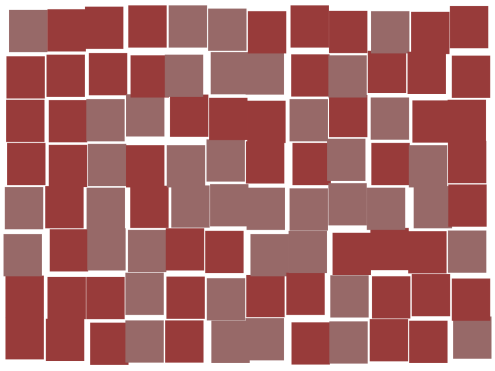
Die folgenden Bilder zeigen bei gegebener **bild_breite** und **bild_hoehe** eine unterschiedliche Zahl **n** an Spalten und **m** an Reihen:



n = 1, m = 2 *n = 3, m = 3* *n = 8, m = 6* *n = 12, m = 4* *n = 4, m = 12*

Der Quadratcharakter der Rechtecke in den Feldern geht verloren wenn für **n** und **m** ungleiche Werte gewählt werden.

Die Veränderung der Bildgröße durch **bild_breite** und **bild_hoehe** liefert wieder einen anderen Bildeindruck:



bild_breite = 1500, bild_hoehe = 1100,
n = 12, m = 8

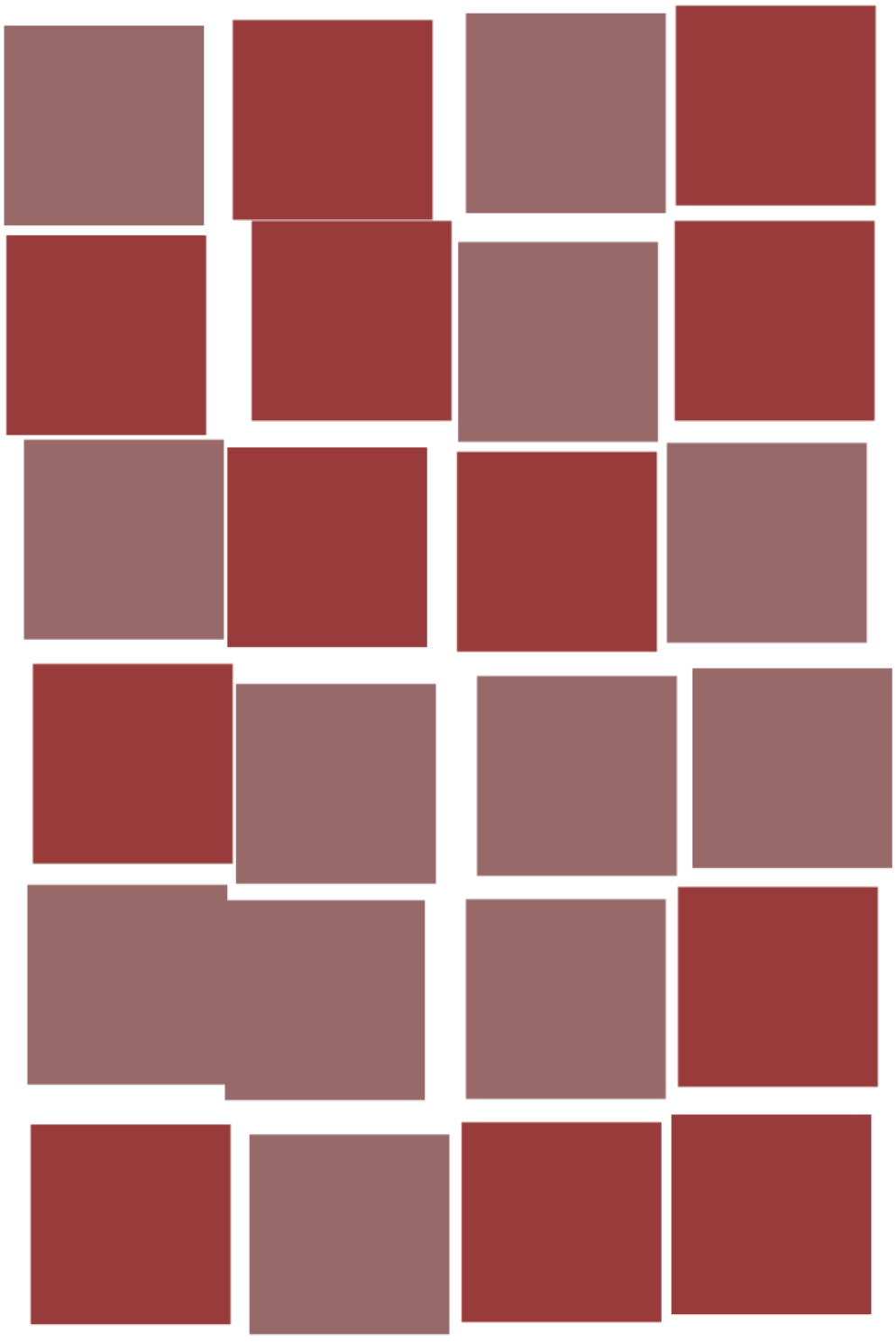
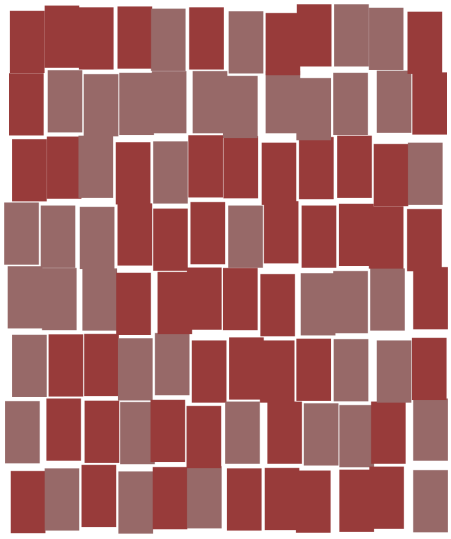


Bild 25: Hommage à Molnar: 25 Quadrate; Remixing als 6*4 Quadrate

Anregung: *Trotz dieser Variabilität sollte nicht vergessen werden, dass der Eindruck des Bildes bei Molnar mehr durch die Farbschattierung als durch Zahl und Anordnung der Quadrate stark geprägt ist. Sie können auch diese leicht ändern. Für* Bild 25 *habe ich das* HSV-Modell *gewählt.*

Die Farbunterschiede der zwei Farbvarianten beruhen auf verschiedenen Sättigungswerten s *bei konstanten Farbwerten* h*, die in der inneren Schleife zufällig ausgewählt werden. Sie können natürlich mit der Variation von Farbton, Sättigung und Helligkeit weiter experimentieren.*





bild_breite = 1000, bild_hoehe = 1200, n = 12, m = 8

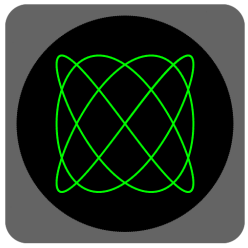
Zwischenfazit: Mit diesem Kapitel ist die Einführung in die Sprachelemente von Snap! nun weitgehend abgeschlossen. Sie werden in den weiteren Kapiteln nur an wenigen Stellen ergänzt, wenn spezielle Anforderungen dies nahelegen. Mit dem Aufbau des Figurenbaukastens und dessen Anwendung in ersten Projekten ist die Grundlage geschaffen, nun die Werke und Personen der frühen Computerkunst noch mehr in den Mittelpunkt zu stellen.

TEIL II: RECODING & REMIXING COMPUTERKUNST

10. VOM ANALOGEN ZUM DIGITALEN

Es ist heute fast in Vergessenheit geraten, dass seit den 50er-Jahren bereits sogenannte elektronische Analogcomputer[50] eine gewisse Verbreitung hatten. Bei ihnen werden kontinuierliche (d.h. analoge) Größen verwendet, die von elektronischen Bauelementen wie beispielsweise von Widerständen, Dioden oder Transistoren erzeugt und miteinander verknüpft werden können. Prinzipbedingt arbeiten die Bauteile parallel und deshalb außerordentlich schnell. In Simulationen - eines der Haupteinsatzgebiete ist das Lösen von Differentialgleichungen - kann deshalb durch einfaches Drehen an Potentiometern unmittelbar das Verhalten bestimmter Komponenten beeinflusst werden und es ergeben sich daraus sofort die so erzeugten Ergebnisse. Dargestellt wurden sie meist auf Oszilloskopen, typischerweise grüne Kurven auf schwarzem Untergrund.

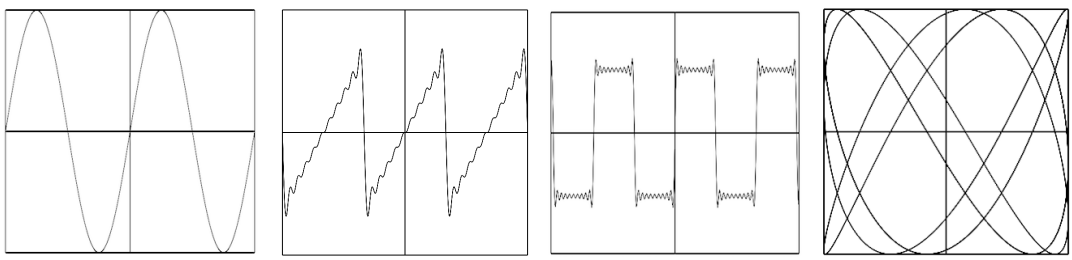
Der zwangsläufig stark technisch geprägte Zugang mit dem dafür notwendigen Aufbau von Schaltungen und der limitierten grafischen Ausgabe auf kleinen, meist runden Bildröhren, hinderte einige experimentierfreudige Pioniere nicht daran, sie für künstlerische Arbeiten zu nutzen.



10.1 Hommage à Laposky

Der Mathematiker und Ingenieur Ben F. Laposky (1914 - 2000) benutzte sein Oszilloskop, das er um Sinuswellengeneratoren und weitere elektronische Komponenten erweiterte, um damit Wellenformen zu erzeugen, abzufotografieren und die Bilder 1952 in einer Galerie auszustellen, später auch in einer Wanderausstellung in vielen Städten der USA. Mit Hilfe von Filtern konnte er sogar Farbbilder erzeugen. In seinem Ausstellungskatalog *electronic abstractions* (Laposky, 1953) bezeichnet er die Bilder als *Oscillons*.

Laposky zeigte in seiner Ausstellung bzw. in seinem Ausstellungskatalog, mit welchen Wellenformen und ihrer Kombination er seine Bilder erzeugte: Sinus-, Sägezahn- und Treppenfunktionen sowie Lissajous-Figuren:



[50] Wer sich für deren Technik, Anwendungen und Beispiele interessiert, findet auf der Website http://www.analogmuseum.org/ (gepflegt von Prof. Dr. Bernd Ulmann) umfangreiche und fundierte Informationen.

Auch Herbert W. Franke stellte mit analoger Technik Linienüberlagerungen her, die er fotografisch festhielt und *Lichtformen* nannte. Ähnlich arbeitete später Otto Beckmann (1908 - 1997), der von der Bildhauerei gekommen war, mit analogen Rechenwerken, dem *Ateliercomputer*[51], der von seinem Sohn speziell für seine künstlerischen Vorstellungen gebaut wurde.

Wir werden uns beim *Recoding* auf die ursprünglichen Lissajous-Figuren konzentrieren.

10.2 Recoding Lissajous

Bei Lissajous-Figuren handelt es sich um die Überlagerung zweier harmonischer Schwingungen, deren Richtungen senkrecht zueinander stehen. Mathematisch werden sie durch Sinusfunktionen beschrieben, deren Parameter ihr Aussehen bestimmen.

Ein Gleichungs-Duo beschreibt die bekannten Lissajous-Figuren, benannt nach dem französischen Physiker Jules Antoine Lissajous (1822 - 1880). Ganz ohne Mathematik geht es an dieser Stelle deshalb nicht (vgl. z.B. Bohnacker et al., S. 348 ff.)[52]:

$$x(t) = A_x \sin(\Omega_x \cdot t + \phi_x)$$

$$y(t) = Ay \sin(\Omega y \cdot t + \phi y)$$

Die verhaltensbestimmenden Parameter sind die jeweilige Amplitude A, die Kreisfrequenz Ω und die Phase ϕ. Grafisch dargestellt wird die Zuordnung der Werte einer Funktion (hier x) zu den Werten einer zweiten Funktion (hier y). Der resultierende Graph wird XY-Diagramm genannt.

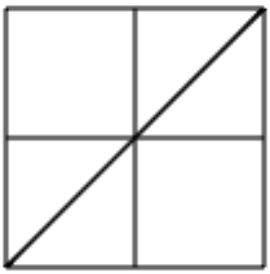
Bei den Lissajous-Figuren gibt es für bestimmte Frequenzverhältnisse typische Figuren, die für das Verständnis komplizierterer Parameterkombinationen[53] hilfreich sind. Um die basalen Eigenschaften vorzustellen, die sich in davon abgeleiteten komplexeren Grafiken später immer wieder finden lassen, sind in der oberen Reihe der folgenden Abbildung die Grundformen für mehrere Frequenzverhältnisse, in der unteren Reihe dieselben Grundformen mit unterschiedlichen Phasendifferenzen dargestellt:

---

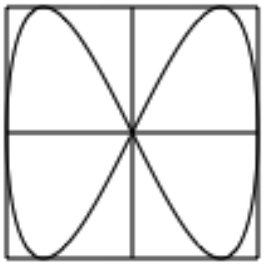
[51] Beckmanns Ateliercomputer: http://www.archiv-otto-beckmann.com/ateliercomputer.html

[52] Zur Beschreibung der Lissajous-Figuren sind unterschiedliche Gleichungspaare zu finden. Die hier verwendeten Sinusfunktionen sind die gängige Darstellungsform. In Wolframs Mathworld sind es zwei Cosinusfunktionen (http://mathworld.wolfram.com/LissajousCurve.html). Beide Formen können ineinander überführt werden, da die Cosinusfunktion auch als phasenverschobene Sinusfunktion ($\pi/2=90^o$) interpretiert werden kann.

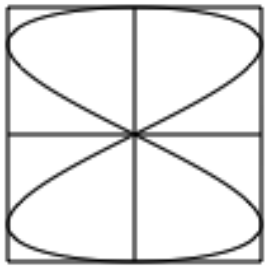
[53] Eine entsprechende Übersicht bietet: http://de.wikipedia.org/wiki/Lissajous-Figur



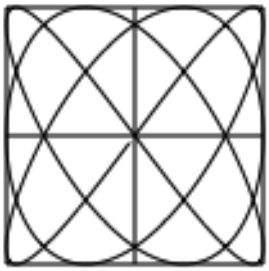
Frequenzen 1:1



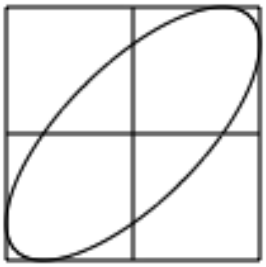
Frequenzen 1:2



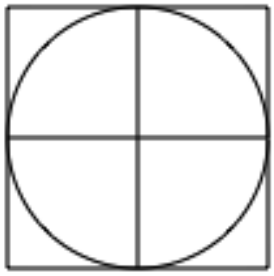
Frequenzen 2:1



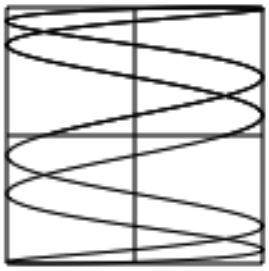
Frequenzen 3:4



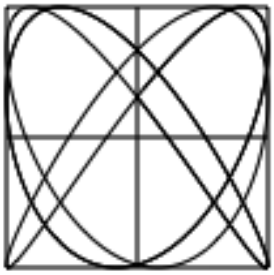
Frequenzen 1:1
Phase y π/4



Frequenzen 1:1
Phase y π/2



Frequenzen 5:1
Phase x π/4

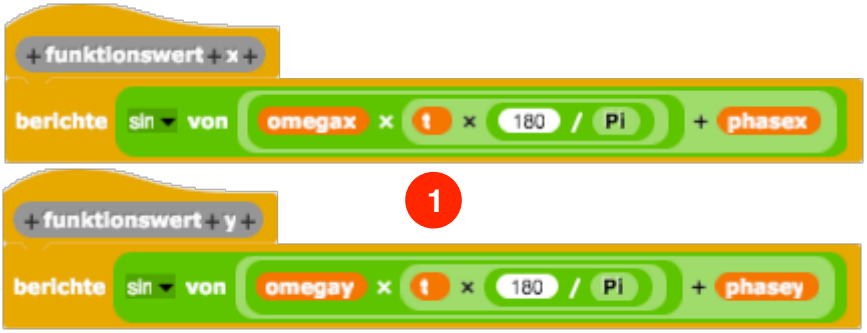


Frequenzen 3:4
Phase y 9π/10

Die Umsetzung in Snap! werden wir in drei Schritten erarbeiten (damit alle Werte an zentraler Stelle variiert werden können, verwenden wir überwiegend globale Variablen):

xstart, ystart: Anfangswerte von x und y für Startpunkt des Zeichenvorgangs
PI: Kreiszahl π als Konstante mit dem Wert 3,14159
omegax, omegay: Kreisfrequenzen
phasex, phasey: Phasen
deltax, delty: Schrittweiten zwischen errechneten Bildpunkten
t: aktueller Wert von t

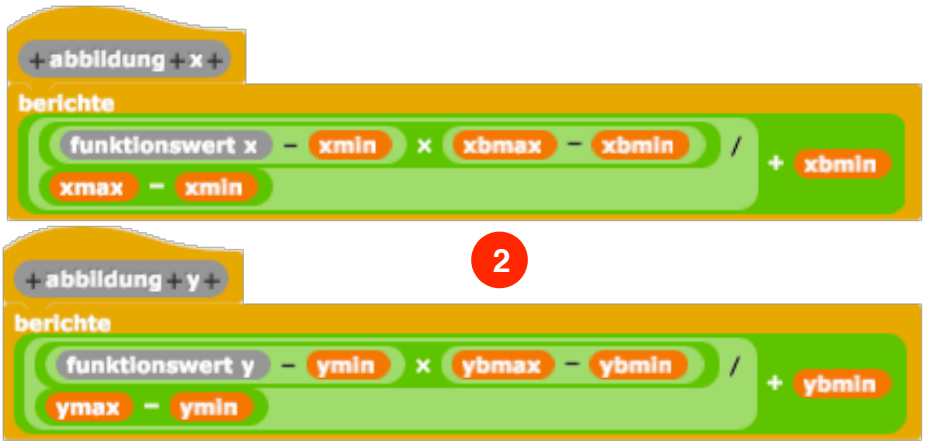
1 Der erste Schritt ist die Formulierung der beiden Gleichungen. Die Reporter, die die Funktionswerte in horizontaler (**funktionswertx**) und vertikaler (**funktionswerty**) Richtung liefern, können z.B. aussehen wie folgt (sie enthalten bereits die Umrechnung des Arguments vom Bogenmaß in Grad):



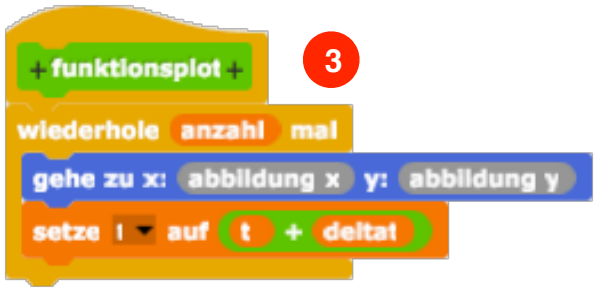
2 Für die grafische Darstellung dieser Gleichungen verlassen wir im zweiten Schritt die eigentliche Schildkrötengrafik und arbeiten im kartesischen Koordinatensystem (vgl. *Orientierung auf der Bühne*). Die oben gezeigten Funktionsgleichungen

liefern uns Werte - gewissermaßen die Weltkoordinaten - die dann jeweils in die entsprechenden Bildschirmkoordinaten umzurechnen sind. Konkret sollen dabei die von uns gewünschten minimalen (**xmin, ymin)** bzw. maximalen (**xmax, y-max**) Weltkoordinaten in die minimalen (**xbmin, ybmin**) bzw. maximalen (**xbmax, ybmax**) Bildschirmkoordinaten abgebildet werden. Erreicht wird dies mit Hilfe einer zentrischen Streckung: Der dritte Strahlensatz (vgl. z.B. Ziegenbalg, 1986, S. 115 ff.) liefert uns die gewünschten Bildschirmkoordinaten.

In der Snap!-Syntax formulieren wir dazu den Aufruf der entsprechenden Reporter:



3 Im dritten Schritt definieren wir schließlich noch die Schleife, in der die gewünschte **anzahl** Bildpunkte errechnet und gezeichnet wird (im Block **funktionsplot**). Zu beachten ist dabei, dass über **deltat** die Schrittweite der Berechnung gesteuert werden kann. Das bedeutet kurz gesagt, je kleiner die Schrittweite, desto genauer ist die Berechnung (hohe Genauigkeit ergibt die oben gezeigten Grundformen).

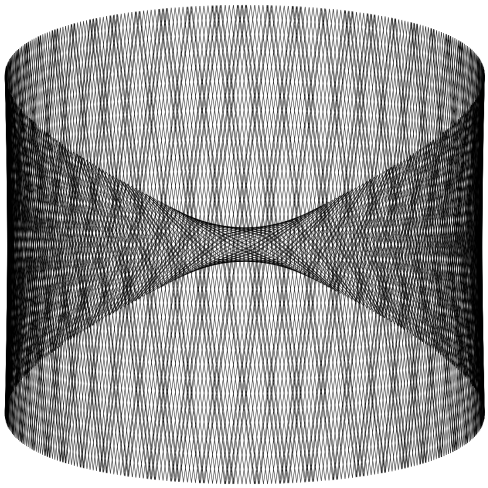




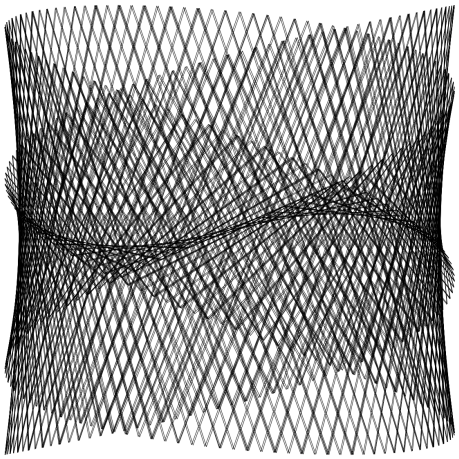


Liefern den Sinus- bzw. Cosinuswert des angegebenen Winkels.

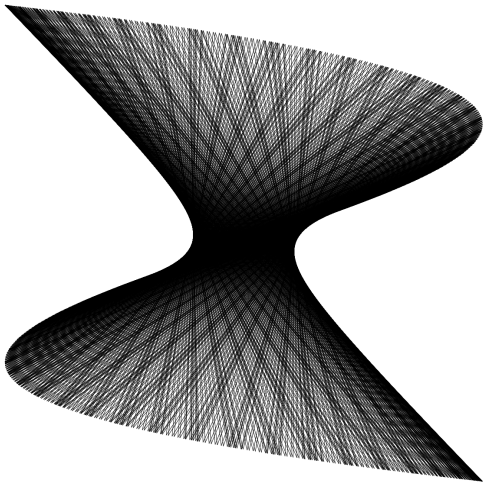
Hinweis: Es ist eine Eigenschaft von Snap!, dass bei allen trigonometrischen Funktionen als Argument der Winkel in Grad anzugeben ist (d.h. z.B. `sin 90` liefert den Wert 1, `cos 180` den Wert -1). Sehr häufig wird aber auch der Winkel im Bogenmaß verwendet (so definiert würde dann also `sin Pi/2` den Wert 1 liefern, `cos Pi` den Wert -1). Wenn wir auch in Snap! diese Werte verwenden wollen, muss in der entsprechenden Prozedur die entsprechende Umrechnung des Arguments `:t` von Bogenmaß in Grad vorangestellt werden: `:t*180/Pi.`



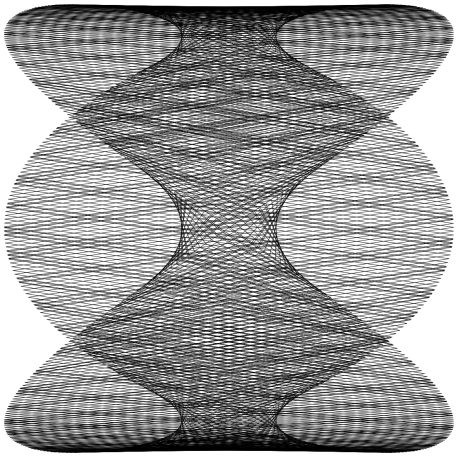
Ω_x = 6, Ωy = 3, deltat = 1, anzahl 1000



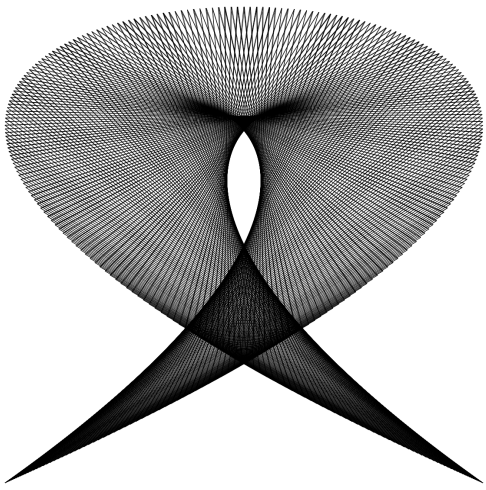
Ω_x = 7, Ωy = 3, d deltat = 1, anzahl 1000



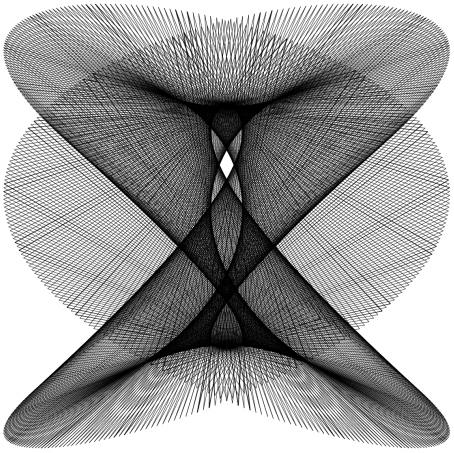
Ω_x = 9, Ωy = 3, deltat = 1, anzahl 1000



Ω_x = 9, Ωy = 6, deltat = 1, anzahl 1000)



Ω_x = 3, Ωy = 2, ϕ_x = 0, ϕy = π/2, deltat = 1, anzahl 800



Ω_x = 3, Ωy = 2, ϕ_x = 10, ϕy = π/2, deltat = 1, anzahl 800

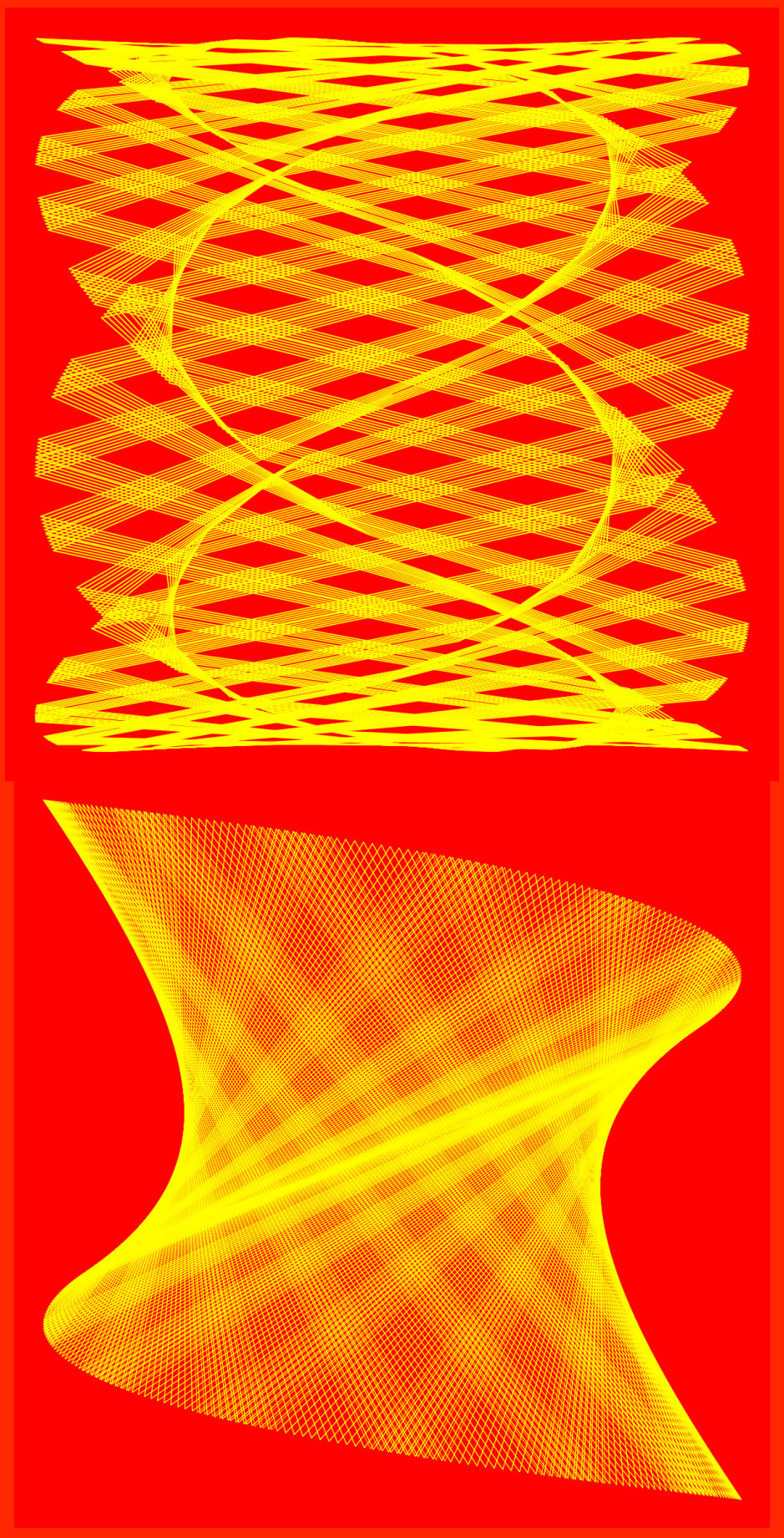


Bild 26: Lissajous farbig I & II

Alle genannten Kennwerte können Sie nun (mehr oder weniger) systematisch variieren. Bei hoher Genauigkeit (also kleinen **delta***-Werten) können Sie sich das grundsätzliche Verhalten der Lissajous-Figuren erschließen.*

Laposky und Franke haben viele ihre Bilder durch Variation der Frequenz- und Phasenverhältnisse erhalten. Das kann hier leicht nachvollzogen werden (siehe die vorangegangene Bildserie).

Ein weiterer Schritt zu ästhetischen Objekten, wie sie vielfach in der Literatur zu finden sind, ist die Vergrößerung der Schrittweiten.

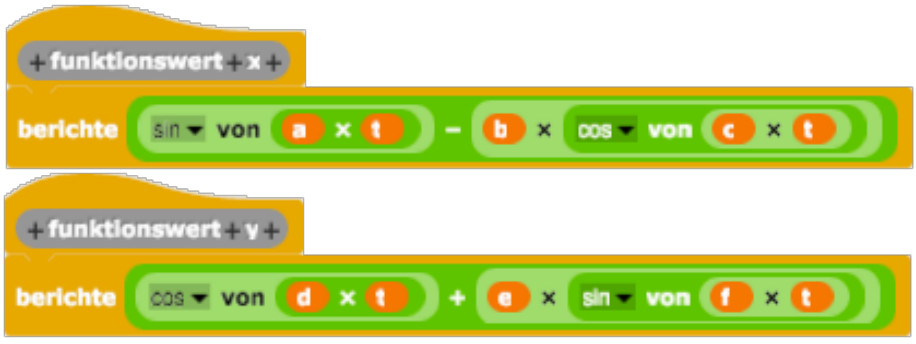
Anregung: *Neben den eigentlichen Kenndaten der Sinusschwingungen bietet vor allem die Variation der Schrittweite (quasi die „Dichte" der durch Linien verbundenen Funktionspunkte) ein unerschöpfliches Reservoir an (oft überraschenden) Ergebnissen. Sie können die Wirkung dieses Effektes erkennen, wenn Sie die delta-Werte langsam steigern. Interessante Effekte erhalten Sie, wenn Sie das für x und y asynchron ermöglichen. Versuchen Sie doch, die hier gezeigten Beispiele nachzuempfinden!*

Einfache Variationen der klassischen Lissajous-Figuren sind auch durch Änderungen der Stiftfarbe *und/oder der* Stiftdicke *möglich. Allein dadurch ergeben sich neue Bildeindrücke.*

Ganz im Sinne von Georg Nees ist es möglich und gilt es, aus der Vielfalt der Ergebnisse unserer *ästhetischen Werkstatt* (Nees, 1995, S. 7) die prägnanten Funde auszuwählen und zu dokumentieren (ein Beispiel dafür liefert Bild 26).

10.3 Remixing Lissajous (I): Sinus-Cosinus-Additionen

Eine mögliche Erweiterung der Darstellung von Lissajous-Figuren ergibt die Addition weiterer Komponenten, d.h. die Überlagerung zusätzlicher Schwingungen[54]. Ein Beispiel als Ausgangspunkt schöner Variationen ist das *Maschennetz*, das Dewdney (1988) vorgestellt hat (die Darstellung erfolgt hier gleich mit Snap!-Blocks):



Mit den von Dewdney verwendeten Parametern ergibt sich ein Ausgangsbild (siehe Abbildung nach Bild 28), das laut seiner Aussage „einer glücklichen Wahl von Konstanten" entsprungen ist (a = 0.99, b = 0.7, c = 3.01, d = 1.01, e = 0.1, f = 15.03, bei Schrittweite 0.5 und 50.000 Punkten).

[54] Ein Einstieg in die Welt der Funktionen (Sinus, Cosinus u.a.) findet sich gut verständlich bei Haftendorn (2010, S. 117 ff.). Für unsere Zwecke besonders nützlich ist ihr „Funktionenbauhof" mit Summe, Produkt und Verkettung von Funktionen (a.a.O., S. 147 ff.).

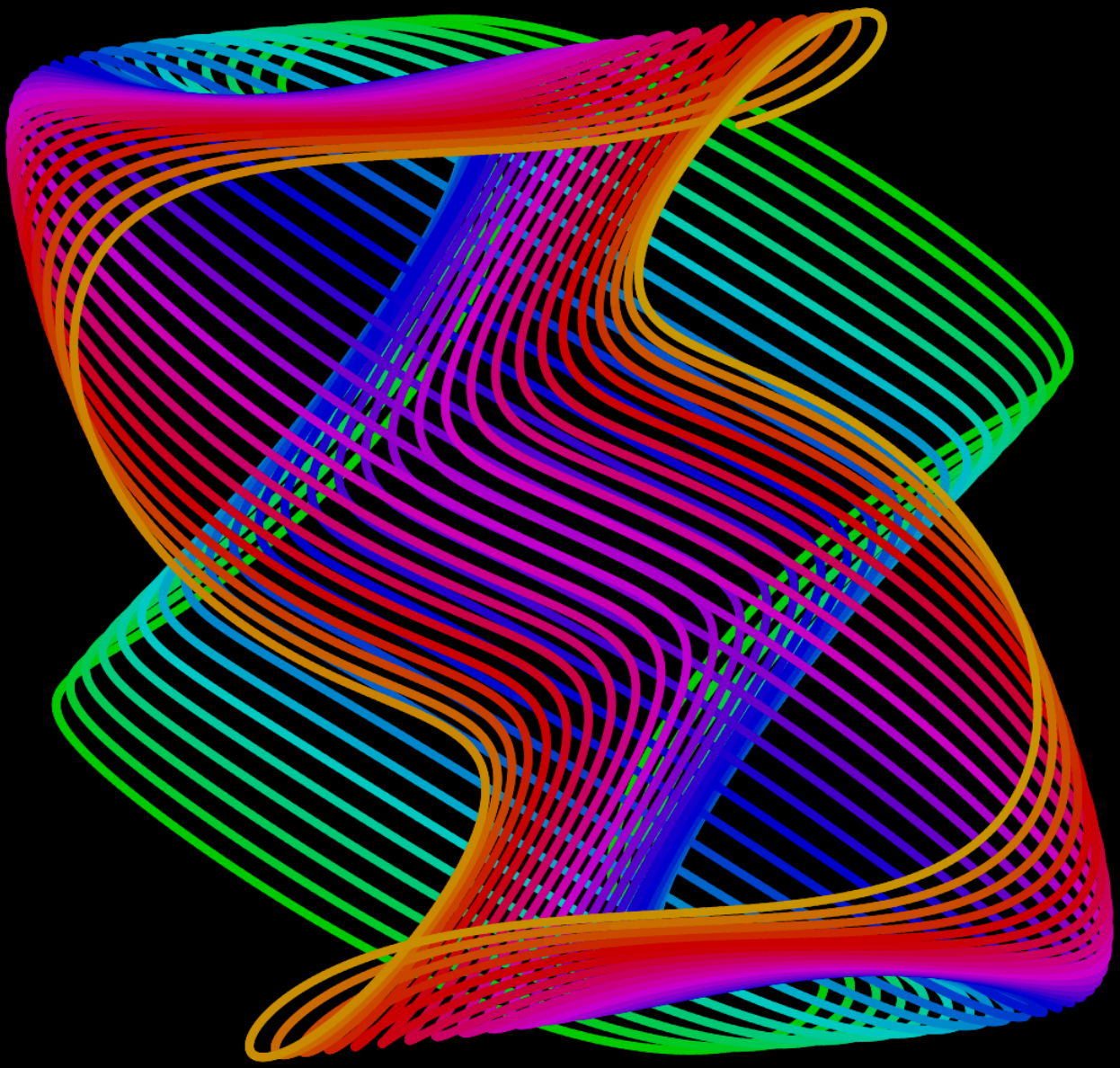
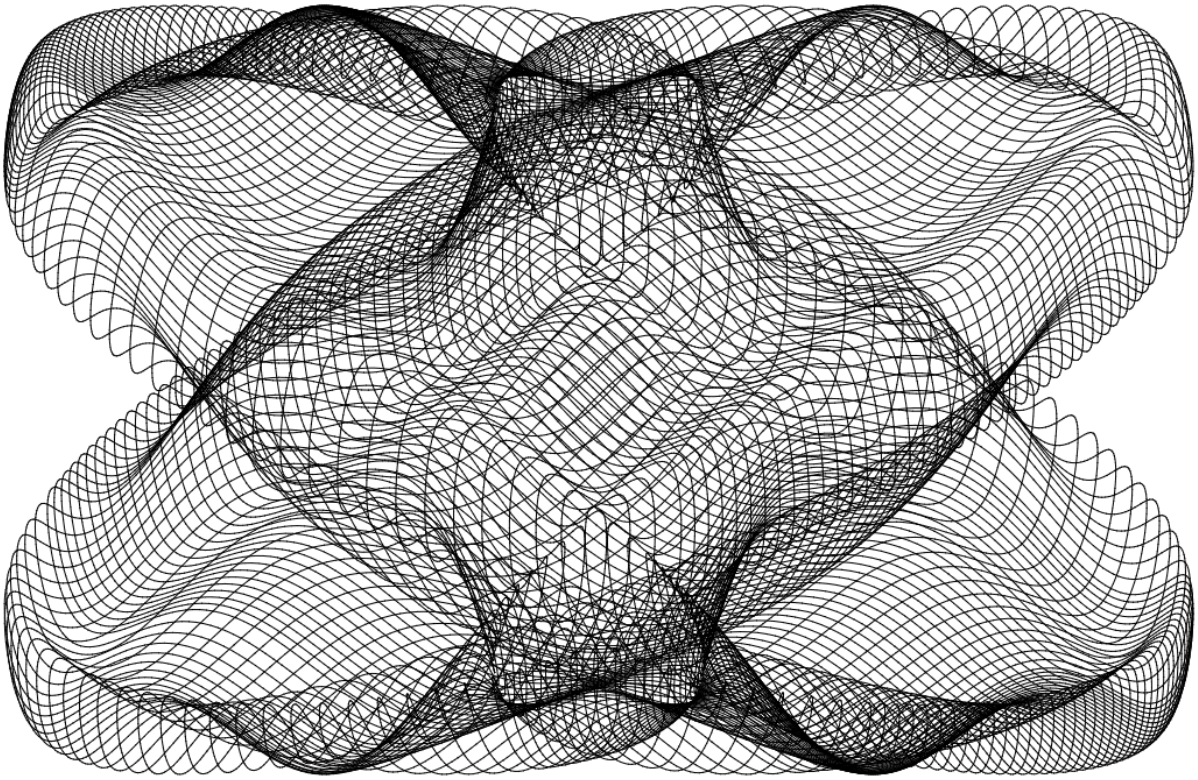


Bild 27: Maschennetz farbig



Bild 28: Sinus-Cosinus-Addition I & II



Neben der Variation der Parameter a - e kann auch hier wieder mit Stiftfarbe und Stiftdicke gearbeitet werden (ein Ergebnis zeigt Bild 27 mit f = 5.03, Schrittweite 0.5 und 20.000 Punkten, bei f = 5.03, Ausgangsfarbe (30,200,30), Farbänderung je Schritt 0.004, ansonsten unverändert).

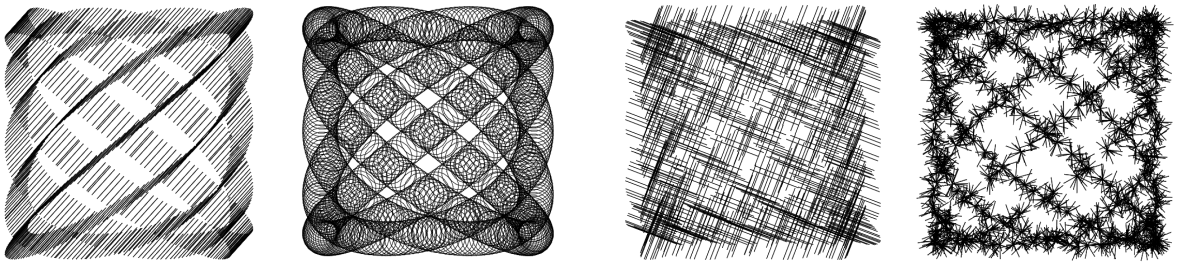
Bei McKenna (2011, 295 ff.) finden sich ähnliche Lissajous-Figuren, die sich jeweils wieder nur durch die gewählten Parameter unterscheiden. Diese waren für mich Ausgangspunkt für weitere Experimente mit Farben und Strichdicke, die u.a. zu der Grafik in Bild 28 geführt haben. In den Beispielen wurden die Verbindungsstücke zwischen den Funktionspunkten durch verdickte Farbsegmente bzw. Farbpunkte ersetzt.

10.4 Remixing Lissajous (II): Anhängsel

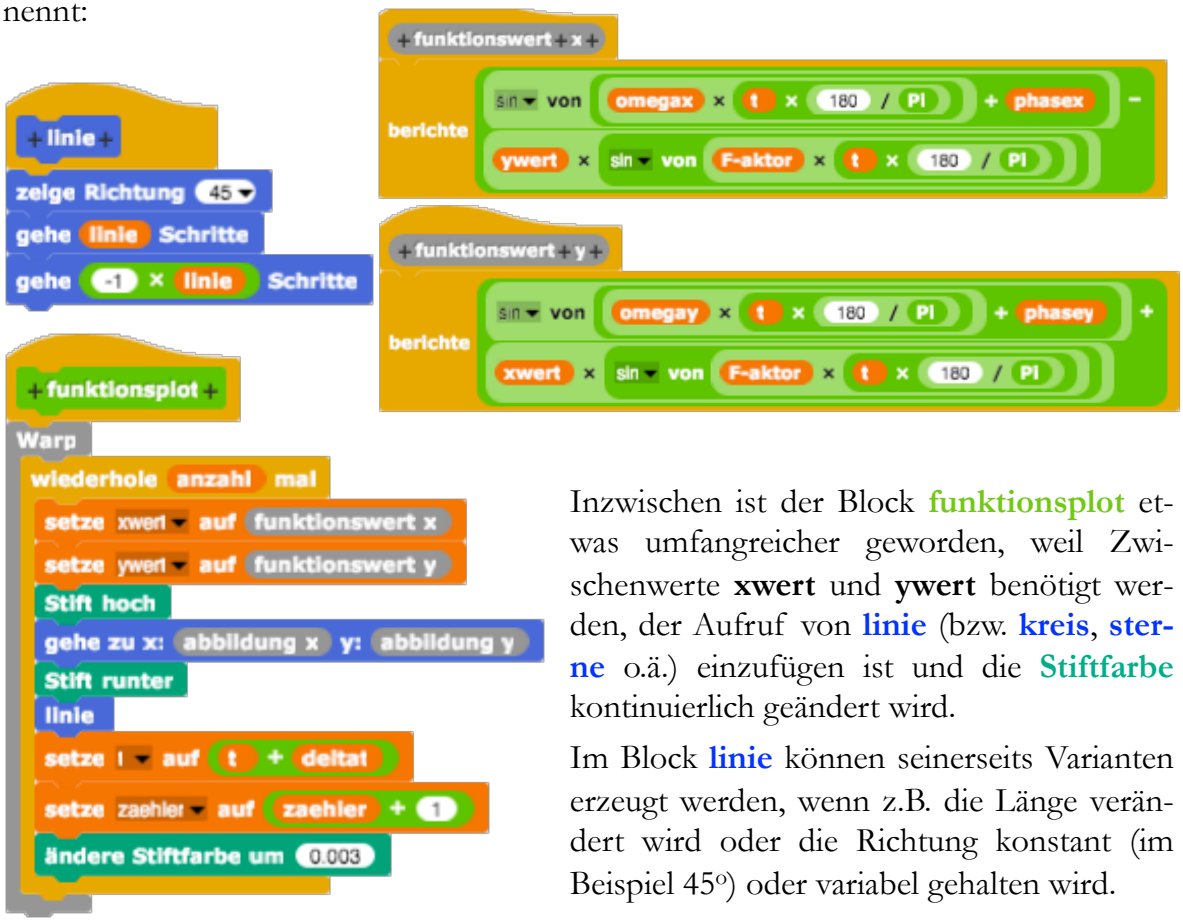
Allein schon die bisherigen Variationsmöglichkeiten liefern ein nahezu unerschöpfliches Reservoir an ästhetischen Objekten. Einige Autoren nehmen die klassischen Lissajous-Figuren als Grundlage für systematische Erweiterungen; ich nenne sie einfach mal *Anhängsel*. Bei Pickover (1991, S. 297 ff.) sind es *Wickelkurven*, die er einzelnen Funktionspunkten anfügt. Bei Brill (2002, 55 ff.) sind es Linien und Kreise (er nennt es *embellishments*), die er bei überlagerten Schwingungen den Punkten anhängt.

Ein Frequenzverhältnis **omegax:omegay** von 5:4 und eine Schrittweite **delta** von 0.06 soll die Bildentstehung veranschaulichen. Zum einen können wir an jeden Funktionspunkt eine Linie anhängen (folgende Abbildung links). Alternativ wird hier an jeden Punkt ein Kreis gekoppelt (rechts daneben).

Die Beispiele können nur andeuten, welch unterschiedliche Ergebnisse zu erzielen sind. Wieder können Effekte mit Farbe und Strichdicke erzeugt werden. Hinzu kommt, dass allein die Ausrichtung der Linien (in obigem Beispiel haben alle die gleiche Richtung) deutliche Effekte hat (folgende Abbildung zweite von rechts). Natürlich können die Linien oder Kreise durch beliebige andere Figuren ersetzt werden (im folgenden ganz rechts sind es Sterne).



Bei Brill (a.a.O., S. 57) finden wir modifizierte Gleichungen, die er *beyond Lissajous* nennt:



Inzwischen ist der Block **funktionsplot** etwas umfangreicher geworden, weil Zwischenwerte **xwert** und **ywert** benötigt werden, der Aufruf von **linie** (bzw. **kreis**, **sterne** o.ä.) einzufügen ist und die **Stiftfarbe** kontinuierlich geändert wird.

Im Block **linie** können seinerseits Varianten erzeugt werden, wenn z.B. die Länge verändert wird oder die Richtung konstant (im Beispiel 45°) oder variabel gehalten wird.

Aus dem **funktionsplot** der klassischen Lissajous-Figuren ist damit ein variables Programm geworden, mit dem sich unendlich viele Varianten erzeugen lassen.





Mit dem Befehl **Warp** kann die grafische Ausgabe beschleunigt werden. Dann werden die grafischen Ergebnisse der umschlossenen Befehlsfolgen nicht nacheinander ausgegeben, sondern erst ihre gesamte Abfolge auf einen Schlag. Andere Skripten werden erst nach deren Beendigung abgearbeitet.

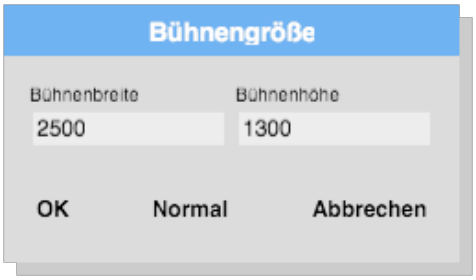
Mit diesem Befehl können bei Bedarf langwierige Ausgaben beschleunigt werden. Auch Animationen werden damit erleichtert.

Mit Lissajous-Figuren, ihren Abwandlungen und Verfremdungen, haben sich nicht nur Computerkünstler befasst, sondern auch etliche Mathematiker und Physiker, die Gefallen an ästhetischen Objekten gefunden haben. Es würde den Rahmen sprengen, sie mit ihren Ergebnissen vorzustellen. Ein paar Verweise auf entsprechende Quellen sollen genügen:

Bei Steller (1992, S. 295 ff.) finden sich ungewöhnliche Verfremdungen durch Hintergrundgestaltung und Linienvariationen. Dass sie auf Lissajous-Figuren basieren, ist ihnen kaum mehr anzusehen.

Ein ganzes Kapitel ihres Buches *Generative Gestaltung* widmen ihnen auch Bohnacker et al. (2009, S. 348 ff.). Sie verändern die Ausgangsfiguren durch Addition bzw. Multiplikation von Funktionen (ähnlich dem *Remixing (I)*). Eine eigene Ästhetik zeigen ihre Figuren, bei denen nicht nur ein Punkt mit dem nächsten Punkt verbunden wird, sondern jeder Punkt mit allen anderen.

Hinweis: *Wenn Sie von besonders gelungenen Grafiken Ausdrucke anfertigen wollen (für mich wäre z.B.* Bild 29 *ein Kandidat dafür), die (evtl. gerahmt) ausgehängt oder ausgestellt werden sollen, dann sollten Sie immer mit einer möglichst großen Bühne arbeiten. Da nur mit Plottern oder Zeichenrobotern (vgl. dazu das Kapitel 26:* Blick über den Tellerrand*) auflösungsunabhängig (mit Vektorgrafiken) gezeichnet werden kann, am Bildschirm aber grundsätzlich Pixelgrafiken entstehen, hilft nur eine möglichst hohe Pixelzahl, die Ausgabequalität zu erhöhen.*



In der Werkzeugleiste von Snap! finden Sie unter dem Werkzeugsymbol die Option **Bühnengröße...**, mit der Sie die **Bühnenbreite** und **Bühnenhöhe** nach Bedarf bzw. den Möglichkeiten Ihres Monitors einstellen können. Für Ausstellungsbilder habe ich an einem 27-Zoll-Monitor eine Auflösung von 2500*1300 Bildpunkten nutzen können. Damit ließen sich Ausdrucke davon angefertigter Bildschirmfotos in guter Qualität bis zum Format A1 (594*841 mm) herstellen. Bei großen Farbflächen ist zu testen, ob das Ausgabegerät (Laserdrucker, Tintenstrahler) streifenfreie Ausgaben garantiert.

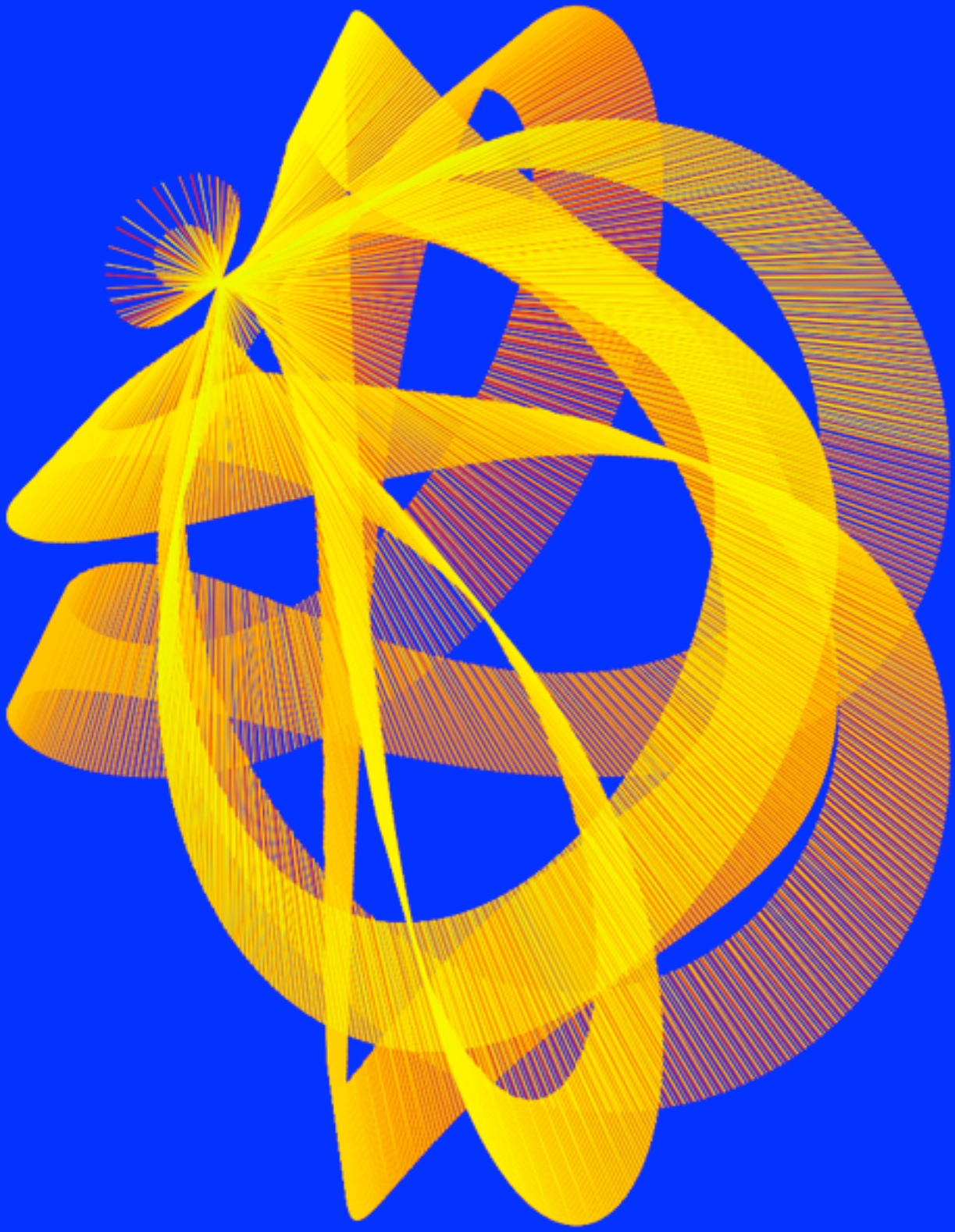
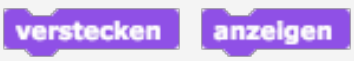


Bild 29: Sinus-Addition

Hinweis: *Wenn es Sie stört, dass bei umfangreichen Zeichenoperationen die Schildkröte sich ständig auf der Bühne bewegt, können Sie sie mit dem Befehl* **verstecken** *unsichtbar machen; die Ausführungsgeschwindigkeit wird aber dadurch nicht erhöht.*



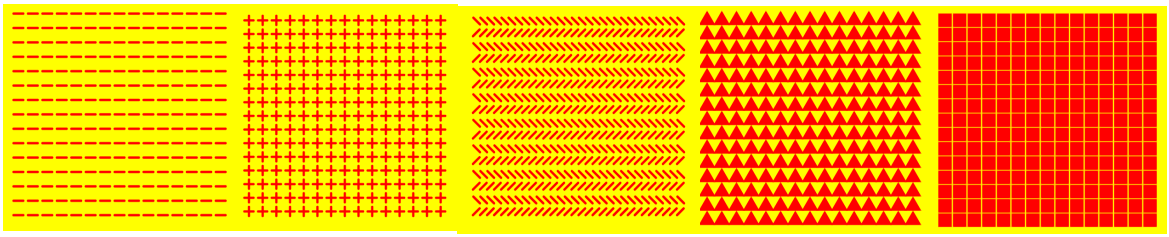


Durch den Befehl **verstecken** wird die Schildkröte unsichtbar gemacht. Mit dem Gegenstück **anzeigen** wird sie wieder sichtbar gemacht.

11. MUSTER AUS FIGUREN

Der Computer ist prädestiniert dafür, Wiederholungen rasch und unermüdlich abzuarbeiten - wie wir an etlichen Beispielen schon gesehen haben (u.a. bei Bild 13). Dementsprechend leicht lassen sich damit *Muster* erzeugen, also sich wiederholende, flächendeckende, grafische Strukturen. Große Bedeutung haben solche z.B. beim Design von Stoffen oder Tapeten, aber auch als Ornamente an Bauwerken. In der islamischen Kunst sind besonders geometrisch konstruierte Muster ausgeprägt, oft in Form lückenloser Kachelungen.

Anregung: *Die folgenden fünf Beispiele können Sie mit simplen Wiederholungsschleifen leicht nachvollziehen. Die drei Linienmuster zeigen das Prinzip der Variation (Kreuze statt Linien bzw. Richtungswechsel der Linien) und zwei Vieleckmuster nach dem Prinzip der Kachelung. Neue Varianten sind leicht zu erzeugen, sowohl durch die manuelle Festlegung der Kenngrößen als auch durch Zufallsprozesse.*



In den bisher vorgestellten Bildbeispielen wurden jeweils wenige figürliche Grundelemente verwendet. Ihre Wirkung und Komplexität haben sie durch deren *Wiederholung* und die *Variation* der charakteristischen Parameter gewonnen. Das wird im Übrigen auch oft von den Vertretern des *konkret-konstruktiven Ansatzes* und der *Konzeptkunst* angewendet. Wenn damit ästhetische Objekte erzeugt werden, kommen Grundelemente der visuellen Gestaltung ins Spiel (siehe z.B. Lewandowsky & Zeischegg, 2002): Bei *Punkten* sind also Charakteristika wie ihre *Position*, *Größe*, *Anzahl*, ihre *Form* und *Farbe* prägend. Bei *Linien* sind es ihre *Länge*, die *Position* und *Richtung*, ihre *Stärke*, die *Anzahl*, ihre *Abstände* und ihre *Farbe*. Bei *Flächen* wiederum sind es die *Formen* sowie die *Proportionen* (wie z.B. Quadrat statt Rechteck, Kreis statt Ellipse) und ihre *Farben*. Den Gegenpol und das Mittel, die Vielfalt wieder einzufangen und ihr gewünschte Struktur zu geben, sind Ordnungssysteme wie die *Symmetrie* der Elemente oder der *Rhythmus* innerhalb der Wiederholungen.

So ist es wenig erstaunlich, dass in der frühen Computerkunst zahllose Beispiele zu finden sind, in denen genau diese Eigenschaften genutzt werden, um Muster und Strukturen zu erzeugen. Viele dieser Beispiele finden sich in Franke (1991), IBM (1978) oder Franke (1984). Es soll in unserem Kontext reichen, sechs Beispiele exemplarisch heraus zu greifen, die die genannten Prinzipien verdeutlichen. Alle sechs zeigen zugleich, dass gerade das Durchbrechen der dekorativen Regelmäßigkeit von Mustern zu Bildern führt, die die Aufmerksamkeit der Betrachter herausfordern und binden kön-

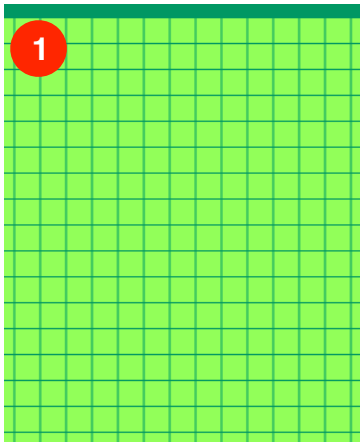
nen. Auch hier werde ich nicht mehr den vollständigen Programmcode vorstellen. Im Mittelpunkt stehen die konzeptionellen Schritte, die schließlich zum gewünschten Muster führen.

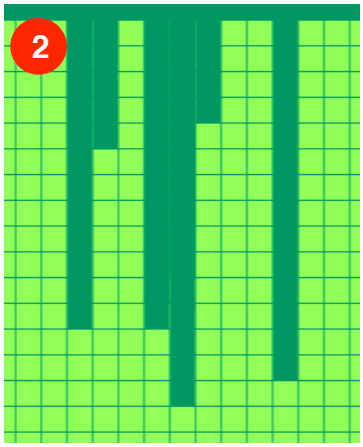
11.1 Hommage à Aaron Marcus: Urbane Nova

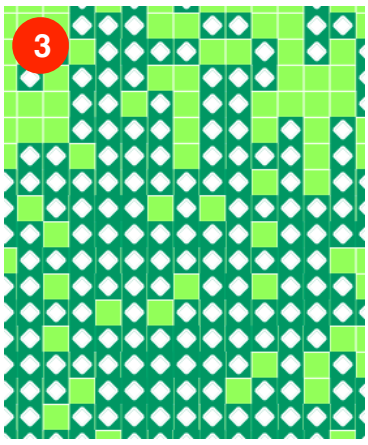
Das erste Vorbild in diesem Kapitel ist *Urbane Nova* (in IBM, 1978, S. 69). Es stammt von Aaron Marcus (geb. 1943 in Omaha, Nebraska), einem amerikanischen Designer und Entwickler von Benutzerschnittstellen, der auch zur Computerkunst beigetragen hat. Bild 30 (oben) orientiert sich daran, ergänzt um eine Farbvariante (unten).

Der Nachvollzug dieses Bildes kann in drei Schritten erfolgen:

1. Vor einem dunkel eingefärbten Hintergrund werden 60*60 Felder hell eingefärbt.
2. Die „Unordnung“ ins Bild bringt eine vom oberen Rand kommende zufällige Anzahl Felder mit der Hintergrundfarbe in zufällig bestimmten Spalten.
3. Abschließend werden 1000 zufällig - mittig zentrierte - ausgewählte Felder mit der Hintergrundfarbe eingefärbt und darüber helle Rauten gelegt.

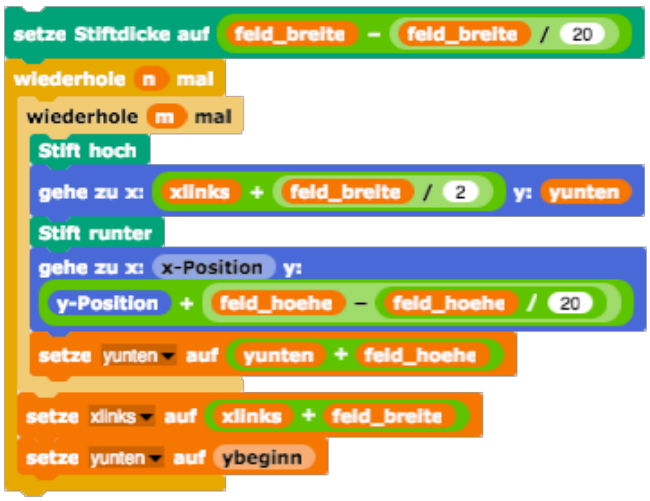






Zur programmtechnischen Umsetzung ein paar Hinweise (die Farbfestlegungen fehlen):

1. Die Felder werden wieder über das m*n-Raster definiert und in einer Doppelschleife (wie beim Bild *25 Quadrate*, hier aber ohne Zufallsabweichungen) mit hellen Quadraten eingefärbt.



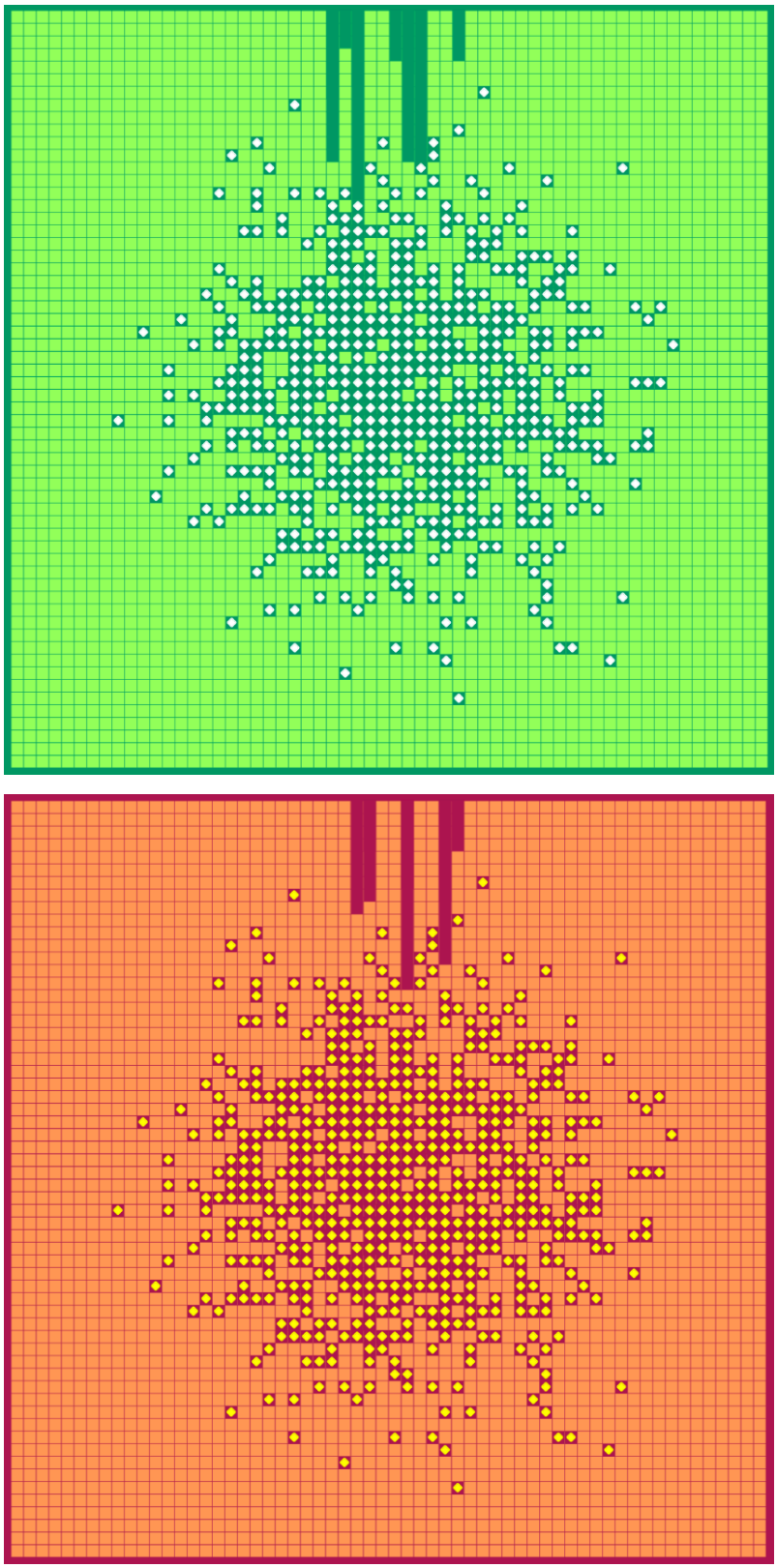
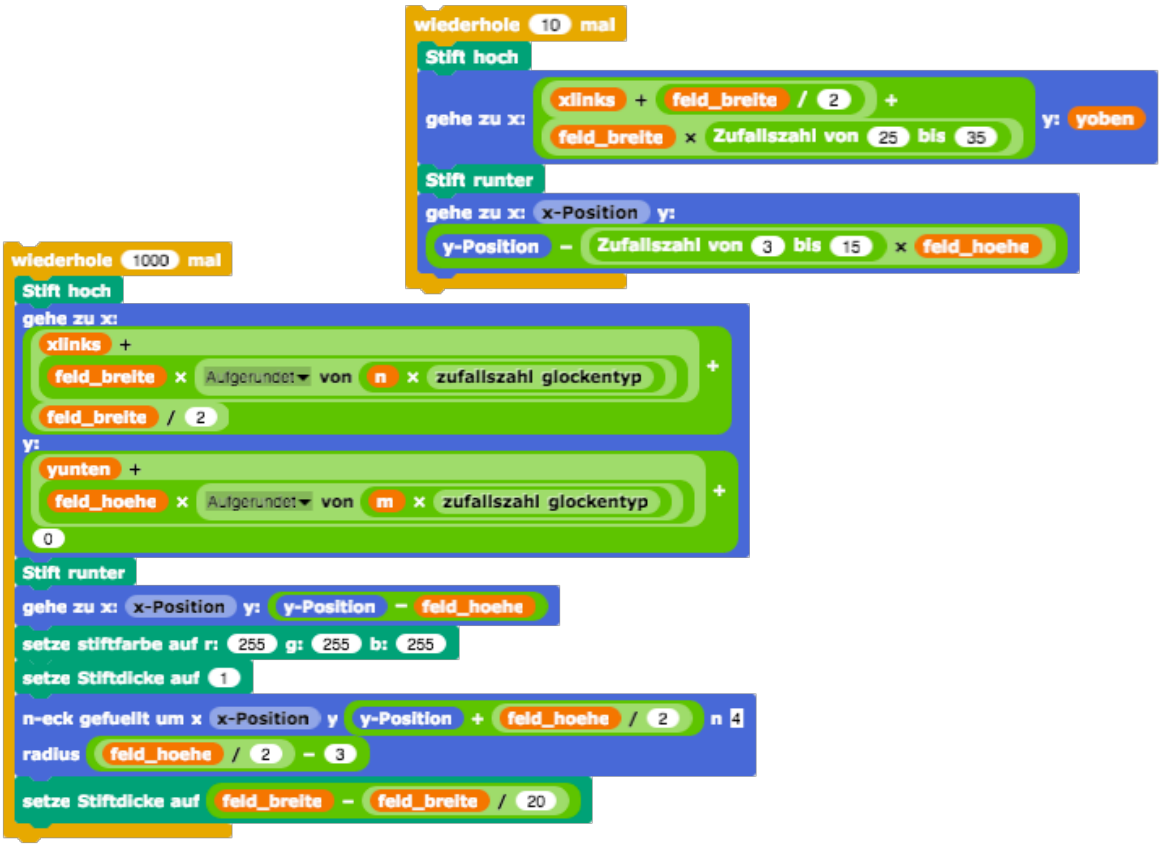


Bild 30: Hommage à Marcus: Urbane Nova (oben), Farbvariante (unten)

2. Die wieder dunkel einzufärbenden Felder am oberen Rand werden zufällig festgelegt.

3. Auch die Felder mit den hellen Rauten werden zufällig festgelegt. Deren mittige Zentrierung wird durch die Verwendung von **zufallszahl glockentyp** erreicht. Nach der dunklen Einfärbung des Feldes wird darüber die Raute mit **n-eck gefuellt um x y n radius** gelegt.



Anregung: *Dass das Bild von Marcus - trotz der Zufallskomponenten - bewusst gestaltet ist, zeigt sich für mich, wenn die von oben hereinragenden Felder weggelassen werden. Ähnlich verändert wird der Charakter mit anderen Zufallsverteilungen. Schließlich bietet sich an, neben Farbvarianten für das Remixing statt der Rauten auch andere Elemente für die Überlagerung zu wählen.*

11.2 Hommage à Komura: Optical Effect of Inequality

Der Japaner Masao Komura (geb. 1943 in Tokio) war Gründer der Computer Technic Group (CTG), deren Spektrum von geometrisch-abstrakten Werken bis zu gegenständlichen Arbeiten reichte. Berühmt sind ihre figurativen Arbeiten, bei denen Figuren ineinander transformiert werden, z.B. *Return to Square*, die Metamorphose eines Quadrats in einen Frauenkopf oder *Running Cola is Africa*, bei der ein Läufer in eine Cola-Flasche und diese dann in einen Afrika-Umriss verwandelt wird - etwas, was wir heute als Morphing bezeichnen würden.

Komuras Arbeit mit dem Titel *Optical Effect of Inequality* nutzt das mathematische Symbol **>** (größer als) zur Erzeugung eines grafischen Musters. Mein *Recoding* führte zu Bild 31, das weitgehend dem Vorbild entspricht (im Original Weiß auf Schwarz).

Die Charakteristika des Bildes sind:

1. In jeder Zeile wird das Symbol von Anfang bis Ende um 360 Grad gedreht.



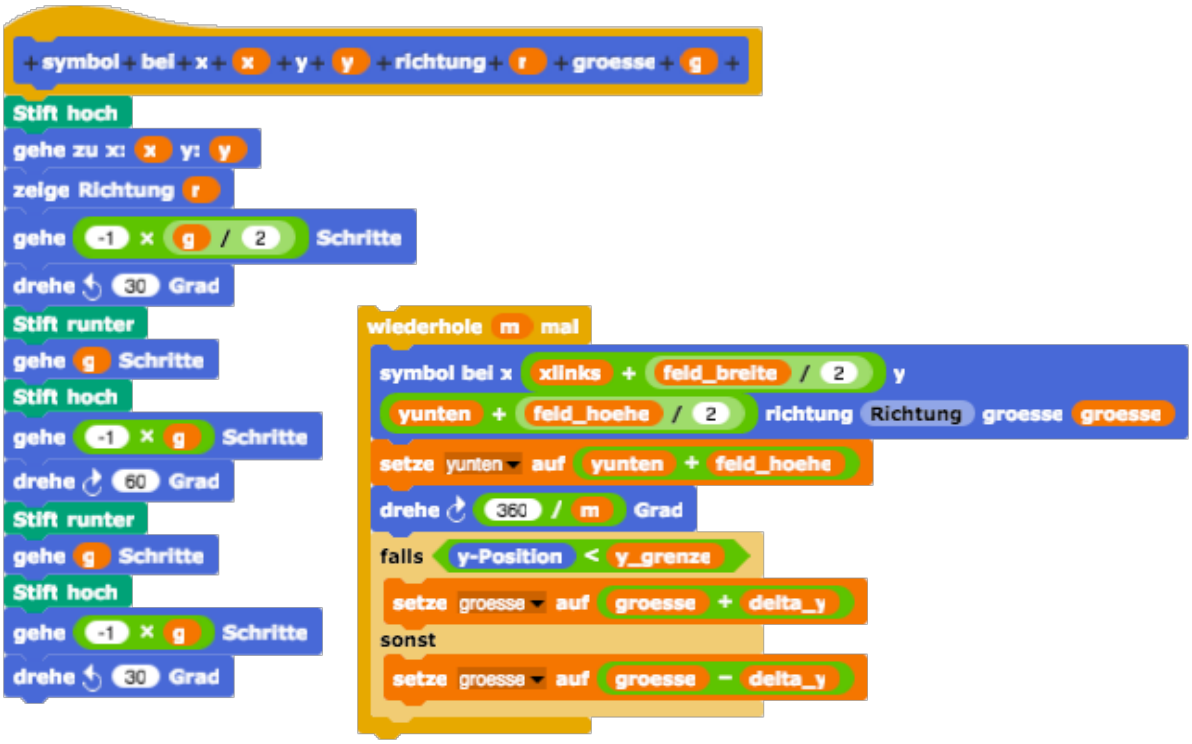
2. In jeder Zeile wird das Symbol von Feld zu Feld vergrößert und ab einem festzulegenden Feld wieder verkleinert.



3. Die Drehung sowie Vergößerung bzw. Verkleinerung gilt auch für jede Spalte.

Für die programmtechnische Umsetzung bildet wieder ein m*n-Raster (hier mit 35*45 Feldern) die Grundlage. Die Anfangsgröße des Symbols ist **laenge**, die wir in **groesse** zwischenspeichern. Zusätzlich legen wir eine **x_grenze** und eine **y_grenze** fest, ab der ein Größenzuwachs **delta_x** bzw. **delta_y** in eine entsprechende Abnahme übergeht.

Das Zeichnen des Symbols verlagern wir in einen Block **symbol**. Für **symbol** wird der Ort mit **x y**, die **richtung** und die **groesse** übergeben. Die unten gezeigte **wiederhole**-Schleife enthält den Aufruf des Blocks **symbol** mit den Kennwerten für das jeweilige Feld. Die aktuelle **y-Position** wird mit der **y_grenze** verglichen und davon abhängig **groesse** erhöht bzw. erniedrigt. Der entsprechende Vergleich der **x-Position** mit



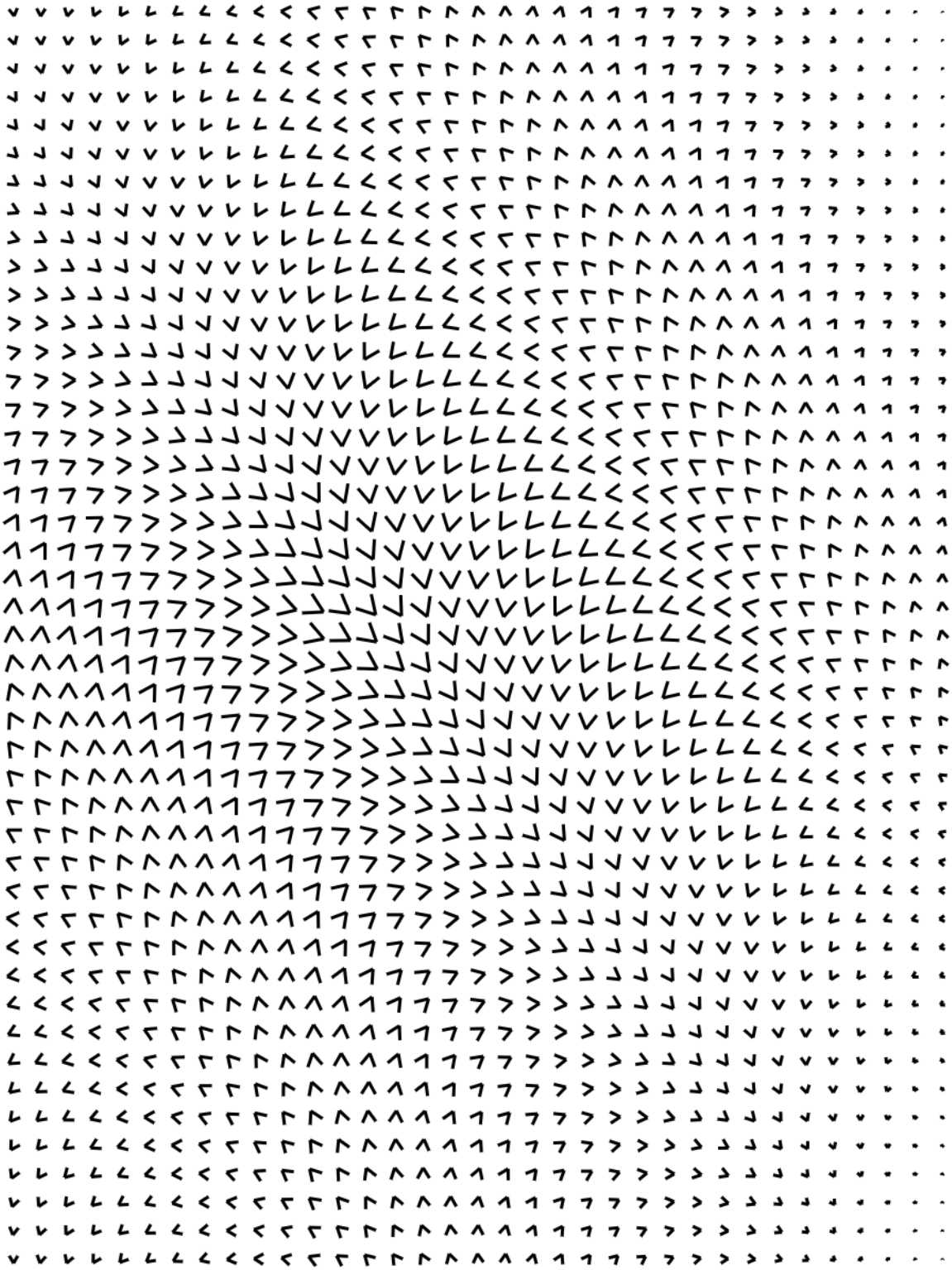


Bild 31: Hommage à Komura: Optical Effect of Inequality

x_grenze erfolgt in der äußeren **n-Schleife** (die hier nicht gezeigt wird), in der außerdem **xlinks** und **yunten** zurück zu setzen sind.

Hinweis: *Der hier gezeigte Block* **symbol** *zeichnet einen Pfeil der Länge* **g**. *Beginn ist immer in der Mitte des Feldes. Nach Ende der Zeichnung befindet sich die Schildkröte wieder am Ausgangspunkt mit der ursprünglichen Richtung.*

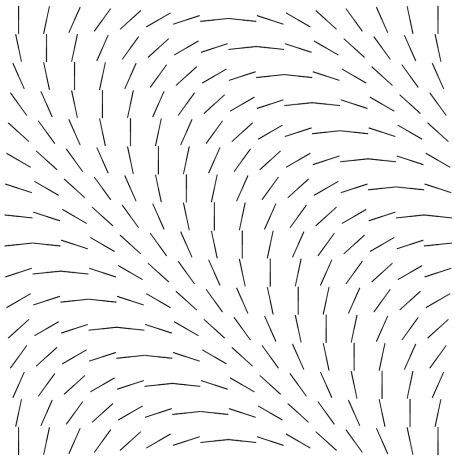
Das Symbol kann bei Bedarf beliebig verändert oder ersetzt werden. Das können einfache Striche sein, aber auch Vielecke und deren farbig gefüllte Pendants.

Mit dem m*n-Raster und dem Block **symbol** haben wir nun eine sehr allgemeine Methode, so dass damit interessanterweise auch Bildserien anderer Vertreter der frühen Computerkunst direkt erzeugt werden können.

11.3 Hommage à Bartnig: 256 ...

Ein Beispiel dafür sind die *Permutationsserien*, mit denen sich Horst Bartnig intensiv befasst hat (siehe auch sein Bild *Entwicklung zum Quadrat* im Kapitel 22: *Noch mehr Computer) Kunst*). Bild 32 zeigt eine seiner Serien (Franke, 1984, S. 45). Vermutlich haben sich Bartnig und Komura nicht gekannt; also kannten sie wohl auch die Algorithmen des Anderen nicht. Dennoch sind sie zu verblüffend ähnlichen Ergebnissen gekommen. Bei uns reicht es, in einem *16*16-Raster* im Block **symbol** statt des Pfeils Linien zu verwenden:

Für das *Recoding* der Serie von Bartnig kommen noch Quadrate und Rechtecke hinzu.



Wie immer bietet sich ein *Remixing* an, hier etwa durch die Veränderung des Rasters, der Größenverhältnisse oder der Verwendung von Farben (Bild 33). In entsprechenden Konstellationen kann sich dabei der Bildcharakter stark ändern. Die folgenden Varianten sind entstanden durch Änderung von **n** und **m** des Rasters, der horizontalen **drehung_x** bzw. vertikalen **drehung_y** (mit denen die Drehung des Symbols **linie** von Feld zu Feld festgelegt wird) sowie der Farben.

Tipp: *Für die folgende Abbildung links oben wurden z.B.* n = m = 64, drehung_x = drehung_y = 12 *und* groesse = feld_breite+100 *verwendet. Für die nächste Abbildung wurde* m *auf 12 reduziert und für die weiteren Abbildungen jeweils* n *und* m *sowie die Drehungswinkel verändert.*

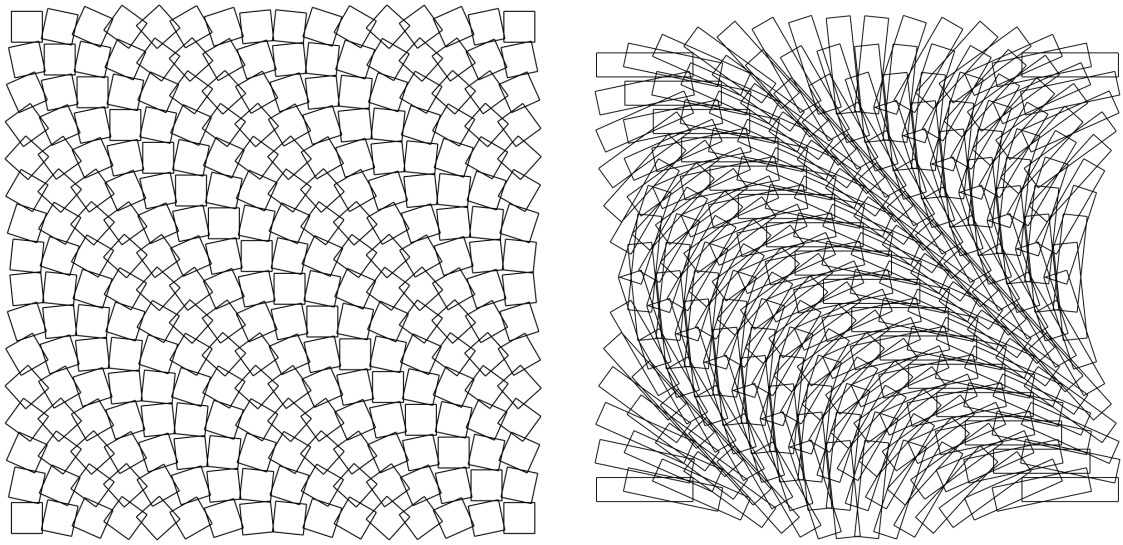


Bild 32: Hommage à Bartnig: 256 Quadrate (links), lange Rechtecke (rechts)

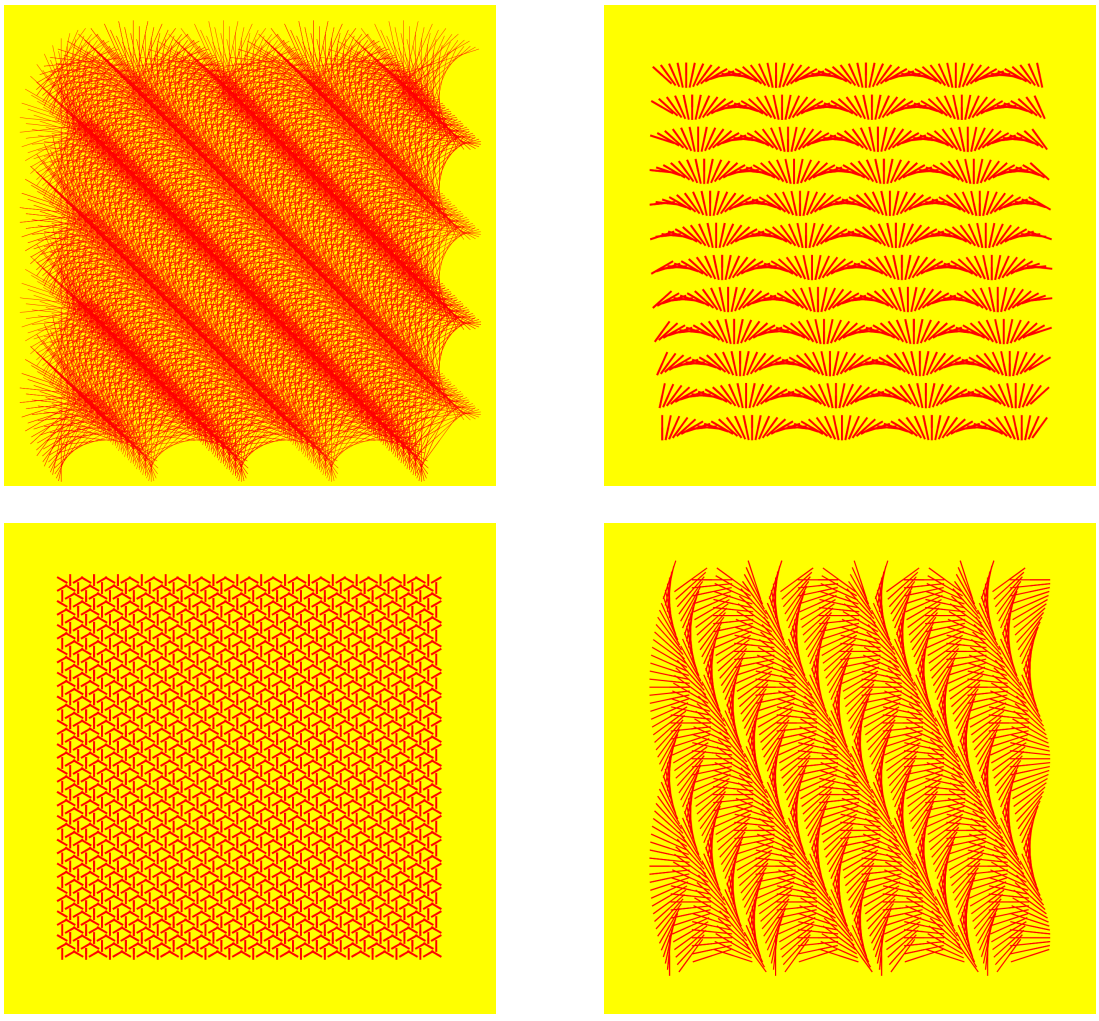


Bild 33: Remixing Permutationsserien

11.4 Hommage à Mohr: Scratch Code

In den 1960er-Jahren gab es nur sehr wenige Maler, die sich der Computerkunst zuwandten. Einer der bekanntesten ist Manfred Mohr (geb. 1938 in Pforzheim), Maler und Musiker, der ab 1968 mit dem Computer einen *„überraschenden neuen Weg gefunden [hat] über [meine] Kunst nachzudenken."* Schon davor ist sein Werk geprägt von geometrischen Strukturen und der Reduktion auf Schwarz/Weiß. Er entwickelte eigene Zeichensysteme, die aus wenigen Grundelementen mit Kenngrößen wie Strichlängen, Abständen, Strichstärken oder Winkeln bestanden. Ab 1969 nutzte er den Computer, um seine grafischen Basiselemente aneinander zu fügen. Eine dieser Bildserien nennt er denn auch *Formal Language*.[55]

Vorbilder für das folgende Projekt finden wir in seiner Serie *Bandstrukturen* (Mohr, 1971, S. 11 ff.). Dabei verwendet Mohr Linienzüge, bei denen die Abweichung von der Horizontalen codiert ist und jeweils zufällig ausgewählt wird. Das Bild *Scratch Code* (Bild 34 oben) ist eines von vielen, bei denen Manfred Mohr so mit eigenen Zeichensätzen Muster erzeugt. Vielfach setzt er diese Zeichen, angeordnet in Reihen und Spalten, zu Linieneinheiten zusammen, so auch in *Scratch Code*:

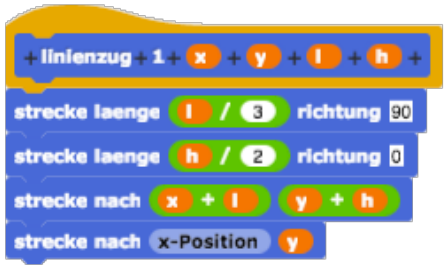


Obwohl der Eindruck übereinander gestapelter Linien entsteht, sind die einzelnen Linien bei genauerem Hinsehen aus wiederkehrenden Einheiten (Linienzügen) aufgebaut. Für das *Recoding* habe ich fünf solcher Einheiten gestaltet:

Der Code für **linienzug 1** (ganz links) besteht aus vier Streckenzügen, beginnend in der linken, unteren Ecke **x y** des jeweiligen Feldes. **l** und **h** sind Breite und Höhe des Feldes.



Entsprechende Blöcke sind für die weiteren Linienzüge zu erstellen (hier nicht dargestellt). Wichtig ist, dass jeder Linienzug in der rechten, unteren Ecke **x+l y** des Feldes endet, um so den direkten Anschluss des nächsten Linienzuges sicher zu stellen.



Ein *Remixing* dieses Beispiels nutzt diese Anschlussfähigkeit der Symbole, nun aber in der Diagonalen. Fünf entsprechende Symbole könnten z.B. aussehen wie nebenstehend gezeigt. Sie ergeben dann Bild 34 unten.



[55] Anlässlich der Ausstellung *Der Algorithmus des Manfred Mohr* ist eine Textsammlung *Texte 1963-1979* erschienen (Rosen, 2013), mit der sich die Denk- und Vorgehensweise von Manfred Mohr gut erschließen lässt.

Innerhalb der Doppelschleife für das m*n-Raster ist für jedes Feld zufällig zu entscheiden, welcher Linienzug verwendet werden soll. Der Einfachheit halber habe ich alle 10 Linienzüge der beiden Bildvarianten in den Entscheidungsprozess einbezogen.

1. Je nach Bild wird deshalb eine Zufallszahl im Bereich 1 bis 5 (Bild 34 oben) bzw. 6 bis 10 (Bild 34 unten) erzeugt.

2. In einer Abfolge von bedingten Anweisungen wird anhand der ermittelten Zufallszahl der entsprechende Linienzug gezeichnet.

3. Am Ende der Schleifen sind die Koordinaten des nächsten Startpunkts anzupassen.



Anregung: *Dieses Programm kann als Gerüst für vielerlei Varianten dienen. Die Linienzüge können beliebig ergänzt oder verändert werden, sofern auf die Anschlussfähigkeit der Linienzüge geachtet wird.*

Werden die Anschlusspunkte verlegt, z.B. mittig gewählt, kann ein ganz neuer Bildeindruck erzeugt werden.

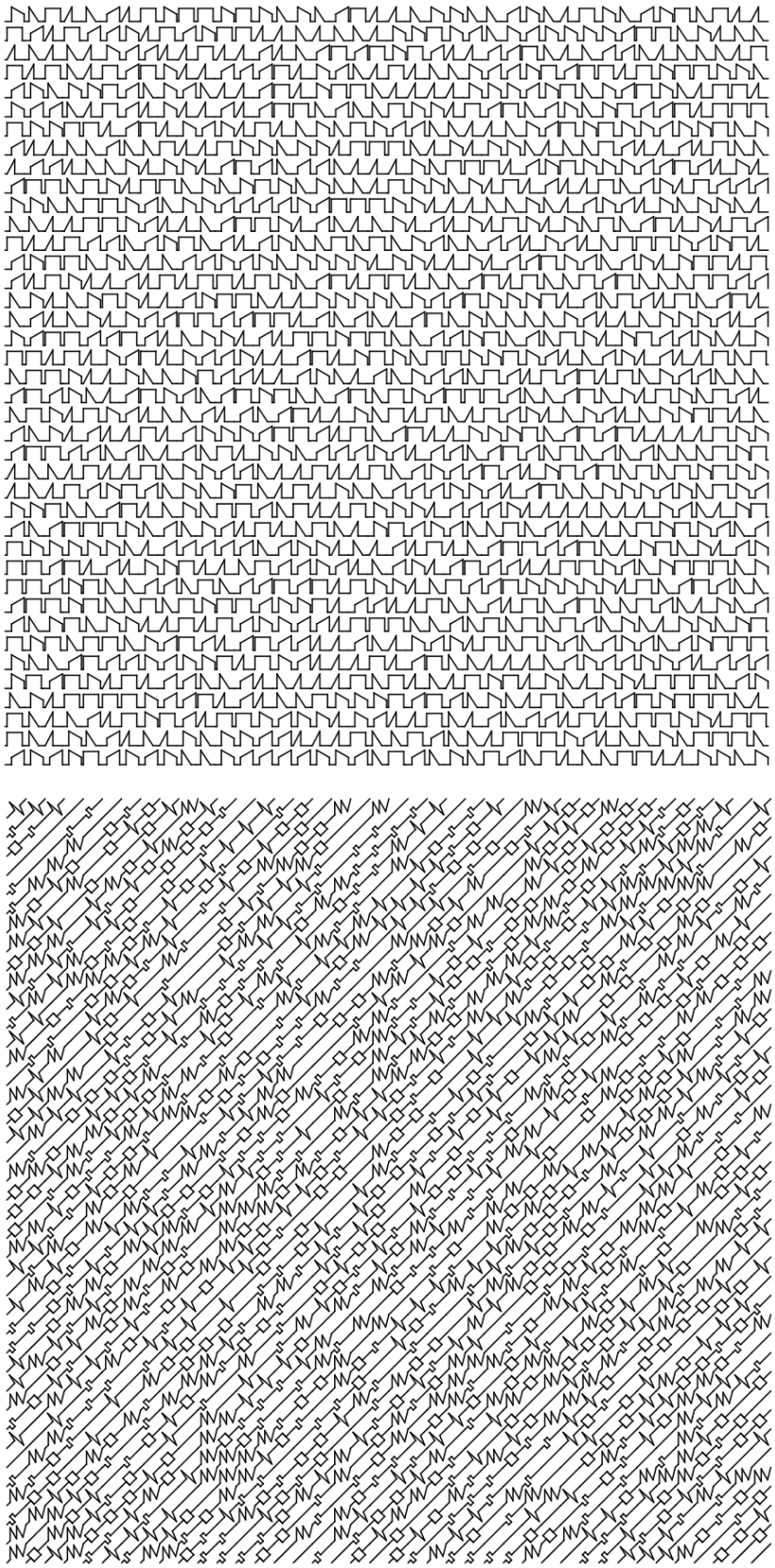
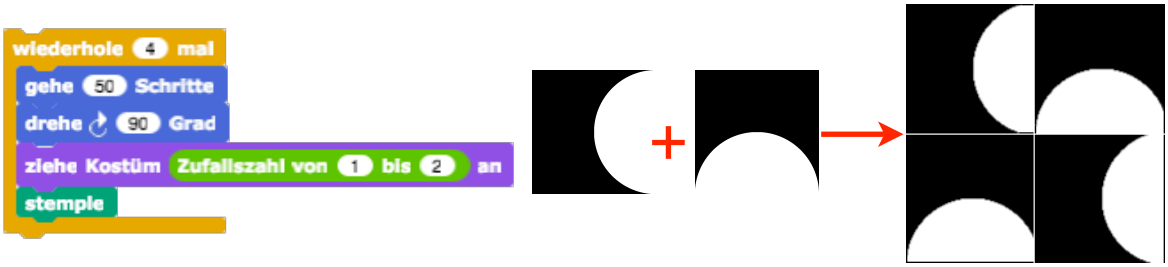


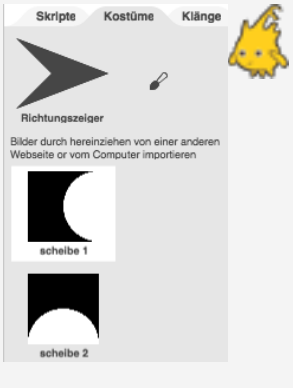
Bild 34: Hommage à Mohr: Scratch Code (oben), Diagonalvariante (unten)

11.5 Exkurs: Stempeln statt Malen

In allen bisherigen Projekten wurden die grafischen Elemente wie Punkte, Linien usw. durch die Bewegung der Schildkröte bei abgesenktem Stift erzeugt. Für einige der folgenden Projekte soll dieses Prinzip aufgehoben werden: Statt der „Spur“ der Schildkröte wird die Schildkröte selbst zum grafischen Element. Bisher wurde sie in unseren Projekten immer als Pfeil dargestellt. Sie kann aber auch andere „Kostüme“ tragen, also unterschiedliche Sprites für die Repräsentation der Schildkröte.

In der folgenden Abbildung hat die Schildkröte z.B. zwei Kostüme, das schwarze Quadrat, bei dem einmal ein weißer Halbkreis von rechts, einmal von unten hineinragt. Aus diesen zwei Kostümen wurde das rechte Bild zusammengefügt, indem die Schildkröte an den gewünschten Ort bewegt, eins der beiden Kostüme zufällig festegelegt und dann dort dauerhaft dargestellt wird, wofür es den Befehl **stemple** gibt.





ziehe Kostüm an | nächstes Kostüm | Kostüm Nr.

Mit dem Befehl **ziehe Kostüm an** wird das aktuelle Kostüm der Schildkröte durch das ausgewählte Kostüm ersetzt. Mit **nächstes Kostüm** wird das aktuelle Kostüm durch das jeweils nächste Kostüm aus der Kostümliste ersetzt.

Die Kostümliste findet sich im Programmbereich unter dem Kartenreiter **Kostüme** (wie im Beispiel rechts).

Mit **Kostüm Nr.** lässt sich die Nummer des aktuell verwendeten Kostüms abfragen.

setze Größe auf 100 % | ändere Größe um 10 | Größe

Über **setze Größe auf x %** kann die Größe des Kostüms prozentual, mit **ändere Größe um x** um x Bildpunkte verändert werden.

Über **Größe** kann die aktuelle Größe in Bildpunkten des Kostüms abgefragt werden.

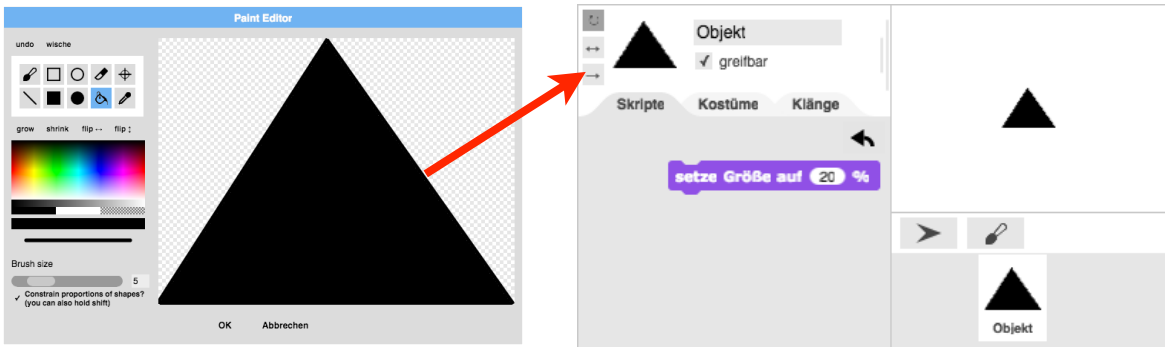
stemple

Mit **stemple** kann das aktuelle Kostüm an der aktuellen Position der Schildkröte „gestempelt“ werden. Es verbleibt dort, auch wenn die Schildkröte weiter bewegt wird.

Die Verwendung solcher Kostüme erspart das wiederholte Zeichnen mit Hilfe entsprechender Blöcke. Das kann durchaus den Code eines Programmes aufblähen und in der Ausführung zeitaufwändiger werden als das simple Stempeln eines Kostüms.

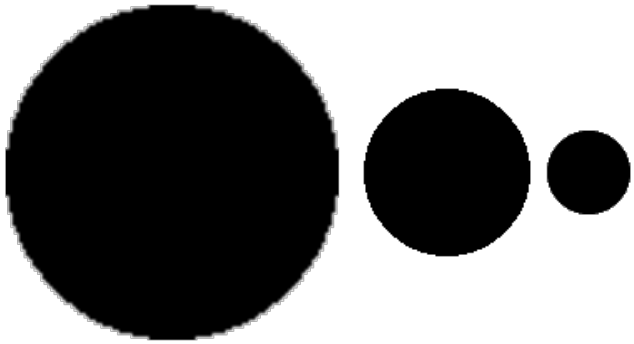
Es gibt mehrere Möglichkeiten solche Kostüme zu erstellen und dann im Programm zu verwenden. Die einfachste davon ist, Kostüme aus der Kostüm-Bibliothek von Snap! zu importieren über die Option **Kostüme...** unter dem Datei-Symbol in der **Werkzeugleiste** (für den Kontext *Computerkunst* bin ich dort allerdings kaum fündig geworden).

Eine weitere Möglichkeit besteht darin, eigene Kostüme mit dem integrierten **Paint-Editor** zu erstellen. Der Aufruf des Editors erfolgt entweder durch Anklicken eines vorhandenen Kostüms mit der rechten Maustaste und Anwahl der Option **Bearbeiten**, oder durch Anklicken des **Pinsel-Symbols** im Programmbereich. Es öffnet sich der Editor, in dem einige der üblichen Werkzeuge zum Malen angeboten werden (Näheres dazu im *Anhang D*).



Ein genaueres und flexibleres Arbeiten ist in der Regel mit gängigen externen Grafikprogrammen möglich, deren Ergebnisse dann problemlos als Kostüme in Snap! importiert werden können. Dieser Import kann über die Option **Importieren...** unter dem Datei-Symbol in der **Werkzeugleiste** erfolgen. Es kann aber auch einfach eine **Bild-Datei** mit dem gewünschten Kostüm unter dem Kartenreiter **Kostüme** in den Programmbereich gezogen werden.

Tipp: *Es ist empfehlenswert, die eigenen Kostüme größer als zunächst benötigt zu erstellen. Sie können dann mit* **setze Größe auf x %** *verlustfrei verkleinert werden. Bei Vergrößerungen enstehen dagegen leicht unschöne Verpixelungen.*



Kostümgröße: 200% *100%* *50%*

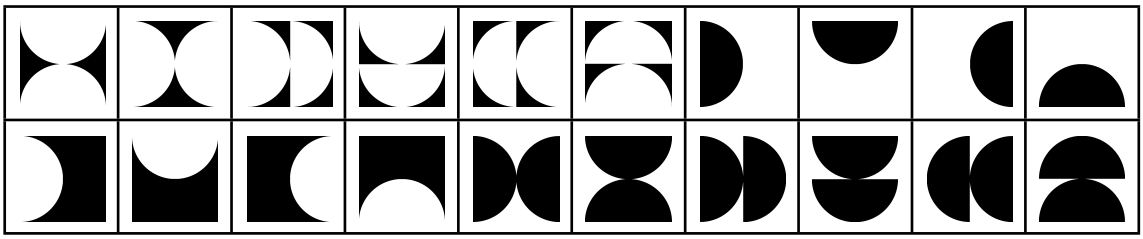
11.6 Hommage à Sýkora: Schwarz-Weiß-Struktur

Die Verwendung des Zufalls schließt nicht aus, dass für den Bildaufbau eingrenzende Regeln gelten. Der Tscheche Zdeněk Sýkora (1920 - 2011), Maler und Bildhauer, experimentierte intensiv mit dem Wechselspiel von Zufall und Regeln. Aus der Zusammenarbeit mit dem Mathematiker Jaroslav Blažek entstanden seine ersten programmierten Strukturen, bei denen nach zuvor festgelegten Regeln alle möglichen kombinatorischen Varianten für die Positionierung mehrerer vorgegebener Elemente verwendet wurden (Rosen, 2011, S. 393).

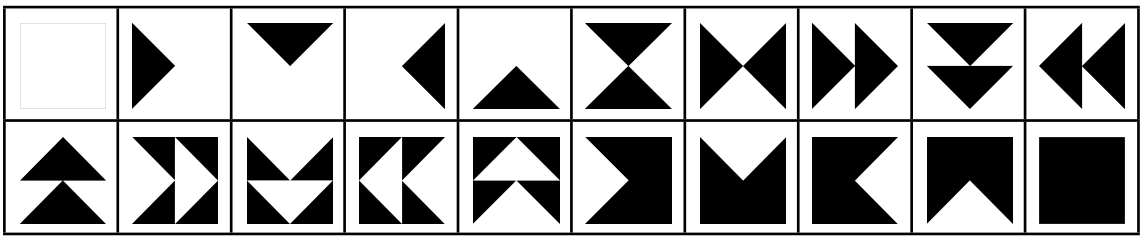
Für frühe Bildserien ab 1964 verwendete er schwarze und weiße Flächen innerhalb von Quadraten. Dabei lässt er 20 Elemente zu, die sich durch Zahl und Lage der Flächen (dies waren zunächst Halbkreisflächen) unterscheiden. Deren Art und Zahl ergibt eine Skala für den Schwarzanteil innerhalb des Quadrats. Über den Schwarzanteil kann die Auswahl entsprechender Elemente gesteuert werden. Sýkora verwendete übrigens die so entstandenen Ausdrucke - ähnlich einer Notenschrift - als Vorlage für danach ausgeführte großformatige Ölbilder.

In Bild 35 ist das Prinzip seiner Bildserie *Schwarz-Weiß-Struktur* nachempfunden. Für das *Recoding* werden sowohl seine zwanzig Halbkreisstrukturen als auch seine zwanzig Dreieckstrukturen, wie unten gezeigt, verwendet. Der Schwarzanteil spielt in Bild 35 keine Rolle, denn die Elemente sind rein zufällig ausgewählt.

Halbkreisstrukturen:



Dreieckstrukturen:



Hinweis: *Es ist sehr zu empfehlen, alle Kostüme mit Transparenz per Alphakanal zu speichern (das verlustfreie PNG-Grafikformat unterstützt diese Option). Das bedeutet z.B. bei den Sýkora-Kostümen, dass statt der Weißflächen transparente Flächen um die schwarzen bzw. farbigen Kostümteile bei den Halbkreisflächen oder Dreiecken gespeichert werden. Nur so können auf der Bühne diese Flächen dann mit einer Hintergrundfarbe angezeigt werden.*

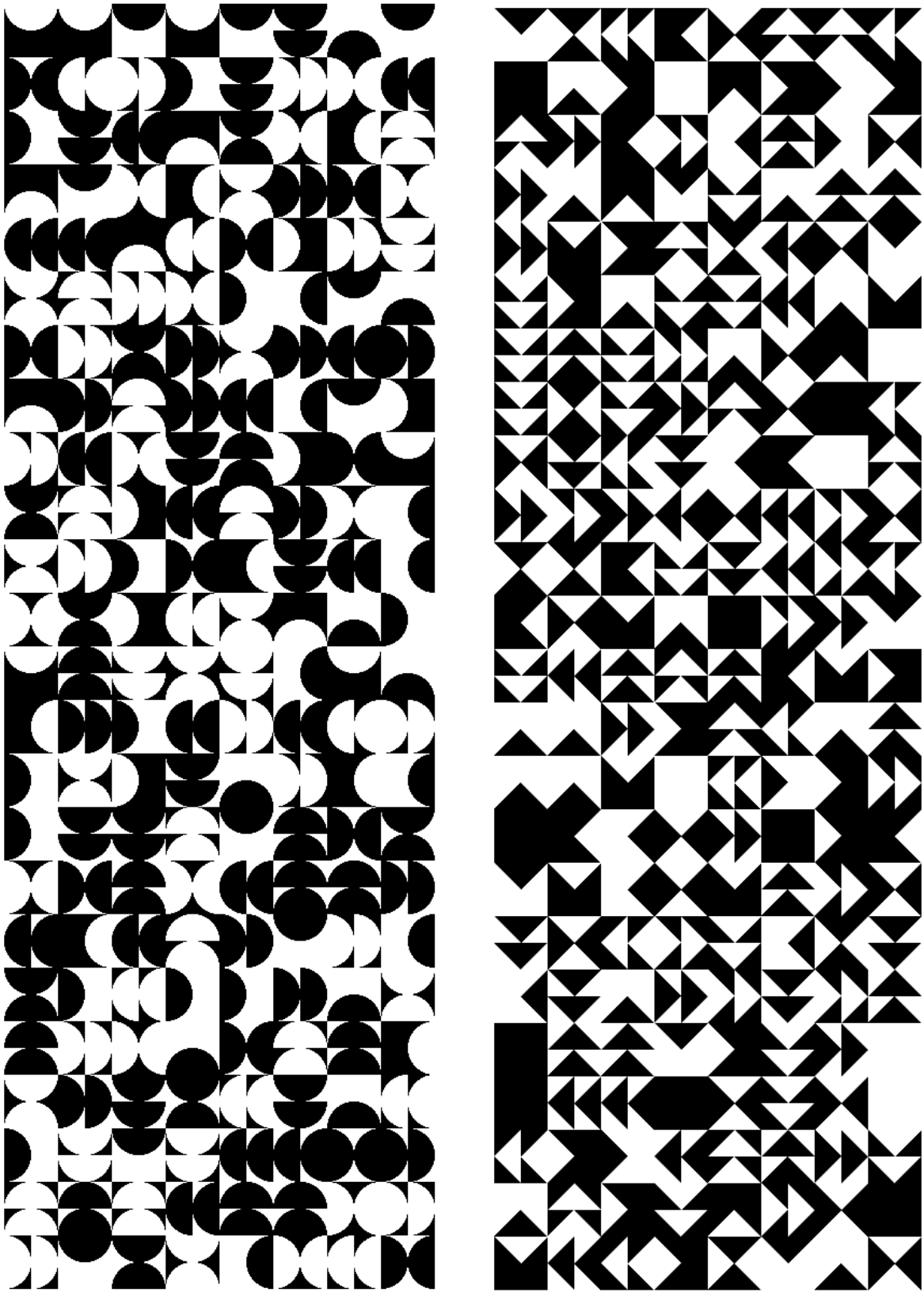
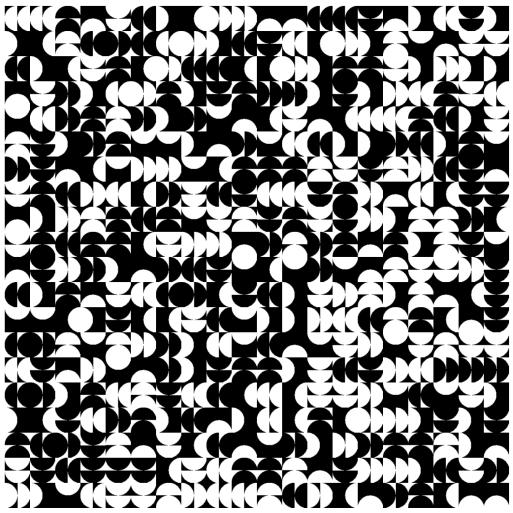


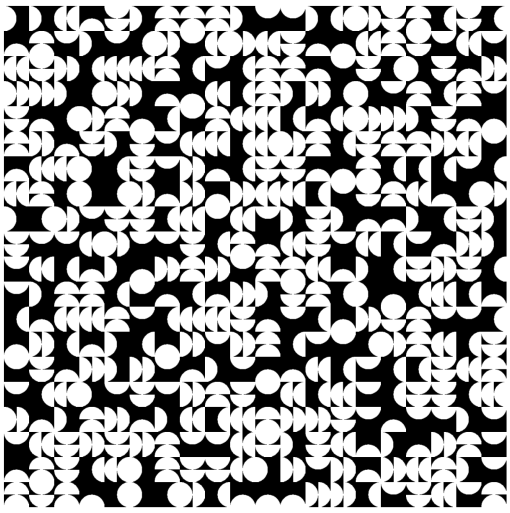
Bild 35: Hommage à Sýkora: Schwarz-Weiß-Strukturen

In seinen Bildern hat Sýkora unterschiedliche Anordnungen gewählt. In den mir zugänglichen Vorlagen gibt es z.B. eine 11*22-Matrix, aber auch 20*20. Für Bild 35 habe ich eine 8*24-Matrix gewählt. Für jedes Feld wird das Kostüm zufällig über den Befehl **ziehe Kostüm Zufallsahl von 1 bis 20 an** festgelegt und mit **stemple** dargestellt.

Hinweis: *Für die Umsetzung kann wieder das* m*n-Raster *verwendet werden. Die Bildbreite und Bildhöhe ist dabei mit der Zahl der* n *Spalten und* m *Reihen sowie der Größe der Kostüme abzustimmen.*

In den folgenden Varianten wurde die Auswahl jeweils bewusst auf wenige Elemente eingeschränkt.





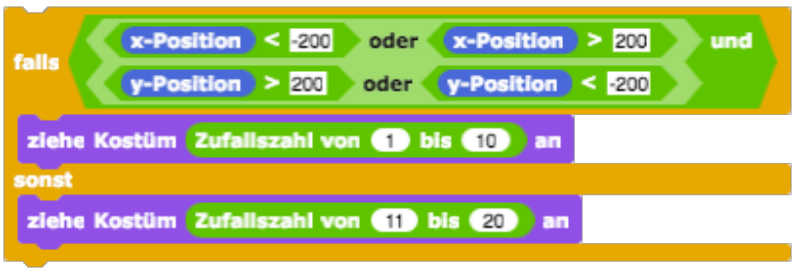
Anregung: *Die Einschränkung auf bestimmte Elemente kann mit einer Umsortierung der Kostüme kombiniert werden. Der Bildcharakter kann damit gezielt beeinflusst werden.*

11.7 Remixing Schwarz-Weiß-Strukturen

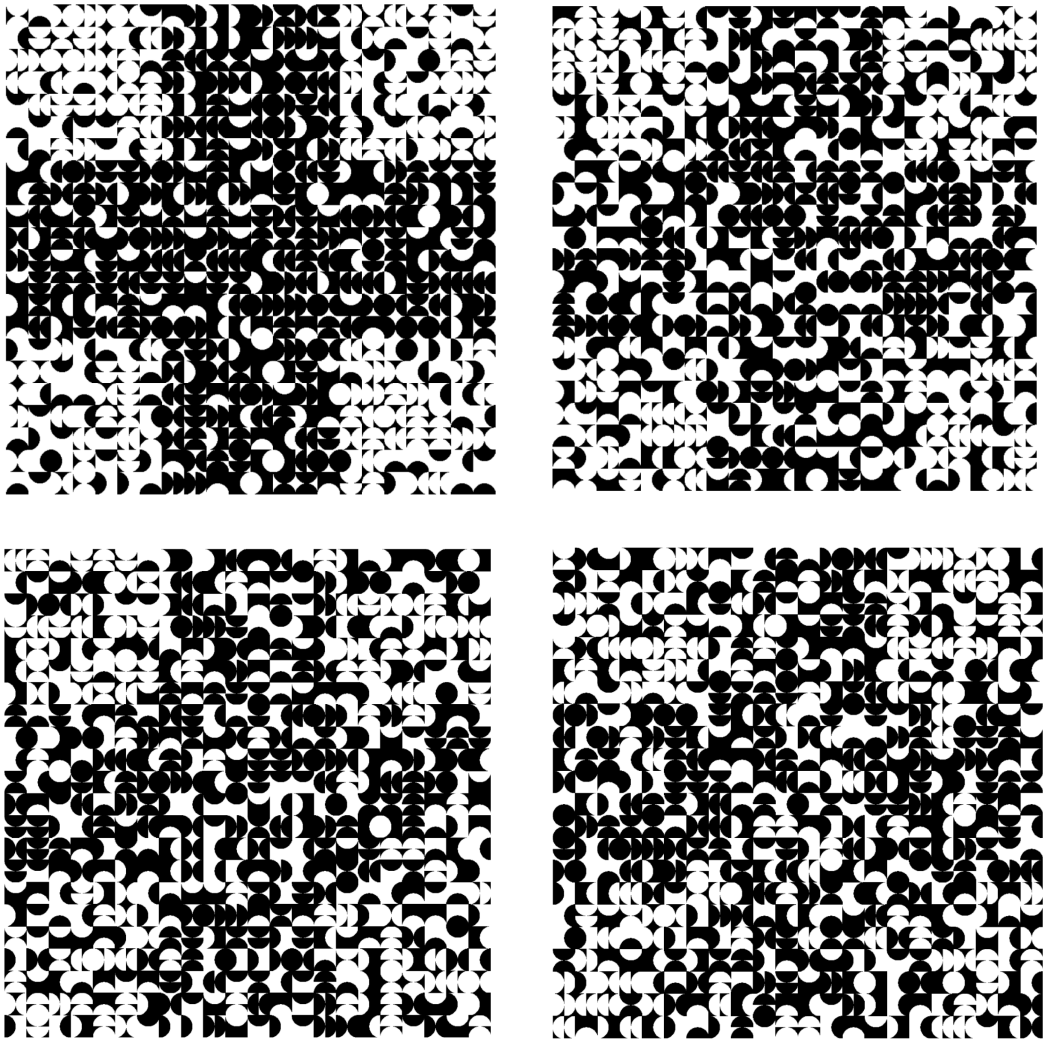
Den Weg zum *Remixing* der *Schwarz-Weiß-Strukturen* hat Sýkora schon selbst beschritten. Es gibt von ihm Varianten, in denen er statt mit Halbkreisflächen und Dreicken mit weiteren geometrischen Formen arbeitet. Ebenso hat er mit Farbvarianten seines Ansatzes gearbeitet und damit gezeigt, welch komplexe Bilder mit den kombinatorischen Anordnungen möglich werden.

Hier sollen zunächst Bilder mit gelenktem Zufall über den Schwarzanteil der Formen gestaltet werden. Dazu wird in einer bedingten Verzweigung festgestellt, ob sich das aktuelle Feld innerhalb vorgegebener Grenzen befindet (im Beispiel der folgenden Abbildung im „Mittelstreifen“, der horizontal und vertikal von -200 bis +200 reicht). Dann wird zufällig eines der „dunklen“ Kostüme gewählt. Außerhalb, also in den vier Eckbereichen, wird zufällig eines der „hellen“ Kostüme gewählt:

Werden die Kostüme in den zwei Bereichen nicht überlappend gewählt (Kostüme 1 - 10 vs. Kostüme 11 - 20), ergibt sich eine deutlich sichtbare Trennung (oben links). Diese löst sich um so mehr auf, je überlappender die Bereiche sind (Kostüme 1 - 14 vs.



Kostüme 7 - 20, oben rechts; Kostüme 1 -15 vs. Kostüme 6 - 20, unten links; Kostüme 1 - 16 vs. Kostüme 5 - 20, unten rechts). Im letzten Fall ist sie kaum noch erkennbar.



Anregung: *Sie können andere Verteilungen auch dadurch erreichen, dass sie den „Streifen" in der Ausdehnung verändern oder seine Lage nicht mittig bestimmen.*

In Bild 36 sind die schwarzen Kostüme der Dreieckstrukturen durch gleichartige Kostüme in den Farben Blau, Gelb und Rot ersetzt bzw. ergänzt worden. Je nachdem, ob nun eine bestimmte Kostümfarbe (1 - 20 ergibt Blau, 21 - 40 ergibt Gelb, 41 - 60 ergibt Rot) oder eine Farbkombination (mit 1 - 60 werden alle Farben berücksichtigt) gewählt wird, ergibt sich in Kombination mit der Hintergrundfarbe eine neue Ausprägung der Dreieckstrukturen.

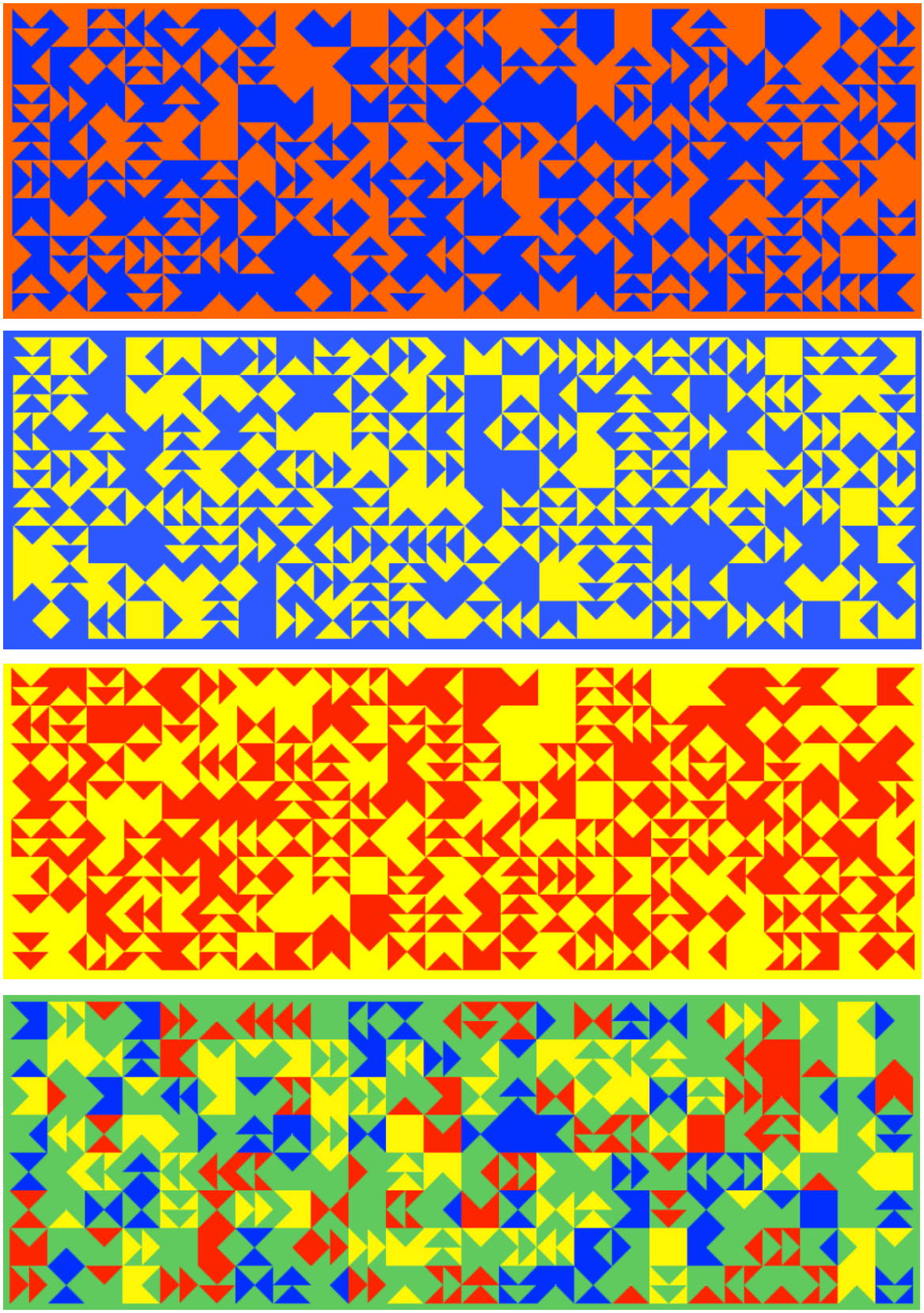
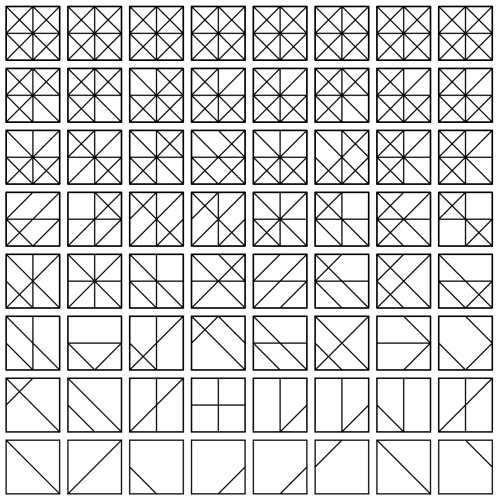


Bild 36: Farbremixing der Schwarz-Weiß-Strukturen

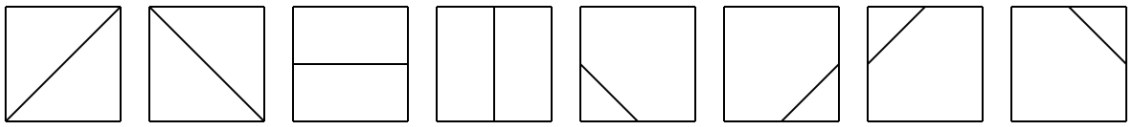
11.8 Hommage à Coqart: Structured Squares

Roger Coqart ist der Künstlername des belgischen Fotografen und Computerkünstlers Roger Kockaerts (geb. 1931 in Wilsele, Belgien). Der Computer half ihm beim *„Streben nach Objektivität im Konstruktionsprozess der geometrischen Abstraktion“*. Ein Schwerpunkt war für ihn die systematische Untersuchung von Gitterstrukturen aus Quadraten, denen er weitere Strukturen überlagerte. Diese Strukturen gewann er durch halbierende und diagonale Linienelemente. Die so generierten Quadrate fügte er wiederum zu einer quadratischen Matrix zusammen, wie die folgende Abbildung demonstriert.

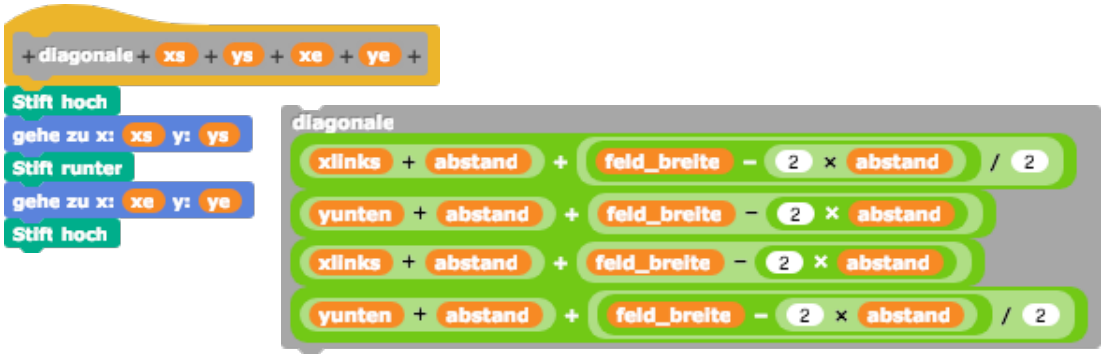


Quadratstrukturen haben wir schon mehrere kennen gelernt (vgl. die Beispiele von Molnar oder Sýkora). Im vorliegenden Fall soll die Erzeugung der Linienelemente durch eigene Prozeduren erfolgen, damit unterschiedliche Strichstärken und Linienfarben einfach durch Parameter gesteuert werden können. Die auch denkbare Verwendung von Kostümen wäre demgegenüber mit deutlichem Mehraufwand verbunden, weil sie ja dann für jede Farbe und Strichstärke vorgehalten werden müssten.

Basis ist ein m*n-Raster, wobei i.d.R. jedes Feld des Rasters von einem Quadrat umschlossen wird. Für die Strukturen in den Quadraten werden insgesamt acht Linienelemente benötigt; vier halbierende Linien und vier diagonale Linien von Quadratseitenmitte zu Quadratseitenmitte:



Für diese Elemente kann eine Prozedur **diagonale** definiert werden, die über entsprechende Wahl der Kenngrößen gesteuert wird, also x- und y-Koordinaten von Startpunkt (**xs**, **ys**) und Endpunkt (**xe**, **ye**) der Linie. Diese Punkte werden über die Verwendung der Kenngrößen **xlinks**, **yunten**, **feld_breite** und **abstand** (wie im Exkurs *Das m*n-Raster* im Kapitel 9: *Figurenbaukasten* definiert) für jeden Linientyp errechnet.



Der Aufruf von **diagonale** fällt dementsprechend länglich aus (im Beispiel für die Linie ganz rechts gezeigt).

Unabhängig davon, ob die Linienelemente nacheinander geordnet oder zufällig aufgerufen werden, ergibt sich bei den entsprechenden Aufrufen ein langer unübersichtlicher Code. Einfacher wäre es, diese Prozeduraufrufe in einer geordneten Liste zusammen zu fassen und einfach die Linienelemente aufzurufen. Das ist in Snap! möglich, wie im Folgenden gezeigt wird.

11.9 Exkurs: Prozeduren als Daten

Listen werden werden zwar erst im Kapitel 15: *Die Sprache G* als leistungsfähiges Konzept vorgestellt, um damit in Feldern Texte und Zahlen sowie ihrerseits Listen abzuspeichern. In Snap! sind aber außerdem auch Prozeduren als Listenelemente erlaubt[56]. So kann hier das Problem der umständlichen Prozeduraufrufe dadurch gelöst werden, dass sie als Elemente einer Liste gespeichert und von dort abgerufen werden.

Damit Prozeduren (oder Reporter oder Prädikate) so verwendet und von Texteingaben unterscheidbar werden, müssen sie von *Ringen* umschlossen werden. Diese werden

Das brauchen wir von **Snap!***:*





Die *Ringe* sind Behälter für Blöcke, Reporter oder Prädikate. Das bedeutet, dass bei ihrem Aufruf der Block, der Reporter oder das Prädikat zurückgegeben wird, nicht deren Ergebnis. So ergibt der Operator im Beispiel die Summe zurück, der Ring um den Operator dagegen den Operator selbst:



Zunächst wird der gewünschte Block in den Ring platziert. Dieser Ring kann dann als Element einer Liste oder entsprechenden Befehlen übergeben werden. Bei Bedarf können dem umringten Block auch Parameter übergeben werden.



[56] Es wird dann von Funktionen höherer Ordnung gesprochen: Funktionen können als Parameter an andere Funktionen übergeben oder als Rückgabewert verwendet werden. Das Arbeiten damit macht Snap! äußerst leistungsfähig und erlaubt die Nutzung weiterer wichtiger informatischer Konzepte, deren Behandlung den Rahmen dieser Einführung aber sprengen würden. Interessenten möchte ich auf die Kapitel *IV. First Class Lists* und *VI. Procedures as Data* des Snap!-Manuals verweisen.

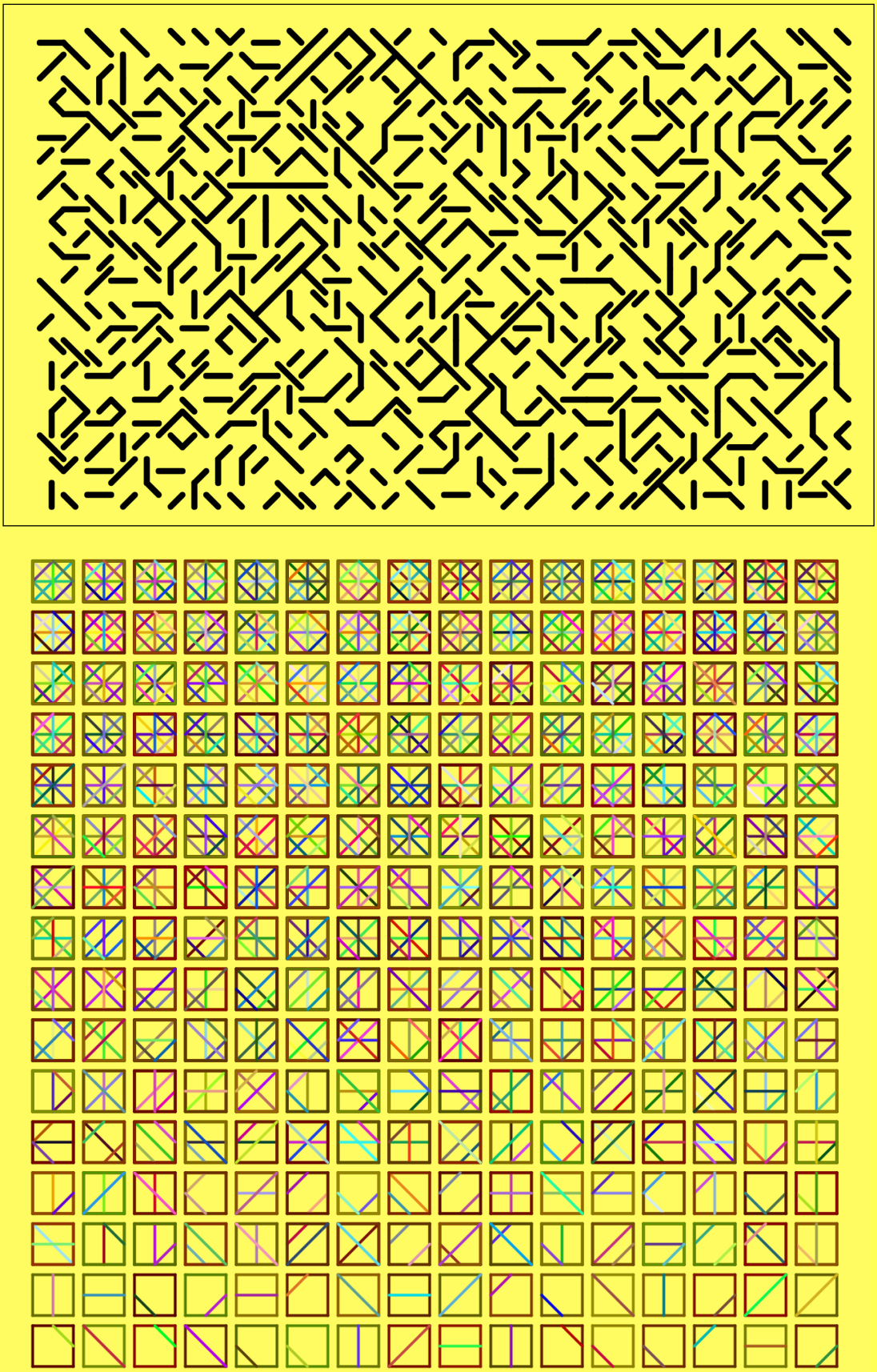


Bild 37: Remixing Coqart: Structured Squares (S/W ohne Raster; Farben)

dann in entsprechenden Blöcken (wie z.B. in **führe ... aus**) als Eingaben akzeptiert und ausgewertet.

In *Structured Squares* wird vor Ausführung der Schleifen für das m*n-Raster mit **mache_liste** eine **strichliste** auf diese Art erzeugt. Es sind dafür jeweils die Prozeduraufrufe **diagonale** mit den errechneten Kenngrößen zu erzeugen und diese Prozeduraufrufe dann zu *umringen*. Dieser Ring kann anschließend der **strichliste** mit dem Befehl **füge diagonale zu strichliste hinzu** eingetragen werden (die Abbildung zeigt dies als Beispiel für die erste oben gezeigte Linie).



Innerhalb der Schleife für das m*n-Raster wird dann der Befehl **führe ... aus** aufgerufen, wobei z.B. ein beliebiges Element der **strichliste** verwendet werden kann.

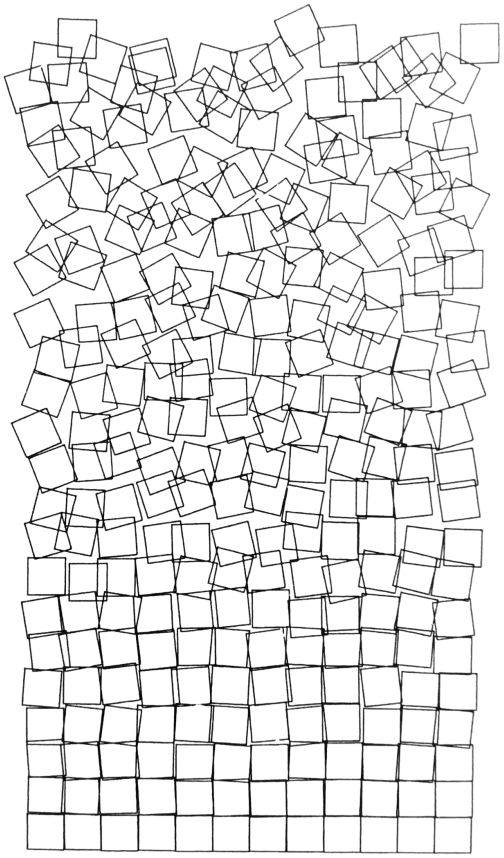


Hinweis: *Wenn - wie in* Bild 37 *- in bestimmten Zeilen des Rasters eine bestimmte Anzahl Linien ohne Wiederholung gezeichnet werden sollen, so sind die gezeichneten Linien jeweils aus der Liste zu entfernen (mit dem Befehl* **entferne Element aus strichliste**). *Am Ende der Schleife wird die ursprüngliche Liste durch* **mache_liste** *rekonstruiert für den nächsten Schleifendurchlauf.*

Anregung: *Die Zahl der in einem Feld gezeigten Linienelemente kann abhängig gemacht werden von der Position in der jeweils aktuellen Reihe bzw. Spalte. Die sich daraus ergebenden Muster sollten in ihrer Struktur den Bildern von Kolomyjec gleichen (vgl.* Bild 42*). Die dort eingeführten Prozeduren können dafür übernommen und angepasst werden.*

12. HOMMAGE À GEORG NEES: SCHOTTER

Mit dem bisher vorgestellten Arsenal an Befehlen und Funktionalitäten von Snap! können wir uns an das *Recoden & Remixen* weiterer konkreter Beispiele der frühen Computerkunst wagen.



Georg Nees: Schotter (aus Nees, 1969, Bild 38, S. 242)

Das nebenstehende Bild entstammt der Dissertation von Georg Nees (1926 - 2016) und trägt den Titel *Schotter* (Nees, 1969, S. 242). Für mich gibt es gleich drei Gründe, diesem Bild ein eigenes Kapitel zu widmen.

Zum einen ist es eines der bekanntesten Bilder der Computerkunst überhaupt, das in sehr vielen Publikationen wiedergegeben ist. Es stammt von einem der Pioniere der Computerkunst, der dazu (übrigens als einer der ganz Wenigen) seinen Algorithmus dokumentiert hat (Nees, 1969, S. 241 u. 186)[57].

Zum anderen war es das erste Werk der Computerkunst, mit dem ich mich überhaupt näher befasst habe - anfangs ohne zu wissen, dass es ein Kunstwerk von Nees war. Entdeckt hatte ich es - dort *Schotterhaufen* genannt - in einer Einführung in die Programmiersprache Logo (Stein, 1984, S. 135). Es ist dort eines von mehreren direkt in Anlehnung an Nees nachprogrammierten Grafikbeispielen[58]. Bei mir entstand daraus jedenfalls eine erste eigenständige Umsetzung mit der *Schildkrötengrafik* (Rauch & Wedekind, 1989, S. 40), einer Logo-Version, die im Rahmen eines Fernstudienprojekts *Lehren und Lernen mit dem Computer* entwickelt und vertrieben wurde.

Erst jetzt - 25 Jahre später - bin ich im Rahmen meiner Arbeiten zur frühen Computerkunst wieder darauf gestoßen. Dabei habe ich überrascht festgestellt, dass das Bild *Schotter* auf ganz unterschiedliche Weise realisiert worden ist. Das ist dann der dritte

[57] Im Kapitel 15: *Die Sprache G* werden wir den generellen Ansatz und die dafür entwickelte Grafikumgebung von Georg Nees kennen lernen.

[58] Das Buch von Nees taucht zwar in den Literaturangaben dieses Buches auf (aber ohne Nennung von Verlag und Jahrgang), im Text wird aber die Herkunft der sich über immerhin 70 Seiten erstreckenden Nachprogrammierung der Nees-Beispiele nicht erwähnt.

Grund für dieses Kapitel: Ich möchte diese unterschiedlichen Vorgehensweisen vorstellen und vergleichen.

Beginnen wir mit dem Originalprogramm von Georg Nees:

```
1   'BEGIN''COMMENT'SCHOTTER.,
2   'REAL'R,PIHALB,P14T.,
3   'INTEGER'I.,
4   'PROCEDURE'QUAD.,
5   'BEGIN'
6   'REAL'P1,Q1,PSI.,'INTEGER'S.,
7   JE1.=5*1/264.,JA1.=-JE1.,
8   JE2.=PI4T*(1+I/264).,JA2.=PI4T*(1-I/264).,
9   P1.=P+5+J1.,Q1.=Q+5+J1.,PSI.=J2.,
10  LEER(P1+R*COS(PSI),Q1+R*SIN(PSI)).,
11  'FOR'S.=1'STEP'1'UNTIL'4'DO'
12  'BEGIN'PSI.=PSI+PIHALB.,
13  LINE(P1+R*COS(PSI),Q1+R*SIN(PSI))
14  'END'.,I.=I+1
15  'END'QUAD.,
16  R.=5*1.4142.,
17  PIHALB.=3.14159*.5.,P14T.=PIHALB*.5.,
18  I.=0.,
19  SERIE(10.0,10.0,22,12,QUAD)
20  'END' SCHOTTER.,
```

```
1   'REAL'P,Q,P1,Q1,XM,YM,HOR,VER,JLI,JRE,JUN,JOB.,
5   'INTEGER'I,M,M,T.,
7   'PROCEDURE'SERIE(QUER,HOCH,XMAL,YMAL,FIGUR).,
8   'VALUE'QUER,HOCH,XMAL,YMAL.,
9   'REAL'QUER,HOCH.,
10  'INTEGER'XMAL,YMAL.,
11  'PROCEDURE'FIGUR.,
12  'BEGIN'
13  'REAL'YANF.,
14  'INTEGER'COUNTX,COUNTY.,
15  P.=-QUER*XMAL*.5.,
16  Q.=YANF.=-HOCH*YMAL*.5.,
17  'FOR'COUNTX.=1'STEP'1'UNTIL'XMAL'DO'
18  'BEGIN'Q.=YANF.,
19  'FOR'COUNTY.=1'STEP'1'UNTIL'YMAL'DO'
20  'BEGIN'FIGUR.,Q.=Q+HOCH
21  'END'.,P.=P+QUER
22  'END'.,
23  LEER(-148.0,-105.0).,CLOSE.,
24  SONK(11).,
25  OPBEN(X,Y)
26  'END'SERIE.,
```

Von Nees wird der Code wie folgt kommentiert:

„[Schotter] wird durch einen Aufruf der Prozedur SERIE erzeugt [...]. Zur Genese der Elementarfigur, die in dem von SERIE gesteuerten Kompositionsprozeß vervielfacht wird, dient die parameterlose Prozedur QUAD. In den Zeilen 4 bis 15 des Generators wird QUAD vereinbart, diese Prozedur

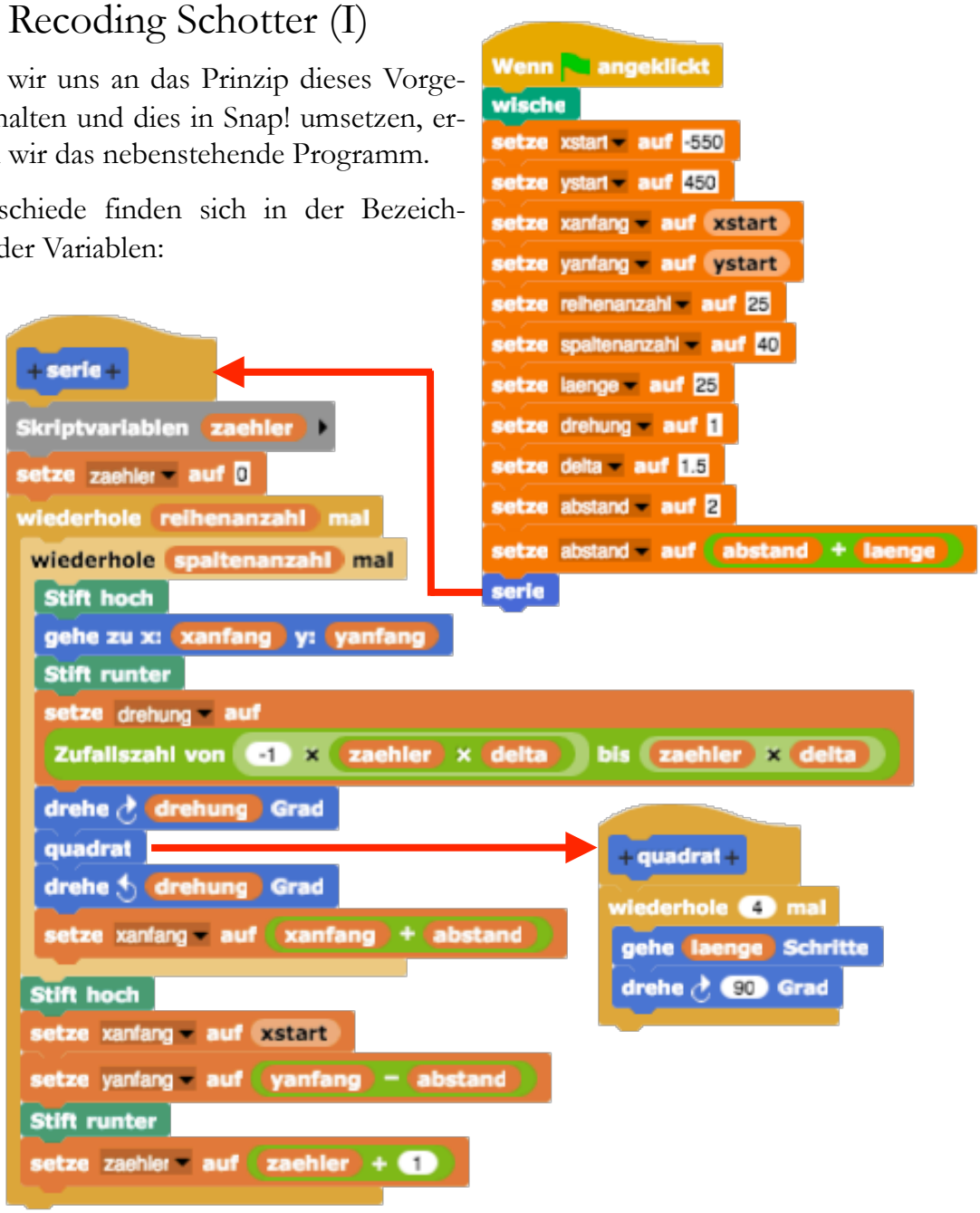
zeichnet Quadrate konstanter Seitenlänge, jedoch zufälliger Lage und Winkelstellung. Man erkennt aus den Doppelzeilen 9 und 10, daß die Position des Einzelquadrats vom Zufallsgenerator J1, die Winkellage von J2 beeinflußt wird. Die sukzessive verbreiterte Streuung der relativen Ortskoordinaten P und Q und des Lagewinkels PSI des einzelnen Quadrats wird durch einen Zähler I gesteuert, der bei jedem Aufruf von QUAD weitergeschaltet wird (siehe Zeile 14).“

Ohne auf die Einzelheiten einzugehen, ist in der Prozedur **SERIE** (grün markiert) leicht ersichtlich, dass in zwei verschachtelten Schleifen (blau markiert) die von der Prozedur **QUAD** (rot markiert) erzeugten Quadrate Reihe für Reihe mit zufälliger Lage und Winkelstellung dargestellt werden.

12.1 Recoding Schotter (I)

Wenn wir uns an das Prinzip dieses Vorgehens halten und dies in Snap! umsetzen, erhalten wir das nebenstehende Programm.

Unterschiede finden sich in der Bezeichnung der Variablen:



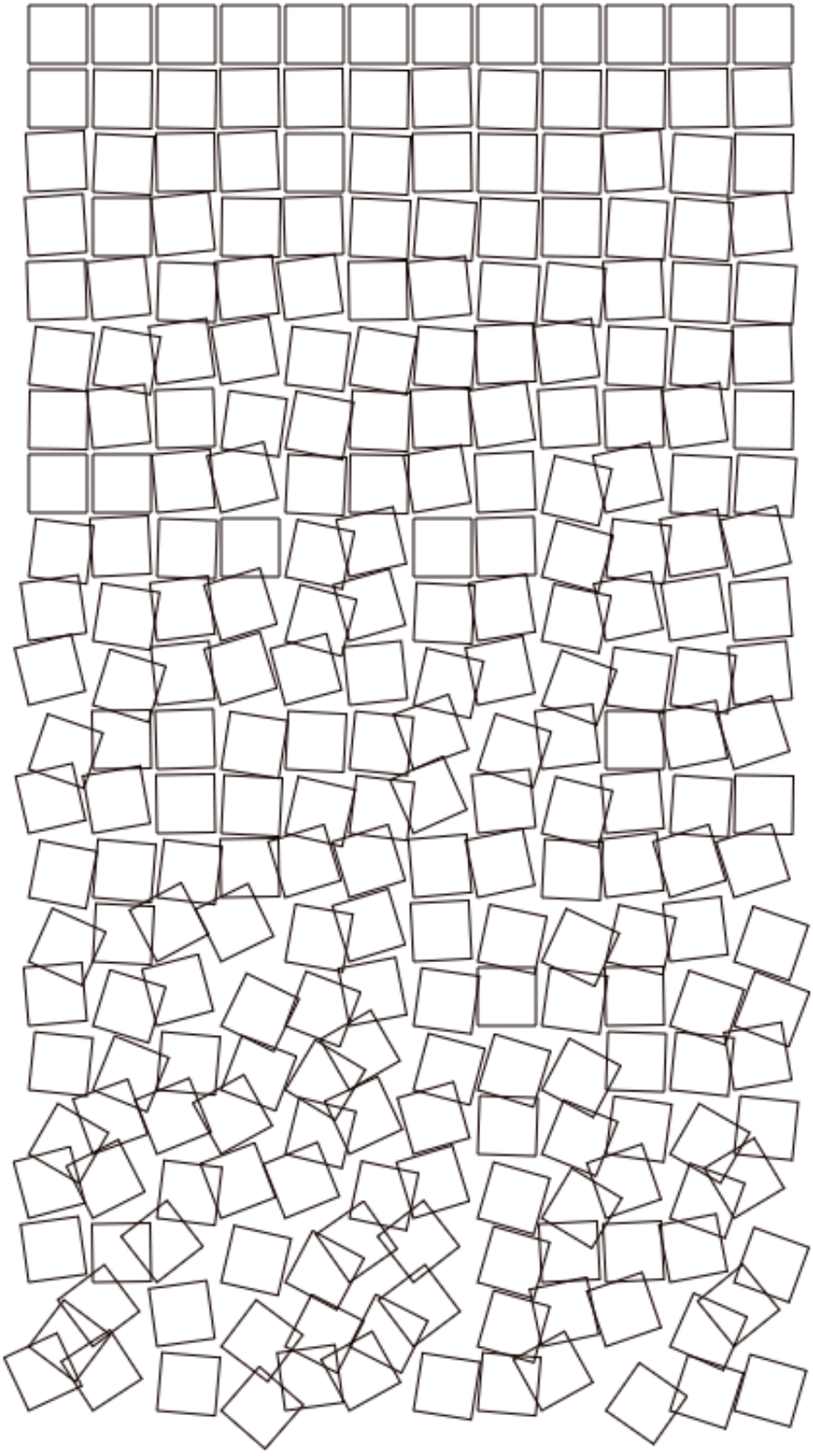


Bild 38: Hommage à Nees: Schotter

- **xstart, ystart**: Anfangspunkt des Zeichenvorgangs
- **xanfang, yanfang**: Anfangspunkt einer Quadratreihe
- **spaltenanzahl**
- **reihenanzahl**
- **laenge**: Seitenlänge der Quadrate
- **drehung**: Drehwinkel eines Quadrats
- **delta**: Grad der Veränderung des Drehungswinkels von Reihe zu Reihe
- **abstand**: horizontaler und vertikaler Abstand zwischen den einzelnen Quadraten

Eingangs werden die Anfangswerte dieser Variablen festgelegt und dann der Block **serie** aufgerufen.

In **serie** werden in der ersten Schleife die Reihen durchlaufen, in der zweiten Schleife die Quadrate pro Spalte gezeichnet. **quadrat** ist die uns seit dem Einführungskapitel wohlbekannte Prozedur (bei Nees wird dies im kartesischen Koordinatensystem gezeichnet).

Trotz der kleinen Abweichungen im Detail erhalten wir ein dem Original sehr ähnliches Bild (Bild 38). Unser Bild wird aber - anders als bei Nees - mit von oben nach unten pro Reihe zunehmend abweichender Winkelstellung der Quadrate gezeichnet. Dadurch wird das Original von Nees quasi auf den Kopf gestellt (weil meine Assoziation dabei ist, wie Schotter von einem Laster gekippt wird).

12.2 Recoding Schotter (II)

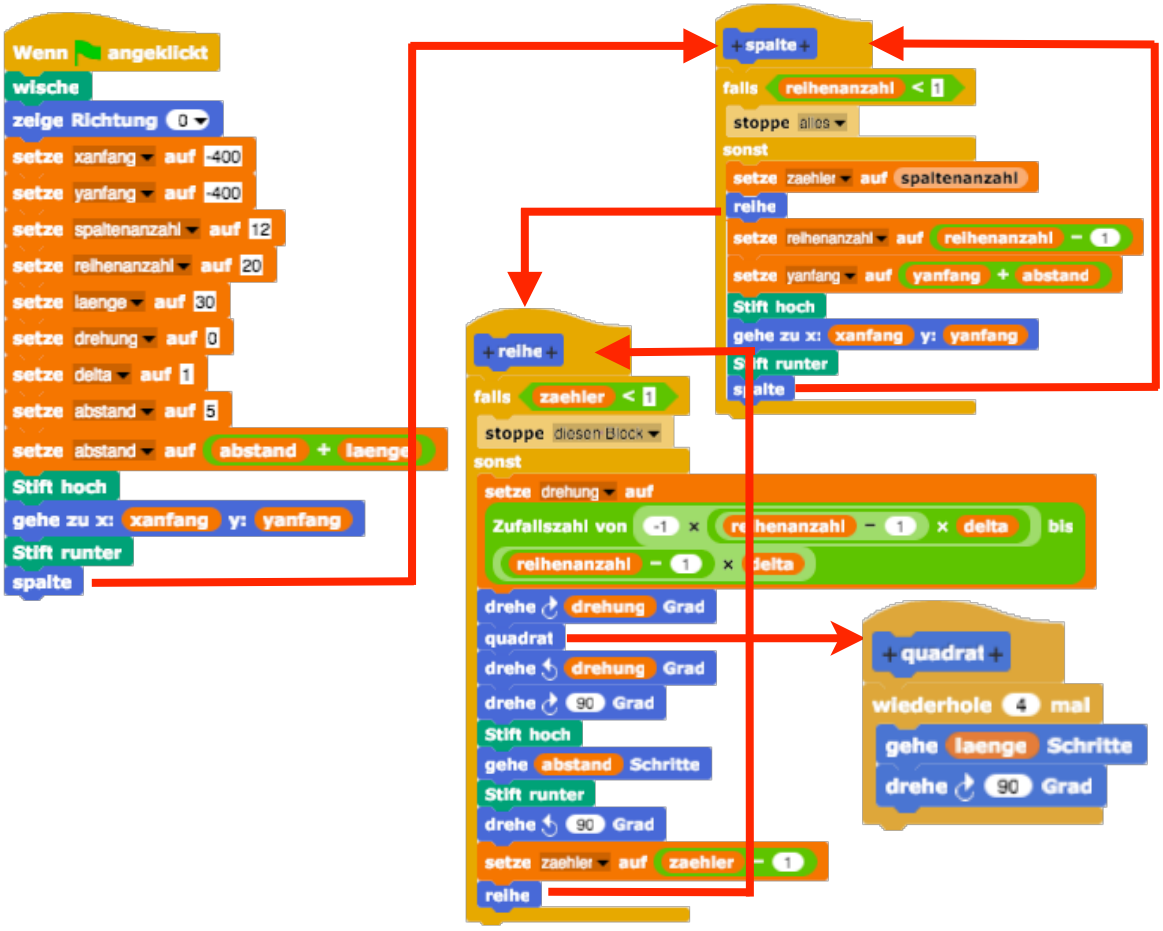
Meine programmtechnisch eigenständige Version des Schotter-Programms sieht etwas anders aus. Bei gleicher Bezeichnung der Variablen ergibt sich eine verschachteltere Abfolge, wie auf der folgenden Seite gezeigt.

Vorab werden wieder die Anfangswerte der Variablen festgelegt. Dann aber wird im *rekursiv* abzuarbeitenden Block **spalte** seinerseits der ebenfalls *rekursiv* abzuarbeitende Block **reihe** aufgerufen (die *Rekursion* erfährt eine ausführlichere Darstellung im Kapitel 13: *Ein Block ist ein Block ist ein Block* ...). Dort erfolgt die Positionierung des Quadrats, das dann wieder vom Block **quadrat** gezeichnet wird.

Das brauchen wir von **Snap!**:



Mit dem Befehl **stoppe alles** wird das aktuelle Programm beendet (entspricht damit dem Drücken des **Beenden**-Schalters in der Werkzeugleiste).
Bei Auswahl von **dieses Skript** bzw. von **diesen Block** kann ein einzelnes Skript oder ein Block beendet werden. Der restliche Programmablauf bleibt davon unberührt.



Sowohl **spalte** als auch **reihe** werden jeweils solange rekursiv aufgerufen, bis das Abbruchkriterium erreicht wurde. Bei **reihe** wird danach der Block beendet und die nächste Spalte gezeichnet. Wird bei **spalte** das Abbruchkriterium erreicht, ist das Programm fertig und wird beendet. Auch diese Umsetzung liefert dann ein Bild, das von der vorherigen Version kaum zu unterscheiden ist.

12.3 Remixing Schotter: Formen und Farben

Von Georg Nees ist mir das Bild *Schotter* vor allem in der Version aus seiner Dissertation bekannt, obwohl mehrere Varianten davon - auch in unterschiedlichen Größen - in Publikationen über ihn bzw. das Bild dokumentiert sind[59]. Sie zeigen alle die Quadrate mit den von Nees vorgegebenen 22 Reihen und 12 Spalten. An diesen Vorlagen orientiert wurde *Schotter* mehrfach von Designern und Programmierern nachprogrammiert[60] und abgewandelt[61].

[59] z.B. bei http://www.medienkunstnetz.de/werke/schotter/ oder http://dada.compart-bremen.de/item/artwork/379

[60] Von Jim Plaxco gibt es ein Tutorial zur Programmierung von *Schotter* mit *Processing*. Dort findet sich auch eine davon inspirierte Bildserie *Cubic Disarray*: http://www.artsnova.com/Nees_Schotter_Tutorial.html

[61] Bei *variart* können online individuelle Varianten erzeugt (und bei Gefallen bestellt) werden; der Programmcode ist auf der Webseite einsehbar. http://variart.io/pieces/schotter



Auch hier sollen einige Möglichkeiten für das *Remixing* angesprochen werden. Eine erste Form ist schon die Änderung der **spaltenanzahl** und/oder der **reihenanzahl**. Neue Ausprägungen des Grundmusters ergeben sich jeweils auch durch die Änderung von **delta**, also dem Grad der zufälligen Veränderung des Drehwinkels von Reihe zu Reihe.

Hinweis: *Für die folgenden Bildbeispiele habe ich weder das Originalprogramm von Nees aus* Recoding Schotter (I) *noch meine rekursive Fassung aus* Recoding Schotter (II) *verwendet. Statt dessen habe ich das bewährte m*n-Raster zu Grunde gelegt. Dadurch wird einerseits die Flexibilität des Programms Schotter deutlich erhöht. Andererseits können wir die Grundstruktur des Programms mit Vorspann und Wiederholungsschleifen direkt übernehmen, z.B. von* Molnar: 25 Quadrate.

Neben den schon bekannten Kenngrößen des m*n-Rasters sind folgende Festlegungen vom *Recoding* für die flexible Gestaltung der Schotterbilder zu übernehmen:

- **delta_d**: Grad der Veränderung des Drehwinkels von Reihe zu Reihe
- **abstand**: horizontaler und vertikaler Abstand zwischen den einzelnen Quadraten

Eine echte Erweiterung des Programs *Schotter* ist der Ersatz der Quadrate durch *Vielecke* (wie in Bild 39 oben). Dafür verwenden wir einen Block **n-eck um x y n radius** (abgeleitet vom Block **punktkreis um x y** im Abschnitt *Punkte* des Kapitels 9: *Figurenbaukasten*). Zur Steuerung sind dafür weitere Kenngrößen festzulegen:

- **n_ecken:** Gibt die Zahl der Ecken eines Vielecks an.
- **delta_xy:** Da mit **n-eck ...** die Vielecke jeweils um den Mittelpunkt eines Rasterfeldes gezeichnet werden, sind hiermit horizontale und vertikale Abweichungen zu steuern.

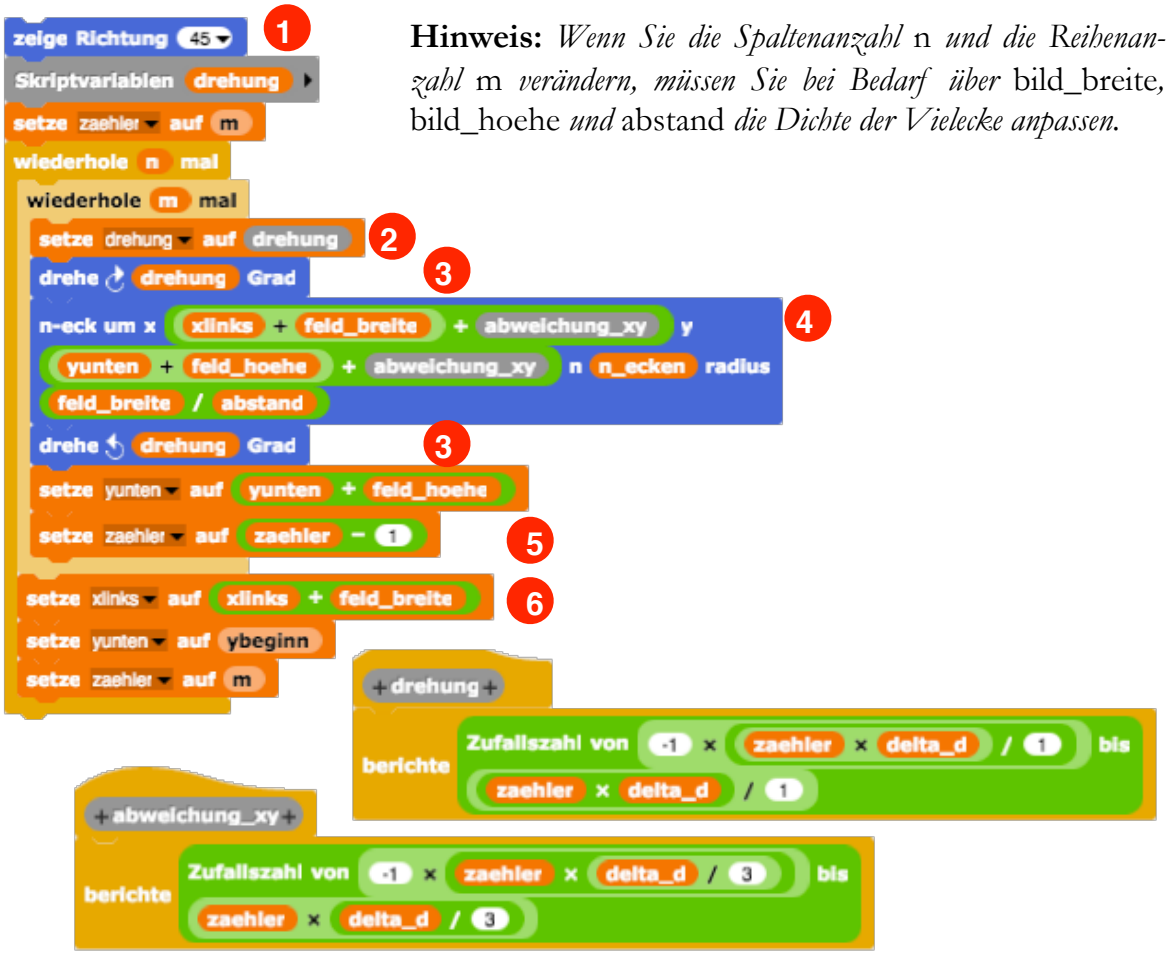
So hat das nebenstehende Beispiel die Kenngrößen **bild_breite = 80, bild_hoehe = 1200**, **n = 3**, **m = 44**, **n_ecken = 4**, **delta_d = 1**, **delta_xy = delta_d/3** und **abstand = 1.5**.

Für die Darstellung der Vielecke in der inneren Schleife des m*n-Rasters sind weitere (Skript-)Variablen als Zwischenspeicher einzuführen:

- **zaehler:** Gibt die aktuelle Reihe an und wird zur Berechnung der Zufallszahlen in Abhängigkeit von der vertikalen Position benötigt.
- **drehung:** Grad der Veränderung des Drehwinkels von Reihe zu Reihe.

Die beiden Schleifen über die Spalten und Reihen enthalten dann die folgenden Befehlsfolgen:

1. Je nach Vieleck kann die Grundrichtung festgelegt werden. Bei **n_ecken = 2** ergibt z.B. **zeige Richtung = 0** senkrechte Linien, **zeige Richtung = 90** waagrechte Linien.
2. Für die **drehung** wird eine Zufallszahl in Abhängigkeit von der aktuell erreichten Reihe ermittelt.
3. Vor dem Zeichnen des Vielecks erfolgt die **drehung**, die nach dem Zeichnen zurück genommen wird.
4. Für das Zeichnen des Vielecks werden **x** und **y** mit zufälligen Abweichungen versehen.
5. Für die Ermittlung der aktuellen Reihe wird der **zaehler** heruntergezählt.
6. Für die folgende Spalte werden Anfangskoordinaten und der **zaehler** zurück gesetzt.



Hinweis: *Wenn Sie die Spaltenanzahl* n *und die Reihenanzahl* m *verändern, müssen Sie bei Bedarf über* bild_breite, bild_hoehe *und* abstand *die Dichte der Vielecke anpassen.*

Die Ermittlung der Zufallszahlen für die zufällige Drehung und die zufälligen horizontalen und vertikalen Abweichungen der Vielecke erfolgt in den Reporter-Blöcken **drehung** bzw. **abweichung_xy**. In beiden Fällen kann das Produkt **zaehler*delta_d** über den Quotienten nach Bedarf nachjustiert werden.

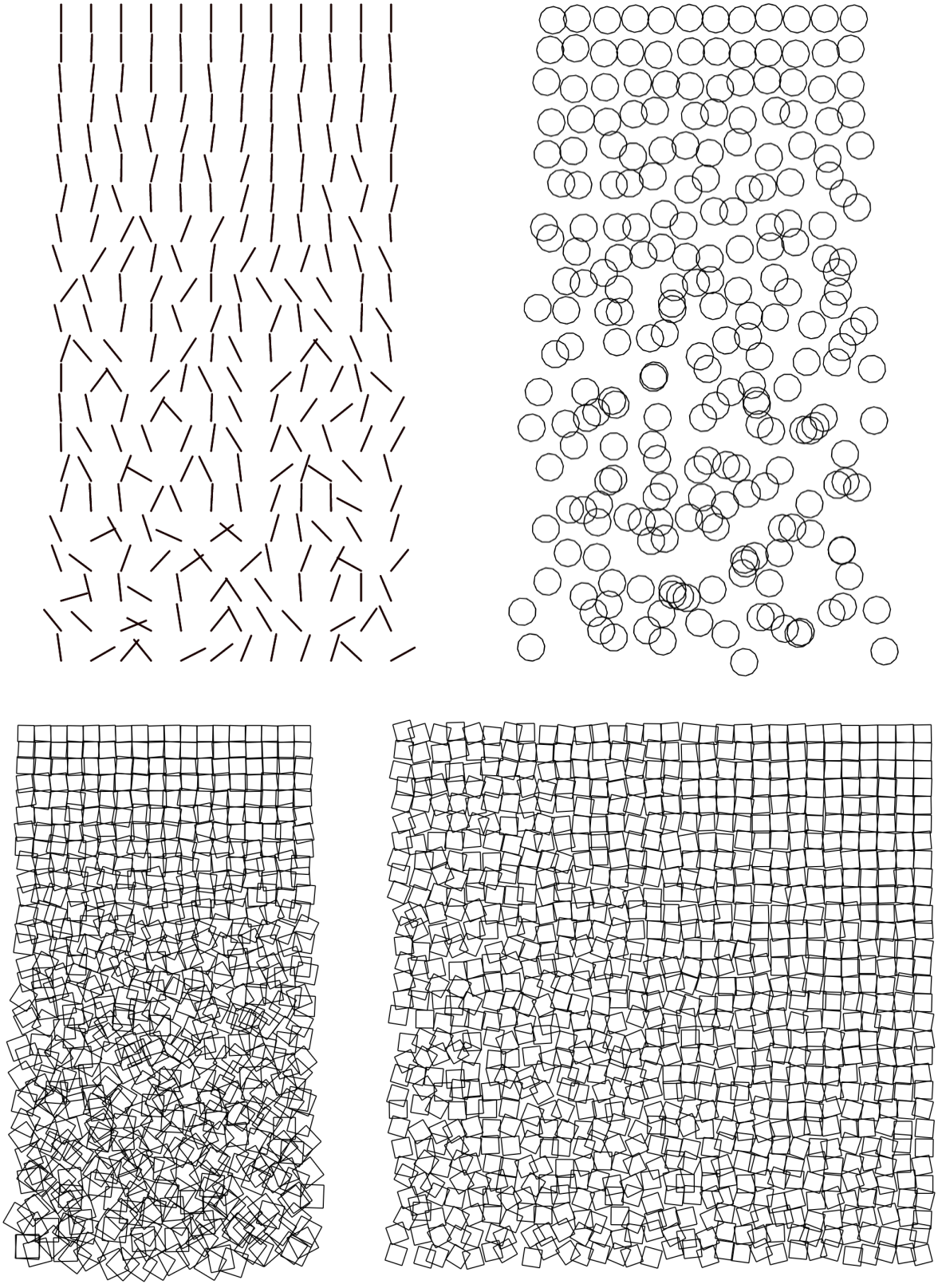


Bild 39: Remixing Schotter

Im vorgestellten Beispiel bleibt der Quotient bei der **drehung** unverändert. Für die **abweichung_xy** wird er gedrittelt. Das „Wackeln" der Vielecke und ihr „Auseinanderdriften" ist darüber präzise zu steuern.

Mit dem Zufallseinfluss kann weiter experimentiert werden. Z.B. kann das Produkt über **zaehler* zahl* delta_d** verändert werden, wobei **zahl** eine beliebige positive Zahl sein kann. Auch der Wert **zaehler** kann eingesetzt werden, wodurch ein quadratischer Einfluss der Zählvariable erreicht wird.

Die Beispiele in Bild 39 wurden mit **n_ecken = 2** (links oben) bzw. mit **n_ecken = 12** und **richtung 0** (rechts oben) erstellt.

Mit *Schotter* hat Georg Nees ein ästhetisches Objekt geschaffen, dessen Aufbau eigentlich sehr einfach ist. Das zentrale Merkmal ist der kontrollierte, graduelle Übergang von starker Ordnung in Richtung zunehmender Unordnung. Dieses einfache Grundkonzept von *Schotter* hat andere Künstler für ein eigenes *Remixing* angeregt.

Für Robert J. Krawczyk bot es einen perfekten Rahmen, um „Unvollkommenheit" zu schaffen. Das *Recoding* und *Remixing* von *Schotter* führte ihn zur Entwicklung dreidimensionaler Skulpturen nach genau diesem Prinzip (Krawczyk, 2002).

Von Krawczyk stammt eine Variation von Schotter, bei der neben den wachsenden Drehungen und horizontalen bzw. vertikalen Abweichungen zusätzlich auch die Größe der Quadrate zunimmt (Bild 39 unten links). Die Ergänzung um eine solche Zunahme (oder auch Abnahme) ist mit unserem Schotter-Programm leicht zu bewerkstelligen (vorige Abbildung unten links). Dasselbe gilt, wenn die Zunahme nicht nur von oben nach unten, sondern z.B. auch diagonal erfolgen soll (Bild 39 unten rechts).

Anregung: *Für die Variante mit diagonaler Zunahme können zwei Zähler (z.B.* zaehler_m *und* zaehler_n*) definiert werden, die aufsummiert und über die Gesamtzahl* m + n *aller Reihen und Spalten normiert werden. Dieser Wert kann dann in* drehung *bzw.* abweichung_xy *verrechnet werden. Für die wachsenden Quadrate reicht es, beim Aufruf von* **n-eck** *den Radius in Abhängigkeit von* zaehler *zu vergrößern.*

Eine weitere Stufe des *Remixing* von *Schotter* ist die Einführung von Farbe. Das wird erreicht durch das Zeichnen farbgefüllter Vielecke, wobei davor jeweils die gewünschte Farbe festzulegen ist. In Bild 40 wurde dazu das RGB-Modell verwendet. Die Farbverläufe (untere Grafik) wurden über den **zaehler** der horizontalen Position gesteuert:



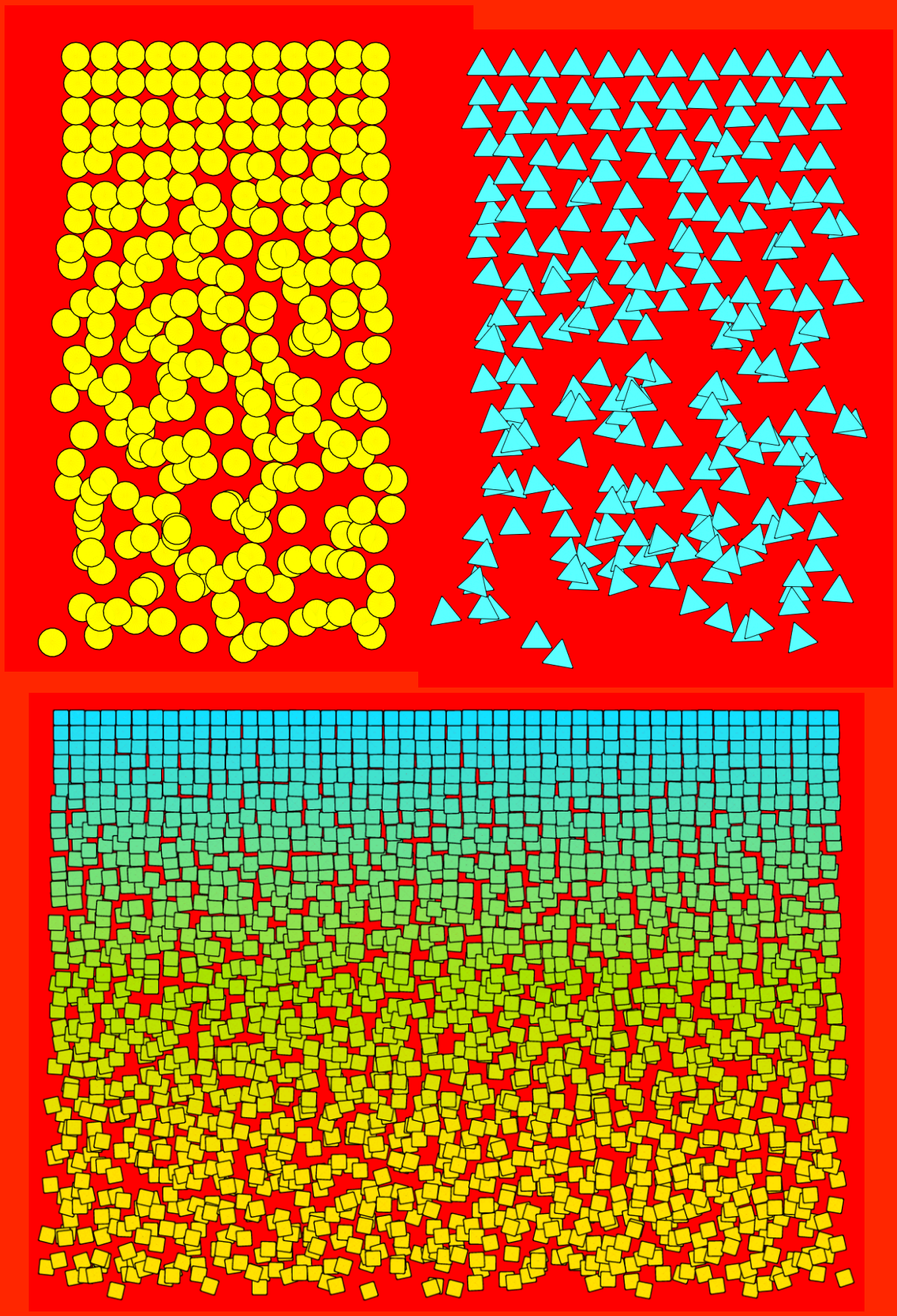


Bild 40: Remixing Schotter: Formen und Farben

Weitere *Schotter*-Varianten, die hier Ausgangspunkt eines eigenen *Remixing* sein sollen, stammen von William Kolomyjek. Für den Nachvollzug der Arbeit von Kolomyjek ist nun wiederum die rekursive Fassung aus *Recoding Schotter (II)* besser geeignet als das Original in *Recoding Schotter (I)*. Im nächsten Kapitel soll deshalb zunächst die *Rekursion* näher betrachtet werden.

Das brauchen wir von **Snap!***:*



Hinweis: Die folgenden Befehle gehören zur Blockbibliothek, die nachgeladen werden kann und in *Anhang C* beschrieben wird.



Mit dem Befehl **n-eck um x y n radius** zeichnet die Schildkröte ein Vieleck mit **n** Ecken um den durch **x** und **y** festgelegten Mittelpunkt mit dem angegebenen **radius**. Die Schildkröte befindet sich nach Zeichnen des Kreises wieder am Ausgangspunkt mit ihrer ursprünglichen Richtung.



Da das Zeichnen eines Vielecks immer am ersten Eckpunkt in Blickrichtung der Schildkröte beginnt (bei oberer Abbildung 0 Grad), ist bei Bedarf über **zeige richtung** die Ausrichtung des Vielecks festzulegen (bei rechter Abbildung 45 Grad)!





Der Befehl **n-eck gefuellt um x y n radius** ist identisch mit dem vorigen Befehl; allerdings wird hier das Vieleck mit der aktuellen Stiftfarbe gefüllt. Die Schildkröte befindet sich nach Zeichnen des Kreises wieder am Ausgangspunkt mit ihrer ursprünglichen Richtung. Der Befehl **n-eck gefuellt ...** verwendet intern den Befehl **male aus**.

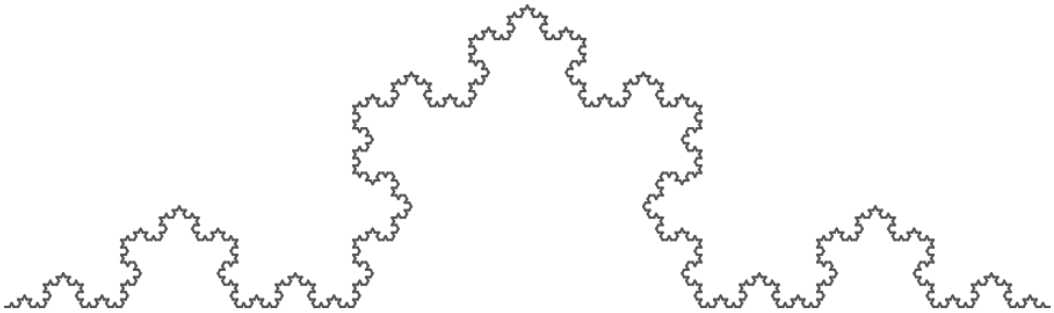




Hinweis: Wenn Linien oder Flächen den Bereich von **n-eck gefuellt um ...** schneiden, endet die Farbfüllung mit **male aus** bereits dort. Um auch in solchen Fällen ein vollständig gefülltes Vieleck zu erhalten, gibt es zusätzlich den Befehl **n-eck-rekursivgefuellt um x y n radius**. Bei diesem Befehl werden Vielecke mit schrumpfendem **radius** um **x y** gezeichnet und so die Farbfüllung sichergestellt.

13. EIN BLOCK IST EIN BLOCK IST EIN BLOCK ...

Das mehrfache Wiederholen eines Befehls oder einer Befehlsabfolge wird als *Iteration* bezeichnet. Davon haben wir bereits mehrfach Gebrauch gemacht und der **wiederhole**-Befehl dient ja genau diesem Zweck. Die Anzahl der Wiederholungen haben wir immer vorher festgelegt. Es gibt aber noch eine andere elegante Möglichkeit - die *Rekursion*, die beim *Recoding* von Schotter bereits zum Einsatz kam. Ihr Prinzip ist nicht immer ganz leicht zu durchschauen. Ich möchte sie deshalb am Beispiel der Koch-Kurve vorstellen, die nicht typisch für die Computerkunst ist, sondern die Visualisierung einer einfachen Konstruktionsregel darstellt. Das Ergebnis einer fünffachen Anwendung dieser Regel zeigt die folgende Abbildung:



Wie kommen wir zu dem Ergebnis? Die Konstruktion erfolgt im folgenden iterativen Prozess:

- Zeichne eine Strecke gegebener Länge.
- Teile die Strecke in drei gleiche Teile.
- Zeichne über der mittleren Strecke ein gleichseitiges Dreieck.
- Wiederhole dieses Vorgehen mit jeder der nun vier Strecken.

Ausgangspunkt ist hierbei also die Bewegung der Schildkröte um eine Strecke bestimmter **Länge** mit dem Befehl **gehe Länge Schritte** (folgende Abbildung links). Diese Bewegung wird dann ersetzt durch einen Umweg, in diesem Fall durch vier gleich lange Strecken, die eine dreieckige Ausbuchtung ergeben (rechts).



Wie kann das erreicht werden? Konkret wird für diese Ausbuchtung der einfache Befehl **gehe Länge Schritte** durch den Befehl **Umweglinie mit Länge** ersetzt.

1. Nach der ersten Teilstrecke
2. wird die zweite Teilstrecke mit den Winkeln eines gleichseitigen Dreickes in zwei Schritten gezeichnet.
3. Daran schließt sich die vierte Teilstrecke an.

Wenn nun bei jeder der Teilstrecken wiederum dasselbe Umwegeprinzip angewandt wird, entsteht die nächste Verfeinerung der Koch-Kurve:





Die Anwendung dieses Prinzips kann weiter fortgesetzt werden - und führt so nach fünf Wiederholungen zum eingangs gezeigten Bild. Die Formulierung dieser Wiederholungen wäre allerdings nicht mehr trivial, müssten doch bei jeder Teilstrecke bis zur gewünschten Verfeinerungstiefe alle Umwege durchlaufen werden.

Bei der Rekursion wird die Aneinanderreihung solcher Schleifen vermieden: Die *Rekursion* bezeichnet die Verwendung einer Prozedur (in Snap! also den Aufruf eines Blocks) durch sich selbst. In der Prozedur **Koch Kurve** werden dazu die vier Befehle **gehe Länge/3 Schritte** der Prozedur **Umweglinie** ersetzt durch den viermaligen Aufruf der Prozedur **Koch Kurve mit Tiefe und Länge**, d.h. Befehlsfolge 1 - 4.



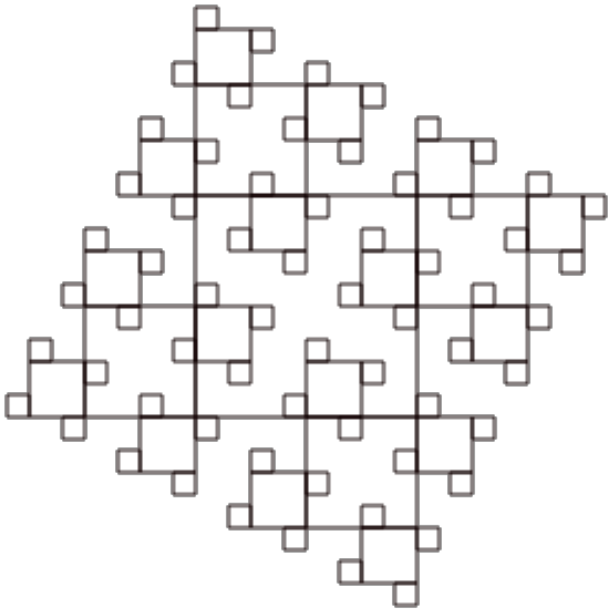
Es gilt, zwei wichtige Besonderheiten zu beachten: Die Zahl der rekursiven Aufrufe wird die *Tiefe* der Rekursion genannt. Bei jedem der Aufrufe ist **Tiefe** um 1 herunterzuzählen, damit alle Verfeinerungsstufen nacheinander durchlaufen werden. Die dabei jeweils notwendige Drittelung der Teilstrecken wird bei jedem rekursiven Aufruf durch den berechneten Wert **Länge/3** erreicht. Zum zweiten erfolgt eingangs des Blocks die

Abfrage, ob die **Tiefe** Null erreicht worden ist, weil dann die Prozedur mit dem Zeichnen der letzten „vierten Teilstrecke“ beendet werden soll.

Wichtiger Hinweis: *Jede rekursive Prozedur muss eine solche Abbruchbedingung enthalten, weil sie sonst endlos weiter läuft! Dabei wird geprüft, ob ein vorgegebenes Abbruchkriterium erfüllt wird, nach dessen Eintreten der Selbstaufruf enden soll.*

Im folgenden Beispiel soll die Rekursion noch einmal - diesmal anhand der Erzeugung einer interessanten Quadratstruktur (Bild 41) - mit der Iteration verglichen werden. Ein ähnliches Beispiel findet sich bei Clayson (1988, S. 92 ff.).





Wir sehen bei der oben gezeigten iterativen Lösung, dass

1. die innere Schleife (mit dem bereits bekannten Zeichnen eines Quadrats) von drei gleichartigen Schleifen umschlossen ist, die jeweils genau dieselben Aktionen durchführen:
2. Die vorgeschaltete Vorwärtsbewegung der Schildkröte, allerdings mit jeweils veränderter Schrittzahl und
3. die anschließende Drehung um 90 Grad.

Mit der Rekursion werden diese gleichartigen Schleifen vermieden und zusammengefasst. Wir erhalten dann folgende Neufassung der Prozedur **quadratmuster**:

1. Wir erkennen in der Schleife wieder das Zeichnen einer Seite und die Drehung um 90 Grad.

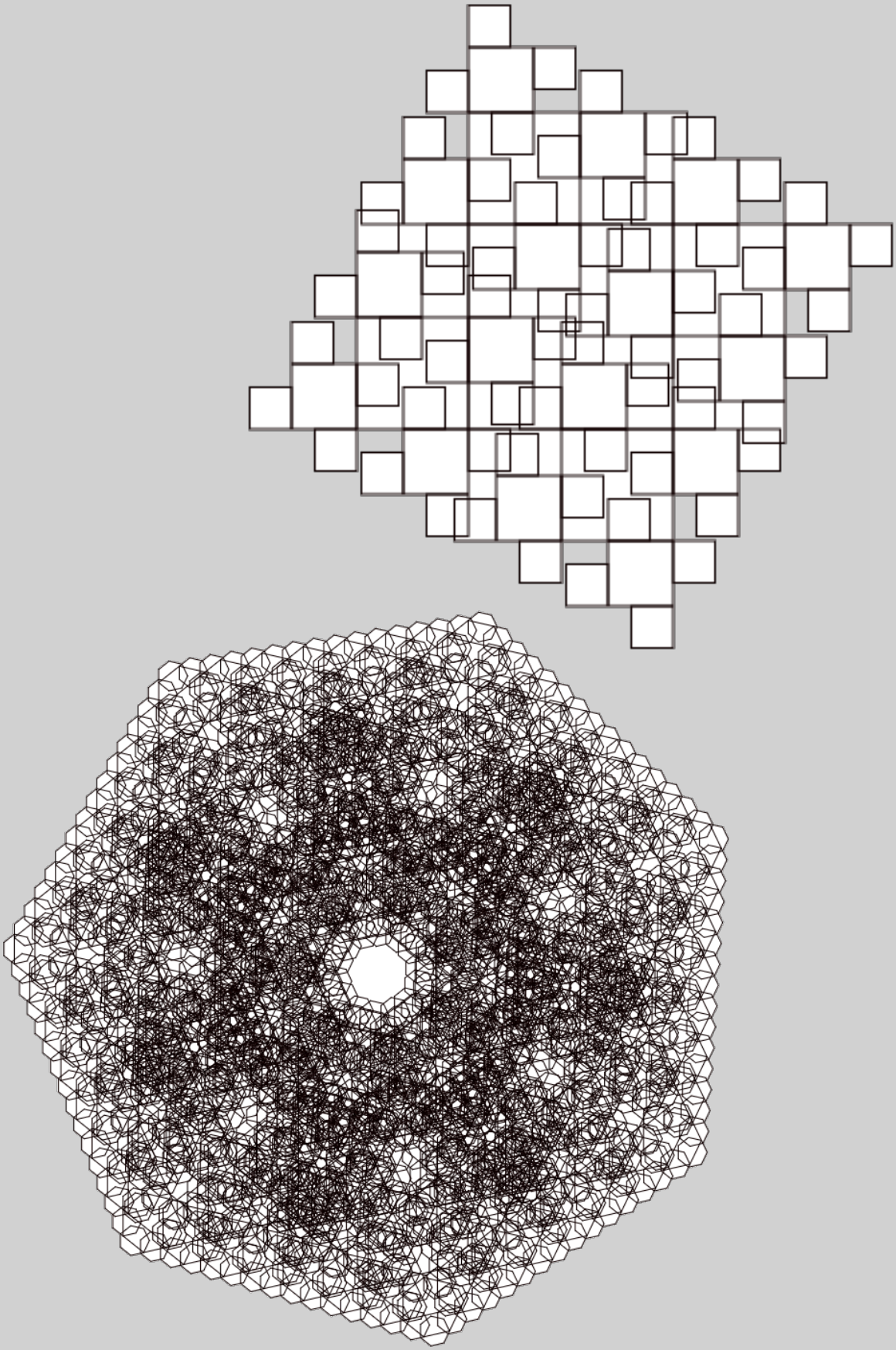


Bild 41: Vielecke rekursiv I & II



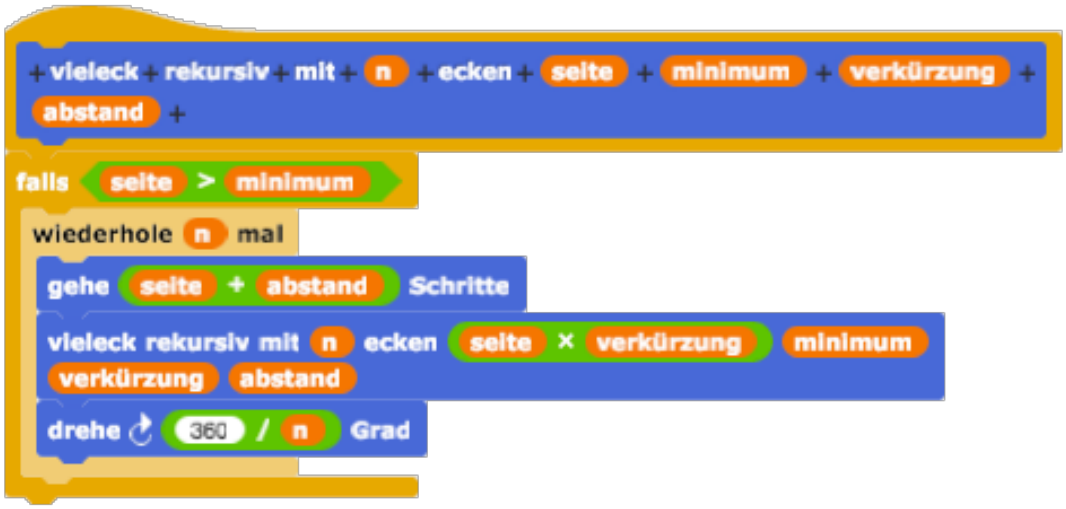
2 Dazwischen erfolgt aber nun der rekursive Aufruf von **quadratmuster**. Die Halbierung der Quadratseiten von Aufruf zu Aufruf wird bei jedem rekursiven Aufruf durch den berechneten Wert **seite/2** erreicht.

3 Eingangs des Blocks erfolgt die Abfrage, ob das vorgegebene Abbruchkriterium erfüllt wird, d.h.: Wird der minimale Wert für **seite** unterschritten, soll die Prozedur beendet werden.

Mit dem Aufruf **quadratmuster rekursiv mit seite** wird die gleiche Figur gezeichnet wie mit **quadratmuster iterativ**. Dies ist ein Beispiel dafür, wie sich im Prinzip jeder iterative Algorithmus in einen rekursiven Algorithmus umwandeln lässt (und umgekehrt).

Die Befehlsfolge wird bei der Iteration mehrmals schrittweise nacheinander zur Ausführung gebracht. Bei der Rekursion wird sie ineinander geschachtelt. Die Rekursion ist ein anspruchvolles Prinzip, das oft die elegantere Formulierung der Lösung erlaubt, dafür aber manchmal gedanklich schwierig nachzuvollziehen ist. Beide Vorgehensweisen stehen für das grundlegende Prinzip der *Zerlegung eines Problems* in möglichst überschaubare *Teilprobleme*.

Die Verallgemeinerung kann durch Parametrisierung nun noch ein wenig weiter getrieben werden. Statt Quadraten sollen nun Vielecke mit **n** Ecken verwendet werden, wobei mit **minimum** die minimale Seitenlänge als Abbruchkriterium und mit **verkürzung** die jeweils neue Seitenlänge festgelegt werden. Mit **abstand** wird außerdem auf jeder Aufrufebene die Seitenlänge vergrößert.



Die beiden Beispiele in Bild 41 geben einen Eindruck, welche Vielfalt der Strukturen sich bereits mit diesem kurzen Algorithmus generieren lassen. Manche weisen durchaus Ähnlichkeiten mit Arbeiten von Auro Lecci auf (siehe Bild 2). Die Beispiele werden mit den Aufrufen **vieleck rekursiv mit 4 ecken 100 10 0.5 10** (oben) bzw. **vieleck rekursiv mit 7 ecken 220 10 0.5 0** (unten) erzeugt.

Tipp: *Wenn Sie Bildvarianten erzeugen wollen, ist es empfehlenswert nur jeweils einen Parameter systematisch zu variieren, damit Sie ein Gefühl für die jeweiligen Auswirkungen bekommen. Bei einer* verkürzung > 0.5 *erhalten Sie sehr „dichte" Muster, bei denen nur schwer eine Struktur zu erkennen ist. Über einen* abstand > 0 *erhalten Sie Asymmetrien.*

Wenn Sie die konkrete Abfolge der Befehle bei der Rekursion besser nachvollziehen wollen, können Sie im aktuellen Beispiel das Abbruchkriterium minimum *auf Werte setzen, die den Abbruch nach einem, zwei usw. Rekursionsschritt(en) bewirken (also z.B. 50, 25, 12 ...).*

13.1 Exkurs: lokale vs. globale Variable

Inzwischen haben wir bereits vielfach *Parameter* bzw. *Variablen* als Platzhalter für Zahlenwerte eingesetzt. Durch diese Parametrisierung haben wir eine Verallgemeinerung und Flexibilisierung der Anwendungen erreicht. Diese Platzhalter habe ich bisher stillschweigend eingeführt und unterschiedlich verwendet. Die notwendige Differenzierung soll hier nachgeliefert werden.

Der Gültigkeitsbereich der Platzhalter war manchmal auf die aktuelle Befehlsfolge oder den Block beschränkt, innerhalb dessen sie angelegt wurden. In einem solchen Fall werden sie als *lokale Variable* bezeichnet.

Vermehrt haben wir auch Variablen verwendet, die automatisch in allen Prozeduren bekannt und gültig sind. Diese werden als *globale Variablen* bezeichnet. Da sie deshalb auch innerhalb von Prozeduren veränderbar sind, werden Programme dadurch leicht unübersichtlich und fehleranfällig. Am besten wäre es also, wenn Prozeduren nur mit Übergabewerten und Reportern Daten untereinander austauschen könnten. Allerdings, wenn mehrere Werte über viele Ebenen geschachtelter Prozeduren durchgereicht werden müssen, bis sie die Prozeduren erreichen, die sie wirklich benötigen, dann sind globale Variablen nützlich und erlaubt.

Das brauchen wir von **Snap!***:*





In der Palette **Variablen** können mit **Neue Variable** *globale* Variablen erzeugt werden, die in allen Prozeduren bekannt und gültig sind. Entsprechend werden mit dem Befehl **Skriptvariablen** *lokale* Variablen erzeugt, die nur temporär innerhalb der aktuellen Befehlsfolge oder innerhalb eines Blocks gültig sind.

Im zugehörigen Dialogfenster kann der Name der Variablen festgelegt werden.



Durch den Befehl **setze Variablenname auf Wert** wird der vorher definierten globalen bzw. lolalen Variablen ein gewünschter Wert zugewiesen. Dies kann auch ein errechneter Wert sein, z.B.:





Mit **ändere Variablenname um Wert** kann eine Variable systematisch um einen Wert geändert werden. Das ist z.B. bei Zählvariablen nützlich.



Mit den Befehlen **zeige Variable** bzw. **verstecke Variable** können die entsprechenden Variablenwerte programmgesteuert in einem Monitorfenster auf der Bühne ein- bzw. ausgeblendet werden.

14. HOMMAGE À KOLOMYJEC: BOXES

Das Bild 42 (oben) mit dem Titel *Boxes I* ist mein *Recoding* eines Bildes von William J. Kolomyjec (geb. 1947 in Detroit). Obwohl dieses und andere seiner Bilder schon 1973 entstanden sind, können sie bereits als Beispiele für ein erstes *Remixing* der frühen Computerkunst gelten. William J. Kolomyjec ist wahrscheinlich einer der ersten, der sich explizit auf diese Vorbilder bezog. Er nahm Nees' *Schotter* als Ausgangspunkt für seine *Boxes I+II* und begründet dies wie folgt (Kolomyjec, 1975):

„It was not uncommon for these early computer image makers to use the computer to duplicate a basic theme by a traditional artist. A classic study is by A. Michael Noll, Computer Compositions with Lines 1965, after Piet Mondrian's Composition with Lines, 1917[62]*. Boxes in turn is my interpretation of a work by George Nees entitled Gravel Stones (Schotter). I feel that it is appropriate to study works of other artists, especially traditional artists and those who established computer art in the fine art realm."*

Der Unterschied der Bilder von Kolomyjec zum *Schotter*-Original liegt darin, dass die Unordnung zur Mitte hin (*Boxes I*) bzw. umgekehrt von der Mitte aus radial zu den Rändern hin (*Boxes II*) zunimmt. Diese Besonderheit bringt beim Recoding die Schwierigkeit mit sich, dass der Zufallsfaktor für die Drehung und die räumlichen Abweichungen nicht stetig in eine Richtung zunimmt, sondern entsprechend der Zentrierung bzw. Auflösung an den Rändern. Die bisher vorgestellten *Schotter*-Varianten in *Recoding Schotter (I)*, *Recoding Schotter (II)* und *Remixing Schotter* sind zwar leicht für andere Varianten zu erweitern (die in *Remixing Schotter* gezeigten Beispiele von Krawczyk belegen das). Für die Kolomyjec-Varianten wären allerdings relativ umständlich mehrfache, bedingte Verzweigungen notwendig, um diese teils gleich-, teils gegenläufigen Veränderungen in den Spalten und Reihen zu verrechnen. Ich bevorzuge deshalb ein einfacheres Verfahren mittels Rekursion.

Mit diesem Projekt möchte ich außerdem noch einmal verdeutlichen, wie mit den Prinzipien der *Parametrisierung*, der *Verallgemeinerung* und der *Rekursion* am Ende ein sehr übersichtlich strukturiertes Programm entsteht. Es ist gleichzeitig so flexibel, dass wieder leicht davon abgeleitete Varianten möglich werden (etwa Farben und Formen). Wie kann das erreicht werden?



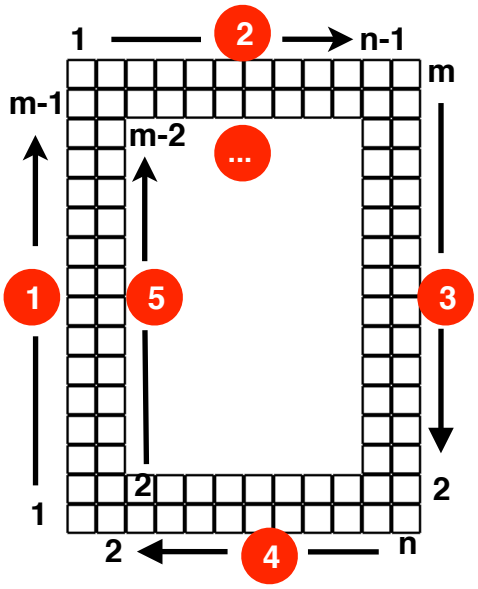
Das Programm *Boxes I* verwendet für alle relevanten Bildeigenschaften globale bzw. lokale Variablen zur Festlegung der entsprechenden Kenngrößen. Das bewährte m*n-Raster wird als Grundstruktur verwendet und dessen Kenndaten übernommen (z.B. von *Molnar: 25 Quadrate*). Von *Schotter* zu übernehmen sind:

- **n_ecken:** Zahl der Ecken eines Vielecks

[62] Dieses Beipiel wird im Kapitel 17: *Hommage à Noll: Stilstudien* vorgestellt.

- **delta_d**: Veränderung des Drehwinkels
- **delta_xy:** horizontale und vertikale Abweichungen
- **abstand**: horizontaler und vertikaler Abstand zwischen einzelnen Vielecken

Im Unterschied zum *Schotter*-Programm soll das Zeichnen der Vielecke nicht spalten- und reihenweise, sondern im Rechteck erfolgen. Begonnen wird außen 1 mit einer senkrechten Linie, dann eine waagrechte Linie 2, gefolgt von einer zweiten senkrechten Linie 3 und zuletzt einer zweiten waagrechten Linie 4. Es folgen die nächsten 4 inneren Linien (5 usw. ...), wobei die Längen der Linien dann horizontal und vertikal um je zwei Vielecke zu reduzieren sind.



Es liegt nahe, diese Abfolge mit einer rekursiven Prozedur **vier_linien** abzuarbeiten. Für deren flexible Steuerung benötigen wir zusätzlich die globalen Variablen

- **n_linien:** Zahl der gezeichneten Linien
- **n_rechtecke:** Zahl der gezeichneten Rechtecke aus den je vier Linien
- **korrektur_ende:** Korrekturfaktor, um bei ungleichen **m** und **n** doppeltes Zeichnen von Linien zu verhindern.

In der rekursiven Prozedur **vier_linien** wird zu Beginn das Abbruchkriterium geprüft, d.h. ob die Zahl der bisher gezeichneten Linien die Zahl der Reihen und Spalten überschreitet. Danach wird viermal der Block **vieleck_linie** aufgerufen.



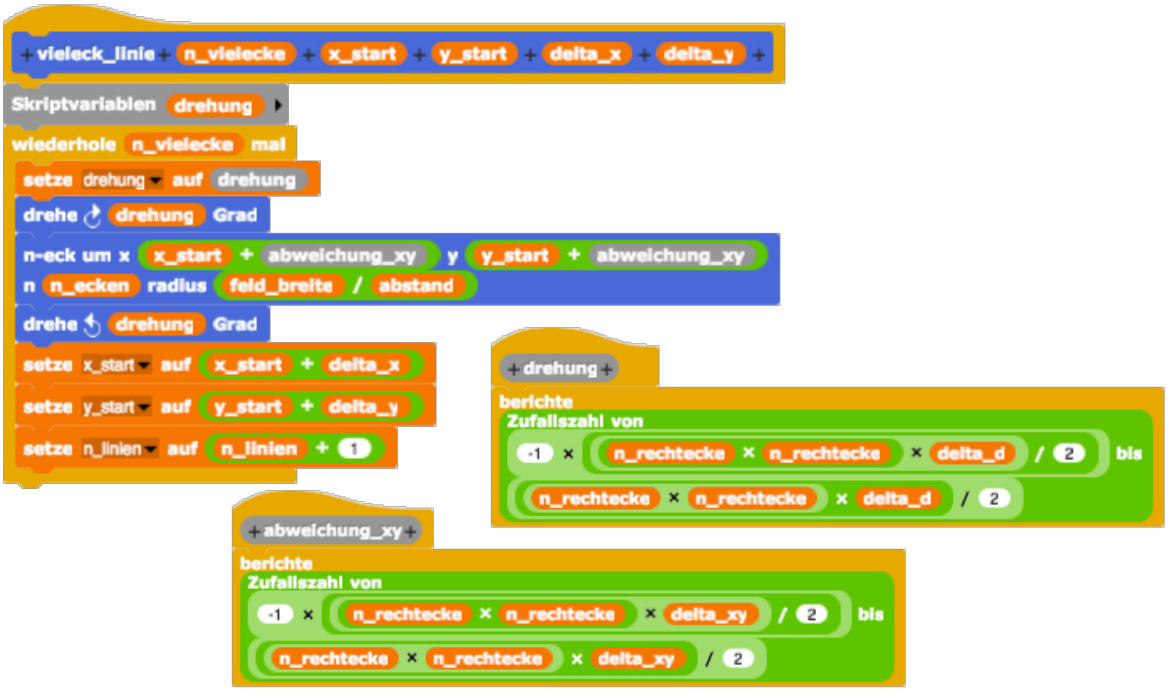
Allein durch die jeweis übergebenen Parameter wird gesteuert, wieviele Vielecke gezeichnet werden sowie ob und wo eine senkrechte oder waagrechte Linie von Vielecken gezeichnet wird. Danach werden die Kenndaten aktualisiert.

Am Ende wird **vier_linien** rekursiv aufgerufen mit entsprechend reduzierten Zahlen für die senkrecht bzw. waagrecht zu zeichnenden Vielecke.

Der Block **vieleck_linie** ist so allgemein gehalten, dass allein durch die Übergabeparameter die Länge und Ausrichtung einer Linie von Vielecken festgelegt wird:

- **n_vielecke:** Zahl der in einer Linie zu zeichnenden Vielecke
- **x-start**, **y_start:** Startpunkt für das Zeichnen des ersten Vielecks
- **delta_x**, **delta_y:** Abstand zwischen zwei Vielecken (bei einer senkrechten Linie ist **delta_x = 0**, bei einer waagrechten Linie entsprechend **delta_y = 0**)

Der Block **vieleck_linie** ruft seinerseits den bereits eingeführten Block **n-eck um x y n radius** auf. Die Drehung und räumliche Abweichungen der einzelnen Vielecke wird von angepassten Reportern für **drehung** und **abweichung_xy** geliefert.



Diese systematische Aufteilung erleichtert die weitere Abwandlung sehr. So soll für *Boxes II* (Bild 42 unten), dem Pendant zu *Boxes I*, der Zufall von der Mitte aus radial zu den Rändern hin zunehmen. Für diese Programmvariante ist eine zusätzliche globale Variable **n_plateau** einzuführen, die angibt, wieviele Rechtecke mit Drehungen und räumlichen Abweichungen gezeichnet werden sollen. Bei den restlichen Rechtecken werden diese unterdrückt, so dass der Eindruck eines „Plateaus“ entstehen kann.

Die Weichenstellung erfolgt im Block **vieleck_linie**. Unterhalb von **n_plateau** wird wieder der Block **n-eck um x y n radius** mit den Block-Reportern für **drehung** und **abweichung_xy** aufgerufen. Diese sind für *Boxes II* angepasst.

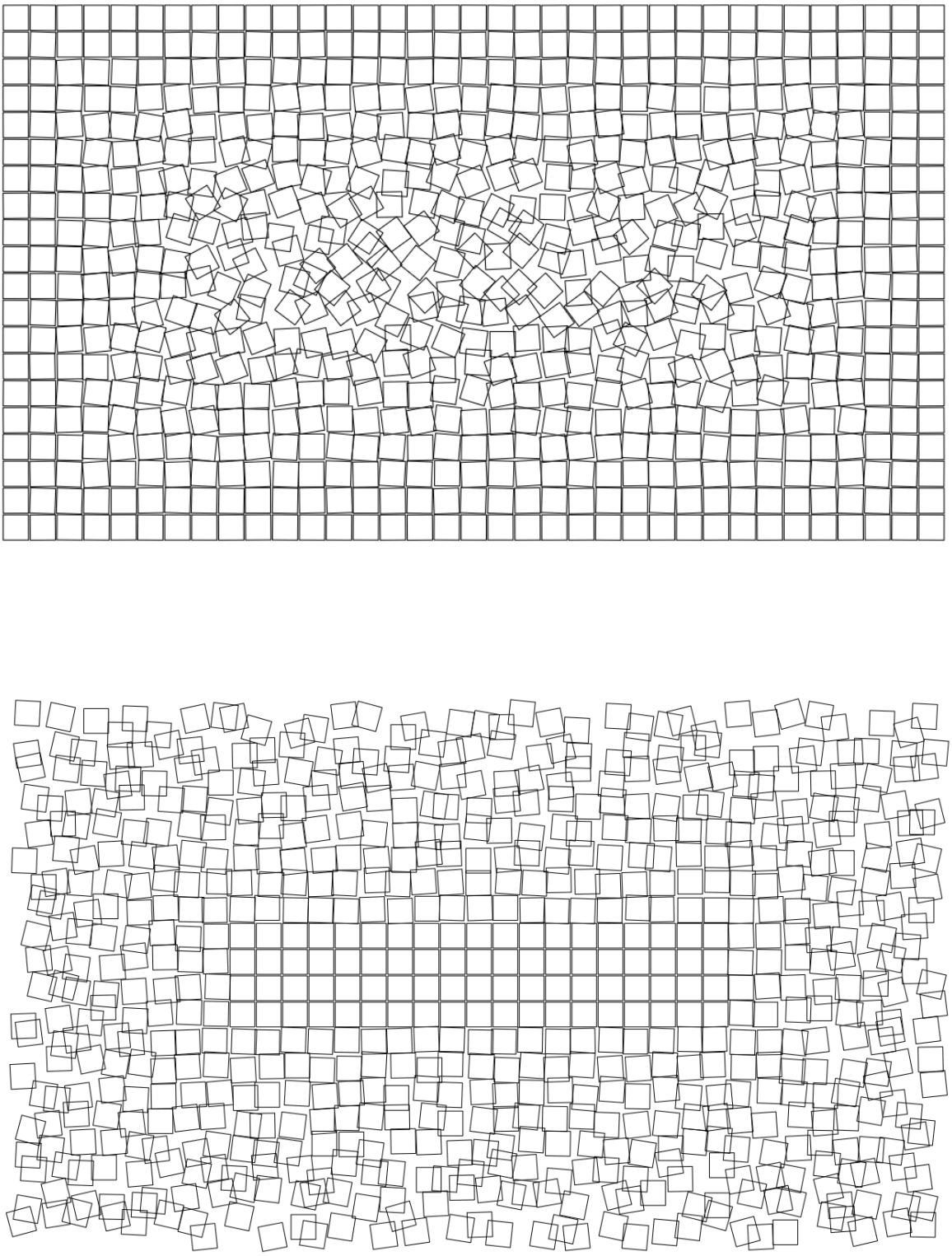
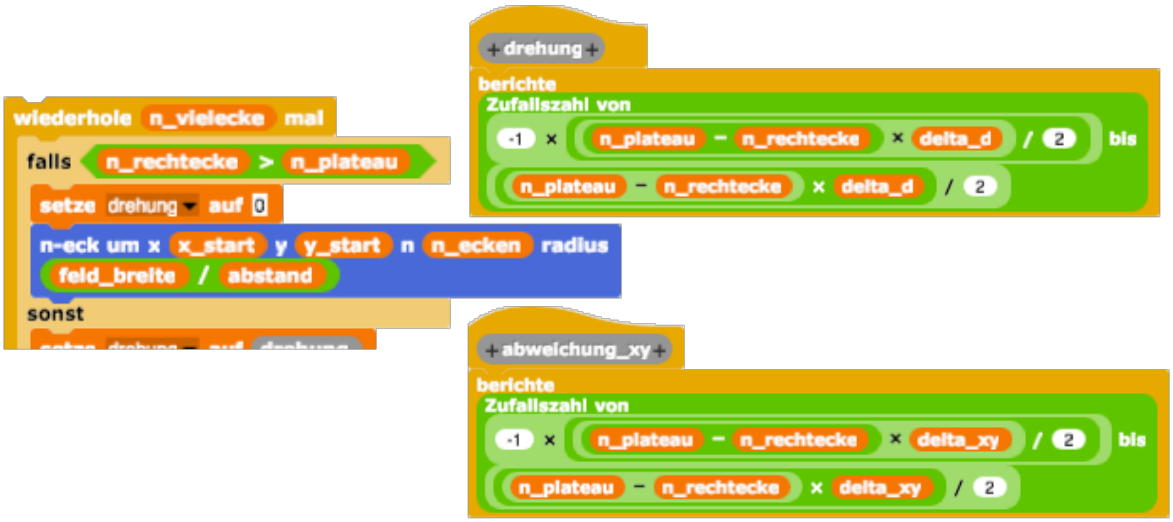


Bild 42: Hommage à Kolomyjec: Boxes I (oben), Boxes II (unten)



Damit sind alle Komponenten besprochen, durch die wir das Ausgangsproblem (*„zeichne Vielecke innerhalb eines m*n-Rasters mit ortsabhängigen Zufallsabweichungen der Richtung und der Lage“*) konsequent in Teilprobleme aufgeteilt und bearbeitet haben:

- Oberste Ebene (Block **vier_linien**): Setze vier Linien (die aus Vielecken bestehen) zu einemViereck der Größe m*n zusammen,.
- Mittlere Ebene (Block **vieleck_linie**): Setze Vielecke zu einer Linie gewünschter Länge aneinander, beginnend an einem festgelegten Startpunkt.
- Untere Ebene (Block **n-eck um x y n radius**): Zeichne innerhalb der aktuellen Linie ein Vieleck an der festgelegten Position.
- Zusatzebene: Reporter-Blöcke (**drehung** und **abweichung_xy**) liefern die zufällig veränderten Werte für die Drehungen und räumlichen Verschiebungen der Vielecke.

Durch die konsequente Modularisierung lassen sich weitere Programmvarianten einfach ableiten.

So entsteht etwa in *Boxes I*, wenn Zufallsabweichungen erst ab einer bestimmten Anzahl von Rechtecken beginnen, ein ebenes „Randplateau“ (Bild 43 links). Dafür reicht es, auch in *Boxes I* die Variable **n_plateau** für die Breite dieses Randes einzuführen und im Block **vieleck_linie** mit einer bedingten Verzweigung unterhalb dieser Grenze die Zufallsabweichungen zu unterdrücken. Bild 43 links ist durch Vielecke mit **n_ecken = 2** entstanden, oben mit **zeige Richtung 90**, Mitte mit **zeige Richtung 45**. Durch ein engmaschiges Raster und größere Stiftdicke entsteht der Eindruck einer Punktwolke. So ist Bild 43 links unten durch Vielecke mit **n_ecken = 4** entstanden. Bild 43 rechts zeigt jeweils die analogen Ergebisse von *Boxes II* zu Bild 43 links.

Anregung: *Allein mit der Variation von* n_ecken, zeige Richtung, abstand *und* n_plateau *lassen sich mit diesen beiden Versionen von Boxes I und Boxes II Bilder mit sehr unterschiedlichen, oft überraschenen Charakteristiken erzeugen.*

Die nächste Form des *Remixing* dieser Vorbilder ist die Einführung von Farbe. Das wird durch das Zeichnen farbgefüllter Vielecke erreicht (Bild 44), wieder unter Verwendung des Blocks **n-eck-rekursivgefuellt**. Da bei der Farbgebung das Randplateau

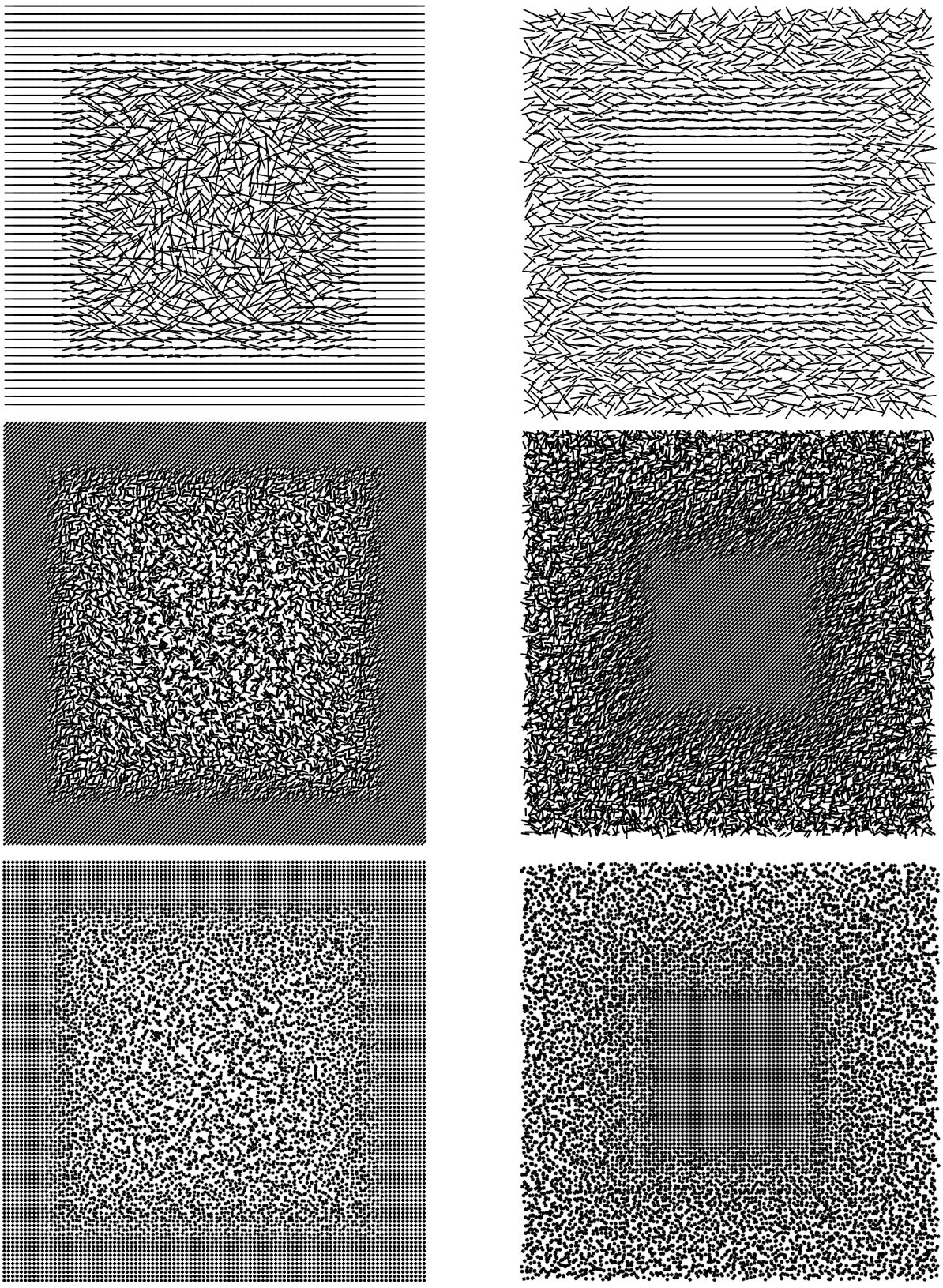
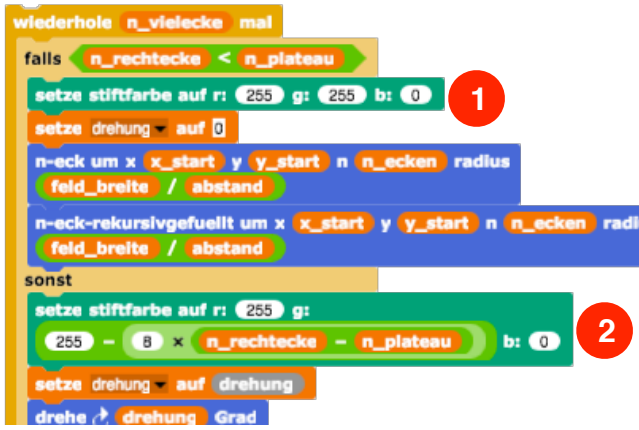


Bild 43: Remixing Boxes I (links) & II (rechts)

in *Boxes I* bzw. das mittige Plateau in *Boxes II* einheitlich eingefärbt werden soll, wodurch der Plateaueffekt verstärkt wird, erfolgt die Farbfestlegung im Block **vieleck_linie**. Die für *Boxes II* eingeführte Weichenstellung für das Plateau kann einfach um die Farbfestlegung ergänzt werden (bei *Boxes I* lautet die entsprechende Abfrage **falls n_rechtecke < n-plateau**).

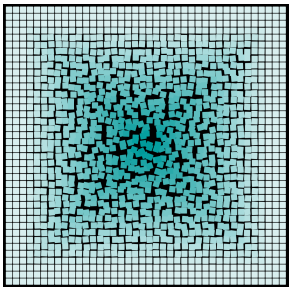
1 Im Beispiel wird als Ausgangsfarbe Gelb festgelegt mit **setze stiftfarbe auf r: 255 g: 255 b: 0**.

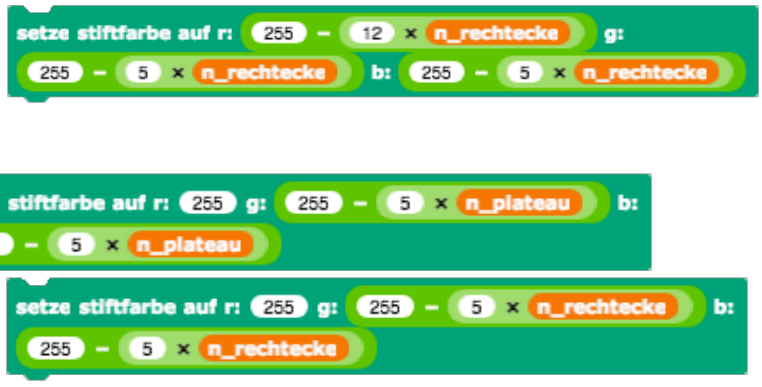
2 Die Ausgangsfarbe wird außerhalb des Plateaus in Abhängigkeit von **n_rechtecke** sukzessive verändert.

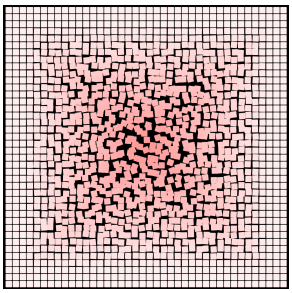


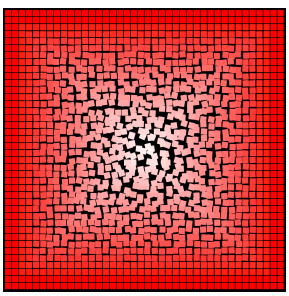
Hinweis: *Die Steuerung der Farbgebung reagiert auch bei kleinen Änderungen der RGB-Werte über das Verhältnis von* n_rechtecke *zu* n_plateau *ziemlich stark. Die folgenden Beispiele dienen als Anregungen und mögliche Ausgangspunkte. Von dem gezeigten zugehörigen Befehlspaar* **setze stiftfarbe ...** *ist bei der bedingten Verwzeigung jeweils der erste Befehl (links) hinter* falls, *der zweite (rechts) hinter* sonst *einzusetzen. Es ist empfehlenswert, davon ausgehend mit kleinen Änderungen zu experimentieren.*













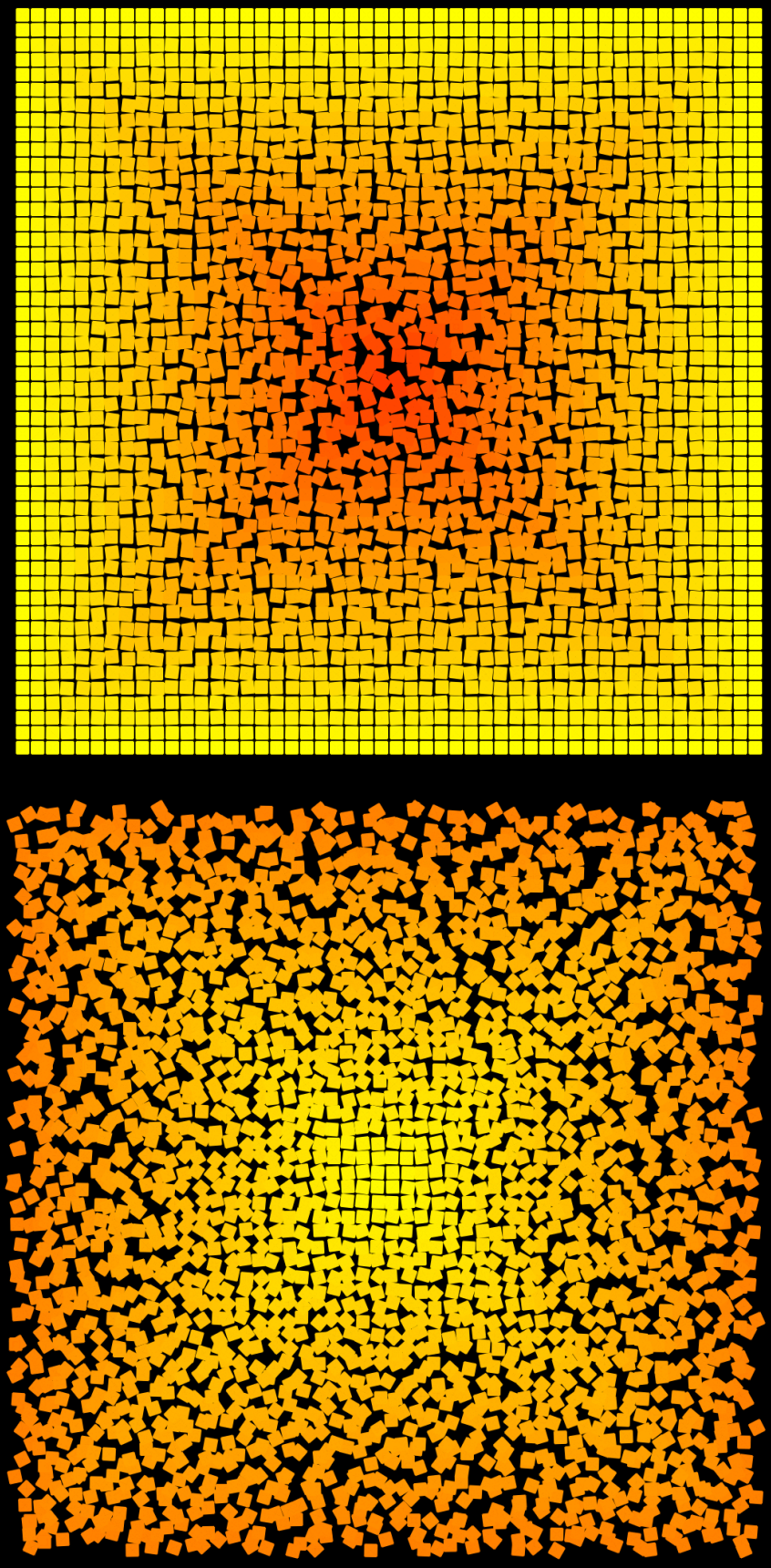
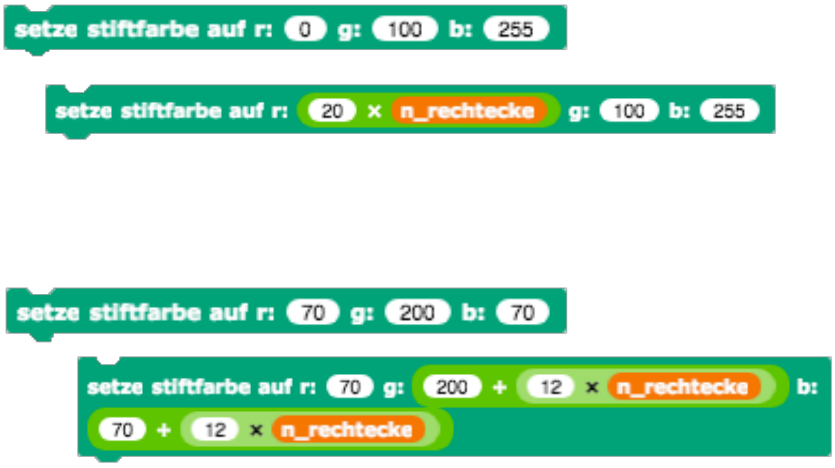
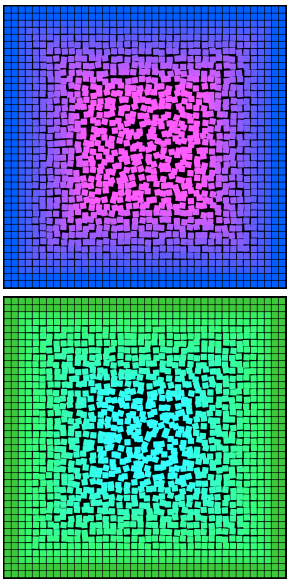


Bild 44: Remixing Boxes I & II





Anregung: *Der Verlauf von Ordnung zu Chaos braucht nicht auf die bisher gezeigten Richtungen oben-unten, außen-Mitte oder Mitte-außen beschränkt werden. Mit bedingten Verzweigungen können Sie z.B. bestimmte (auch mehrere) Regionen festlegen, innerhalb derer Unordnung zugelassen ist. So lassen sich dann auch asymmetrische Anordnungen erreichen.*

15. HOMMAGE À NEES: DIE SPRACHE G

Georg Nees hat für mich eine besondere Stellung in der frühen Computerkunst. Er war nicht nur einer der ersten, der solche Werke schuf und ausstellte, ihm war auch von Anfang an der innovative Charakter seiner Grafiken bewusst: „*Als ich Figur nach Figur unter dem Schreibstift hervorquellen sah, lief es mir kalt den Rücken hinunter. Ich dachte: ‚Hier ist etwas, das nicht wieder verschwinden wird.'* "(Nees, 2005). Er programmierte sie bewusst unter ästhetischen Gesichtspunkten und verstand sie als Modelle des künstlerischen Produktionsprozesses (Klütsch, 2007, S. 110).

Nees ist gleichzeitig einer der Wenigen, die ihre den „ästhetischen Objekten" zu Grunde liegenden Algorithmen publiziert haben (wie wir im Kapitel 12: *Hommage à Nees: Schotter* bereits gesehen haben). In seiner Dissertation *Generative Computergraphik* (1969)[63] behandelt er das Thema Computergraphik zugleich aus informatischer und philosophisch-ästhetischer Sicht. Er führt mit konkreten Beispielen in seine Denkweise ein und illustriert mit der Entwicklung exemplarischer Programme das Zusammenspiel von prozeduraler Abstraktion und (gesteuertem) Zufall. Damit hat er für viele weitere Akteure die Tür zur digitalen Computerkunst und Aktivitäten in diesem Feld aufgestoßen.

In seinem Buch hat Georg Nees eine *problemorientierte Sprache G für die generative Graphik* entwickelt und beschrieben. Sie ist in eine Trägersprache eingebettet - in seinem Fall in die Sprache ALGOL 60. Er greift dabei auf ALGOL-Erweiterungen (von Horst Glasauer, Siemens AG) zurück, die die Ansteuerung des verwendeten Plotters erlauben. Es sind dann eigentlich recht wenige Sprachelemente (genau genommen gerade mal acht), die Nees benötigt, um damit sein ästhetisches Laboratorium einzurichten. Sie ähneln den Snap!-Blöcken, die im Kapitel 9: *Figurenbaukasten* erarbeitet wurden. Für die Projekte in diesem Kapitel reichen sogar zwei seiner Sprachelemente: **LEER (X,Y)** für die Bewegung zu einem Punkt ohne das Zeichnen einer Linie; **LINE (X,Y)** für dasselbe mit dem Zeichnen einer Linie.

Die Snap!-Programmierumgebung enthält bereits alle grundlegenden grafischen Sprachelemente. So ist es sehr leicht möglich, die Elemente der Sprache G direkt durch Snap!-Sprachelemente zu ersetzen bzw. durch eigene Blöcke nachzubilden. Bei Nees sind die Sprachelemente an der Ansteuerung des Plotters Zuse Graphomat Z 64 ausgerichtet, die Snap!-Blöcke dagegen an der Schildkrötengrafik.

Es geht mir in diesem Kapitel darum, an drei Beispielen zu zeigen, wie Nees mit einfachsten Mitteln *Figurenkollektive* erzeugt. Ihn interessiert und er untersucht das Zusammenspiel von Ordnung und Zufall. Das zeigt sich in Bildfolgen, in denen er mit der Wirkung von Zufallsgeneratoren vielfältige Variationen von Grundmustern erzeugt.

---

[63] Die Dissertation von Georg Nees ist seit 2006 in einer Neuauflage als E-Book erhältlich. Eine lohnende Lektüre!

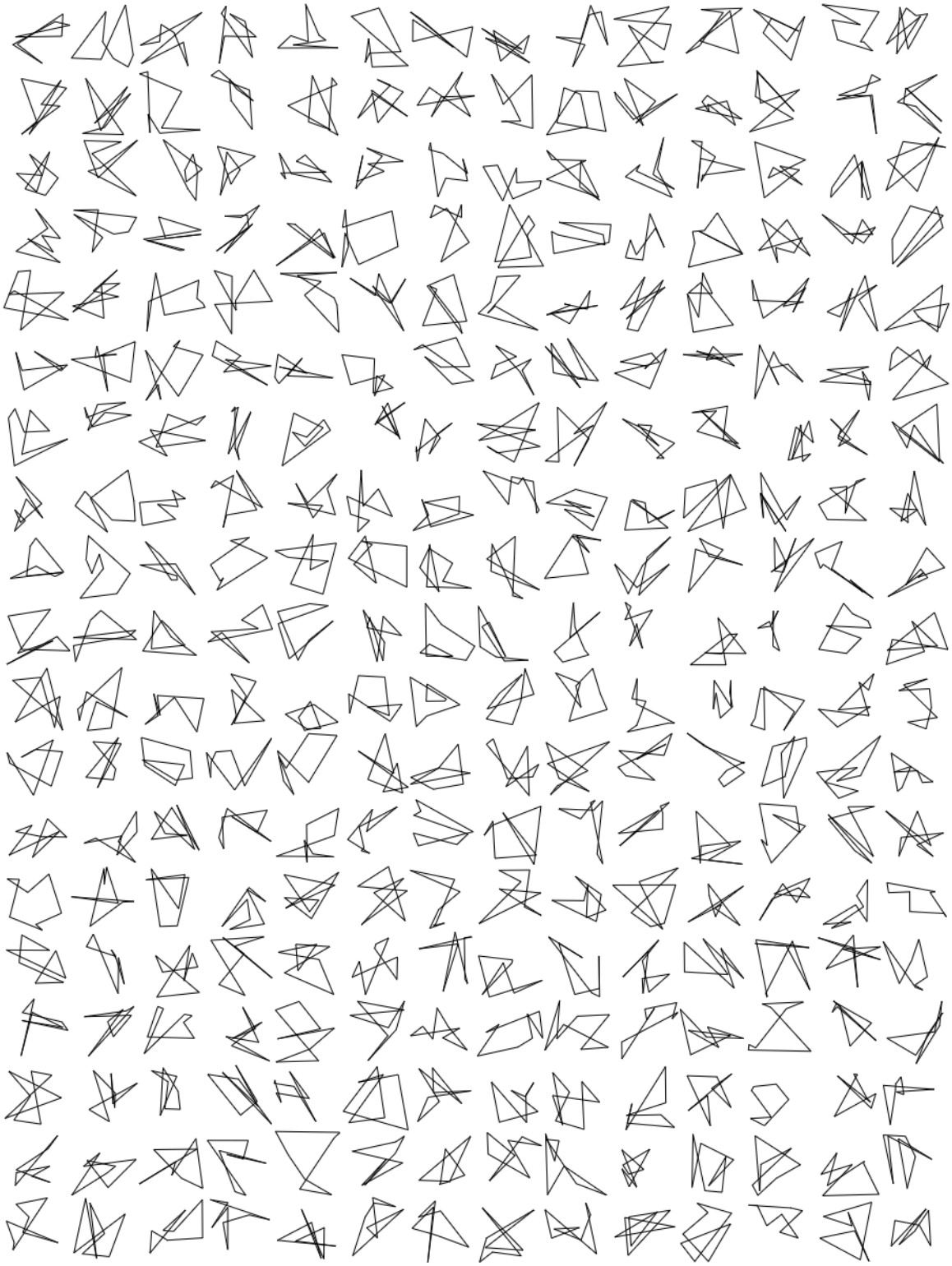


Bild 45: Hommage à Nees: 8-Ecke

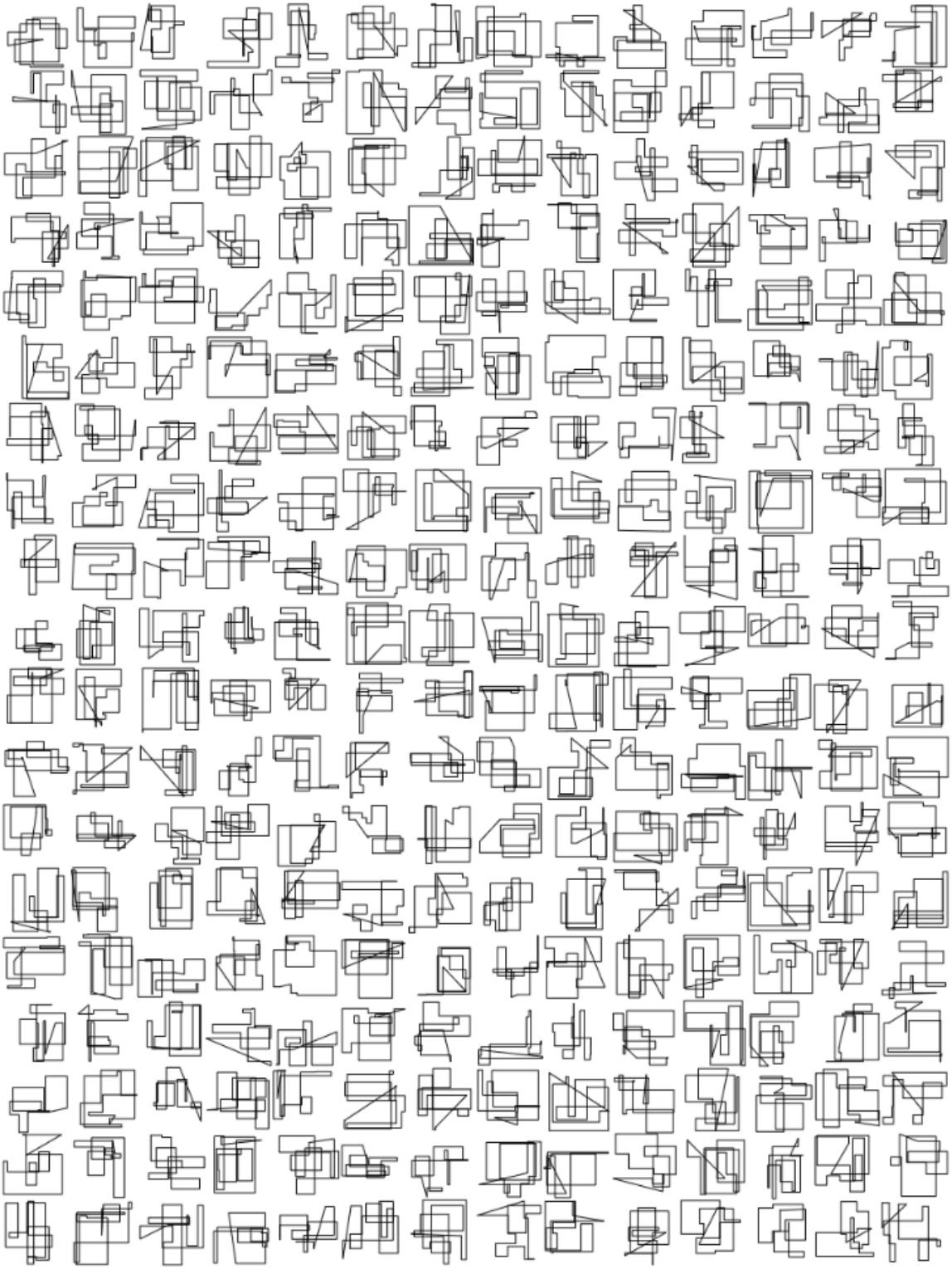


Bild 46: Hommage à Nees: 23-Ecke (rechtwinklig)

Da er dabei den Zufall in starkem Maße kontrolliert, entstehen seine Bilder nicht zufällig; die Ergebnisse werden nur mit Hilfe des Zufalls erreicht. Die Vorbilder für Bild 45 und Bild 46 gehörten zu seinen Exponaten der ersten Ausstellung von Computerkunst 1965 in Stuttgart. Sie verdeutlichen ein Paradox: Beide sind einerseits sehr einfach (das gilt auch für die zugehörigen Programme; siehe unten), andererseits lassen sie die Vielfalt erahnen, die uns das Konstruktionsprinzip eröffnet.

Dies lautet für Bild 45: *„Streue acht Punkte in das Figurquadrat und verbinde sie durch einen geschlossenen Streckenzug."* (Nees, 1965). Die Bildfläche wird also in ein Quadratraster unterteilt (bei uns ist es das bekannte m*n-Raster) und in jedem Quadrat acht zufällig gesetzte Punkte miteinander verbunden.

In der Snap!-Umsetzung (unter Verwendung von **LEER** und **LINE** aus der Sprache G) erkennen wir in der **m-Schleife** des Rasters die Skriptvariablen **x0** und **y0**, mit denen der Ausgangspunkt eines Streckenzugs gespeichert wird, die zufällige Festlegung von **x0** und **y0** und die Positionierung der Schildkröte an diesem Punkt. In einer weiteren Schleife werden die Strecken zu den nächsten sieben Punkte gezogen. Danach wird die Strecke zurück zum Ausgangspunkt gezogen.



Das Konstruktionsprinzip für Bild 46 ist ein wenig komplexer wegen der rechten Winkel zwischen den Strecken: *„Zeichne, im Figurquadrat irgendwo beginnend, einen abwechselnd horizontalen und vertikalen – in der horizontalen zufällig nach links oder rechts, in der vertikalen zufällig nach oben oder unten – innerhalb des Figurquadrats verlaufenden Streckenzug mit 23 Teilstrecken zufälliger Länge. Verbinde Anfangs- und Endpunkt des Streckenzugs geradlinig."* (Nees, 1965).

In der Snap!-Umsetzung dieser Variante wird die Schildkröte in jeweils zwei Schritten in rechten Winkeln zum nächsten Punkt bewegt. Abschließend wird die Strecke zurück zum Ausgangspunkt gezogen.



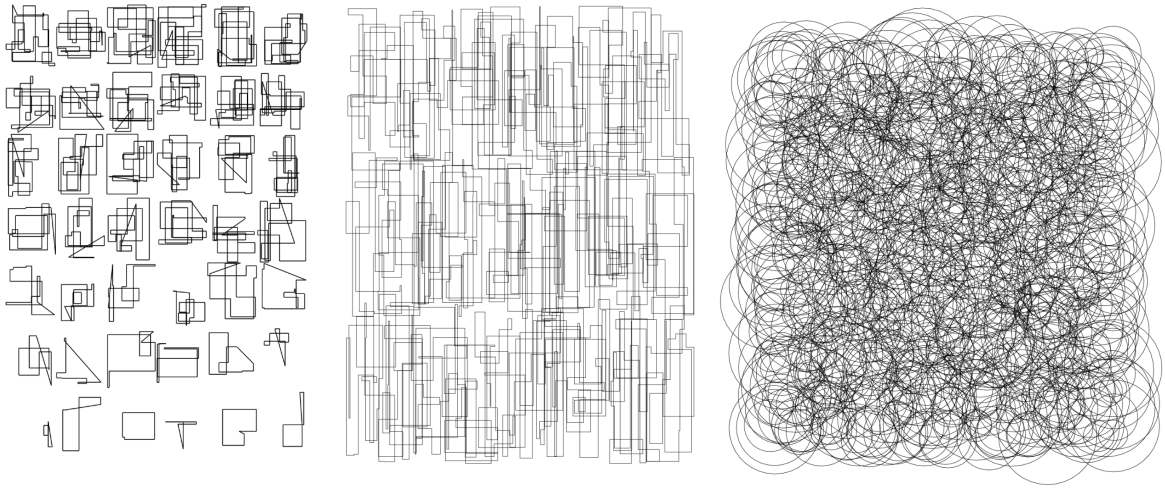
Mit seinen Bildern demonstriert Nees das Wechselspiel von Redundanz - der Wiederholung von Grundmustern im Raster - und Vielfalt - der ständigen Abwandlung der Grundmuster durch Zufallseinflüsse. So kurz und einfach die Programme für Bild 45 und Bild 46 auch sind, so ermöglichen sie doch sehr unterschiedliche Ergebnisse.

Anregung: *Durch leichte Änderungen bzw. Ergänzungen lassen sich aus den beiden Programmversionen sehr unterschiedliche Bilder erzeugen. Die folgenden Abbildungen sind wie folgt entstanden und sollten leicht nachvollziehbar sein:*

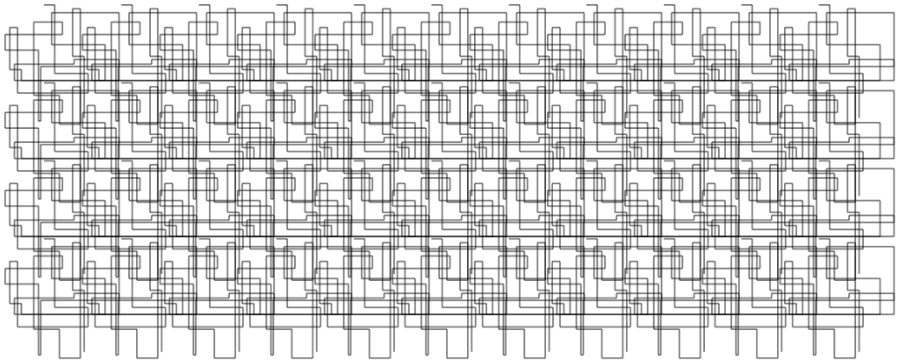
links: *Die Anzahl der gezeichneten Strecken innerhalb eines Feldes nimmt von unten nach oben zu.*

Mitte: *Die Strecken können die Grenzen eines Feldes um eine vorgebene Länge überschreiten. Dadurch können sich die Streckenzüge nebeneinander liegender Felder überlappen.*

rechts: *Statt senkrecht aneinander gefügter Strecken werden hier Kreise mit unterschiedlichen Radien in die Felder gezeichnet. Die Kreise nebeneinander liegender Felder können sich überlappen.*



In der untenstehenden Abbildung ist zu erkennen, dass die (sich wieder überlappenden) Streckenzüge diesmal in allen Feldern die gleiche Form haben. Trotz des gleichen Grundschemas wie die bisherigen Varianten, lässt sich diese Version aber nicht - bzw. nur sehr umständlich - mit den bisherigen Mitteln realisieren. Für die Wiederholung der gleichartigen Form wird es nämlich notwendig, alle Punktkoordinaten des Streckenzugs im erstem Feld zu speichen, damit dann in allen anderen Feldern darauf wieder zugegriffen werden kann. Die Möglichkeit dafür bieten *Felder* und *Listen*, die im folgenden Abschnitt vorgestellt werden.



15.1 Exkurs: Felder und Listen

Bei allen bisherigen Projekten wurden Variablen (global oder lokal) als „Behälter“ für Werte eingeführt, auf die später über den Namen der Variablen zugegriffen werden konnte. Die Erweiterung dieses Konzepts sind *Felder*, in denen unter dem Namen des Feldes gleich mehrere Werte abgespeichert werden können. Der Zugriff auf einzelne Werte dieses Feldes erfolgt über ihren „Index“, d.h. ihre Position innerhalb des Feldes. Felder können auch mehrdimensional sein und besitzen dann entsprechend mehrere Indizes. Der Zugriff auf einzelne Werte erfolgt über die Angabe aller Indizes. Benötigen wir z.B. für einen Streckenzug die Koordinaten von drei Punkten, so können sie in einem zweidimensionalen Feld gespeichert werden, etwa [x1,y1] [x2,y2] [x3,y3]. Die x-Koordinate des dritten Punktes erhalten wir dann über den Index [3,1].

Felder sind in Snap! als *Listen* verfügbar. Diese haben den Vorteil, dass sie als Elemente Zahlen und/oder Texte und sogar Listen enthalten können; ein Leistungsmerkmal, dass wir im aktuellen Beispiel ausnutzen werden. Während bei Feldern vorab für jede Dimension ihre Länge festgelegt werden muss (d.h. wieviele Elemente sie enthalten kann), lassen sich Listen dynamisch verändern, d.h. sie können während des Programmablaufs nach Bedarf verlängert oder verkürzt werden. Snap! bietet die dafür notwendigen Befehlsblöcke.

Das brauchen wir von **Snap!**:





Der Reporter **Liste** zeigt den Inhalt einer neu erzeugten Liste an. Mit den Pfeilen **<>** kann die Zahl der Elemente verändert werden. **Liste >** erzeugt eine leere Liste ohne Elemente.





Für das Erzeugen einer Liste gibt es keinen speziellen Befehlsblock. Dies erfolgt durch Erzeugen einer neuen Variablen, hier z.B. **name_liste**, der eine Liste als Wert zugeordnet wird. Mit **setze name_liste auf Liste** wird eine leere Liste erzeugt. Mit **setze name_liste auf Liste ... <>** können (beliebig viele) neue Elemente der Liste hinzu gefügt werden. Diese Elemente können Zahlen, Texte oder andere Listen sein.



Durch den Befehl **füge Element zu Liste hinzu** wird ein neues Element am Ende der Liste hinzugefügt. Mit **füge Element als n. in Liste ein** wird ein neues Element an der n-ten Stelle in die Liste eingefügt. Die weiteren Elemente werden entsprechend nach hinten verschoben.



Über den Reporter **Element n von Liste** kann gezielt auf das Element an der n-ten Stelle in der Liste zugegriffen werden.

Für die Umsetzung der vorigen Abbildung verwenden wir als Vorspann erneut das m*n-Raster. Die Punktkoordinaten des Streckenzugs im ersten Feld speichern wir nun in einer Liste, auf die dann in allen Feldern des Rasters zugegriffen werden kann. Dafür führen wir zusätzliche Variablen ein:

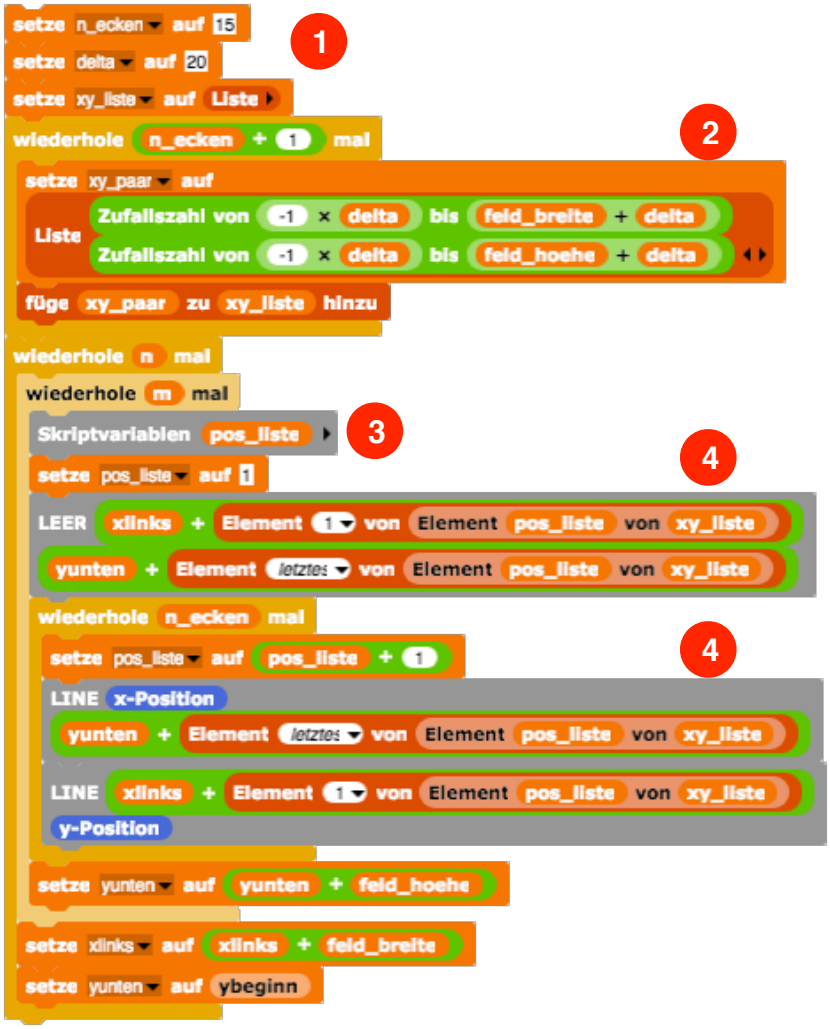
- **n_ecken:** Zahl der Ecken des Streckenzugs.
- **delta:** Überschneidung der Felder in Anzahl der Pixel.
- **xy_liste:** Liste mit den Koordinatenpaaren der Ecken.
- **xy_paar:** Liste mit den x- und y-Koordinaten eines einzelnen Punktes des Streckenzugs.

1 Als Erstes wird für den Streckenzug die gewünschte Zahl **n_ecken** und **delta** festgelegt, sowie die leere Liste **xy_liste** angelegt.

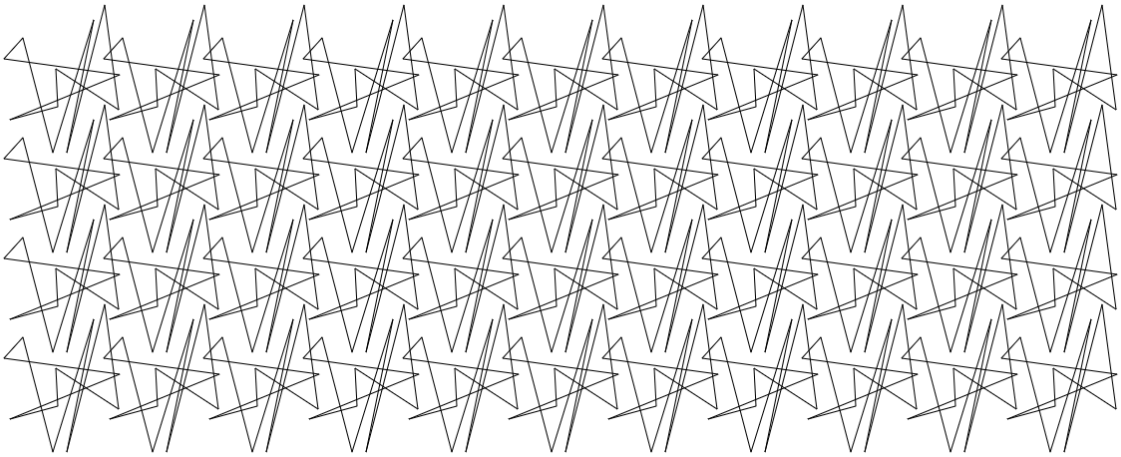
2 Die Koordinaten der Ecken werden zufällig festgelegt und in **xy_paar** abgelegt. Dieses Paar wird der **xy_liste** hinzugefügt.

3 Mit **pos_liste** wird ein Zähler für die Position von **xy_paar** innerhalb **xy_liste** eingeführt.

4 Zunächst wird die Schildkröte zum Startpunkt des Streckenzugs geschickt. In der darauf folgenden Schleife werden die weiteren Punkte abgerufen und die resultierenden Teilstrecken gezeichnet.



Anregung: *Die folgende Abbildung lässt sich durch die Kombination der Schleifen für* Bild 43 *mit den Listen für den Streckenzug innerhalb eines Feldes aus dem vorigen Programm kombinieren.*



15.2 Gestörte Gewebe

Das folgende Projekt (Bild 47) basiert auf einem Vorbild von Georg Nees, das er *Gewebe* genannt hat, weil er *„die Vertikallinien mit Ketten-, die Horizontallinien mit Schußfäden vergleichen kann“* (Nees, 1969, S. 260). Auch wenn die Grundstruktur wieder auf dem m*n-Raster beruht und oberflächlich gesehen einige Ähnlichkeit mit dem Bild *Schotter* (Bild 38) aufweist, so ist doch der Weg dorthin programmtechnisch deutlich abweichend (vgl. Code rechte Seite).

1. Zunächst werden alle Kreuzungspunkte des Gewebes berechnet und in einer Liste gespeichert (in der Reihenfolge von unten nach oben und von links nach rechts).
2. Gezeichnet werden zunächst alle vertikalen Linien, und
3. danach in gleicher Weise alle horizontalen Linien.

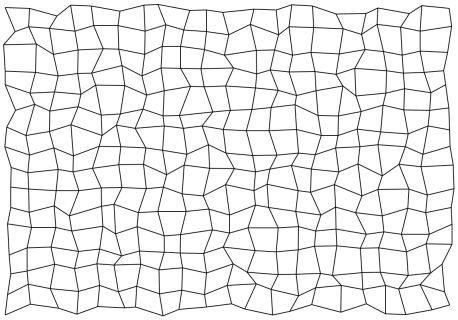
Beim Zugriff auf die Koordinaten ist auf die korrekte Reihenfolge zu achten: Bei den vertikalen Linien wird nacheinander auf die Elemente **xy_liste** zugegriffen, gesteuert über **pos_liste**, das entsprechend bei jedem Schleifendurchlauf um 1 erhöht wird. Bei den horizontalen Linien erfolgt die Steuerung über **pos_liste_v**, das bei jedem Schleifendurchlauf um **m+1** zu erhöhen ist, weil in die nächste „Ebene“ gewechselt wird.



Die „Störung“ im Gewebe hängt von der (wieder über **delta** gesteuerten) zufälligen Abweichung der Koordinatenpunkte von der Ideallage gleichseitiger Rechtecke ab. Anschaulich wird die Entstehung des Bildes, wenn die zufällige Abweichung **delta** zunächst eingeschränkt wird. Bei Beschränkung auf die Spalten entsteht ein Bild mit geraden Senkrechten (links), bei Beschränkung auf die Reihen entsteht ein Bild mit geraden Horizontalen (unten).



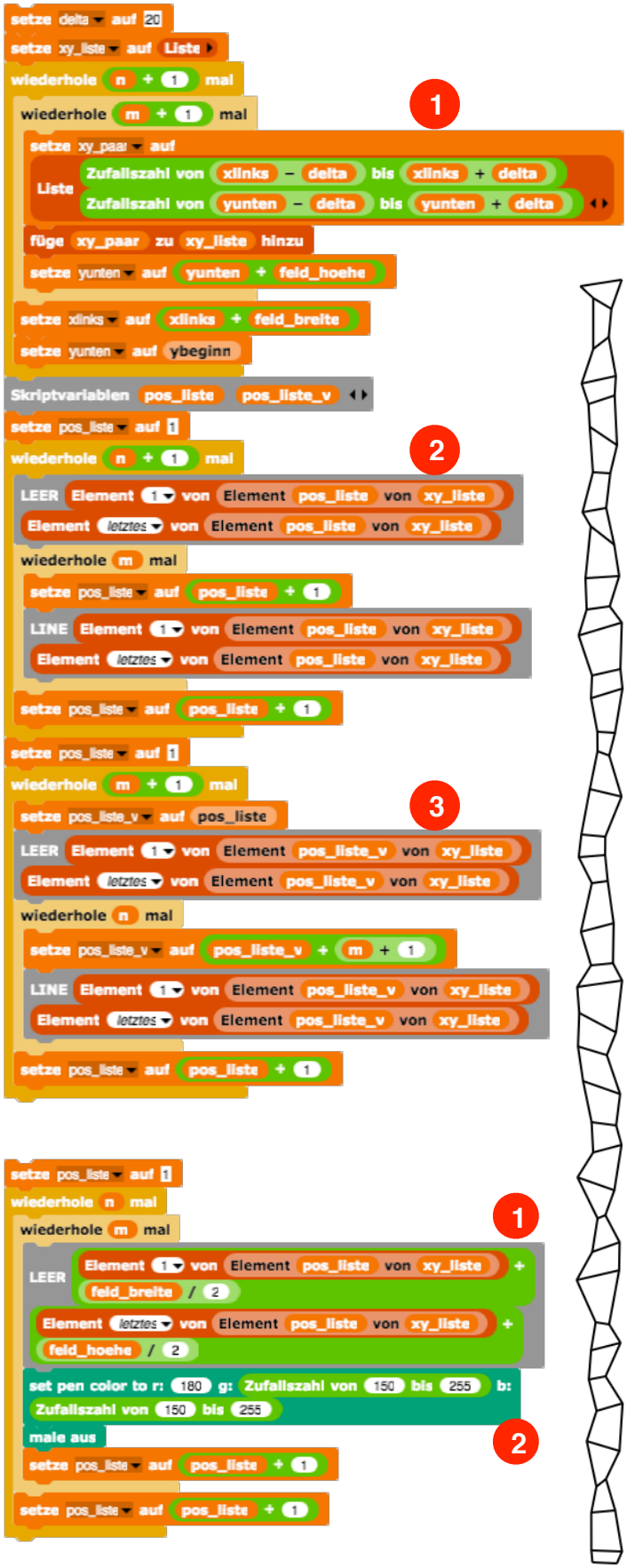
Ihre Kombination ergibt dann das „gestörte Gewebe“ (Abbildung unten und Bild 47). Die Variante mit nur einer Säule (rechts) hat Nees *Ästhetische Unruhe* genannt.



Für das *Remixing* dieses Projekts bietet sich das Füllen der Vierecke mit Farben an. Ein mögliches Ergebnis zeigt Bild 48, das ich *Patchwork* genannt habe. Bei der Umsetzung kann z.B. für jedes Viereck

1. die Schildkröte an den (ungefähren) Mittelpunkt geschickt werden,
2. die gewünschte Farbe bzw. das gewünschte Farbspektrum (über Zufallszahlen) bestimmt und damit das Viereck mit **male aus** gefüllt werden.

Hinweis: *Da beim „gestörten Gewebe“ wegen der zufälligen Abweichungen keine rechtwinkligen Vierecke auftreten, ist die gezeigte Mittelpunktbestimmung umso ungenauer, je stärker die Zufallsabweichung* delta *gewählt wird. Es kann dabei auch zu Überlappungen der Vierecke kommen (wie z.B. in Bild 47), die* **male aus** *dann nicht mehr korrekt ausfüllen kann.*



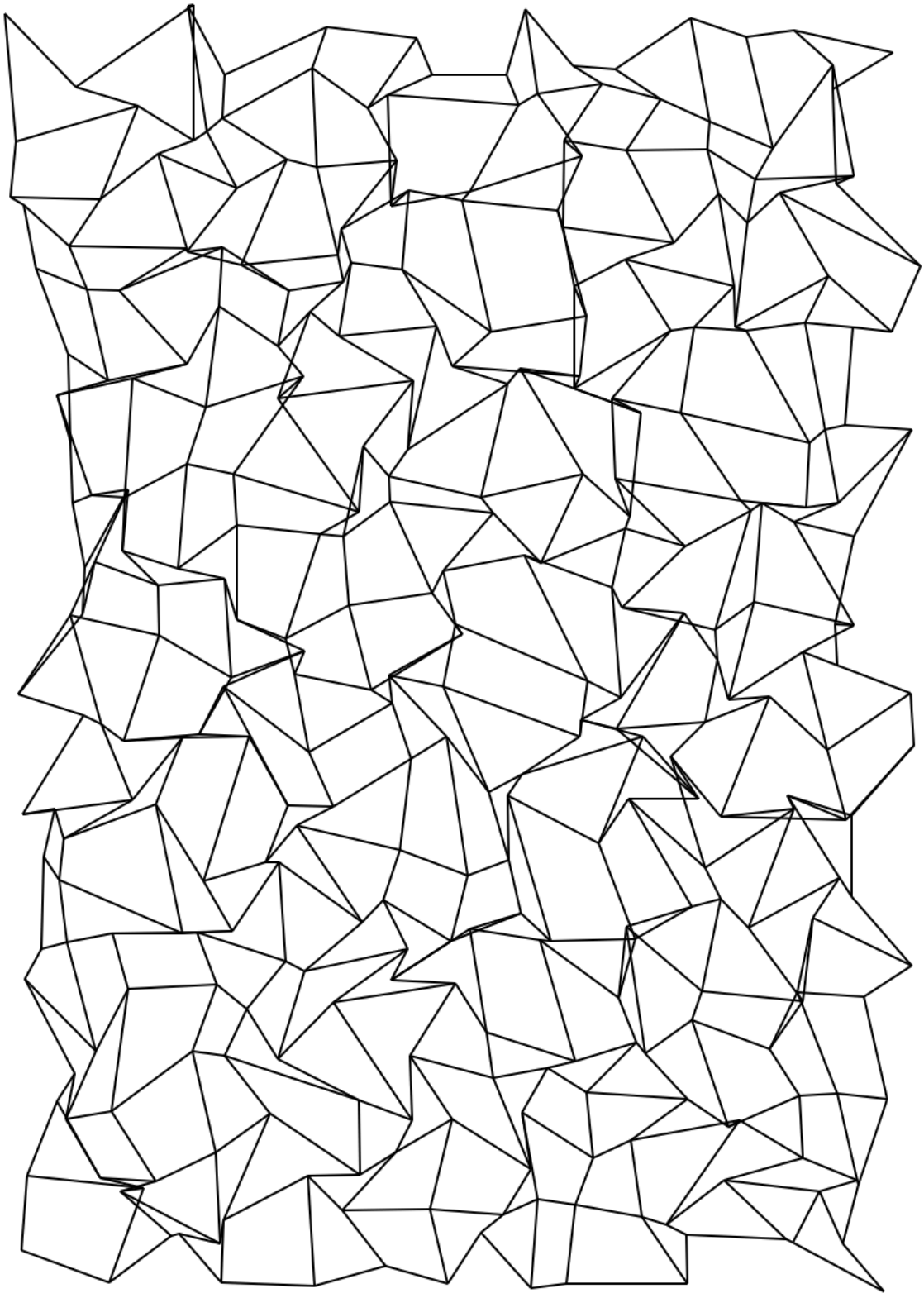


Bild 47: Hommage à Nees: Gestörtes Gewebe

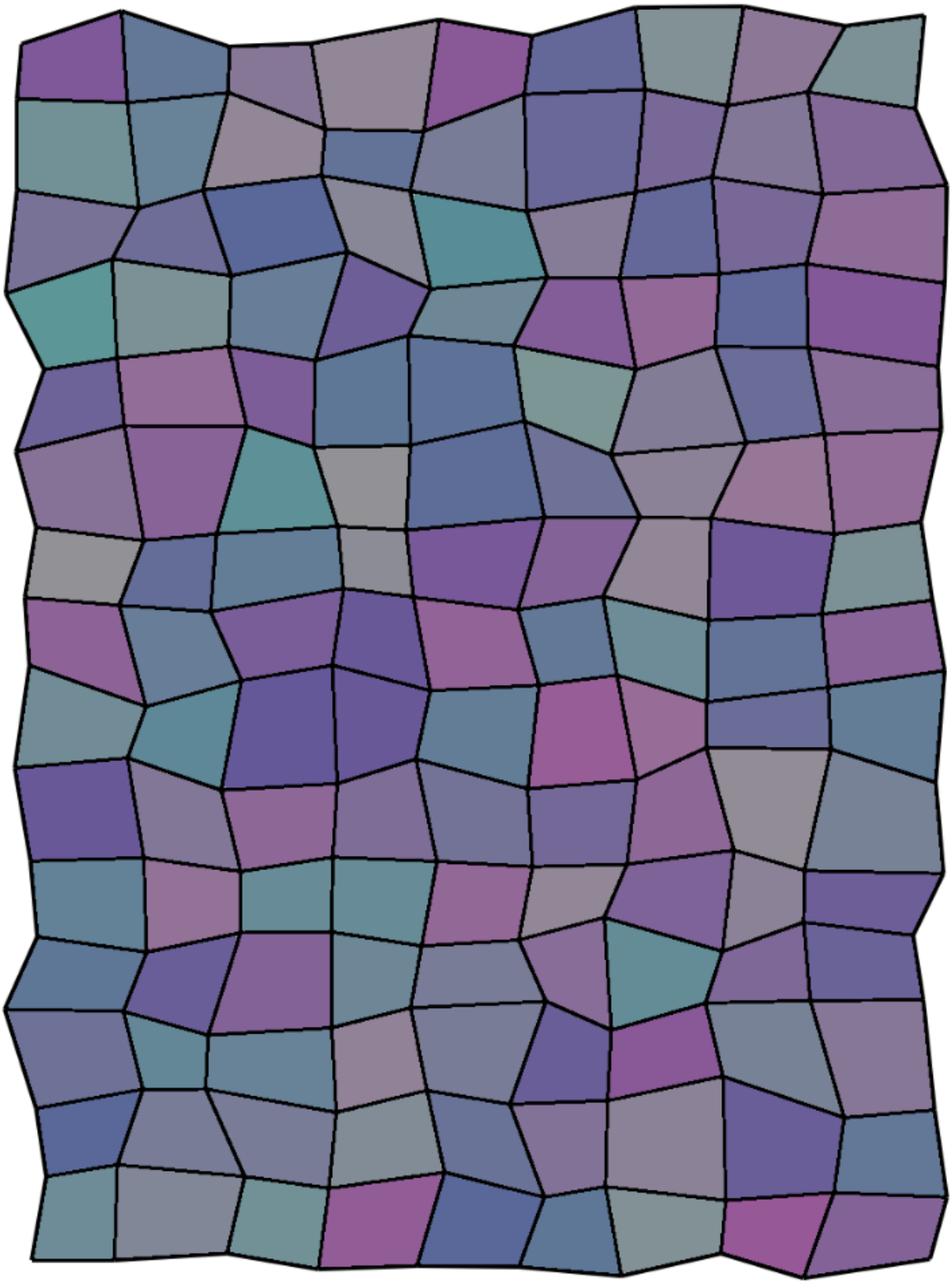
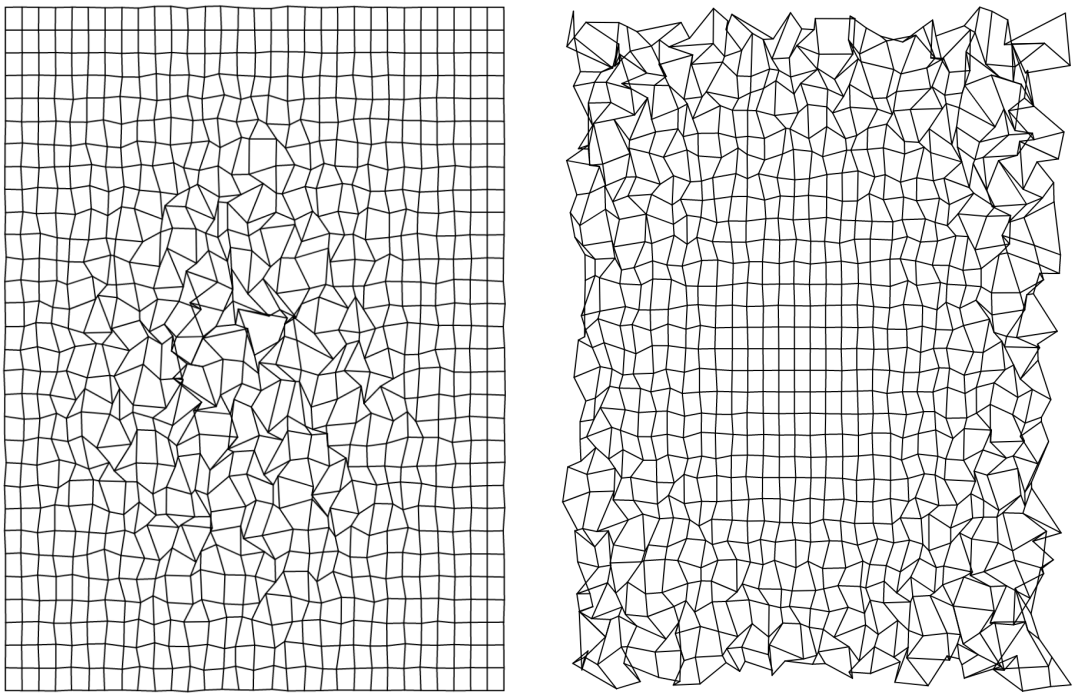


Bild 48: Patchwork

In gewisser Weise hat Georg Nees das *Remixing* seines *Schotter* durch W. J. Kolomyjec (Bild 42) schon selbst vorweg genommen. Allerdings hat er dafür nicht seinen *Schotter* als Ausgangspunkt genommen, sondern das *Gewebe*. Den dortigen zufälligen Abweichungen der Koordinatenpunkte des Gewebes fügt er die Abhängigkeit von der Entfernung vom Mittelpunkt hinzu. Wieder gibt es zwei Varianten: Die Abweichung kann mit der Annäherung an den Mittelpunkt zunehmen (in der folgenden Abbildung links) oder sie kann mit der Entfernung vom Mittelpunkt zunehmen (in der Abbildung rechts).



Anregung: *Für die Umsetzung kann das Gerüst von* Gewebe *genommen werden. Statt der Erstellung der Liste der Koordinatenpunkte in* **wiederhole***-Schleifen habe ich in diesem Fall* **Für***-Schleifen gewählt. Diese erlauben eine flexiblere Steuerung als die* **wiederhole***-Schleifen. Die Zufallsabweichungen lassen sich über die Abstände vom Mittelpunkt ermitteln. Das Plateau bzw. der ebene Außenbereich können über bedingte Verzweigungen berücksichtigt werden.*

Das brauchen wir von **Snap!***:*



Hinweis: Der folgende Befehl gehört zur Blockbibliothek, die nachgeladen werden kann und in *Anhang C* beschrieben wird.



Mit dem C-förmigen Block **für .. schritt .. bis** kann eine Abfolge von Befehlen zusammengefasst werden, die solange wiederholt wird, wie durch die Variable **i** angegeben. In jedem Durchlauf wird **i** um den Wert von **schritt** erhöht, **bis** der Endwert erreicht oder überschritten wird.

16. HOMMAGE À NAKE: EINFACHE GRAFIKEN

„Eines Tages kam mein Chef und sagte zu mir »Herr Nake, wir kaufen eine Zeichenmaschine, haben dafür aber keine Software.« Dann hat er mich gefragt »Würden Sie das machen?« und ich sagte nur »Ja.« Im Rückblick finde ich das klasse, denn eigentlich hätte ich ehrlich antworten müssen, dass ich keine Ahnung hatte. Jetzt sollte ich plötzlich etwas programmieren, das Bilder hervorbrachte, nicht Zahlen. Ich sollte das Analoge schlechthin – also die Zeichnung – digital machen. Beim Testen meiner Software sind dann die ersten Grafiken entstanden. Das waren einfache Versuche, bestimmte Vektoren zu Papier zu bringen." (Burmeister, 2013, S. 13)

So beschreibt Frieder Nake (geb. 1938 in Stuttgart) in einem Interview, wie er mehr oder weniger zufällig zur Computerkunst gekommen ist. Er entwickelte daraufhin in Maschinensprache die Steuerungsroutinen für den Zuse Graphomat Z 64 und ein Programmpaket compART ER56. Damit produzierte er ganze Serien von Grafiken, bei denen er selbst von *ad-hoc-Programmen* und *einfachen Grafiken* spricht. Er hat mit ihnen aber markante Beispiele der frühen Computerkunst geschaffen, die auf vielen Ausstellungen vertreten waren (und heute wieder sind). Mit seinem Bild *Labyrinth* (vgl. dazu Bild 50) war er 1967 Preisträger beim *Computer Art Contest* der Zeitschrift Computers and Automation.

Mit einer Notiz in *PAGE* (The Bulletin of the Computer Arts Society): *There should be no computer art* (Nake, 1971) verabschiedete sich Frieder Nake zunächst aus politischen Gründen von der Computerkunst. Die Lektüre lohnt sich, weil er in der Notiz grundlegende Probleme zur Positionierung der Computerkunst im Kunstbetrieb anspricht. Inzwischen - seit Ende der 80er-Jahre - hat er als Informatik-Professor für grafische Datenverarbeitung und interaktive Systeme an der Universität Bremen das Thema wieder aufgenommen. Mit compArt hat er ein Projekt ins Leben gerufen, mit dem *ästhetische Objekte interpretiert und generiert* werden sollen. Zentrales Element ist die Datenbank daDA, die die wohl umfassendste Informationssammlung zum Thema Digital Art darstellt.

Die Datenbank ist u.a. auch Fundort für seine *einfachen Grafiken*. Für einige davon gibt es umgangssprachliche Beschreibungen, die wir für die programmtechnische Umsetzung, das *Recoding*, verwenden können. Die drei folgenden Beispiele werden das illustrieren.

16.1 Rechteckschraffuren

Das Bild *Rechteckschraffuren* von Frieder Nake diente im Kapitel 1: *Computerkunst* als ein Beispiel, mit dem typische Bildelemente und Darstellungsformen illustriert werden konnten (siehe Bild 1). Es war auch - zusammen mit einer Reihe durchnumerierter Varianten mit dem gleichen Titel - Bestandteil der Ausstellung *Herstellung von zeichnerischen Darstellungen, Tonfolgen und Texten mit elektronischen Rechenanlagen*, die 1966 am Deutschen Rechenzentrum in Darmstadt präsentiert wurde. Anlässlich dieser Ausstellung entstand eine der ersten Publikationen, die sich explizit der Computerkunst widmete (Nake et

al., 1966). In dieser eher technisch orientierten Broschüre beschreibt Nake die Anforderungen an ein Programm, das diese Rechteckschraffuren erzeugen soll, wie folgt (a.a.O., S. 3 ff.):

„Das gemeinsame Thema dieser Bilder ist das Verteilen von horizontal oder vertikal schraffierten Rechtecken, deren Seiten parallel zu den Bildrändern sind. Es ist hierfür ein Programm aufzustellen, das die Aufgabe erfüllt, die Klasse aller Zeichnungen zu produzieren, die aus Schraffuren in Rechteckgestalt parallel zu den Bildrändern bestehen. Um dies zu ermöglichen, müssen alle jene Elemente einer Zeichnung dieser Klasse festgelegt werden, die zufällig (nicht fest vorgegeben, variabel, der Intuition unterworfen) sein sollen. Das sind bei den Bildern:

1. *Anzahl N der Schraffuren pro Bild.*
2. *Ort x, y pro Schraffur (z. B. bestimmt durch die Lage der linken unteren Ecke des Rechtecks).*
3. *Länge l und Höhe h pro Schraffur.*
4. *Anzahl M der Striche pro Schraffur.*
5. *Linienführung L pro Schraffur (z. B. sei nur vertikal oder horizontal zulässig).*
6. *Zeichenstift s pro Schraffur (Strichstärke, Farbe).“*

Außer der Bildgröße werden alle Kennwerte zufällig festgelegt. Der Vorschlag zur programmtechnischen Umsetzung hält sich eng an die Beschreibung, d.h. es werden gleichlautende Variablennamen und Blöcke verwendet.

1. Einleitend werden diese Kennwerte festgelegt.
2. In einer **wiederhole**-Schleife werden **N** Schraffuren vorbereitet und gezeichnet.
3. In der Prozedur **startpunkt** wird die Schildkröte an den Ausgangspunkt einer Rechteckschraffur geschickt.
4. In der Prozedur **schraffur** wird sie dann gezeichnet.



In beiden Prozeduren wird der Befehl **springe nach x y** verwendet.

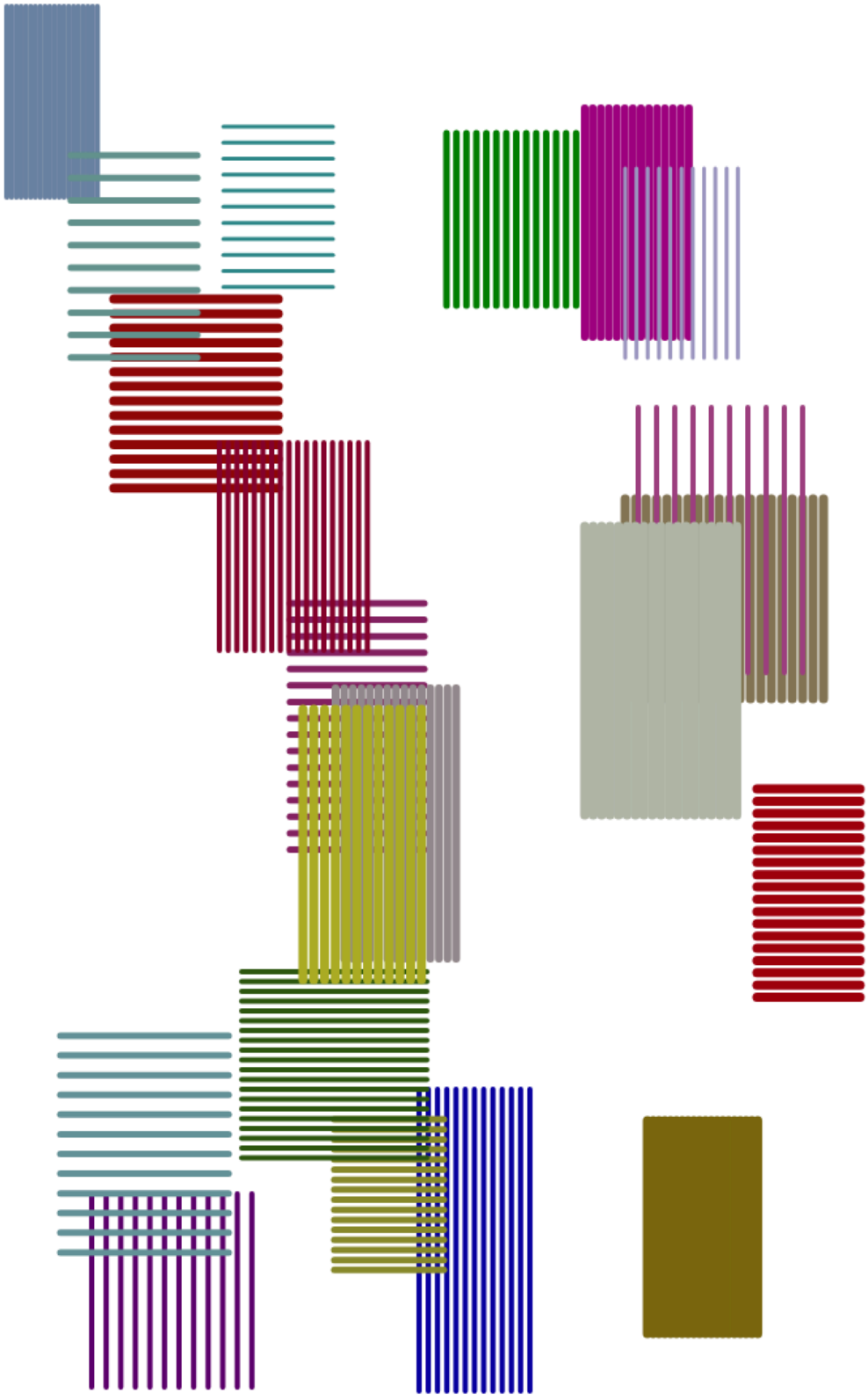
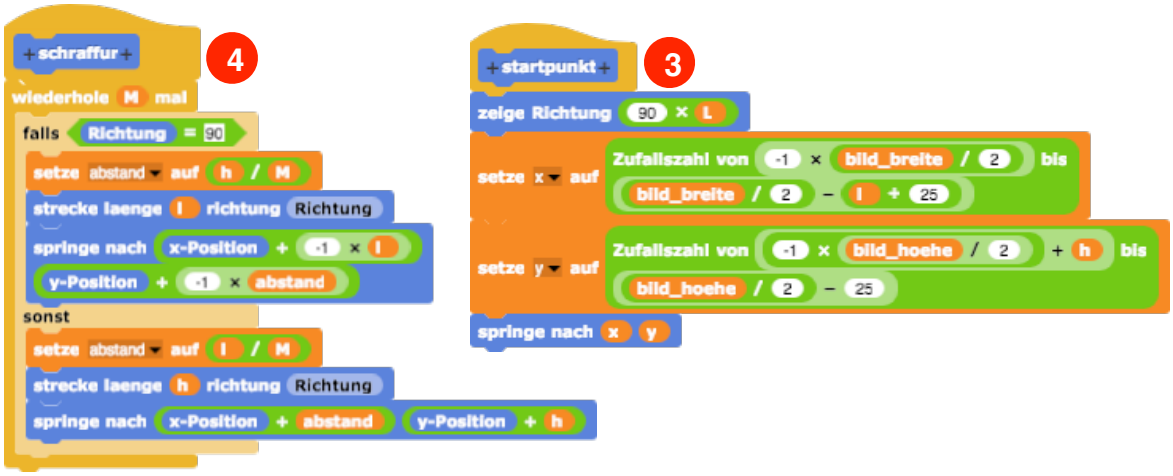


Bild 49: Remixing Rechteckschraffuren (Farbe)



Mit der gezeigten Fassung können Varianten von Bild 1 mit schwarzen Linien vor weißem Hintergrund erzeugt werden. Bild 49 ist ein *Remixing*, bei dem die Stiftfarbe zufällig festgelegt wird. Im RGB-Modell können dabei die Farbkomponenten mit **Zufallszahl von ... bis ...** festgelegt werden.

Anregung: *Durch Veränderung der zulässigen Größenbereiche für Strichlänge, Abstände und Anzahl der Striche pro Schraffur lassen sich weitere Varianten erzeugen. Ein veränderter Bildcharakter entsteht, wenn die Linienführung nicht nur horizontal oder vertikal erfolgt, sondern auch „schräge" Linien zulässig sind. Das* **springe nach** *den Anfangspunkten der Linien einer Schraffur wird dann etwas aufwändiger.*

Das brauchen wir von **Snap!***:*



Hinweis: Der folgende Befehl gehört zur Blockbibliothek, die nachgeladen werden kann und in *Anhang C* beschrieben wird.



Mit dem Befehl **springe nach x y** bewegt sich die Schildkröte von ihrer aktuellen Position zu der Position, die durch **x** und **y** festgelegt wird. Dabei ist der Zeichenstift angehoben und es wird keine Linie gezeichnet.

Hinweis: der Befehl **springe nach x y** ist in der Wirkung identisch mit dem Befehl **LEER (X,Y)** der Sprache G (vgl.. Kapitel 15: *Hommage à Nees: Die Sprache G*).

16.2 Labyrinth

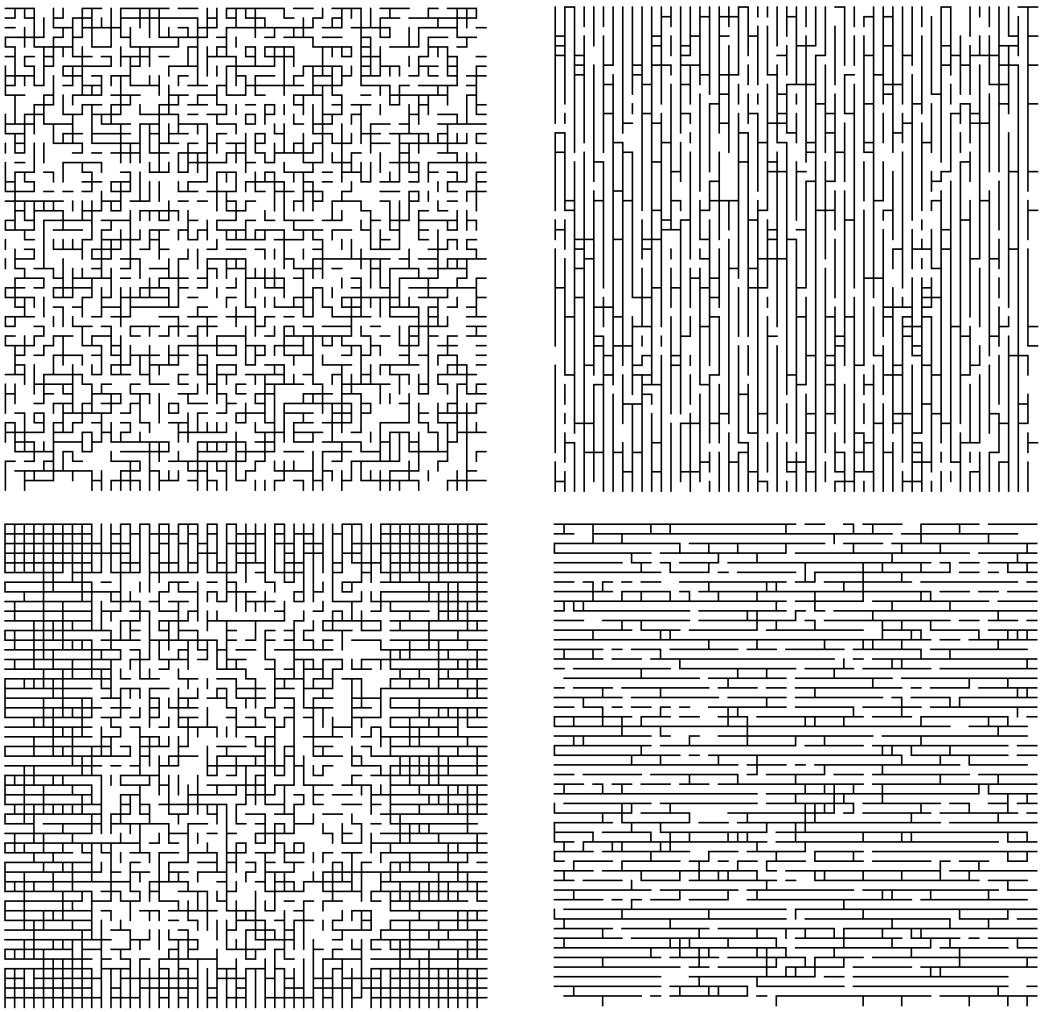
Das nächste Projekt bezieht sich auf das preisgekrönte Bild *Labyrinth*, von dem Nake etliche Varianten vorgestellt hat (zu finden z.B. in Franke, 1971, S. 31). Das Ausgangsbild dafür (wie in der folgenden Abbildung links oben) zeigt in einem 50*50-Raster senkrechte und horizontale Linien, die zufällig - bei Verwendung gleichverteilter Zufallszahlen - verteilt sind. Für die Erzeugung verwendet Nake ein frei wählbares Zeichenrepertoire. Im gezeigten Beispiel sind es drei Zeichen, die über die Felder des Grundrasters verteilt werden: Es kann eine horizontale Linie sein, eine vertikale Linie

oder ein Leerzeichen, d.h. der Verzicht auf eine Linie. Es können auch beide Linien in einem Feld vorkommen, da sie unabhängig voneinander erzeugt werden.

Die Umsetzung erfolgt - wie schon mehrfach praktiziert - in einer Doppelschleife, in der in jedem Feld ermittelt wird, ob eine horizontale oder vertikale Linie gezeichnet werden soll. Abweichungen von der im Ausgangsbeispiel gezeigten Gleichverteilung der Linien können dann über Grenzwerte für die Zufallszahlen gesteuert werden.

In der folgenden Abbildung oben rechts wird z.B. bei einer Zufallszahl < 90 eine senkrechte und bei einer Zufallszahl < 15 eine waagrechte Linie gezeichnet. Entsprechend seltener tauchen waagrechte Linien auf; das Bild wird von den senkrechten Linien geprägt. In der Abbildung darunter verhält es sich genau umgekehrt.

Die Steuerung des Zufalls lässt sich vielfältig verfeinern. So lassen sich auch Verteilungen erzeugen, die zu Bildern führen, die stark an die *Komputerstrukturen* von Peter Struycken erinnern (vgl. Bild 20). In der Abbildung unten links wird so eine mittige Ausdünnung erreicht.



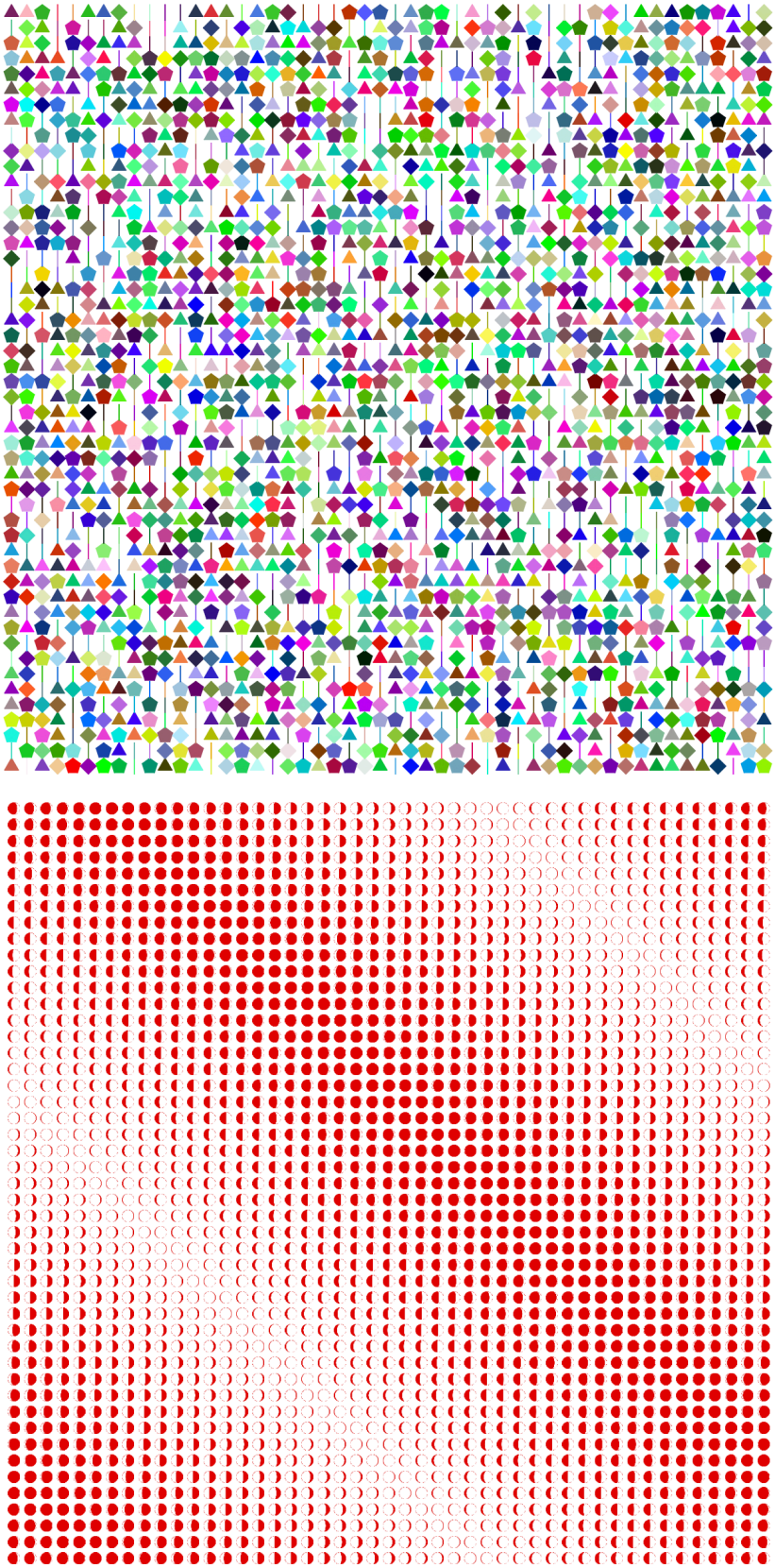


Bild 50: Remixing Labyrinth (Zeichen und Farbe)

Nake spricht bei *Labyrinth* bewusst allgemein von einem ‚frei wählbaren Zeichenrepertoire'. Seine Liniengrafiken stellen also nur eine spezielle Ausprägung der Klasse möglicher *Labyrinth*-Bilder dar. Für ein *Remixing* liegt es nahe, Nake beim Wort zu nehmen, und alternative Zeichenrepertoires auszuprobieren. Durch ein anderes Zeichenrepertoire wird der Bildcharakter bei gleichem Grundschema deutlich verändert.

Das Zeichenrepertoire können Sie etwa mit importierten Kostümen oder durch direktes Zeichnen anderer Grafikelemente erweitern. In Bild 50 (oben) wird statt dem Zeichnen einzelner Linien das Zeichnen von n-Ecken (mit dem Block **n-eck-rekursivgefuellt um x y n radius**) verwendet. Die Zahl **n** der Ecken variiert zwischen 2 und 5 und die Farben wurden im RGB-Farbmodell gewählt.

Für Bild 50 unten habe ich Kostüme importiert (die später auch im Kapitel 21: *Alles in Bewegung* zur Animation benötigt werden), die verschiedene Mondphasen darstellen. Das Bild entsteht - ganz ohne Zufallselemente - allein durch Setzen der aufeinander folgenden Kostüme in den Wiederholungsschleifen für Spalten und Reihen.

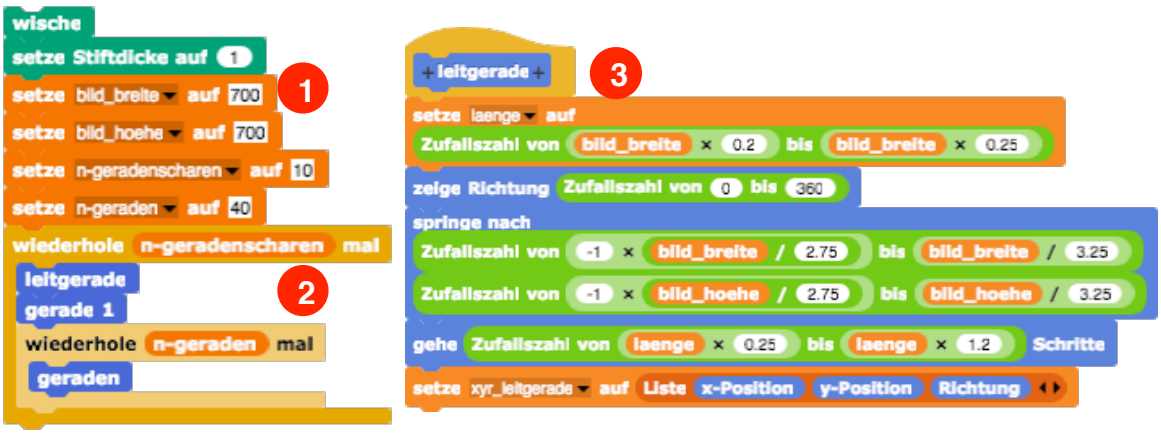
16.3 Geradenscharen

Die Bildserie *Geradenscharen* von Frieder Nake gehört für mich zu den eindruckvollsten Beipielen der frühen Computerkunst. Auch wenn wir die Entstehung der Bilder algorithmisch nachvollziehen und Recoden können, so zeichnen sie sich doch durch immer wieder überraschende Konstellationen aus, die von einer eigenen Dynamik geprägt sind. Für die programmtechnische Umsetzung folgen wir einer Beschreibung von Frieder Nake (Nake, 1974, S. 221):

„Es wird eine Zahl n gewählt, die als Anzahl von Geradenscharen interpretiert wird. Für jede solche Schar wird dann eine Leitgerade gewählt (zufälliger Ort, zufällige Steigung). Auf der Leitgeraden wird ein erster Punkt (x,y) gewählt. Durch diesen wird ein erstes Geradenstück mit Steigung α und Länge l gezeichnet, wobei (x,y) Mittelpunkt ist. Von diesem ersten Geradenstück aus werden weitere bestimmt: man schreitet auf der Leitgeraden um die zufällige Distanz d vom Punkt (x,y) aus weiter, ändert den letzten Winkel α um den zufälligen Betrag Δα und die letzte Länge l an beiden Enden um die zufälligen Beträge Δl1 bzw. Δl2. Dabei bleibt der Schnittpunkt von Geradenstück und Leitlinie i.a. nicht Mittelpunkt des Geradenstücks. Eine zufällige Anzahl solcher Geradenstücke werden pro Leitlinie gezeichnet und definieren eine Schar."

Die verschiedenen Schritte können in mehrere Blöcke aufgeteilt werden.

1. Vorbereitend werden die ersten Kenngrößen festgelegt:
   - **bild_breite:** Breite des Gesamtbildes, an dem sich die Geradenlängen ausrichten werden, um ein Überschreiten der Bildgrenzen zu verhindern.
   - **bild_hoehe:** Höhe des Gesamtbildes.
   - **n-geradenscharen:** Anzahl der Geradenscharen.
   - **n-geraden:** Anzahl der Geraden einer Geradenschar.

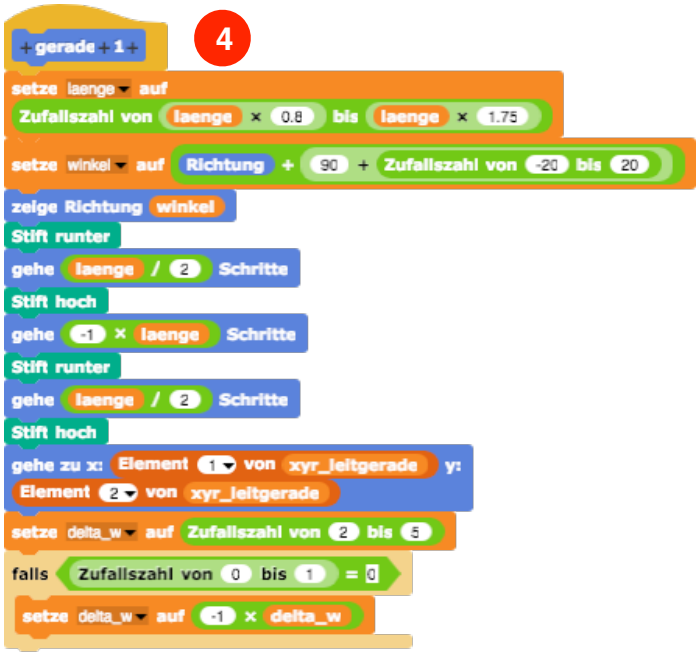


2 In einer äußeren Schleife wird die **leitgerade** bestimmt und daran ausgerichtet die erste Gerade **gerade 1**, an der wiederum sich die restlichen Geraden einer Geradenschar ausrichten; das erfolgt in einer zweiten inneren Schleife.

3 Da die **leitgerade** lediglich eine unsichtbare Hilfsfunktion übernimmt und in der fertigen Zeichnung nicht zu sehen sein wird, können wir die Vorarbeit für das Zeichnen der Geradenschar darauf reduzieren, die Richtung, den Anfangspunkt und den Startpunkt der Geraden zu bestimmen und in der Liste **xyr_leitgerade** zu speichern.

4 Im Block **gerade 1** wird **laenge** und **winkel** der ersten Gerade festgelegt und diese gezeichnet. Danach wird die Schildkröte an den Ausgangspunkt auf der Leitgeraden zurück geschickt und danach der Winkel für die folgende Gerade bestimmt.

5 Im Block **geraden** wird die Schildkröte zum nächsten Ausgangspunkt verschoben, die Verkürzung der Gerade an beiden Enden mit **delta_l1** und **delta_l2** ermittelt und die resultierende Gerade gezeichnet.



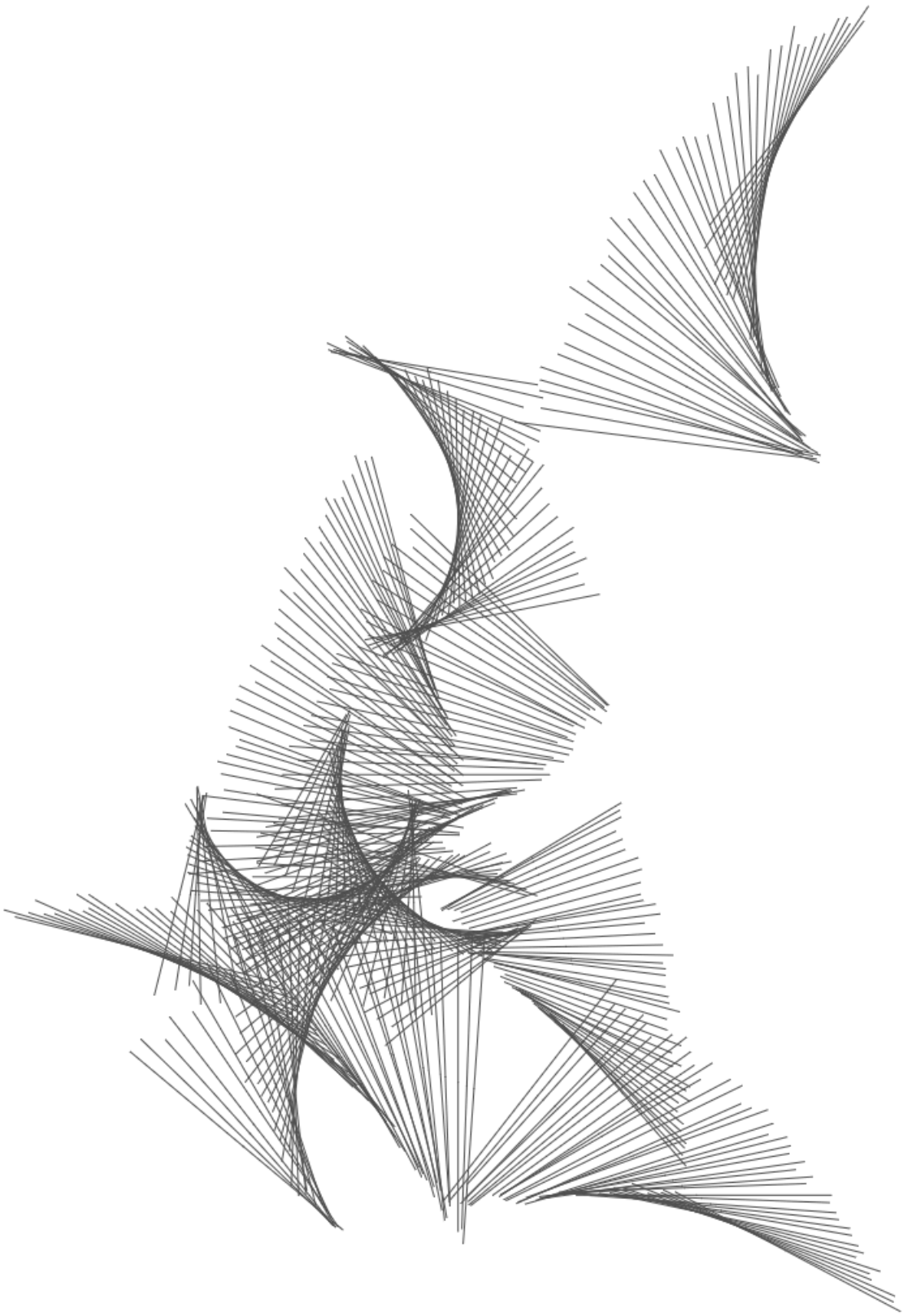
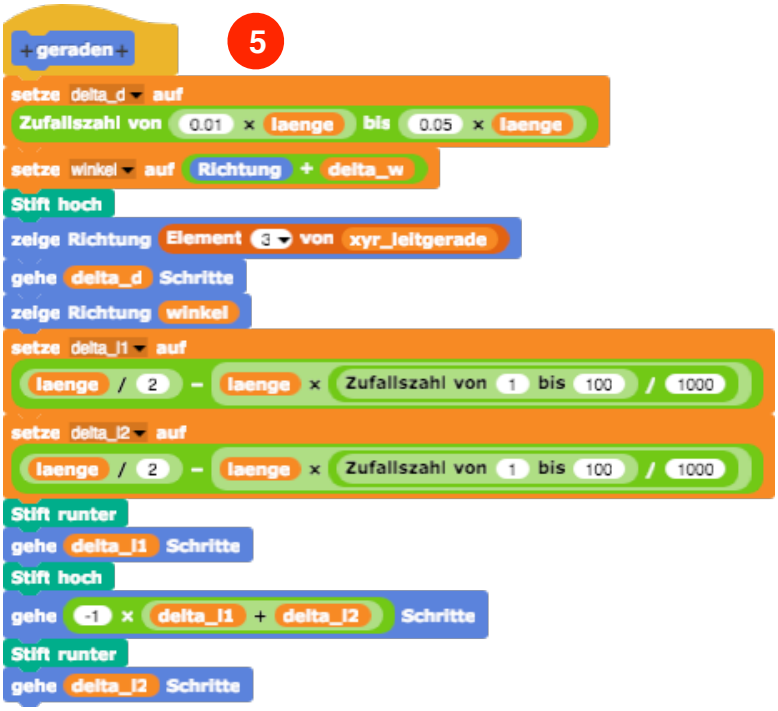


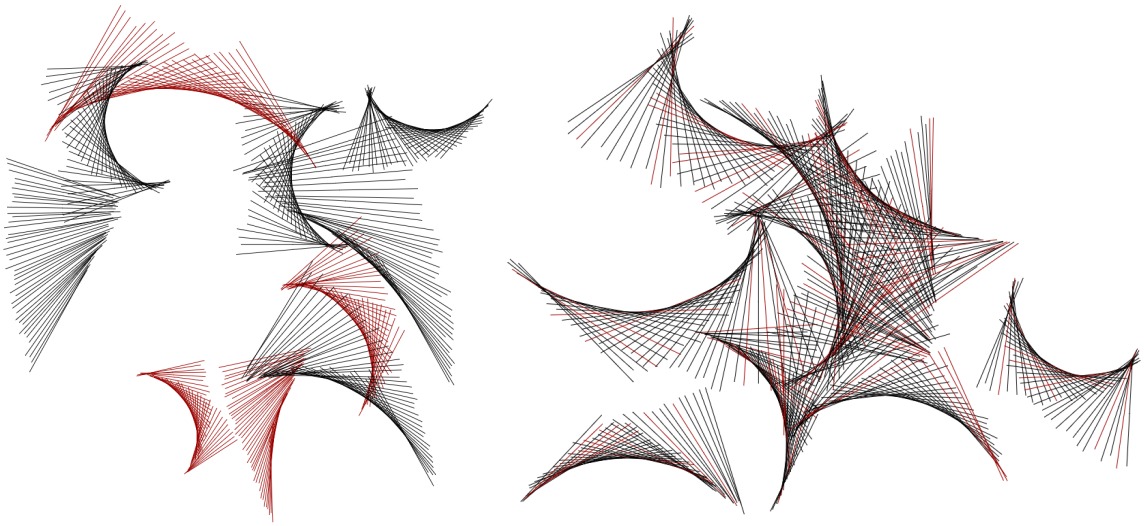
Bild 51: Hommage à Nake: Geradenscharen



Ein Ergebnis dieses Programms ist Bild 51, das stark den Originalen ähnelt.

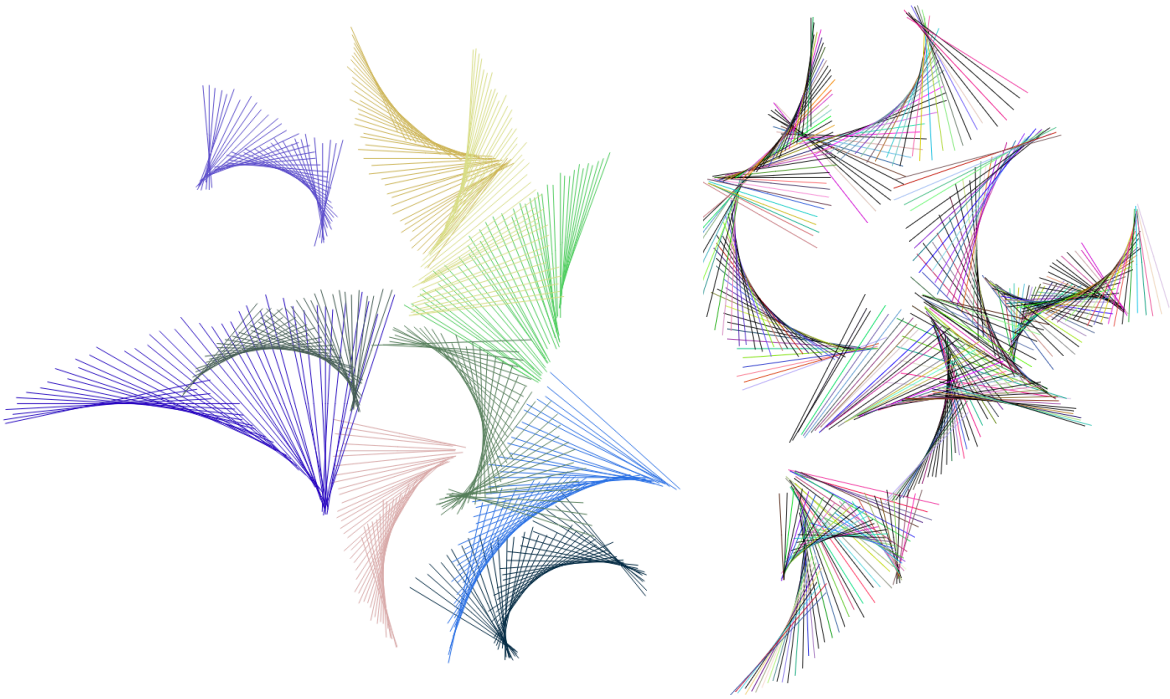
Anregung: *Über die Variation der Kenngrößen lassen sich sehr unterschiedliche Geradenscharen erzeugen. Im einfachsten Fall reicht dafür die Änderung der Anzahl der Geradenscharen* n-geradenscharen *und/oder der Zahl der Geraden pro Geradenschar* n-geraden. *Auch kann deren Zahl für jede einzelne Geradenschar neu bestimmt werden. Daneben ist natürlich auch die Änderung der Geradenverschiebungen* delta_d*, der Geradenverkürzungen* delta_l1 *und* delta_l2 *sowie der Winkelveränderung* delta_w *möglich.*

Wie schon mehrfach praktiziert, liegt es nahe, für das Remixing zusätzlich mit Farben zu arbeiten. Bereits bei Nake finden sich Varianten seiner Geradenscharen, bei denen er (sehr dezent) Farbe einsetzt, wobei einzelne Scharen innerhalb des Gesamtbildes eingefärbt werden.



Die vorige Abbildung links entspricht der sparsamen Farbgebung einzelner Scharen bei Nake. In der Abbildung rechts sind in gleicher Weise einzelne Geraden innerhalb einer Schar eingefärbt.

Hinweis: *Die sparsame Farbverwendung kann ausgeweitet werden, indem mehr Geraden bzw. Geradenscharen eingefärbt werden und indem das ganze Farbspektrum ausgeschöpft wird. Die folgenden Abbildungen verdeutlichen dies, wieder mit gleichfarbigen Scharen (links) bzw. unterschiedlich gefärbten Geraden innerhalb einer Schar (rechts).*



17. HOMMAGE À MICHAEL NOLL: STILSTUDIEN

Für drei der Pioniere, die den Computer für die Erzeugung von Kunst einsetzten, wird gerne das Kürzel „die großen 3N“ verwendet. Dazu zählen Georg Nees und Frieder Nake, von denen wir bereits einige Arbeiten in eigenen Kapiteln kennen gelernt haben. Als Dritter gehört zu den 3N der amerikanische Ingenieur und Professor für Telekommunikation A. Michael Noll (geb. 1949 in Newark, USA). Als Forscher an den Bell Labs - einem Forschungslabor der Bell Telephone Laboratories Inc. - hatte er Gelegenheit, sich mit den Möglichkeiten des Computers für den künstlerischen Einsatz zu beschäftigen. Seine ersten Bilder stammen wohl bereits aus dem Jahr 1962. Er war - zusammen mit Bela Julesz - 1965 an der ersten Computerkunst-Ausstellung in den USA in der New Yorker Howard Wise Gallery beteiligt.

Das Spektrum der grafischen Arbeiten von Michael Noll war sehr breit. Eine seiner ersten und bekanntesten haben wir im Kapitel 9: *Figurenbaukasten - Linien und Strecken* schon recoded: *Gaussian Quadratic* (Bild 23). Eine besondere Rolle kommt Noll auch deswegen zu, weil er mit einigen Stilkopien berühmter Künstler experimentiert hat. Zwei davon sind Thema der folgenden Abschnitte.

17.1 Recoding Noll, Remixing Riley: Sinusoide

Für die Vertreter der frühen Computerkunst spielte die Analyse bestehender Kunst eine große Rolle. Das gilt besonders für A. Michael Noll, der sich u.a. mit der Op-Art auseinandersetzte. So entstand seine Grafik *Ninety Parallel Sinusoids With Linearly Increasing Period* (vgl. Bild 3) wohl als Reaktion auf die Ausstellung *The Responsive Eye*, die 1965 im Museum of Modern Art in New York stattfand (Seitz, 1965). Diese überaus populäre Ausstellung - u.a. mit Bildern von Albers, Morellet, Riley, Stella und Vasarely - konzentrierte sich auf Wahrnehmungsaspekte in der Kunst, z.B. Bewegungsillusionen, Moiré-Muster oder die Wechselwirkung von Farben.

Es ist fraglich, ob für Noll die optischen Wirkungen entscheidend waren oder nicht doch eher die Möglichkeit, über eine einfache mathematische Beschreibung (in diesem Fall mit Sinuslinien) ein Bild von Riley zu reproduzieren; wenn man so will, war das bereits eine frühe Form des *Recoding* von Kunst:

„*Op-art was a big thing back then – Bridget Riley and all that. It occurred to me too, inspired by Bridget Riley I just thought to make sort of computer version out of that, which was just a bunch of parallel sinus waves with increasing period and than I plotted it. People looked at this and I had critics who said that is better than Bridget Riley, because Bridget Riley cropped it. So since I showed the whole thing some art critics thought I was better, which made me wondering about art critics in general – notorious incompetent, subjective nonsense. I thought if you are doing Op-Art, rather than sitting there drawing it, which was the way she was doing it, the computer was a natural for doing Op-Art.*“ (nach Klütsch, 2007, S. 177)

Die programmtechnische Umsetzung des *Recoding* der *Sinusoide* ist denkbar einfach. Wir können auf etliche im Kapitel 10: *Vom Analogen zum Digitalen* für die *Lissajous*-Figuren eingeführte Blöcke zurückgreifen, nämlich **funktionswert x** und **funktionswert y** sowie **abbildung x** und **abbildung y**. Da wir bei den Sinusoiden keine Schwingungsüberlagerung, sondern nur eine einfache Sinusfunktion benötigen, vereinfacht sich **funktionswert x** auf den Reporter **berichte x** für die aktuelle x-Position und auf den reporter **berichte sin ...** für den Funktionswert von y.



Außerdem übernehmen wir die Kenngrößen **xbmin**, **xbmax**, **ybmin**, **ybmax**, **xmin**, **xmax**, **ymin**, **ymax**, **amplitude** sowie **anzahl**. Hinzu kommt **abstand**, mit dem der Abstand zwischen den 90 Sinuskurven bestimmt wird. Das Bild entsteht dann einfach durch das Erzeugen einer Sinuskurve und deren Wiederholung untereinander mit vorgegebenem Abstand.

Schon mit Blick auf weitere Formen des *Remixing* der *Sinusoiden* soll für deren Erzeugung ein etwas anderer Weg aufgezeigt werden. Denn warum eigentlich zigmal die Sinuskurve neu berechnen? Einfacher - und mit mehr Flexibilität - geht es, wenn die Kurve einmal berechnet und diese Kurve dann als **Kostüm** abgespeichert wird. Snap! bietet genau diese Möglichkeit mit dem Befehl **erstelle kostuem aus stiftspuren**, wobei **stiftspuren** die aktuelle grafische Darstellung auf der Bühne beinhaltet (das kann im Bedarfsfall durchaus eine sehr komplexe Grafik sein).

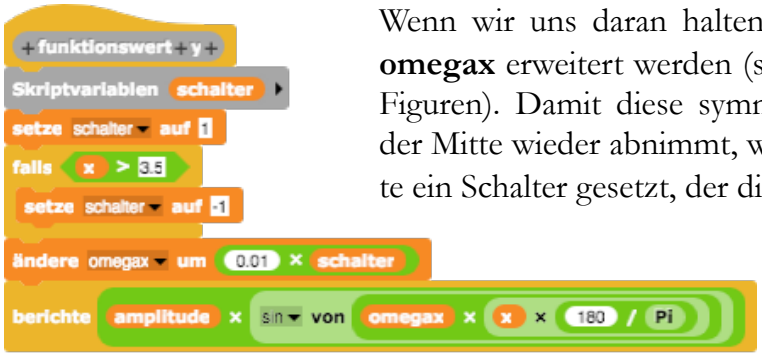
Daraus ergibt sich folgender Programmablauf:



Nach Festlegung der Kenngrößen und eines Startwerts für x (hier nicht gezeigt) wird die Schildkröte

1. zum Augangspunkt geschickt,
2. in einer **wiederhole**-Schleife die **anzahl** Bildpunkte der Sinuskurve errechnet und aus den **stiftspuren**
3. das Kostüm **schwarz** erstellt.
4. Dieses Kostüm wird ausgewählt und damit -
5. ausgehend von einem frei wählbaren Startpunkt - können nun in einer **wiederhole**-Schleife die 90 Kurven im gewünschten Abstand durch **stemple** dargestellt werden.

Es ist festzustellen, dass das damit entstehende Bild (mit konstanter Periode des Sinus) nicht ganz der Vorlage von Noll entspricht, denn bei ihm soll sich ja die Periode kontinuierlich erhöhen (wie bei Bild 3). Allerdings, wenn wir die ursprüngliche Vorlage ***Current*** von Bridget Riley heranziehen (siehe z.B. Klütsch, 2007, S. 176), ist zu bemerken, dass auch Noll etwas unterschlagen hat, nämlich die Symmetrie ihrer Vorlage.



Wenn wir uns daran halten, muss die Sinusfunktion um **omegax** erweitert werden (siehe dazu wieder die *Lissajous*-Figuren). Damit diese symmetrisch zunächst zu- und ab der Mitte wieder abnimmt, wird bei Überschreiten der Mitte ein Schalter gesetzt, der die Zu- bzw. Abnahme steuert.

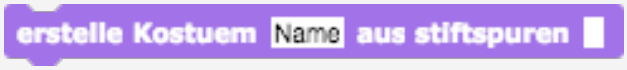
Mit dieser kleinen Erweiterung nähert sich das Ergebnis nun ziemlich genau der Vorlage *Current* von Bridget Riley an (Bild 52).

Hinweis: *Kostüme sind mit* **stiftspuren** *nur einmal zu erzeugen, ansonsten kommt es zu Dopplungen mit mehreren Namen (z.B.* Name, Name(2) *usw.), was beim Zugriff leicht zu Verwechslungen führen kann! Beim Sichern eines Projekts werden die erzeugten Kostüme mit abgespeichert, müssen beim erneuten Laden und Ausführen also nicht noch einmal erzeugt werden!*

Das brauchen wir von **Snap!**:



Hinweis: Der folgende Befehl gehört zur Blockbibliothek, die nachgeladen werden kann und in *Anhang C* beschrieben wird.



Mit **erstelle Kostuem aus stiftspuren** kann die aktuelle grafische Ausgabe auf der Bühne als Kostüm gesichert werden. Dieses ist dann unter dem angegebenen Namen verfügbar und kann mit allen Befehlen der Blockpalette Bewegung verarbeitet werden.



Der Reporter **stiftspuren** liefert die aktuelle grafische Ausgabe auf der Bühne, die mit **erstelle kostuem** gesichert werden kann.

17.2 Recoding Noll, Remixing Mondrian: Composition With Lines

Der Ingenieur A. Michael Noll verstand sich als Grenzgänger zwischen den Wissenschaften und als Mittler zwischen zwei Kulturen - den Geistes- und Naturwissenschaften. Als solcher war er selbstbewusst genug, einen Artikel über Kunst in einer psychologischen Fachzeitschrift zu veröffentlichen: *Human or Machine: A Subjective comparison of*

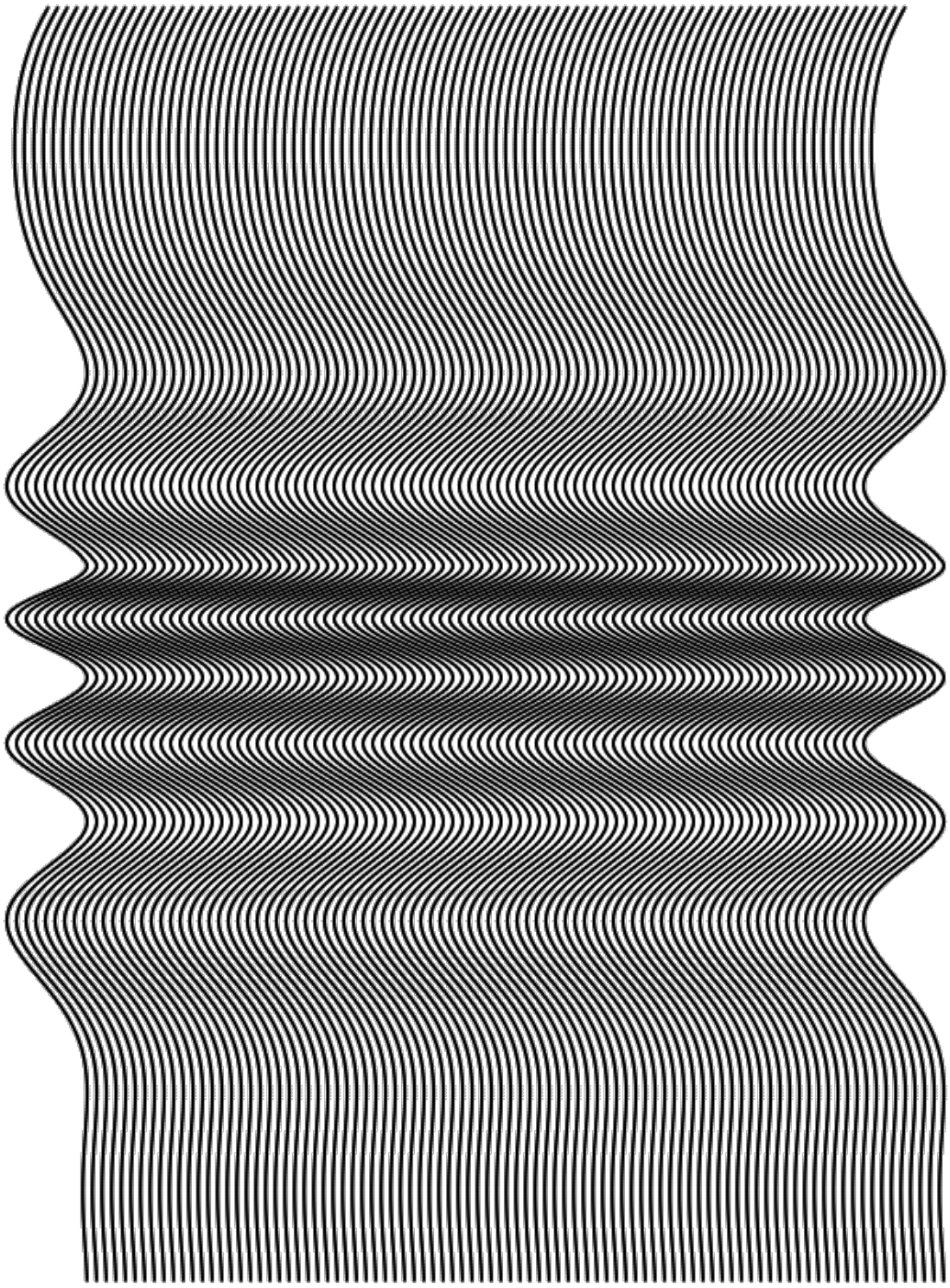
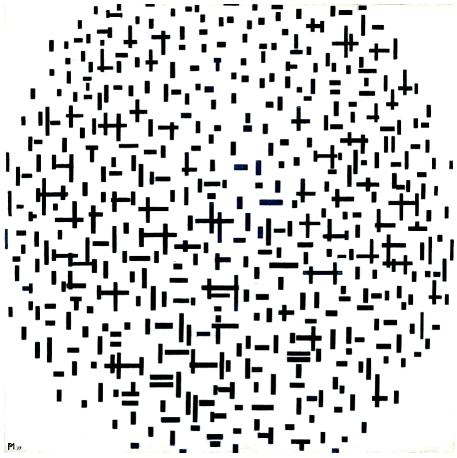


Bild 52: Hommage à Riley: Recoding Current (im Original Querformat)

Piet Mondrians »Composition with Lines« (1917) and a computer- generated Picture (Noll, 1966). Gegenstand des Artikels ist eine empirische Untersuchung an hundert Testpersonen. Diesen legte er Kopien des Bildes *Composition with Lines* von Piet Modrian (unten links) sowie das Ergebnis seines *Recoding* dieses Bildes *Computer Composition with Lines*, also eine computergenerierte Grafik (unten rechts), vor. Die Testpersonen sollten dann entscheiden, welches Bild von Modrian sei und welches vom Computer erzeugt wurde.



Mondrian: Compositie in lijn (1916)



Noll: Computer Composition with Lines (1965)

Interessanterweise bevorzugte die Mehrheit seine Computerversion, schrieb aber dessen Urheberschaft Mondrian zu. Noll notierte, dass diese Teilnehmer das Computerbild als ordentlicher, vielfältiger, phantasievoller, beruhigender und abstrakter als das Mondrian-Bild beschrieben. Noll schloss daraus, dass sie die zufällige Anordnung im computergenerierten Bild mit menschlicher Kreativität verbanden und die geordnete Verteilung der Striche bei Mondrian eher der Maschine zuordneten.

Das Vorgehen Nolls beim *Recoding* des Mondrian-Bildes gleicht der Arbeitssweise, die auch wir bei unseren Projekten vefolgt haben. Er beginnt mit einer genauen Bildanalyse (Noll, 1966, S. 1):

(a) „The outline of the painting is a circle that has been cropped at the sides, top, and bottom;
(b) The vertical and horizontal bars falling within a region at the top of the painting have been shortened in length; and
(c) The length and width of the bars otherwise seem to he randomly distributed.
(d) and fourth, the placement of the bars is not random but seems to follow some scheme so that the entire space is almost uniformly covered."

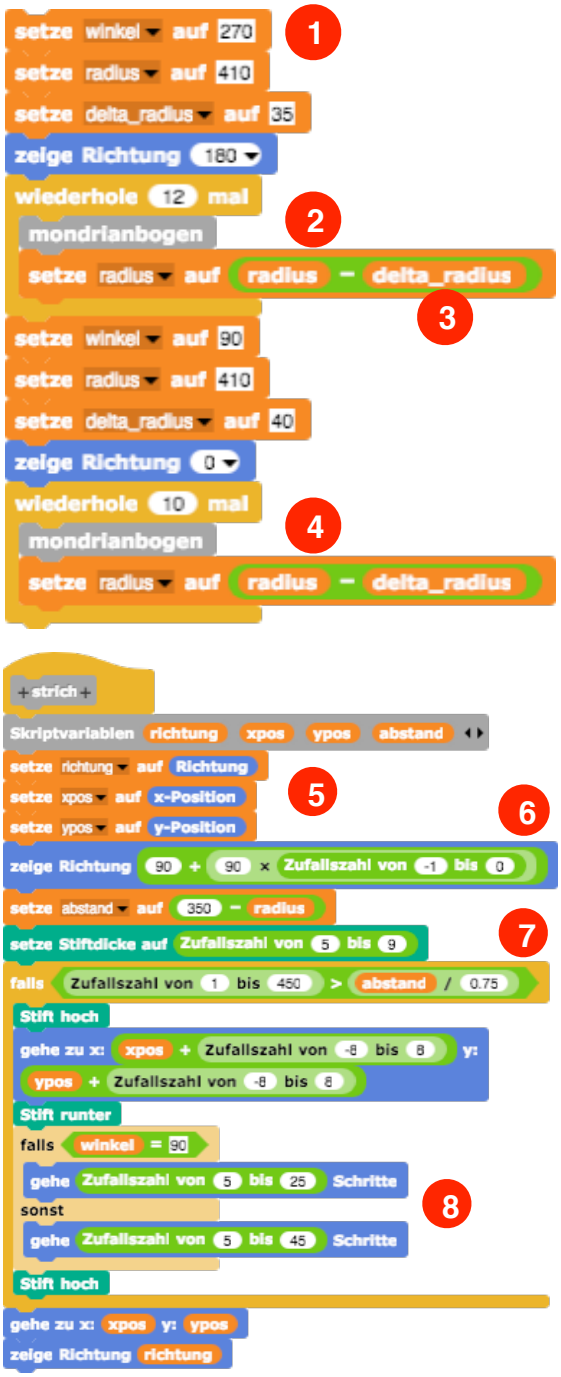
Daraus entwickelt Noll einen Algorithmus für ein Computerprogramm (a.a.O., S. 3):

„*The vertical and horizontal bars in* Computer Composition with Lines *were produced as a series of parallel line segments that were closely enough spaced to slightly overlap each other. Although Mondrian apparently placed his bars in a very orderly manner, the computer was programmed to place*

the bars randomly within a circle of radius 450 units so that all locations were equiprobable. The choice between vertical bar or horizontal bar was equally likely, and the widths of the bars were equiprobable between 7 and 10 lines; the lengths of the bars were equiprobable between 10 and 60 points.

If a bar fell inside a parabolic region at the top of the picture, the length of the bar was reduced by a factor proportional to the distance of the bar from the edge of the parabola. A try-and-error approach was used to insure that the effect of the picture was reasonably similar to Mondrian's Composition with Lines.“

Diese Beschreibung ist allerdings noch nicht präzise genug für die Umsetzung in Snap!.



1 Für die Annäherung an das Original von Mondrian sieht meine Umsetzung das Zeichnen der Linien auf konzentrischen Kreisbögen mit einem **winkel** von 270 Grad vor (mit Ausrichtung nach unten).

2 Für die Kreisbögen greife ich auf den Block **rundbogen um x y radius winkel** zurück, der hier als **mondrianbogen** leichte Anpassungen erfährt (s.u.).

3 Der **radius** wird von Kreis zu Kreis um **delta_radius** reduziert.

4 Die von Noll identifizierte parabolische Region nähere ich mit weiteren konzentrischen Kreisbögen mit 90 Grad an (dieses Mal mit Ausrichtung nach oben).

Im neuen Block **mondrianbogen** werden statt der einzelnen Kreispunkte bei jedem Kreispunkt der Block **strich** aufgerufen (übrigens ein vergleichbares Vorgehen wie bei *Recoding Lissajous (II): Anhängsel*)..

5 Im Block **strich** sind die aktuelle Position und die Richtung zwischenzuspeichern.

6 Zufallsgesteuert wird entweder eine waagrechte oder senkrechte Richtung der Linie eingestellt.

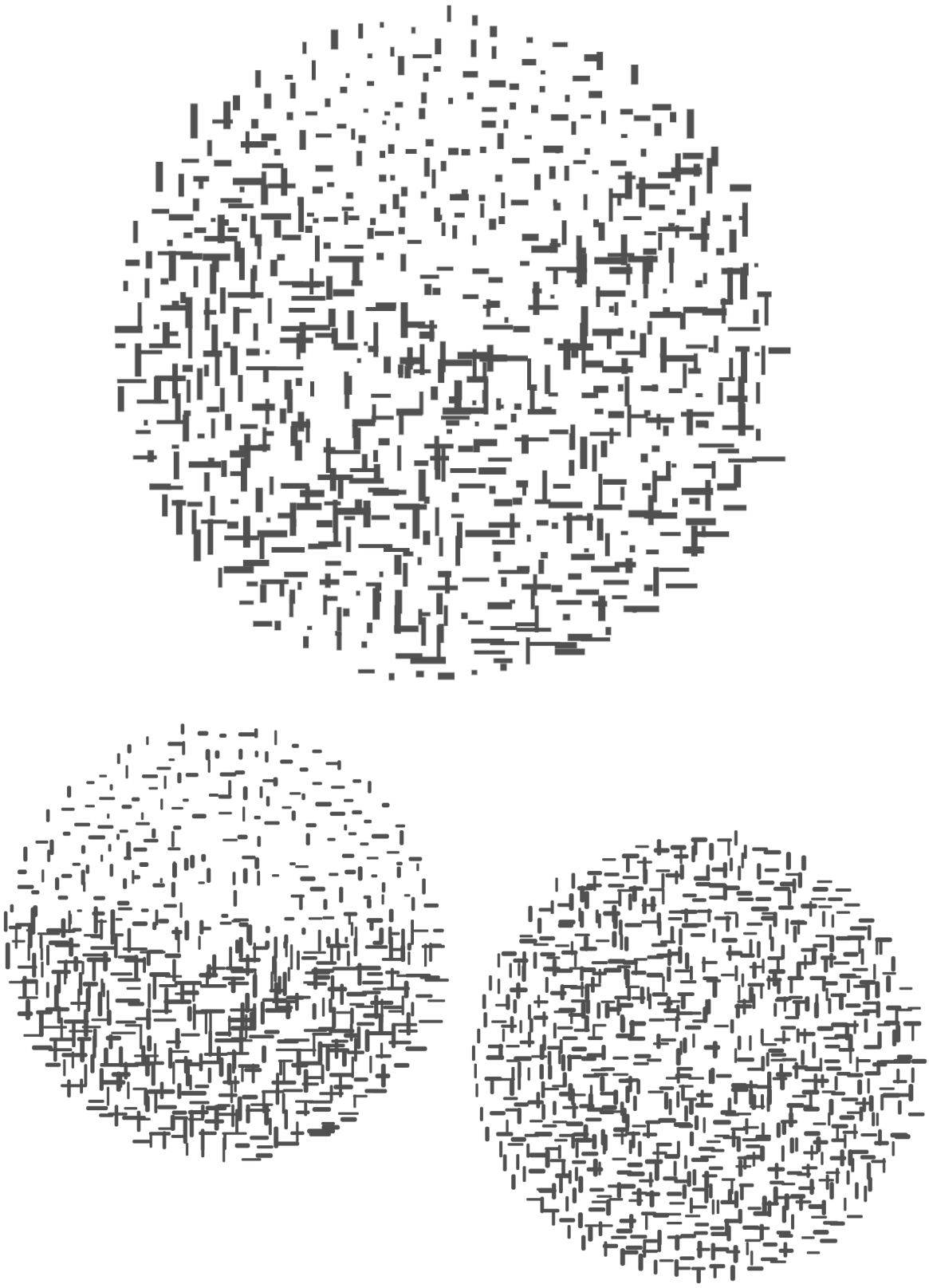


Bild 53: Recoding & Remixing Noll: Mondrian

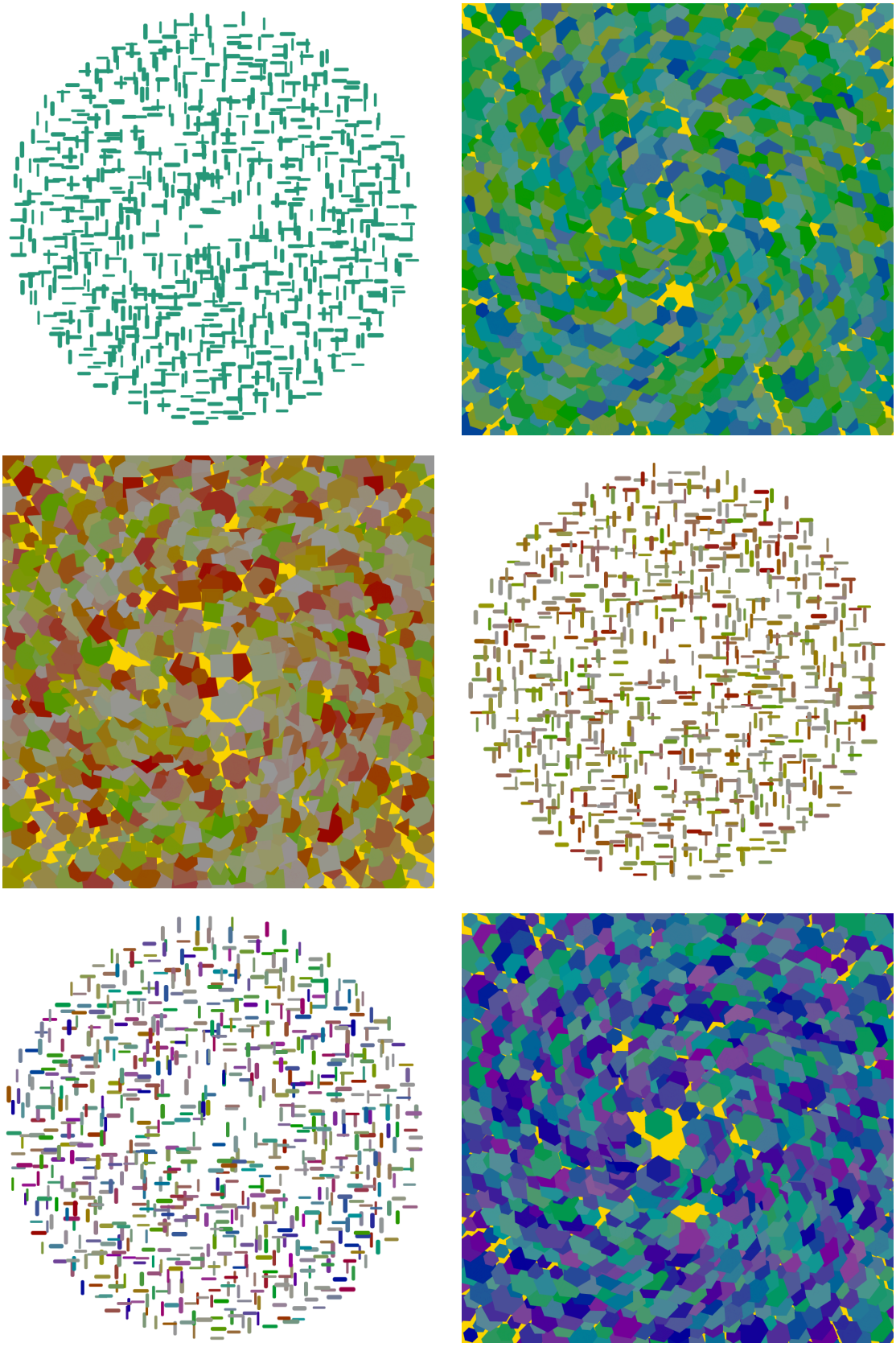


Bild 54: Remixing Noll & Mondrian: Farben und Flächen

7 Je geringer der **abstand** vom Mittelpunkt und je kleiner damit der Kreisumfang wird, desto weniger Linien sollen gezeichnet werden. Das wird über den Grenzwert der Zufallszahlenfunktion erreicht.

8 Abhängig davon, ob das obere oder untere Kreissegment betroffen ist, werden unterschiedlich lange Linien gezeichnet.

Bild 53 oben ist das Ergebnis dieses *Recoding*. Es zeigt große Ähnlichkeit sowohl mit dem Vorbild von Mondrian als auch mit der Arbeit von Noll.

Noll benötigte für seine Untersuchung verschiedene Linienvarianten, orientiert an der Struktur des Mondrian-Bildes. Auch mit der hier erarbeiteten Version ist es leicht, solche Varianten zu erzeugen. Bei Bild 53 unten links ist die parabolische Region auf 180 Grad ausgeweitet, bei Bild 53 unten rechts dagegen auf Null reduziert.

Anregung: *Unabhängig vom möglichst genauen Nachvollzug der Vorlagen von Noll bzw. Mondrian, bietet es sich an, für ein Remixing das inzwischen bekannte Repertoire an Veränderungsmöglichkeiten auszuschöpfen. Wie verändert sich der Bildeindruck, wenn farbige Linien verwendet werden? Welchen Einfluss hat dabei dieVerwendung gleichfarbiger bzw. bunter Linien? Und was bewirkt der Ersatz der Linien durch Polygone?*

Die Beispiele in Bild 54 *können es nur andeuten ... jedenfalls machen z.B. die gefüllten Polygone deutlich, dass oft durch sehr kleine Programmänderungen (hier eben allein der eines einzelnen Befehlsblocks, nämlich Ersatz des Ziehens einer Linie durch das Zeichnen eines Polygons) bildliche Veränderungen erreicht werden können, die die Verwandtschaft zum Ausgangsbild kaum noch erahnen lassen.*

18. HOMMAGE À VERA MOLNAR: UNORDNUNGEN

Unter den Pionieren der Computerkunst nimmt Vera Molnar (geb. 1924 in Budapest) in mehrfacher Hinsicht eine Sonderstellung ein. Zunächst ist sie eine der ganz wenigen Frauen, die sich in dieser Kunstrichtung profiliert haben. Vor allem aber ist sie wohl die erste ausgebildete Künstlerin gewesen, die sich den Computer für ihre Kunst erschlossen hat. Als Malerin hatte sie von Beginn an eine große Affinität zur Konkreten Kunst, was sich in einer frühen Systematisierung und Reduktion ihrer Darstellungsmittel äußerte. Ihre Arbeiten sind gegenstandslos und geometrisch angelegt. Sie gehörte 1960, u.a. zusammmen mit François Morellet, zu den Gründungsmitgliedern der Groupe de Recherche d'Art Visuel (GRAV). Deren Experimente mit Symmetrie/Asymmetrie, Wiederholungen und Zufallseinflüssen erlaubten ihr, eine neue Bildsprache zu entwickeln.

Ihre Versuche einer strengen Systematisierung führten sie zum Prinzip der *machine imaginaire*:

„*Ich stellte mir vor, ich hätte einen Computer. Ich entwarf ein Programm und dann, Schritt für Schritt, realisierte ich einfache, begrenzte Serien, die aber in sich abgeschlossen waren. [...] Ich dachte mir damals, dass die Verbindung von einer strengen Systematik bei gleichzeitiger Freiheit Kunst hervorbringen könnte.*" (Molnar, 1990, S. 16 f.)

Erst 1968 hatte sie im Forschungslabor des Computerherstellers Bull dann die Gelegenheit, tatsächlich mit dem Computer zu experimentieren, also zur *machine réelle* zu wechseln. Zu den ersten dieser Arbeiten gehört ihre Serie *Unordnungen*, aus der hier zwei Projekte übernommen werden.

18.1 Recoding & Remixing: Ein Prozent Unordnung

Bei der Verwendung des Zufalls am Computer verzichtet Vera Molnar auf einen korrigierenden Eingriff, um auf diese Weise *„ungeahnte Bilder zu finden, die man zuvor nie zuvor gesehen hat"* (Molnar, a.a.O.). Bei ihrer Bildserie *Ein Prozent Unordnung* fehlen auf jedem Blatt bei insgesamt 250 Quadraten (in einer 5*5-Matrix) zwischen einem und drei Quadraten. Die Fehlstellen werden zufällig bestimmt. In Bild 55 findet sich oben links die Ausgangssituation ohne Fehlstellen, rechts daneben eine Situation, wie von Molnar beschrieben (mit drei Fehlstellen).

Die programmtechnische Umsetzung brauche ich hier nicht im Einzelnen ausführen. Es reichen zwei geschachtelte Wiederholungsschleifen für die jeweils fünf Reihen und Spalten, in denen wiederum eine Schleife für die einzelnen Quadrate ausgeführt wird. Dabei wird per Zufall (mit vorgegebener Prozentgrenze) bestimmt, ob ein Quadrat gezeichnet oder ausgelassen wird. In Bild 55 sind so die Varianten in der mittleren und unteren Reihe entstanden. Auch ein *Remixing* dieser Vorlage ist denkbar einfach. In Bild 56 werden dazu farbige Quadrate gezeichnet. Statt der Leerstellen werden hier die zufällig ermittelten Stellen mit andersfarbigen Quadraten ausgefüllt.

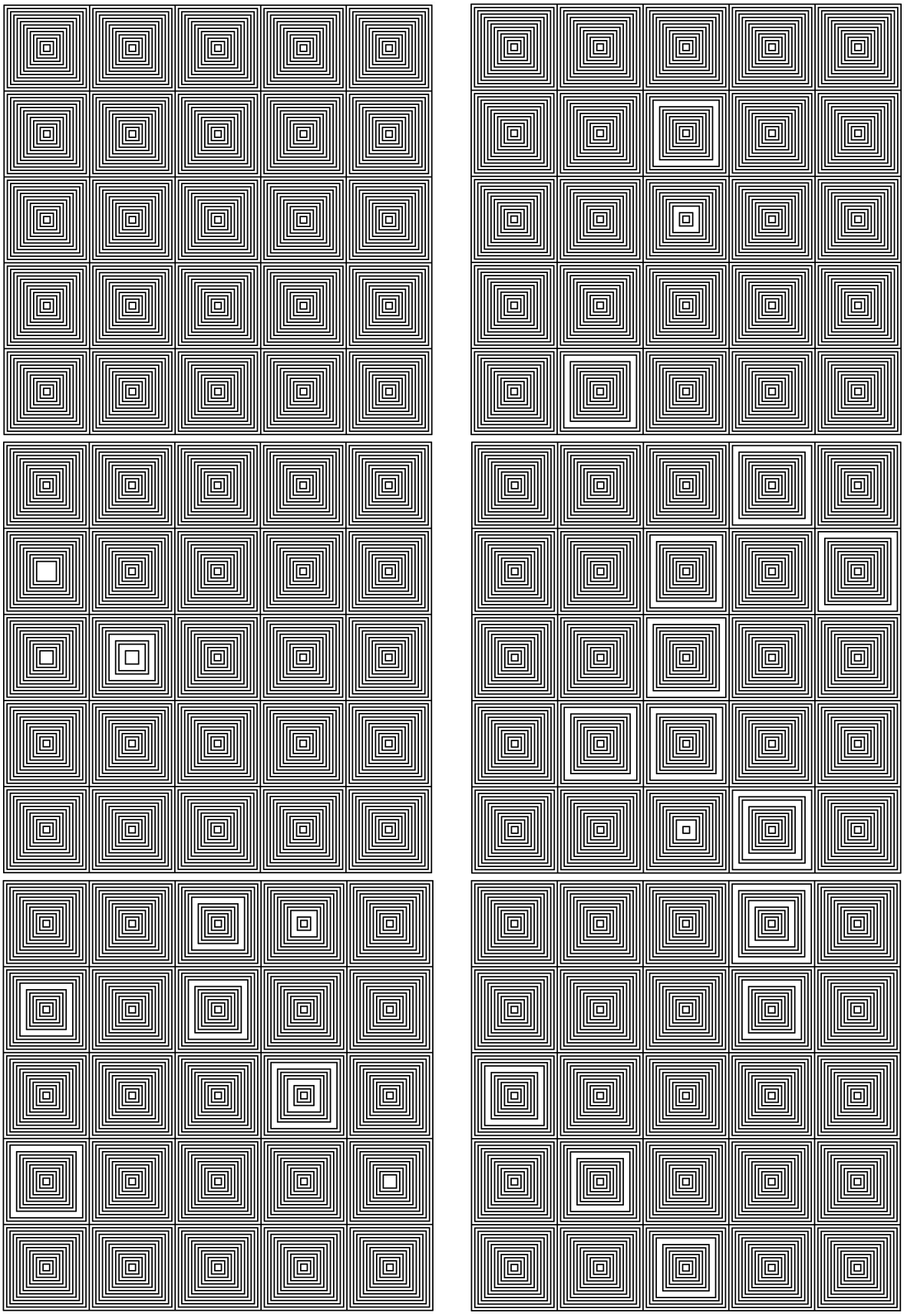


Bild 55: Hommage à Molnar: Ein Prozent Unordnung

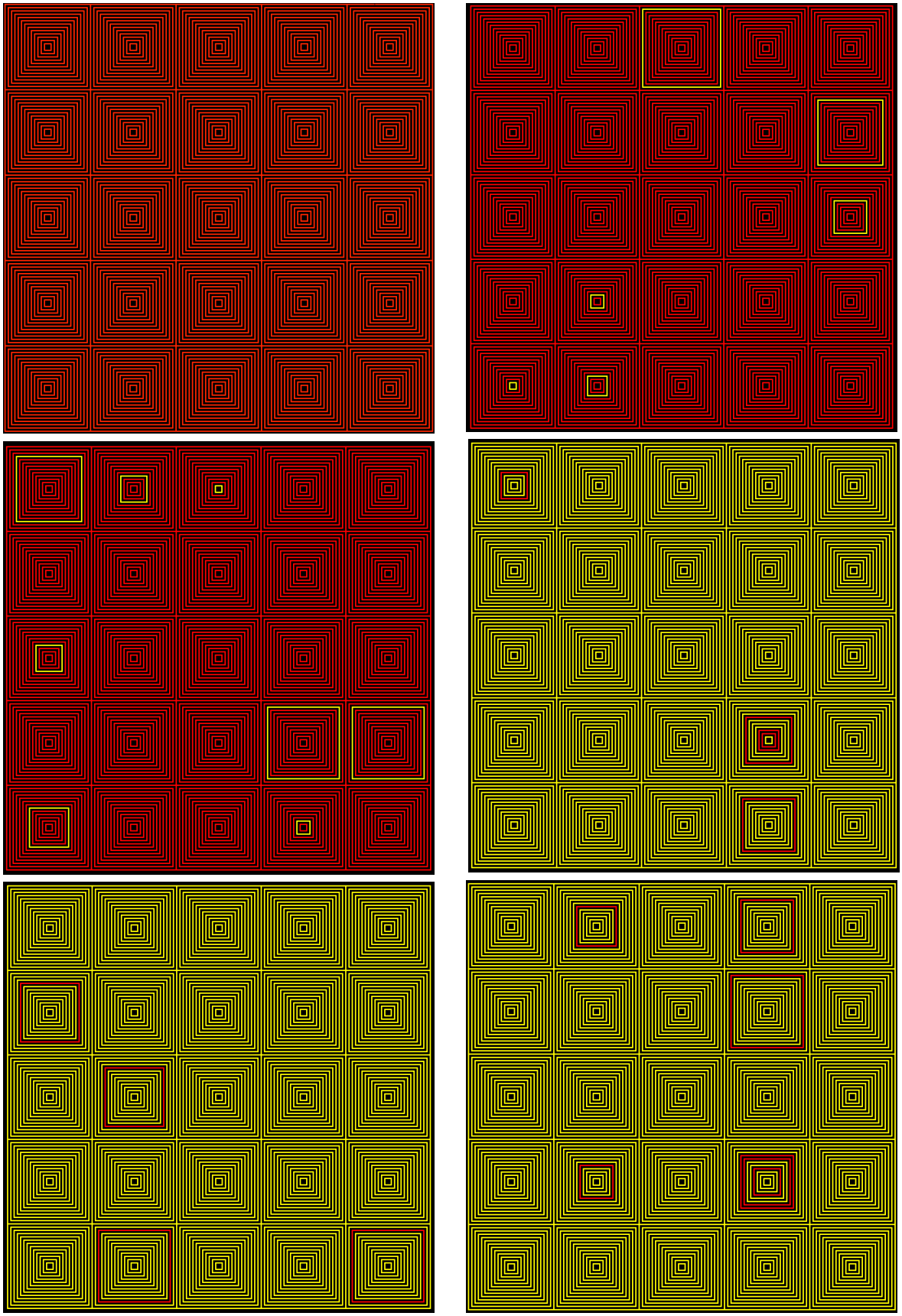


Bild 56: Remixing Molnar: Ein Prozent Unordnung (Farbe)

Hinweis: *Es lohnt sich, diese Bilder möglichst groß anzufertigen bzw. auszudrucken, denn je kleiner das Format, desto eher treten flirrende Effekte oder sogar Bewegungsmuster auf.*

18.2 Recoding & Remixing: (Un)Ordnungen

Die Bildserie *(Un)Ordnungen* (im Original quadratisch) unterscheidet sich von der vorigen Serie gleich doppelt. Zugrunde gelegt wird eine feiner aufgelöste Matrix, hier 30*55. In den Feldern der Matrix werden dann zufallsgesteuert goße Quadrate (jeweils dunkel gefärbt) zentriert eingezeichnet. Ebenfalls zufallsgesteuert werden dann zusätzlich jeweils horizontal bzw. vertikal leicht versetzt kleinere Quadrate in weiteren Farbvarianten gezeichnet. Dem Ganzen wird ein einheitlich eingefärbter Hintergrund gegeben. Die Grundelemente haben dann im Prinzip folgendes Aussehen:

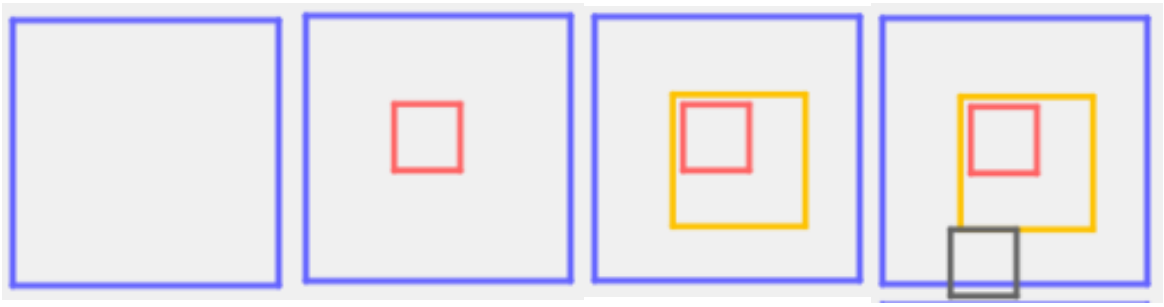


Bild 57 ist ein *Remixing* der originalen Vorlage von Molnar (vgl. Lieser, 2009, S. 40 f.) durch Ändern der Farbgebungen (Original mit Gelbtönen und im Querformat).

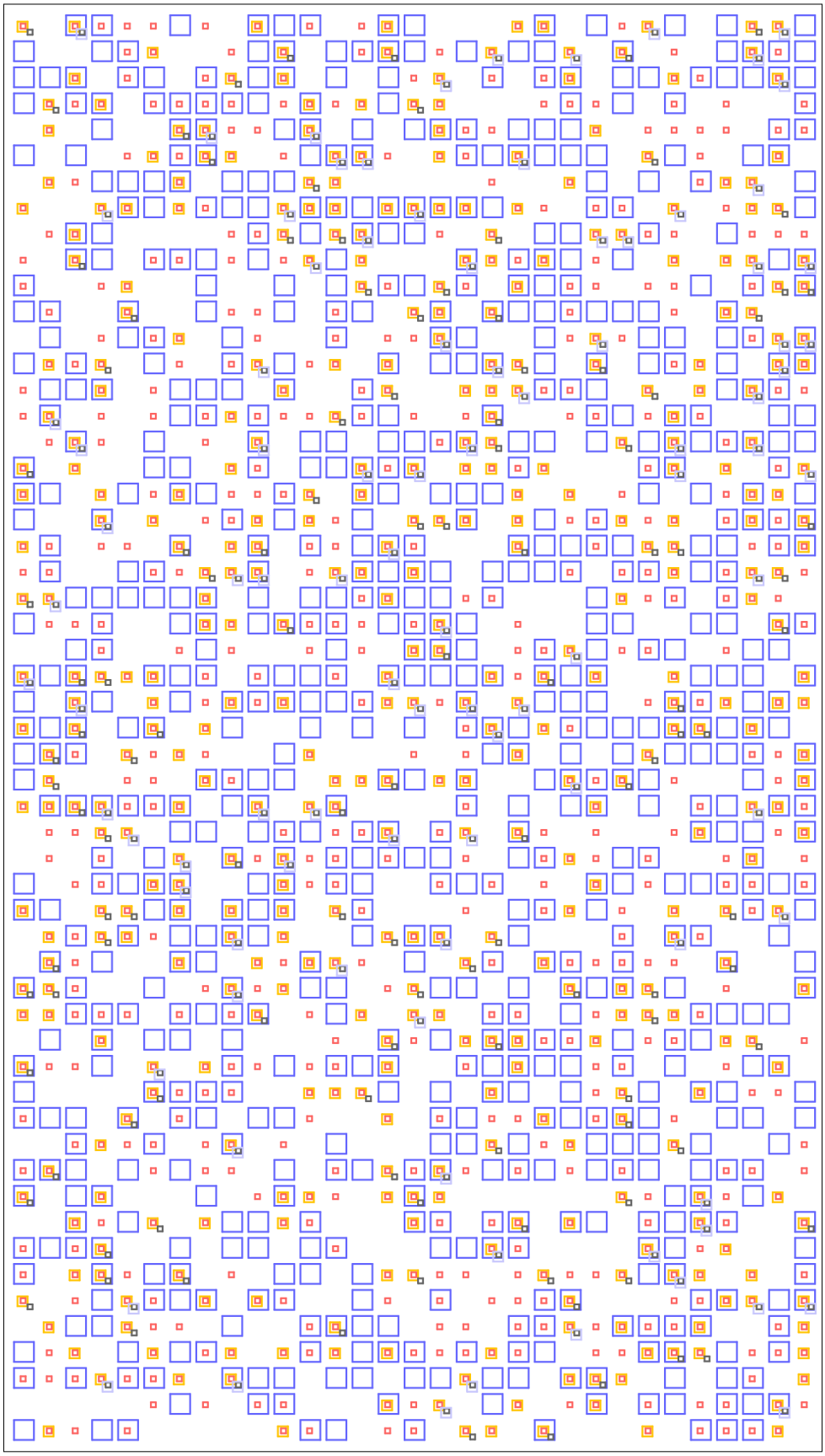


Bild 57: Remixing Molnar: Ein Prozent (Un)Ordnungen

19. HOMMAGE À SCHNEEBERGER: SNE KAO

Das erste Lehrbuch zur Computerkunst trägt den Titel *Computergrafik. Ein Lehr- und Lernbuch* (Limbeck & Schneeberger, 1979). Der Titel gibt wieder, dass sich das Buch als praktische Einführung in die Computergrafik versteht, mit dem Ziel: *Laßt den Computer unter Anwendung von künstlerischen Kriterien Grafiken erzeugen, die dann unter dem Aspekt der Kunst zu betrachten sind* (a.a.O., S. 13).

Das Buch stammt von dem Computerkünstler Reiner Schneeberger, der seinen Ansatz auch *Parameterkunst* bzw. *Computer Minimal Art* nannte. Sein Buch ist Ergebnis einerseits von Lehrveranstaltungen, die Schneeberger als Lehrbeauftragter für Computergrafik ab 1976 durchführte, andererseits der Entwicklung eines Grafiksystems SNE COMP ART. Damit sollte es möglich werden, Computergrafiken ohne Programmierkenntnisse zu generieren (und ist so eigentlich ein Vorläufer von Scratch/Snap!). SNE COMP ART ist daher auch Gegenstand des praktischen Teils des Buches. Das System baut auf FORTRAN[64] auf und soll auch zur Programmierung in FORTRAN selbst hinführen.

Etliche der im Buch vorgestellten Beispiele zeigen Ähnlichkeiten mit den Projekten, die in den vorangegangenen Kapiteln vorgestellt wurden. Sie basieren in SNE COMP ART auf Routinen, die jeweils bestimmte strukturelle Bildaspekte unterstützen. Die Routine SNE KAO etwa dient der Erzeugung waagrecht oder senkrecht schraffierter Rechteckfelder. Bild 58 zeigt ein dafür typisches Beispiel. Das soll im Folgenden nachvollzogen werden.

Für SNE KAO hat Schneeberger Parameter festgelegt (Schneeberger, 2013), von denen einige denen der bekannten m*n-Matrix, die auch hier wieder bestens geeignet ist, zugeordnet werden können:

- **n:** Anzahl der Felder waagrecht
- **m:** Anzahl der Felder senkrecht
- **feld_breite:** Länge eines Feldes
- **feld_höhe:** Länge eines Feldes

Hinzu kommen bei Schneeberger:

- **anzahl_striche:** Anzahl der Striche in einem Feld
- **prozent_waagrecht:** Prozentsatz der Felder mit waagrechten Strichen (die restlichen Felder enthalten senkrechte Striche; leere Felder sind nicht vorgesehen)

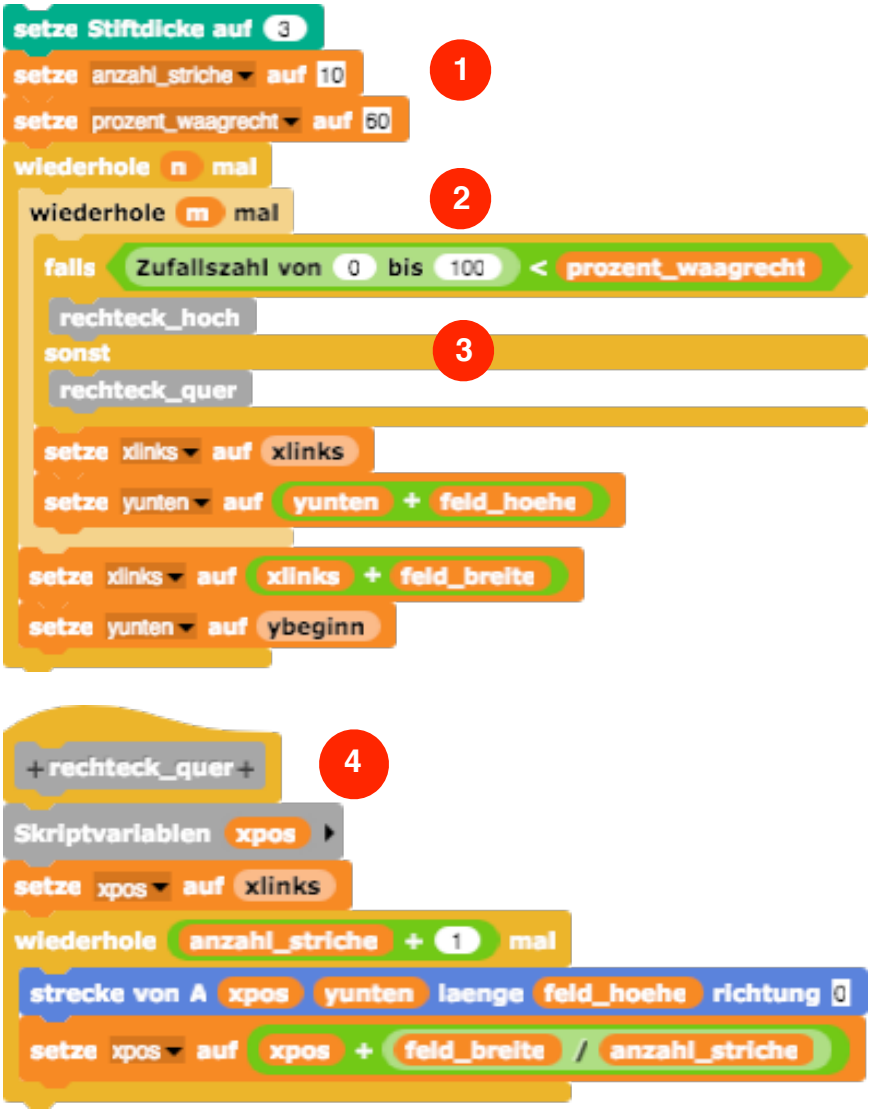
Zusätzlich wird eingeführt:

- **stiftdicke:** Dicke des Zeichenstifts

[64] FORTRAN steht für FORmula TRANslation und ist die erste höhere Programmiersprache (marktreif 1957). Ihr Anwendungsschwerpunkt war und ist bis heute in Wissenschaft, Technik und Forschung.

Damit wird nochmals die Flexibilität des Programms erhöht. Die programmtechnische Umsetzung sollte keine Probleme bereiten. In einem Vorspann können die Festlegungen für die m*n-Matrix erfolgen.

1. Danach sind die hinzu gekommenen Parameter zu bestimmen
2. und in einer Doppelschleife die Felder der Matrix mit den Linienrechtecken zu füllen.
3. Je nach ermittelter Zufallszahl wird dann der entsprechende Block zum Zeichnen eines Feldes mit waagrechten Strichen oder senkrechten Strichen aufgerufen.
4. Im Block **rechteck_quer** werden die Querstriche gezeichnet. Der Block **rechteck_hoch** (hier nicht gezeigt) ist identisch, nur werden dort die y-Werte entsprechend festgelegt.



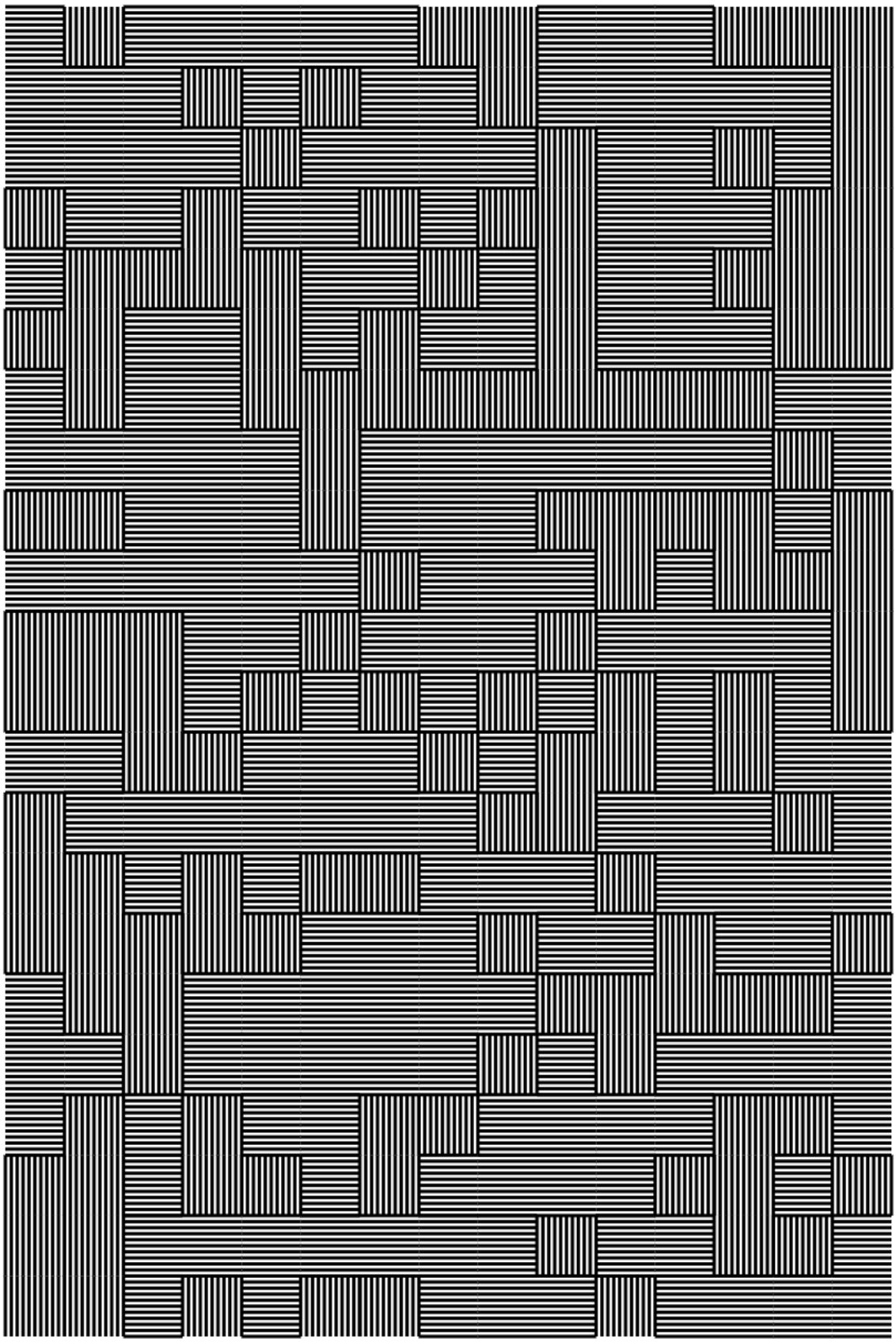


Bild 58: Hommage à Schneeberger: SNE KAO

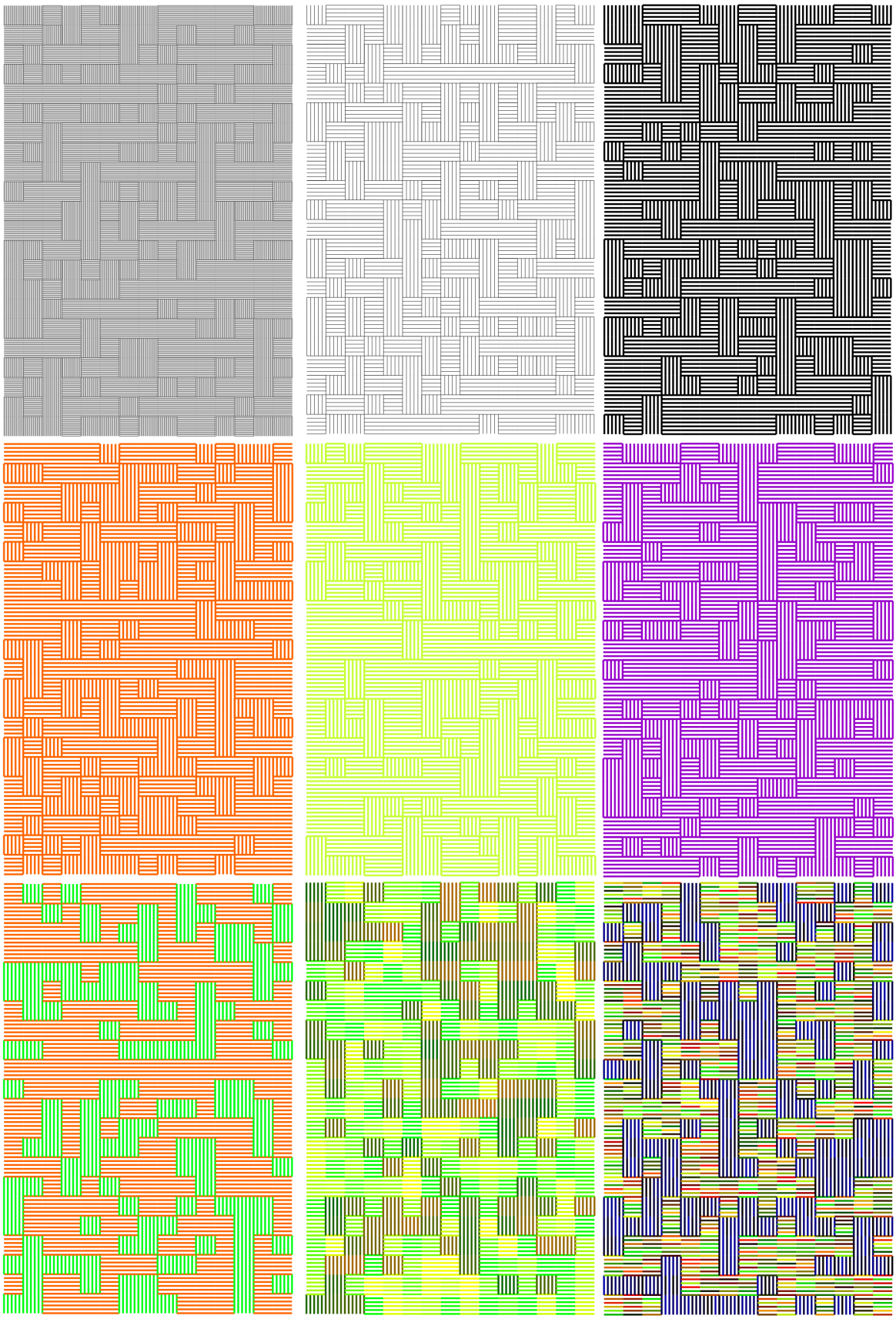


Bild 59: Remixing SNE KAO

Für Schneeberger bietet eine tabellarische Zusammenfassung der Parameter für mehrere Bildvarianten Vorteile (a.a.O.), was mit der folgenden Tabelle geprüft werden kann.

„*Das Gestaltungssystem meiner "Parameterkunst" kann man bei einem Blick auf ein paar nach SNE KAO erstellte Bilder noch nicht erkennen. Erst wenn man die Parametertabelle sieht, wird das Konzept klar. Allerdings bleibt auch dann der (die Verteilung von waagrechten und senkrechten Kästchen steuernde) Zufallsprozess unbestimmt. Zufallsprozesse spielen jedoch in SNE ART eine tragende Rolle.*“

	58	**59-1**	**59-2**	**59-3**	**59-4**	**59-5**	**59-6**	**59-7**	**59-8**	**59-9**
m	22	22	22	22	22	22	22	22	22	22
n	15	15	15	15	15	15	15	15	15	15
feld_breite	54	54	54	54	54	54	54	54	54	54
feld_höhe	54	54	54	54	54	54	54	54	54	54
striche	10	15	5	5	5	5	5	5	5	5
%	60	60	60	60	60	60	60	60	60	60
stiftdicke	3	1	5	5	5	5	5	5	5	5
Farben	S/W	S/W	S/W	S/W	oran-ge	hell-grün	lila	5-6	6-5	bunt

20. HOMMAGE À SÝKORA: LINIEN MALEN

Der tschechische Maler und Bildhauer Zdeněk Sýkora wurde im Kapitel 11: *Muster aus Figuren* mit seiner *Schwarz-Weiß-Struktur* bereits vorgesetllt. Waren es dort geometrische Figuren, die er im Raster anordnete, hat er später einen völlig anderen Stil entwickelt mit dynamischen, kurvigen Elementen. Wie bei seinen Rasterbildern hat er die Bildentwürfe zunächst mit dem Computer generiert. Die endgültige Umsetzung erfolgte dann mit Ölfarben auf Leinwand.

Ein solches Beispiel zeigt Bild 60, das sich an der Vorlage *Linien Nr. 18* orientiert (Groos & Froitzheim, 2017, S. 97). In seiner Bildserie kombiniert Sýkora Kurvenelemente mit geraden Verbindungselementen. Die Bildkomposition wird darüber hinaus durch Farben und unterschiedliche Strichstärken bestimmt. Die Bilder vermitteln einen sehr spontanen Eindruck. Für Sýkora stellen aber auch diese Bilder Ergebnisse seines Versuchs dar, *„strukturelle Elemente in eine präzise Ordnung zu bringen"* (nach Franke, 1984, S. 142).

Wie zeigt sich das beim *Recoding*? Grundlage wird eine leicht modifizierte (gekürzte, vgl. *Anhang C*) Prozedur **kreisbogen**, weil die Schildkröte diesmal nicht zum Ausgangspunkt zurück geführt wird. Sie soll hier jeweils am Ende des Kreisbogens verbleiben, da dieses dann wieder Ausgangspunkt des nächsten Elements (Gerade oder Kreisbogen) wird. Die einzelne Linie wird sich also aus geraden Elementen und Kreisbögen zusammen setzen. Die Kreisbögen können im bzw. im entgegengesetzten Uhrzeigersinn gezeichnet werden. Die resultierende Linie kann dann typischerweise aussehen, wie in der Abbildung rechts:



Für die programmtechnische Umsetzung können wir auf Bestandteile der Block-Bibliothek zugreifen, hier die Linien und die Kreisbögen. Das vollständige Programm ist (wieder einmal) überraschend einfach. Wir benötigen einige wenige Kenngrößen und Zwischenwerte:

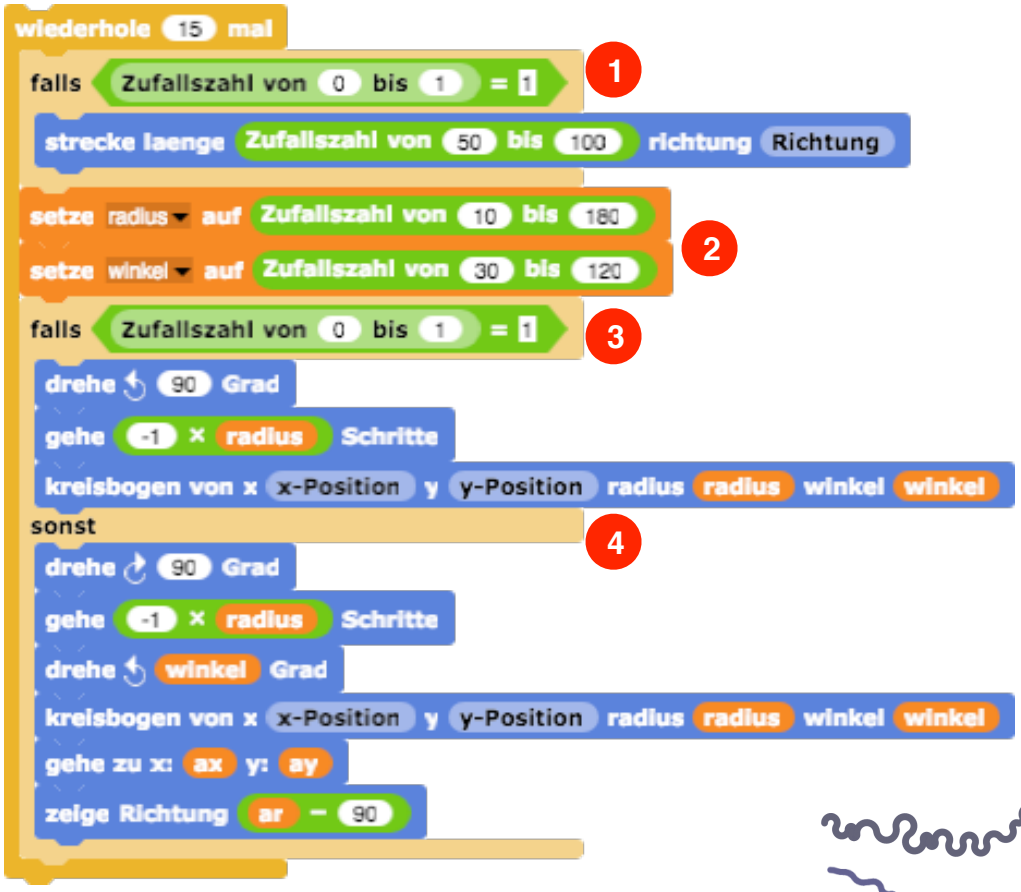
- **radius:** gibt den Radius eines Kreisbogens an
- **winkel:** gibt den Winkel für den Kreisbogen an
- **ax, ay, ar:** geben die Koordinaten und die Richtung für den Ausgangspunkt nach einer Rechtsdrehung an

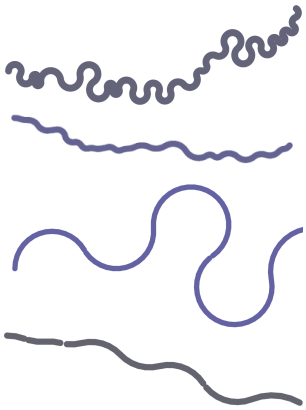
In einer äußeren Schleife (hier nicht gezeigt), die mit der Anzahl der Linien wiederholt wird, können Stiftdicke, Stiftfarbe, Anfangspunkt der Linie und Richtung vom Anfangspunkt aus zufällig festgelegt werden. In einer inneren Schleife, die mit der Anzahl der Linienbestandteile wiederholt wird,

1. ist zu entscheiden, ob eine gerade Strecke gezeichnet werden soll.
2. Danach werden **radius** und **winkel** des zu zeichnenden Kreisbogens zufällig festgelegt.

3 Es wird entschieden, ob der Kreisbogen im Uhrzeigersinn gezeichnet werden soll. In diesem Fall wird die Schildkröte zum Ausgangspunkt bewegt und von dort der Kreisbogen gezeichnet. Die Schildkröte befindet sich danach am Ende des Kreisbogens.

4 Soll der Kreisbogen im entgegengesetzten Uhrzeigersinn gezeichnet werden, wird die Schildkröte zum entsprechenden Ausgangspunkt bewegt und von dort der Kreisbogen gezeichnet. In diesem Fall wird die Schildkröte danach zum Ende des Kreisbogens bewegt.





Anregung: *Die Kombination gerader und kurviger Elemente bestimmt den Grundcharakter der Bildserie. Es bietet sich an, die Kenngrößen systematisch zu variieren. Bei Beschränkung auf die kurvigen Elemente ergeben so z.B. die Kombination kleine Winkel/ kleine Radien die Linie rechts oben, große Winkel/ kleine Radien die zweite Linie von oben, große Winkel/ große Radien die dritte Linie von oben und schließlich kleine Winkel/ große Radien die Linie unten.*

In Bild 61 oben *sind so große Winkel und kleine Radien kombiniert mit einer erhöhten Linienzahl. Des Weiteren kann das Farbspektrum eingegrenzt werden und dann mit variablen Hintergrundfarben hinterlegt werden, wie in* Bild 61 unten.

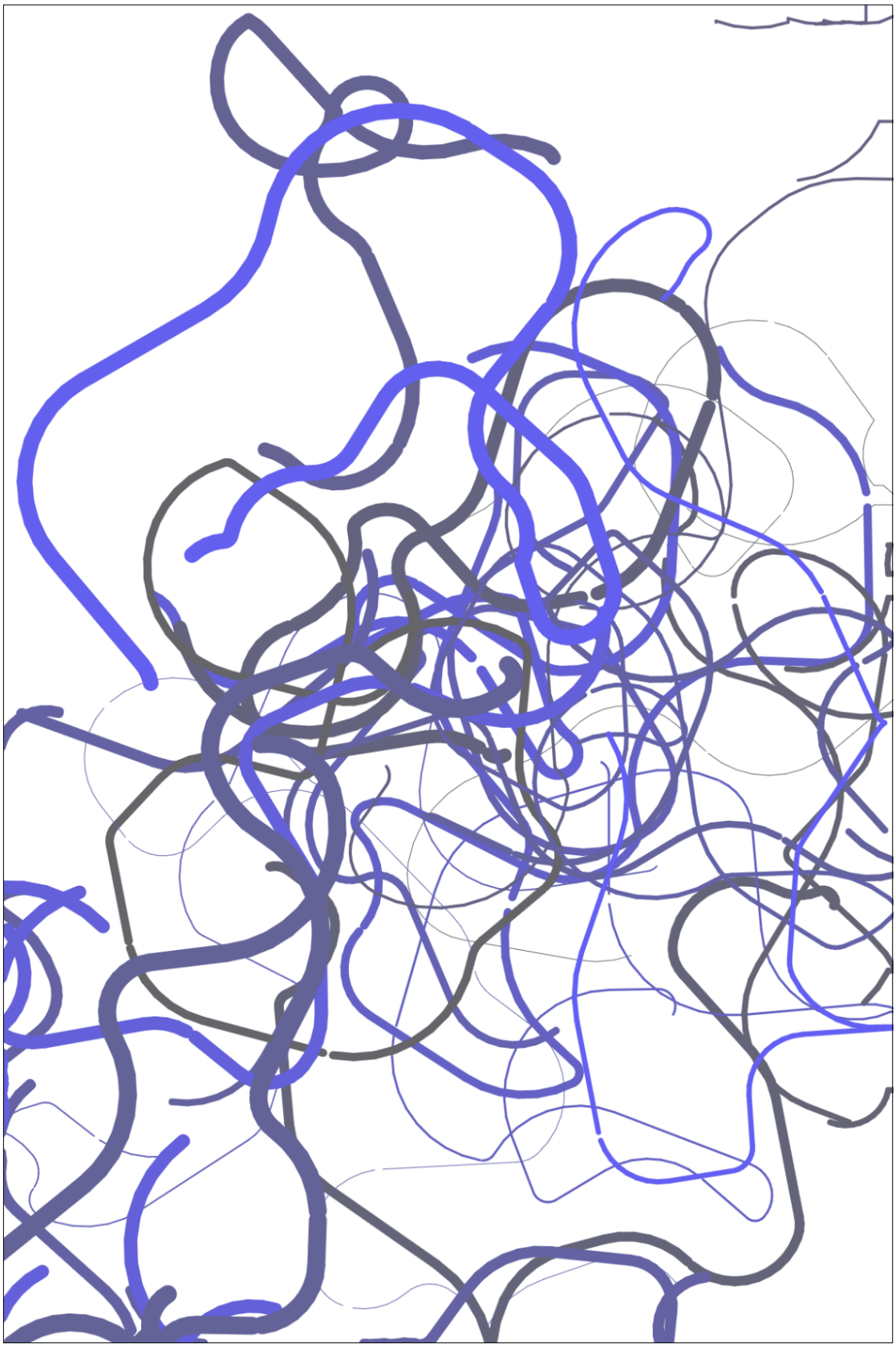


Bild 60: Hommage à Sýkora: Linien Nr. 18

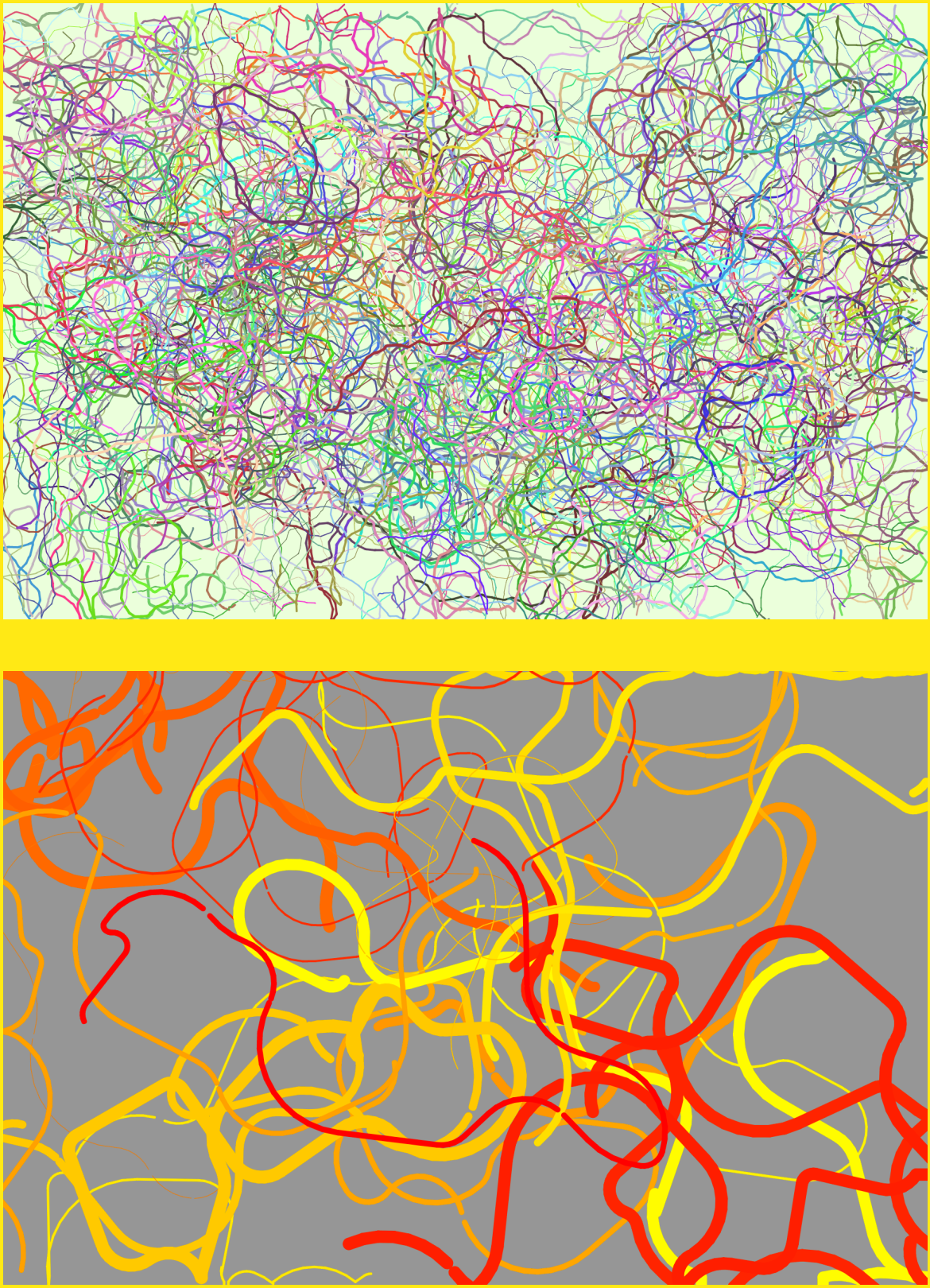


Bild 61: Remixing Sýkora: Linien

21. ALLES IN BEWEGUNG

Filme bestehen aus Einzelbildern, von denen beim professionellen Film typischerweise 24 Bilder pro Sekunde, beim 8mm-Amateurformat 16 Bilder pro Sekunde gezeigt werden, um so den Eindruck realistischer Bewegungen zu erzeugen. *Computeranimationen* sind nun Filme oder entsprechende in Echtzeit ablaufende Bildfolgen, die vollständig am Computer erzeugt und dargestellt werden. In Spielfilmen haben Computeranimationen inzwischen einen festen Platz erobert. Zu den Meilensteinen gehören *Tron* (1982), der erste Film mit längeren rein computergenerierten Sequenzen, und *Toy Story* (1995), der erste vollständig am Computer erstellte Langfilm für das Kino.

Auch in der Computerkunst gehörten von Beginn an Computerfilme zum Repertoire der realisierten Formen. Trotz der anfangs sehr beschränkten technischen Möglichkeiten gelangen damit Umsetzungen in der Tradition des abstrakten Films eines Oskar Fischinger, Viking Eggeling oder Walther Ruttmann.

Zu den Pionieren in der Computerkunst gehörte John Whitney, der ab Ende der 60er-Jahre abstrakte Kurzfilme erstellte, in denen einfache geometrische Objekte sich überlagern und umwandeln. Seine visuellen Effekte hat er in Catalog (1961) zusammen gestellt; als Höhepunkt seiner Arbeit gilt Arabesque (1975). Der Professor für Computergrafik, Kenneth Knowlton, entwickelte eine Programmiersprache BEFLIX für die Produktion von Computerfilmen. Zusammen mit den Künstlern Stan Vanderbeek und Lilian Schwartz entstand damit die Serie Poem Fields (1967) mit zehn computergenerierten Animationen.

Die erwähnte Lilian Schwartz kombinierte in ihren Animationen unterschiedliche digitale Techniken, Collagen und handgemalte Bilder. Mit ihren vielfach ausgezeichneten Arbeiten, die auf ihrer Website unter Digital Animations und Films nach wie vor einsehbar sind, hat sie ganz wesentlich zur Akzeptanz digitaler Kunstformen beigetragen. Für unseren Kontext empfinde ich *Pixillation* (1970) und *Mutations* (1972) als besonders sehenswert und anregend.

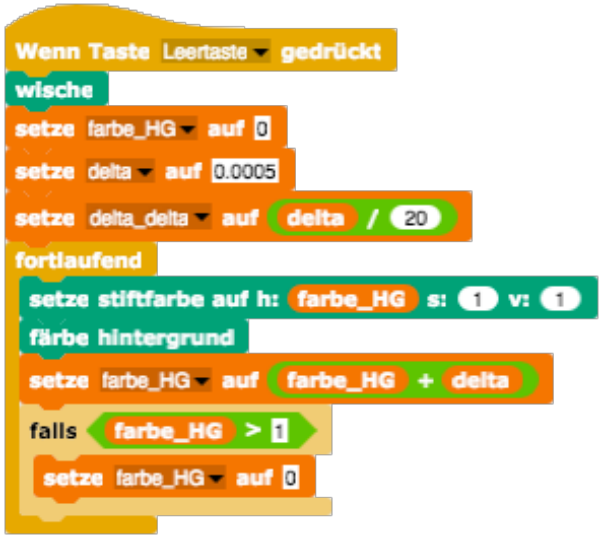
Zu nennen ist auch Charles Csuri, der als Professor of *Art Education*, wie auch als Professor of *Computer Information Science*, die Computerkunst und die Computeranimation von künstlerischer wie auch technischer Seite mit geprägt hat. Seine Sinuskurven-Portraits zählen zu den unverwechselbaren Werken der frühen Computerkunst. In seiner preisgekrönten Animation Hummingbird (1968) vollzieht er grafische Manipulationen mit der zeichnerischen Darstellung eines Kolibris. Bemerkenswert ist, dass für Csuri der Computerfilm kein fertiger, vorführbarer Film sein soll, sondern einer, der in *Interaktion* mit dem Benutzer entsteht: *„Auch der Zuschauer kann sich in gewissem Maße als Künstler erleben, wenn er sich ans „Lenkrad" des Computers setzt. Selbst auf der Grundlage eines vorgegebenen Kontextes sind die Möglichkeiten nämlich fast unendlich groß."* (Csuri, 1978, S. 62 f.).

Zum *Remixing* soll deshalb auch gehören, die Computerkunst *interaktiv* in *Bewegung* zu setzen. In den folgenden Abschnitten werden dafür verschiedene Techniken gezeigt.

21.1 Farbinstallation: Malen ... Wischen ... Malen

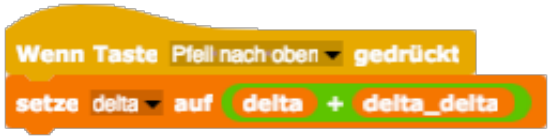
Mit einem einfachen Beispiel sollen die ersten Komponenten einer interaktiven Animation eingeführt werden. Für ein kleines *Installationsprojekt* hatte ich als bewussten Kontrapunkt zu schnellen Bildwechseln und Bewegungen grafischer Objekte, wie sie in vielen heutigen Multimedia-Installationen vorherrschen, eine extrem langsame Veränderung der gleichmäßigen Einfärbung des Monitors vorgenommen. Ein vollständiger Farbdurchlauf dauerte etwa eine Minute. Die Umsetzung ist äußerst simpel; Startpunkt ist die Einfärbung der Bühne (bzw. des Hintergrunds).

Zu Beginn werden die Startwerte für **farbe_HG**, d.h. die Farbe der Bühne, und für **delta**, d.h. die Veränderung des Farbwerts bei jedem Wiederholungsschritt, festgelegt. Mit **delta_delta** kann **delta** interaktiv angepasst werden. Bei einer Zunahme von **delta** kann der Vorgang gewissermaßen im Zeitraffer ablaufen bei einer Abnahme wird er verlangsamt.



In einer Endlosschleife wird **farbe_HG** kontinuierlich um den konstanten Wert **delta** erhöht und damit der Hintergrund eingefärbt (es wird der Farbton h aus dem HSV-Modell verwendet; vgl. Kapitel 7: *Alles schön bunt hier*). Da der Farbwert sich zwischen 0 und 1 bewegen kann, wird er bei Überschreiten von 1 wieder auf 0 gesetzt[65].

Für meine Installation habe ich daraus ein eigenständig ausführbares Programm mit *Snapp!* (siehe *Anhang A*) compiliert und im Vollbildmodus vorgeführt. Weiterhin habe ich eine interaktive Komponente eingeführt: Mit der Taste **Pfeil nach oben** kann der Wert von **delta** erhöht, mit der Taste **Pfeil nach unten** entsprechend wieder erniedrigt werden.





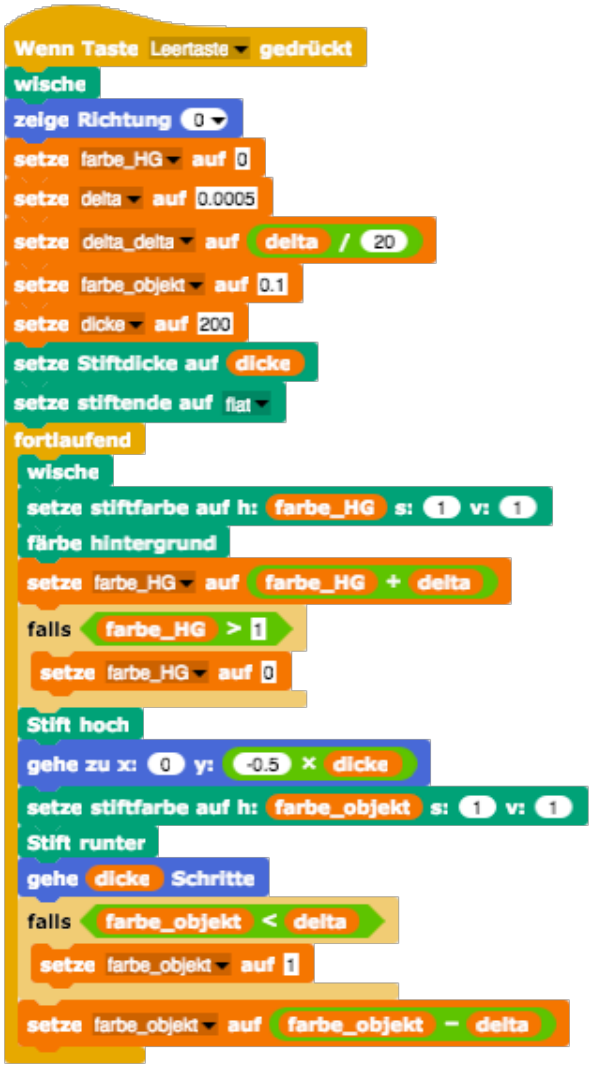
Hinweis: *Bei großen* **delta***-Werten (etwa ab* **delta** *> 0.03) kann es zu unangenehmen Flacker-Effekten kommen. Bei sehr kleinen Werten (etwa ab* **delta** *< 0.00009) sind die Änderungen so verlangsamt, dass sie von Wiederholungsschritt zu Wiederholungsschritt fast nicht mehr erkennbar sind.*

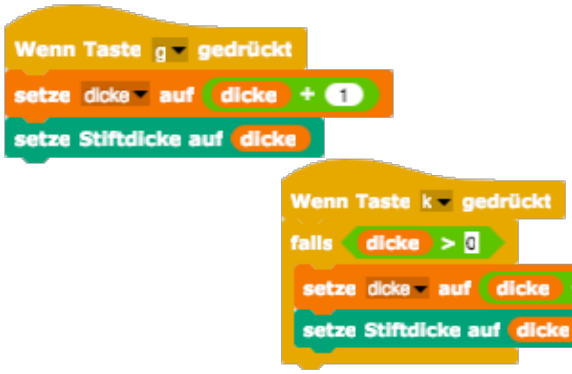
Eine kleine Erweiterung führt dazu, dass neben dem Farbverlauf des Bühnenhintergrunds ein zweites Objekt (hier ein Quadrat) einen ebensolchen, aber gegenläufigen

[65] Dieses Zurücksetzen ist nicht zwingend notwendig, da sich die Farbfolge ab dem Wert 1 wiederholt.

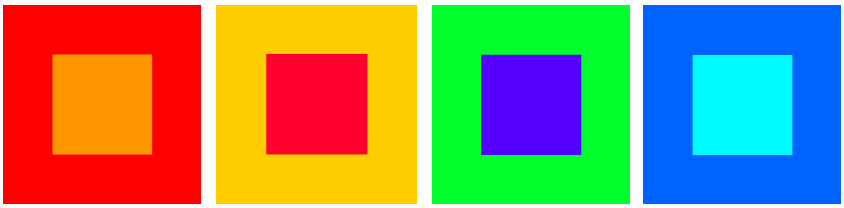
Farbverlauf vollzieht. Es werden zusätzliche Startwerte benötigt für **farbe_objekt**, d.h. die Farbe des Quadrats, und für **dicke**, d.h. die Dicke des Strichs, mit dem das Quadrat gezogen wird. Der Farbverlauf des Quadrats wird analog wie beim Hintergrund gesteuert. Weil die Quadrate auch mit unterschiedlicher **dicke** gezeichnet werden können sollen, muss zusätzlich bei jedem Wiederholungsschritt mit **wische** die Bühne geleert werden, da es sonst zu Farbüberlagerungen kommen kann.

Als weitere interaktive Komponente kommt hier hinzu: Mit der Taste **g** kann der Wert von **dicke** vergrößert werden (Vergößerung des Quadrats), mit der Taste **k** entsprechend wieder verkleinert werden (Verkleinerung des Quadrats).





Je nach Kombination der Farben von Hintergrund und Objekt kann es zum Simultankontrast kommen, einer optischen Täuschung, bei der Übergangsfarben an den Grenzen zwischen Objekt und Hintergrund wahrgenommen werden. Die Abbildung zeigt die (hier nur statisch darstellbare) Abfolge der gegenläufigen Änderung von Farbhintergrund und Quadrat.



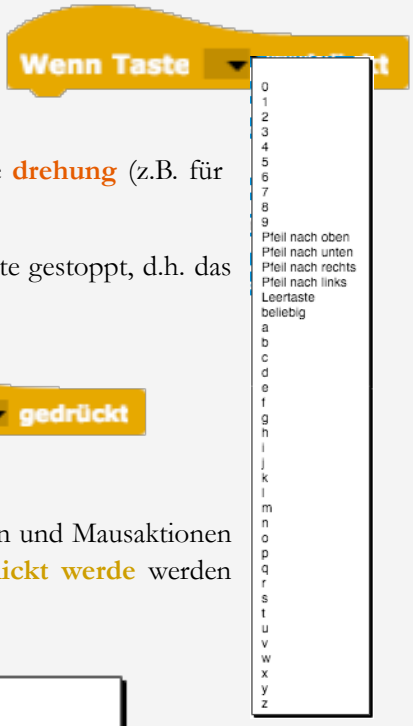
Hinweis: *Weitere Farbnunacen ergeben sich, wenn statt dem Farbton die Sättigung und die Helligkeit - wieder jeweils getrennt für Bühne und Objekt - variiert werden.*

Hinweis: *Auch hier gilt, dass es bei großen* **delta***-Werten zu Flacker-Effekten kommen kann und bei sehr kleinen Werten die Änderungen fast nicht mehr erkennbar sind.*

Das brauchen wir von **Snap!***:*

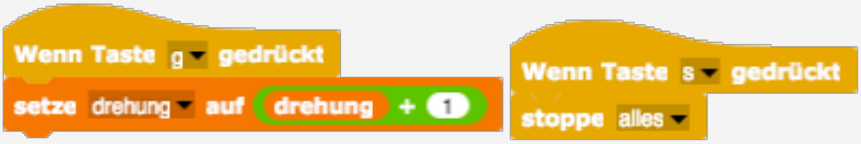


Mit dem Hut-Block **Wenn Taste gedrückt** wird abgefragt, ob und welche Taste gedrückt wurde. Dabei stehen alle Zahlen und Buchstaben, die Leertaste sowie die Richtungspfeile zur Auswahl. Wenn eine dieser Tasten erkannt wird, werden alle an diesen Hut-Block folgenden Befehle ausgeführt.

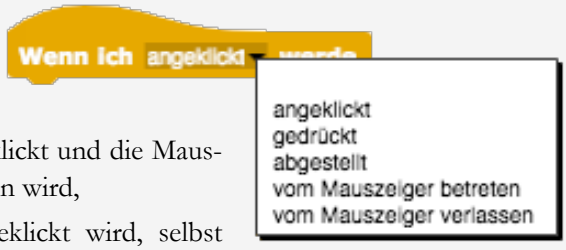


Im Beispiel links wird bei Drücken der **g**- Taste die Variable **drehung** (z.B. für einen Drehwinkel) um 1 erhöht.

Im Beispiel rechts werden bei Drücken der **s**- Taste alle Skripte gestoppt, d.h. das Programm beendet.



Ganz analog zu Tastendrücken können auch Mausbewegungen und Mausaktionen ausgewertet werden. Mit dem Hut-Block **Wenn ich angeklickt werde** werden diese abgefragt:

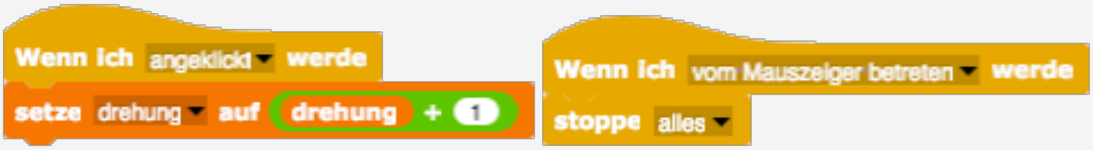


- wenn ein Sprite angeklickt und die Maustaste wieder losgelassen wird,
- wenn ein Sprite angeklickt wird, selbst wenn die Maustaste noch nicht wieder losgelassen wurde,
- wenn ein Sprite angeklickt, mit gedrückter Maustaste bewegt und anschließend die Maustaste wieder losgelassen wird,
- wenn der Mauszeiger einen Sprite-Bereich betritt,
- wenn der Mauszeiger einen Sprite-Bereich verlässt.

Mit der jeweiligen Maus-Aktion werden alle an diesen Hut-Block folgenden Befehle ausgeführt.

Im Beispiel links wird bei Anklicken eines Sprites die Variable **drehung** (z.B. für einen Drehwinkel) um 1 erhöht.

Im Beispiel rechts werden bei Bewegen der Maus in einen Sprite-Bereich (oder z.B. die Bühne) alle Skripte gestoppt, d.h. das Programm beendet.



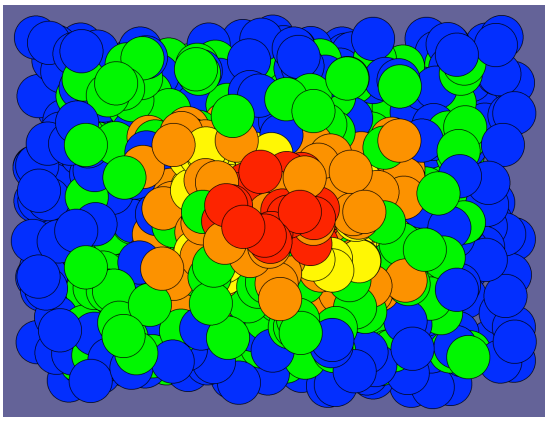
21.2 Exkurs: Das Arbeiten mit Objekten (I) - Sprites

In den bisher vorgestellten Projekten wurden die Grafiken dadurch erzeugt, dass sich immer genau *eine* Schildkröte über die Bühne bewegte und sie dabei eine Spur hinterließ, aus der am Ende die Grafik entstanden ist. Allgemein gesprochen haben wir dabei mit einem *Objekt* (der Schildkröte) gearbeitet, das mit bestimmten *Eigenschaften* ausgestattet war (Farbe, Größe, Position usw.). Wir haben *Methoden* (Skripte, d.h. Befehlsfolgen, Prozeduren) entwickelt, mit denen das *Verhalten des Objekts* gesteuert wurde. Das sind bereits Merkmale der objektorientierten Programmierung[66], einem Programmierparadigma, das sich in den meisten modernen Programmiersprachen findet[67].

Es gibt in Snap! zwei Möglichkeit, *mehrere* Objekte zu definieren und *gleichzeitig* auf der Bühne zu bewegen: *Sprites* und *Klone*.

Das Arbeiten mit *Sprites* bietet mehrere Vorteile:

- Neue Sprites können leicht hinzugefügt werden. Jedes Sprite hat seine eigenen Eigenschaften und Skripte (wird ein Sprite im **Objektbereich** angewählt, werden diese zugehörigen Skripte im **Programmbereich** angezeigt).
- Vorhandene Sprites können verdoppelt oder vervielfacht werden.
- Sprites können Skripte von anderen Sprites übernehmen (*„erben“*). Übernommene Skripte können bei Bedarf verändert und neue Skripte können hinzu gefügt werden.
- Die Skripte aller Sprites werden *zeitlich parallel* und *unabhängig voneinander* abgearbeitet.
- Sprites können miteinander *kommunizieren*, d.h. auf Nachrichten anderer Sprites *reagieren*.



Ein einfaches Beispiel soll einiges davon verdeutlichen (es orientiert sich an einer Vorlage *Ghost Disks* von Andrew Lassner, 2010, 362 ff.). Dabei sollen Scheiben unterschiedlicher Farbe (die als Kostüme verfügbar gemacht werden können) zufällig, aber mittig übereinander - bei abnehmender Breite des abgedeckten Bereichs - gezeichnet werden.

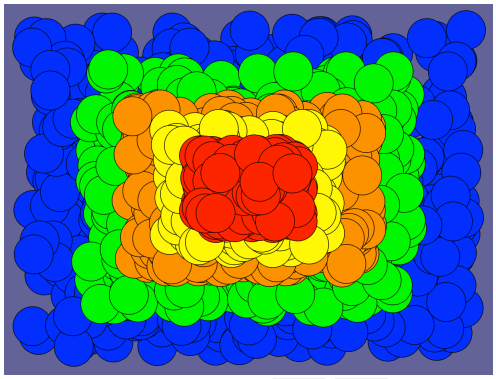
In einem ersten Anstatz soll versucht werden, das Ergebnis mit einer einzelnen Schildkröte (Sprite) zu erhalten. Es liegt nahe, zunächst die Scheiben einer Farbe zu zeichnen und danach der Reihe nach die Scheiben in den anderen Farben darüber zu zeichnen.

---

[66] Die objektorientierte Programmierung zeichnet sich durch eine Reihe von Konzepten aus, die sie von anderen Programmierparadigmen unterscheidet. Da die meisten für das *Recoding & Remixing* der Computerkunst nicht elementar sind, würde ihre Behandlung hier den Rahmen sprengen.

[67] Snap! fehlten zur Zeit der Abfassung dieses Buches einige der konstitutiven Konzepte, weshalb Jens Mönig - Chefentwickler von Snap! - lieber von objekt*basierter* Programmierung sprach. Echte Objektorientierung gibt es in Snap! ab Version 4.1.

1 Bei Beschränkung auf eine einzelne Schildkröte wird für jede Farbe (mit Hilfe von fünf Kostümen) eine Schleife abgearbeitet, in der (mit abnehmender Zahl) die Scheiben zufällig innerhalb des jeweiligen Bühnenbereichs gezeichnet werden. Duch diese Abfolge liegen die Farbstapel quasi übereinander (siehe Abbildung rechts). Um das Vorbild zu erreichen, muss aber eine bessere „Durchmischung“ erreicht werden.





1



2 Dafür werden in der folgenden Variante mehrere Sprites verwendet. Die zusätzlichen Sprites entstehen als Duplikate der ersten (hier der blauen) Scheibe. Das Kostüm wird jeweils durch eine anderfarbige Scheibe ersetzt. Jedes Sprite hat nur dieses eine Kostüm.



2

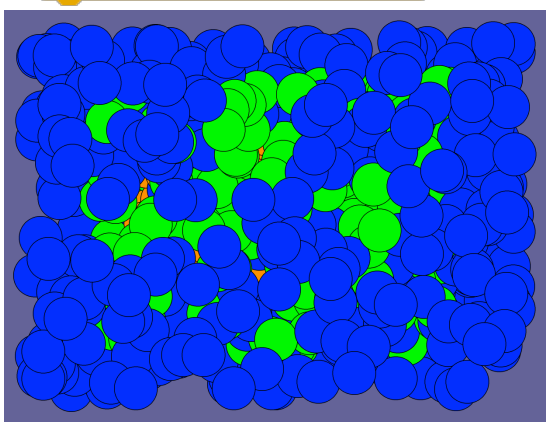
3 Das Verhalten des ersten (blauen) Sprites wird mit dem gezeigten Skript beschrieben. Dabei wird auch die Hintergrundfarbe und mit **n** die Zahl der farbigen Scheiben festgelegt. Dieses Skript kann per Drag & Drop auf die anderen Sprites übertragen werden.





4 Es ist dort für das jeweilige Sprite anzupassen. Hintergrundfarbe und Scheibenzahl brauchen hier nicht wiederholt werden und sind zu löschen.

Dagegen sind die Zufallszahlen für die x- und y-Koordinaten für den jeweiligen Bereich anzupassen.



Wird nun das Projekt mit gestartet, werden die Skripte *aller* Sprites *parallel* ausgeführt. Das hat zur Folge, dass dieses Mal nicht die Farbstapel übereinanderliegen, sondern dass einzelne farblich verschiedene Scheiben übereinander gezeichnet werden. Das Ergebnis in der nächsten Abbildung unterscheidet sich deshalb vom oben gezeigten Ergebnis des sequentiellen Zeichnens mit einem einzelnen Sprite.

Auch dieses Ergebnis ist noch nicht sehr befriedigend, da nur die blauen und teilweise noch die grünen Scheiben sichtbar sind. Das liegt daran, dass das Zeichnen aller Scheiben gleichzeitig beginnt und die blauen und grünen Scheiben deshalb schlicht aufgrund ihrer größeren Anzahl die Scheiben in den restlichen Farben fast völlig überdecken.

Wie erhalten wir daraus ein Ergebnis, bei der wie gewünscht die Scheiben aller Farben zu sehen sind?



Die Lösung ist, immer nur dieselbe Anzahl unterschiedlich gefärbter Scheiben gleichzeitig zu zeichnen:

5 Das Sprite **blau** zeichnet zunächst nur eine bestimmte Anzahl blauer Scheiben (hier z.B. **n/4**). Erst danach wird das Zeichnen der grünen Scheiben gestartet. Dazu wird die Nachricht **sende Liste** *blau fertig / gruen* an das Sprite **gruen** gesendet. Zeitgleich beginnt das Zeichnen des Rests der blauen Scheiben.

6 Erhält das Sprite **gruen** diese Nachricht über **Wenn ich** *blau fertig* **empfange**, wird hier ebenfalls zunächst ein bestimmter Teil (der um **n/5** reduzierten Anzahl) grüner Scheiben gezeichnet und danach eine entsprechende Nachricht **sende Liste** *gruen fertig / orange* **an** das Sprite **orange** gesendet. Danach beginnt das Zeichnen der orangenen Scheiben und des Rests der grünen Scheiben.

Dieser Ablauf wird so lange fortgeführt, bis das letzte Sprite (hier das Sprite **rot**) mit dem Zeichnen begonnen hat.

Das Ergebnis ist nun ein Bild, das der eingangs gezeigten Vorlage entspricht.

Hinweis: *Natürlich wäre dieses Ergebnis auch mit einem einzelnen Sprite erreichbar gewesen. Das entsprechende Skript wäre aber durch die Abfolge der getrennt bzw. der gleichzeitig zu zeichnenden Scheiben unterschiedlicher Farbe deutlich länger und unübersichtlicher geworden!*



Das brauchen wir von **Snap!**:



Neue **Sprites** können mit zwei unterhalb der Bühne befindlichen Schaltern erzeugt werden:



Neues Sprite hinzufügen: Damit wird ein neues Sprite erzeugt und mit zufälliger Farbe, zufälliger Position und zufälliger Richtung auf der Bühne platziert.



Neues Sprite zeichnen: Es wird der Paint-Editor geöffnet und es kann eine Figur für das neue Sprite gezeichnet werden. Nach Schließen des Editors wird das neue Sprite mittig auf der Bühne platziert.



Mit dem Befehl **sende** *nachricht* **an alle** wird die *nachricht* an alle Sprites und die Bühne verschickt.



Mit **Liste nachricht sprite** kann eine *nachricht* gezielt an ein einzelnes zu benennendes *sprite* verschickt werden (obwohl der Befehl **sende ... an alle** lautet).



Das Sprite erwartet die *nachricht* und führt nach deren Eintreffen die nachfolgenden Befehle aus.

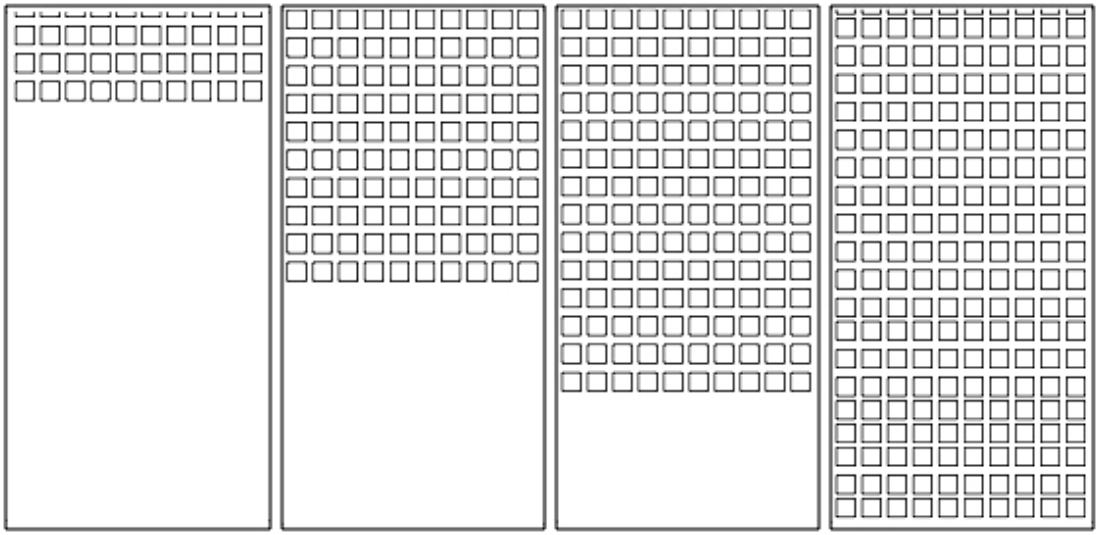
Anregung: *Interessante Varianten ergeben sich, wenn die Größe der Scheiben zufällig in einem vorgegebenen Bereichs variirt wird (z.B. über* **setze Größe auf Zufallszahl von 20 bis 75 %**).

Hinweis: *Für die parallele Abarbeitung von Skripten unterschiedlicher Sprites ist es wichtig, die Aktivierung der Skripte mit einem Hut-Block auszulösen, also z.B.* **Wenn angeklickt**, **Wenn**

Taste x gedrückt *oder* **Wenn ich** *nachricht* **empfange**, *da sonst immer erst das entsprechende Sprite ausgewählt und dessen Skript angewählt werden müsste.*

21.3 Remixing Schotter (II) - Klone

Bei der Nachprogrammierung von *Schotter* (vgl. Kapitel 12: *Hommage à Georg Nees: Schotter*) hatte ich bereits die ursprüngliche Bildrichtung auf den Kopf gestellt, weil ich den Eindruck von Schotter, der vom Laster geschüttet wird, erzeugen wollte. Es liegt also nahe, diesen Eindruck noch zu verstärken, indem der Schotter animiert wird. Dafür werden sehr viele „Steine" benötigt, die sich dann nebeneinander von oben nach unten bewegen. Für die Steine, die am unteren Rand die Bühne verlassen, müssen am oberen Rand immer neue Steine hinzu kommen.



In der obigen Abbildung (die naturgemäß nur eine statische Abfolge darstellen kann) zeigen die Steine zunächst weder seitliche Auslenkungen noch Drehungen. Programmtechnisch müssen jedenfalls immer neue Steine erzeugt und oben auf die Bühne gebracht sowie unten wieder herausgenommen werden. Für diese dynamischen Änderungen ist die händische Vervielfachung von Sprites denkbar ungeeignet. Snap! bietet dafür das *Klonen* von Sprites an.

Beim Klonen mit dem Befehl **klone** *mich* werden eine oder beliebig viele Kopien eines Sprite erzeugt. Klone übernehmen zunächst lediglich das Kostüm des jeweiligen Sprites und sind nicht sichtbar (d.h. sie werden erst nach dem Befehl **anzeigen** sichtbar). Ihr Verhalten muss danach mit einem eigenen Skript - immer eingeleitet mit dem Hut-Block **Wenn ich geklont werde** - beschrieben werden.

Für das Projekt ergibt sich daraus folgendes Vorgehen:

1. Zuerst werden die Kenngrößen für den *animiertenSchotter* festgelegt.

   - **groesse:** Größenangabe für das Sprite (das als Vorlage für die Klone dient)
   - **abwaerts:** Schrittweite (in Pixeln) für die Abwärtsbewegung des Schotters

- **seitlich:** seitliche Auslenkung der Bestandteile des Schotters.
- **drehung:** Drehwinkel der Schotter-Bestandteile
- **warten:** zeitlicher Abstand beim Erzeugen der aufeinanderfolgenden Klone



1

2 Dann wird die Vorlage für das Klonen im Block **zeichne Vorlage** erzeugt und diese fortlaufend zeilenweise im Block **klone Steine groesse** vervielfältigt. Zwischen dem zeilenweisen Erscheinen gibt es eine zeitliche Verzögerung der Länge **warten**.



2

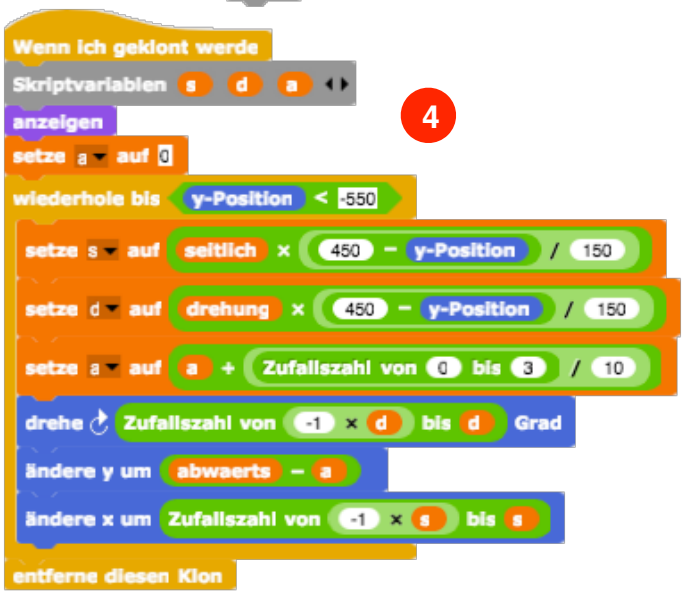
Im Block **zeichne Vorlage** wird mit vorgegebener Farbe ein **n-eck** und daraus mit **ziehe Kostüm stiftspuren an** das Kostüm des aktuellen Sprites erzeugt. Die Vorlage wird mit **wische** gelöscht und **Stift hoch** gesetzt, damit die Klone keine Spur hinterlassen.

Im Beispiel wird ein Quadrat als Vorlage gezeichnet. Durch Austausch des Befehls **n-eck necken seite** kann aber bei Bedarf leicht eine andere Vorlage erzeugt werden. Alternativ können natürlich auch vorgefertigte Kostüme verwendet werden.



3

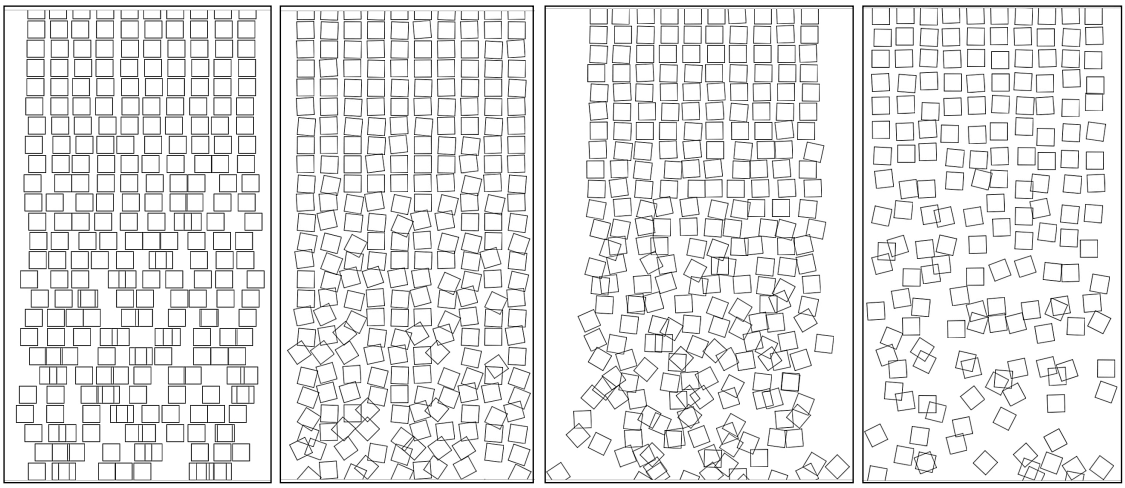
3 Im Block **klone Steine** wird die gewünschte Zahl der Klone erstellt, und zwar als eine Reihe mit leichtem Abstand waagrecht nebeneinander liegender Steine. Der Ausgangspunkt liegt etwas außerhalb der Bühne, damit der Eindruck des Hereinfallens entsteht.



4

4 Sobald Klone erzeugt sind, werden alle zugehörigen Skripte ausgeführt, die mit dem Hut-Block **Wenn ich geklont werde** beginnen. Im Beispiel werden alle Klone solange **abwaerts** bewegt, bis sie die Bühne verlassen. Dann werden sie

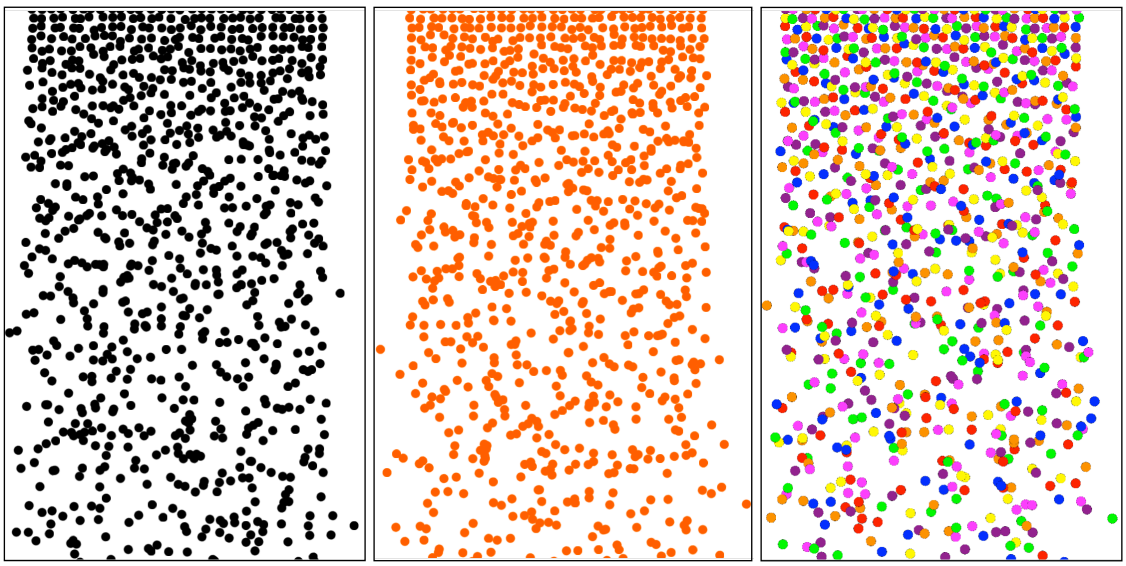
mit **entferne diesen Klon** gelöscht (sonst würden immer mehr Klone entstehen, wodurch sich das Programm zunehmend verlangsamen würde). Mit den Skriptvariablen **s**, **d** und **a** wird das Programm sehr flexibel, weil damit seitliche Auslenkung, Drehwinkel und Abwärtsbewegung jeweils mit zunehmender „Fallhöhe" vergrößert werden können.



In der obigen Abbildung nimmt jeweils von oben nach unten die seitliche Auslenkung zu (ganz links), die Drehung (zweite von links), Auslenkung plus Drehung (zweite von rechts) und schließlich Auslenkung plus Drehung sowie Abwärtsbewegung.

Hinweis: *Wird das Programm mit dem Schalter*  *in der Werkzeugleiste beendet, werden alle Klone gelöscht und verschwinden von der Bühne!*

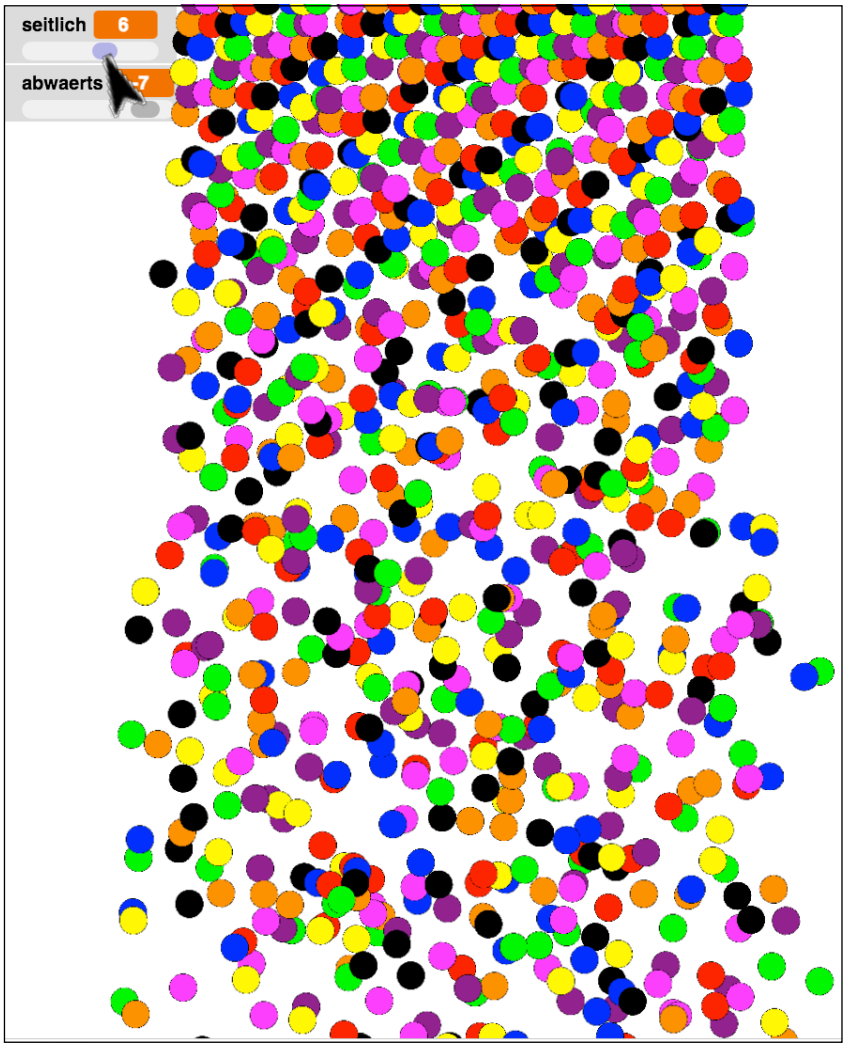
Der Eindruck herabfallender Steine kann nochmals verstärkt werden, wenn schwarz ausgefüllte (oder andersfarbige) Sprites verwendet werden. Dafür bietet es sich an, entsprechende Kostüme vorzusehen (im folgenden Beispiel sind es farbige Scheiben), die dann im Block **klone Steine** in der **wiederhole**-Schleife mit dem Befehl **nächstes Kostüm** den einzelnen Klonen zugewiesen werden können.



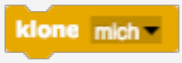
Was diesen Animationen nun noch fehlt ist *Interaktivität*. Das bedeutet, die Betrachter sollen auch hier die Möglichkeit erhalten, den Ablauf der Animation während des Ablaufs zu beeinflussen. Dafür bieten sich die Kenngrößen für seitliche Auslenkung, Drehwinkel (bei runden Kostümen allerdings überflüssig) und Abwärtsbewegung an.

Werden diese Kenngrößen in der Blockpalette **Variablen** angeklickt, werden sie auf der Bühne angezeigt. Diese Anzeige kann mit einem Regler versehen werden, mit dem die Betrachter dann während des Programmablaufs die Werte verändern können. Das Programm greift immer auf die aktualisierten Wert zu, reagiert also sofort auf die Eingaben der Betrachter.

Damit die Betrachter die Regler mit einer Maus oder mit dem Finger auf einem Touchscreen oder einem Tablet sicher treffen und bewegen können, empfiehlt es sich, die Variablenanzeige zu vergrößern. Das ist mit der Option **Blöcke vergrößern ...** möglich, die in der Werkzeugleiste unter dem Werkzeugsymbol zu finden ist. Die Vergrößerung der Blöcke wirkt sich dann nämlich auch auf die Regler auf der Bühne aus.



Das brauchen wir von **Snap!***:*





Mit dem Befehl **klone** *mich/Objekt* wird das aktuelle Sprite (*mich*) oder das angegebene Sprite (*Objekt*) vervielfacht.



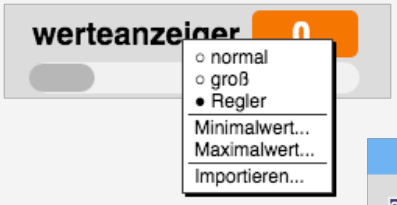
Mit dem Hut-Block **Wenn ich geklont werde** wird ein Skript eingeleitet, dass nach dem Erzeugen eines Klons ausgeführt wird. Es können auch mehrere solche Skripte parallel das Verhalten eines Klons bestimmen.

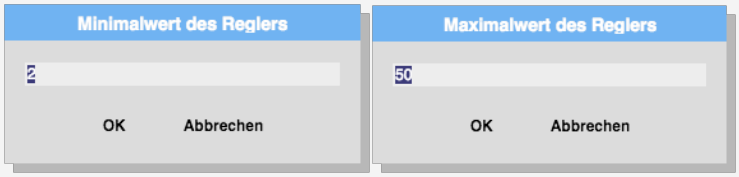


Mit dem Befehl **entferne diesen Klon** wird ein Klon gelöscht, ist also von keinem Skript mehr ansprechbar und wird von der Bühne entfernt.

Variablenwerte interaktiv regeln:

Alle auf der Bühne **angezeigte Variablen** können mit einem **Regler** zum Einstellen der Werte versehen werden. Ein Rechtsklick auf den Anzeiger öffnet ein Kontextmenü, in dem die Darstellungsoptionen angezeigt werden. Nachdem **Regler** gewählt wurde, sind in zwei weiteren Schritten der gewünschte **Minimalwert...** bzw. **Maximalwert...** festzulegen, zwischen denen der Schieberegler bewegt werden können soll.

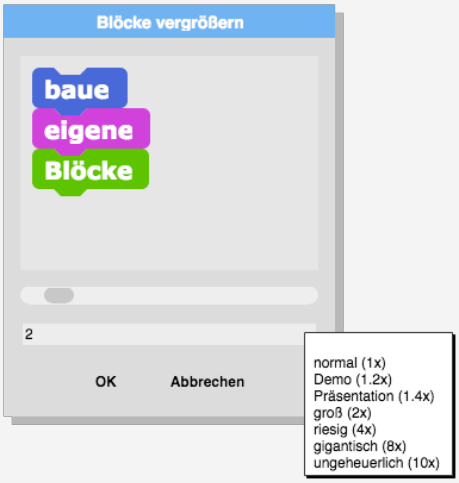




Um die Darstellung der Blöcke zu vergrößern gibt es in der **Werkzeugleiste** unter dem **Werkzeugsymbol** die Option **Blöcke vergrößern ...** Die Vergrößerung wirkt sich auch auf die Regler auf der Bühne aus.

Die Blockgröße kann entweder mit den Werten aus dem Klappmenü belegt werden oder eigenständig zwischen 1 und 12 variiert werden.

Es ist zu beachten, dass die gewählte Blockgröße auch beim Laden eines anderen Programms beibehalten wird.

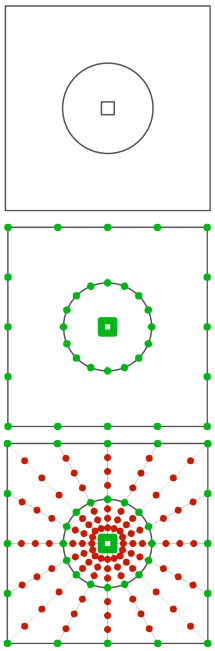


21.4 Hommage à CTG: Return to Square

Hinter der Computer Technique Group (CTG) stand eine japanische Gruppe von Kunst- und Technikstudenten, die ein Design-Studio eröffneten. Ihre theoretischen Arbeiten erschienen in Zeitschriften, ihre Werke wurden in Galerien und Ausstellungen weltweit gezeigt.

Ihre Vorbilder für die folgenden Animationen sind eigentlich statische Bildfolgen. In mehreren Bildern - die zu den Kassikern der frühen Computerkunst gehören - haben sie Möglichkeiten aufgezeigt, wie Bilder durch Transformationen ineinander überführt werden können. Berühmt ist zum einen *Return to Square* (siehe Franke, 1971, S. 35)), bei dem ein größeres Quadrat zunächst durch Verkleinerung und Verzerrung in ein menschliches Profil verwandelt wird und dieses anschließend wieder in ein kleineres Quadrat. Nach dem gleichen Prinzip wird bei *Running Cola is Africa* verfahren, bei dem der Umriß eines Läufers in eine Cola-Flasche und diese dann in eine Kontur von Afrika überführt wird (siehe Franke, 1971, S. 34). Dargestellt werden die Umwandlungsprozesse mit der statischen Überlagerung zahlreicher Zwischenstadien.

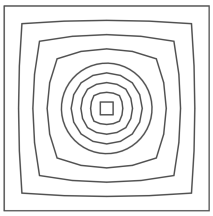
Das *Recoding* soll diesen Prozess dynamisieren. Mit ihren Beispielen hat die CTG nämlich etwas vorweg genommen, das einige Jahre später als Morphing bekannt wurde. Ende der 80er-Jahre konnte es mit professioneller Software in Videos und Kinofilmen eingesetzt und popularisiert werden. Stilbildend wirkte 1991 das Musikvideo Black & White von Michael Jackson (ab Min. 5:28). Wenig später gab es Morphing-Werkzeuge auch für PCs und erlaubte „Morphing für jedermann“. Im Folgenden geht es aber nicht um das Nachprogrammieren solcher Werkzeuge, sondern um einen einfachen Weg, vergleichbare Effekte zu erzeugen. Das erste Beispiel soll diesen Weg verständlich machen.



1. Es soll zunächst ein äußeres Quadrat in einen innenliegenden Kreis überführt werden. Dieser Kreis wiederum soll in ein weiteres darin liegendes kleineres Quadrat überführt werden.

2. In allen drei Objekten sind zunächst die jeweils korrespondierenden Punkte zu bestimmen. Im Beispiel sind es 16 (hier grün gezeichnete) Punkte. Die jeweiligen Punktkoordinaten werden in drei Listen gespeichert.

3. Mit dem Befehl **zeige auf ...** und dem Reporter **Entfernung von ...** kann der Abstand zwischen Ausgangs- und Endpunkt ermittelt und daraus dann auch die Abstände für die korrespondierenden Punkte auf den Zwischenlinien berechnet werden. Die Koordinaten der resultierenden Punkte (hier rot gezeichnet) werden in Listen für die Zwischenlinien gespeichert.



4 Werden nun die Punkte für eine Zwischenlinie aus der entsprechenden Liste abgerufen und die Punkte miteinander verbunden, wird das nebenstehende Gesamtbild dargestellt.

Die Ermittlung der Punkte auf den Zwischenlinien in 3 kann im Einzelnen so erfolgen:

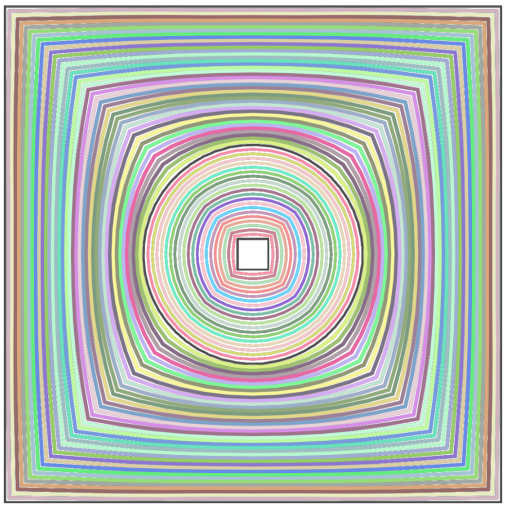


1 In der inneren Schleife werden als Erstes für die **n** Punkte einer Zwischenlinie die korrespierenden Zielpunkte ermittelt.

2 Wird nun auf das **ziel** gezeigt, kann die **entfernung von ziel** und daraus der **abstand** der korrespondierenden Punkte auf den Zwischenlinien ermittelt werden.

3 Wird die Schildkröte zu diesen **n** Punkten geschickt, können dort die Koordinaten in **x-y-koord** zwischengepeichert und diese Liste wiederum in der Liste **zwischenlinie** gespeichert werden.

4 Die **zwischenlinie** wird dann der **liste_zwischenlinien** hinzugefügt.



Für komplexere Figuren ist ein höheres **n**, d.h. eine höhere Auflösung vorteilhaft. Es sind dann dementsprechend mehr korrespondierende Punkte vorzusehen. Auch eine höhere Anzahl Zwischenlinien kann für fließendere Übergange zwischen Ausgangs- und Zielfigur sorgen.

Unser erstes einfaches Beispiel, erweitert um farbige Linien und variable Strichstärken, erzeugt z.B. Ergebnisse, wie die nebenstehende Abbildung.

Das bis hierher geschilderte allgemeine Prinzip kann für beliebige Ausgangs- und Zielfiguren verwendet werden. Allerdings, es auf komplexere Grafiken anzuwenden (wie Läufer, Cola-Flasche und Afrika-Umriss bei *Running Cola is Africa*) erfordert etwas Handarbeit.

Als Ausgangsmaterial werden die Punktlisten der Ausgangs- und Zielgrafiken benötigt. Anders als im vorigen Beispiel mit Viereck und Kreis können sie in der Regel nicht automatisch berechnet werden, sondern sind händisch bzw. halbautomatisch zu erstellen.

Eine Möglichkeit ist, die aktuelle Mausposition im Sekundentakt (oder einer anderen der eigenen Geschicklichkeit angepassten Zeitspanne) abzufragen und diese Koordinaten in eine Liste **x-y-koord** als Zwischenspeicher einzutragen. Mit **stemple** kann diese Position für die bessere weitere Orientierung markiert werden.

Die Liste **x-y-koord** wird dann der Liste **punktliste** hinzugefügt und anschließend wieder geleert.



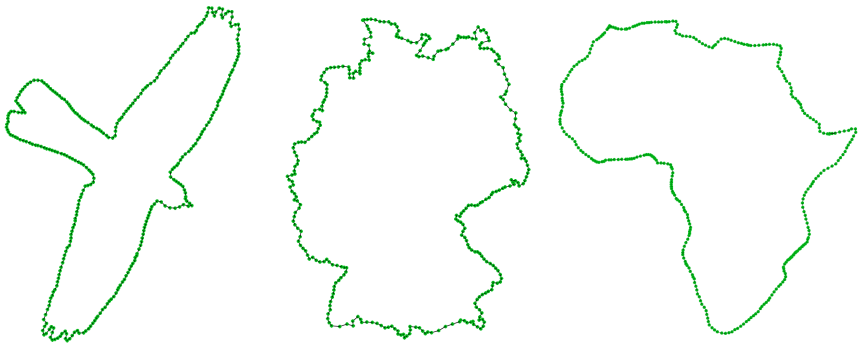
Die Erstellung der Punktlisten ist für alle vorgesehenen Zielgrafiken zu wiederholen. Am Ende müssen die Punktlisten natürlich jeweils die gleiche Anzahl an Punkten enthalten. Abweichungen können bei Bedarf manuell korrigiert werden:



- Durch Anklicken des Punkts rechts neben einem Listeneintrag können Listeneinträge gelöscht werden.
- Durch Doppelklick kann ein Eintrag markiert und dann editiert werden.
- Der Rechtspfeil (linke untere Ecke) erlaubt das Hinzufügen eines Listenelements.

Zum Glück ist dieses aufwändige Vorgehen nur einmal notwendig, denn die so erstellten Punktlisten werden immer zusammen mit dem Programm abgespeichert.

Für das folgende Beispiel, das ich *Zugvogel Deutschland Afrika* nenne, habe ich auf diese Weise drei Zielgrafiken (Zugvogel, Deutschland- und Afrika-Umriss) mit jeweils 330 Punkten erstellt.



Das weitere Vorgehen ist dann wie oben bereits praktiziert: Errechnen und Zeichnen der Zwischenlinien.

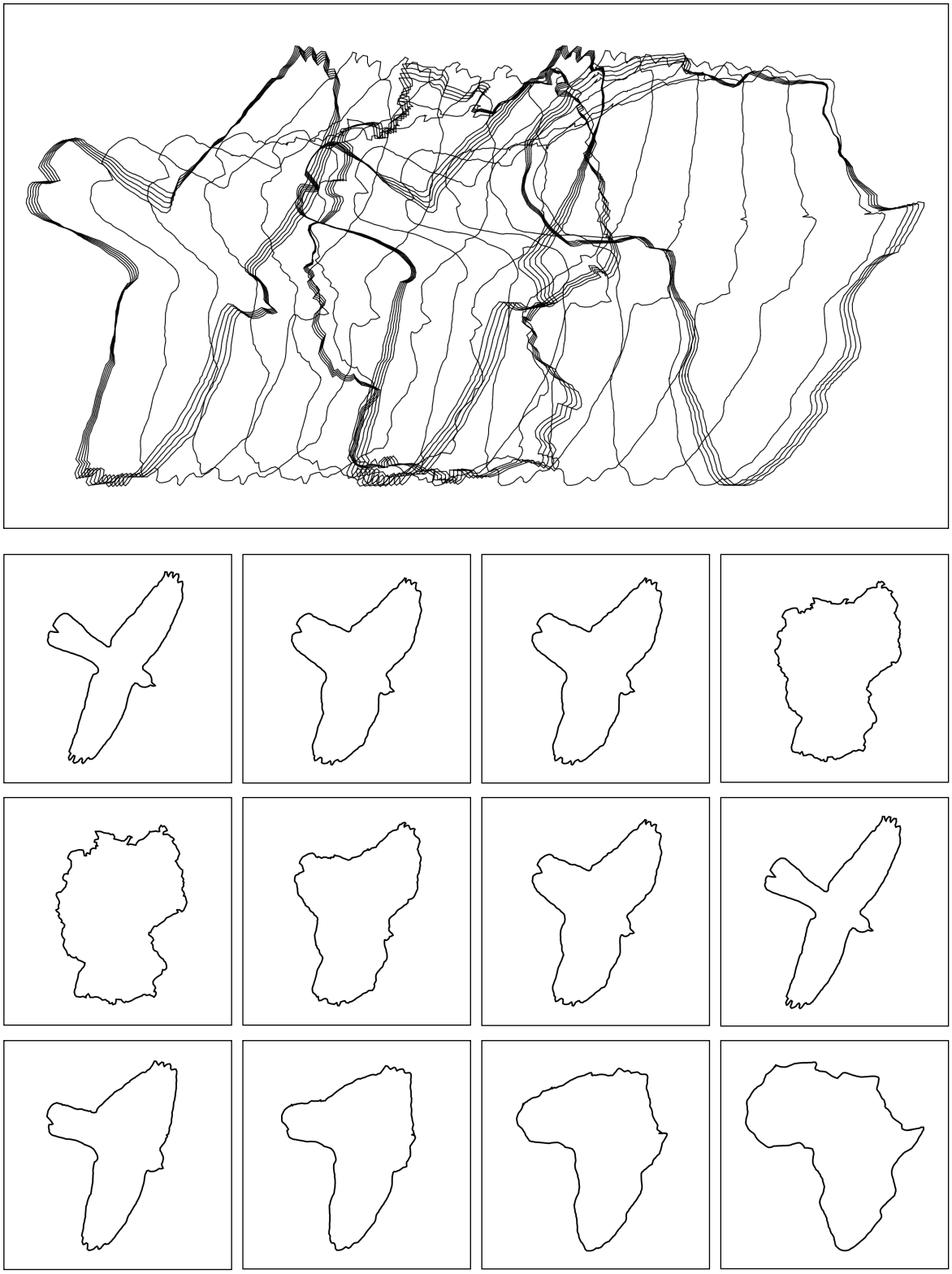


Bild 62: Hommage à CTG: Zugvogel - mit Zwischenstadien (oben), statische Abfolge Animationsphasen (unten)

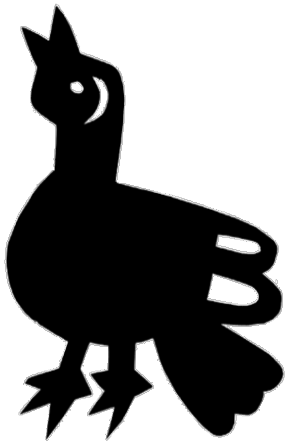
Für die Darstellung in Bild 62 (oben), die sich stark am Vorbild *Running Cola is Africa* orientiert, werden im Anfangs- und Endstadium einer Grafik vier Zwischenlinien in engem Abstand gezeichnet, dazwischen aber nur jede vierte Zwischenlinie. Dadurch werden die Konturen der Ausgangs- und Ziegrafiken deutlicher erkennbar.

Der wohl einfachste Weg, daraus nun eine richtige Animation zu erstellen, ist das Zeichnen jeweils einer Zwischenlinie und das anschließende Abspeichern dieser Zwischenlinie als Kostüm mit **erstelle kostüm ... aus stiftspuren**. Im vorliegenden Fall habe ich z.B. 30 Zwischenlinien vorgesehen, d.h. es sind auf diese Weise 30 Kostüme für den Übergang vom Vogel zum Deutschland-Umriss und weitere 30 Kostüme für den Übergang vom Vogel zum Afrika-Umriss entstanden.

Bild 62 (unten) zeigt die statische Abfolge der Wandlung des Vogels in den Deutschland-Umriss, zurück zum Vogel und dann in den Afrika-Umriss.

21.5 Hommage à Csuri: Hummingbird

Die Idee bei diesem Projekt ist, in Anlehnung an Hummingbird von Charles Csuri, eine eigene Animation zu erstellen. Wie in der Vorlage soll ebenfalls ein Vogel (oder eine andere Bildvorlage) in Bruchstücken auseinanderfliegen, die Bruchstücke sich zufällig über die Bühne bewegen und auf Tastendruck zu ihrer Ausgangsposition zurückkehren. Sie nehmen dort ihre Ausgangsposition ein, so dass am Ende das Ausgangsbild wieder hergestellt ist.



Vogel (Scherenschnitt Inge Wedekind, 2009)

Zu Beginn benötigen wir das Ausgangsbild (hier ein stilisierter Vogel).

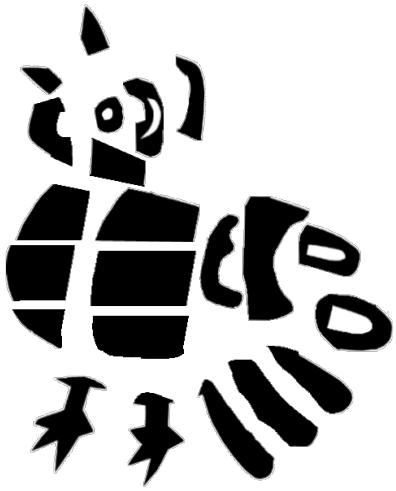
Diese Vorlage ist dann in die gewünschte Anzahl der Einzelteile zu zerlegen[68]. In meinem Beispiel sind so 22 Teile entstanden. Jedes einzelne Teil kann dann als Kostüm eines Sprites gespeichert werden. Für die bessere Zuordnung erhält jedes Teil einen aussagekräftigen Namen.



Wichtig: Ein Kostüm wird nach dem Import auf der Bühne an einer zufälligen Position und mit zufälliger Blickrichtung positioniert. Die Einzelteile sind deshalb auf der

[68] Das Zerlegen kann mit dem Sprite-Editor in Snap! vorgenommen werden. Einfacher und genauer ist das Arbeiten mit einem externen Grafik-Editor (z.B. PikoPixel für Mac OS X, Microsoft paint.net für Windows).

Bühne als Erstes zu der obigen Ausgangsfigur zusammen zu puzzeln! In dieser Lage müssen dann für jedes einzelne Teil die Positionsdaten (**x-Position** und **y-Position**) und **Richtung** festgehalten werden. Diese Werte werden benötigt, damit am Ende die Ausgangsfigur wieder hergestellt werden kann. Sie werden den Skriptvariablen **x-anfang**, **y-anfang** und **richtung-anfang** zugewiesen. Des Weiteren führen wir eine Variable **delta** ein, mit der das Tempo der Bewegung auf der Bühne in einem vorgebenen Zufallsfenster gesteuert wird (was bei Bedarf für jedes Teil unterschiedlich festgelegt werden kann).





Das eigentliche Programm besteht aus einer simplen Dauerschleife, in der das jeweilige Teil bewegt und gedreht wird. Wird der Bühnenrand erreicht, prallt das Teil zurück.



Zu Beginn jeden Schleifendurchlaufs wird geprüft, ob der Betrachter die **Leertaste** gedrückt hat; damit wird der Abschluss der Animation eingeleitet: Das Teil gleitet innerhalb von zwei Sekunden zur Ausgangsposition und dreht sich nach einer kurzen Wartezeit in die ursprüngliche Richtung. Insgesamt erhalten wir so am Ende wieder das Ausgangsbild.

Das brauchen wir von **Snap!**:





Mit **pralle vom Rand ab** wird geprüft, ob die Schildkröte den Rand der Bühne erreicht oder überschritten hat. Ist dies der Fall, prallt sie vom Rand ab und ändert ihre Richtung gemäß Einfallswinkel gleich Ausfallswinkel.



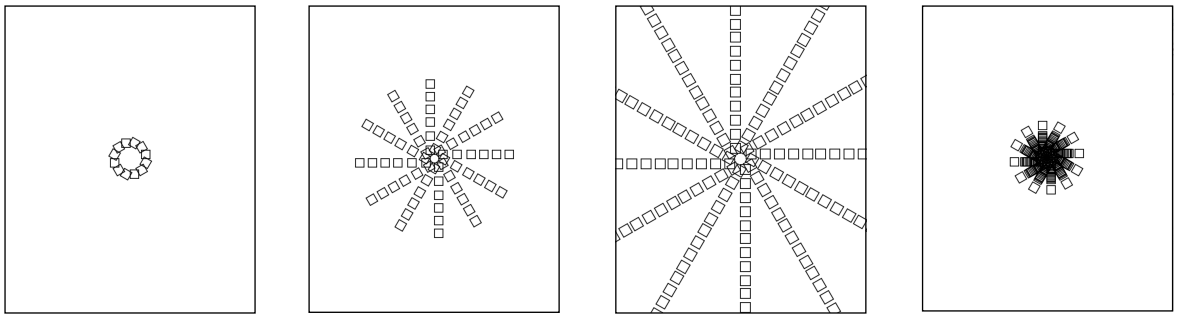
Ausgangsbild *Zwischenstadium Animation* *Rückkehr Ausgangsposition* *Abschlussbild*

Hinweis: *Der Programmcode ist für jedes Sprite derselbe. Es macht also Sinn, den Code für das erste Teil zu erstellen und diesen dann jeweils in die weiteren Sprites zu kopieren. Wenn unterschiedliche Bewegungen der Teile gewünscht werden, kann deren* **delta** *oder die Drehrichtung natürlich auch individuell festgelegt werden.*

21.6 Hommage à Resch: Netzstrukturen

Das folgende Projekt geht auf zwei statische Bilder zurück, die zwei Phasen eines Films von Ronald Resch zeigen (in Franke, 1971, S. 96). Im Begleittext wird beschrieben, dass sich im Film aus einem einzelnen Quadrat eine Netzstruktur entwickelt, die am Ende wieder in das Quadrat zurück verwandelt wird.

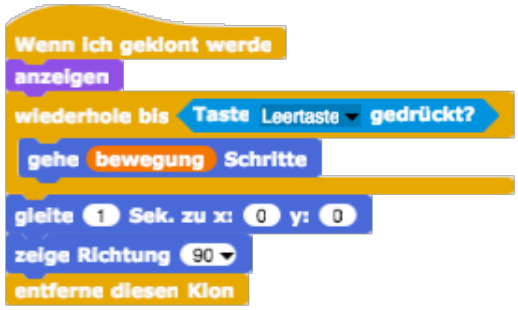
Aus dieser kargen Beschreibung ist bei mir eine Animation entstanden, die ebenfalls ein Quadrat zum Ausgangspunkt nimmt. Dieses Quadrat wird x-fach geklont und die Klone in geradlinige (oder auch rotierende) Bewegungen versetzt. Am Ende gleiten alle Klone zurück in die Ausgangsposition des ursprünglichen Quadrats:



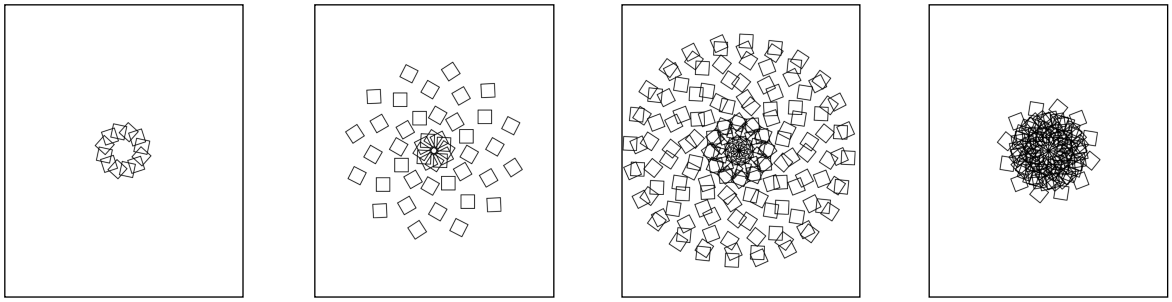
Für die Erzeugung des Quadrats und der Klone können die Blöcke vom *animierten Schotter* übernommen werden, wobei in **klone Steine** die Schleife **360/drehung** mal durchlaufen wird und das Ausgangsquadrat innerhalb der Schleife mit **drehe drehung Grad** gedreht wird.

Im Skript **Wenn ich geklont werde** gehen die erzeugten Klone solange geradlinig um die Strecke **bewegung**, bis vom Betrachter die **Leertaste** gedrückt wird. Danach glei-

ten die Klone in die Ausgangsposition zurück und werden dann gelöscht.



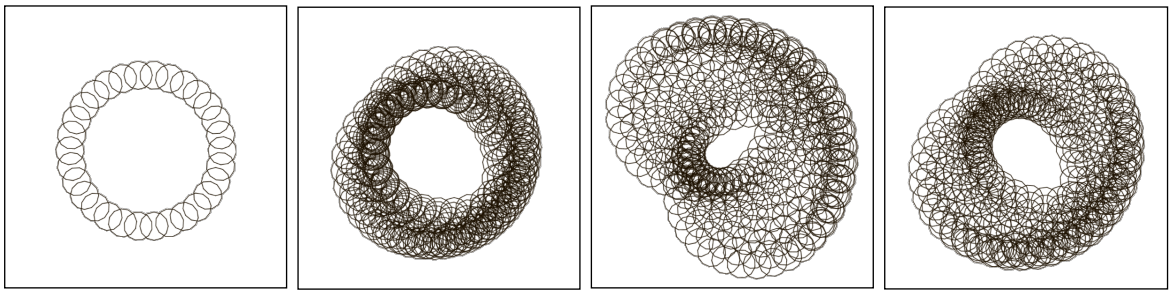
Davon ausgehend können leicht Änderungen und Erweiterungen vorgenommen werden, die den Charakter der Animation deutlich bis drastisch verändern können. Das alleinige Einfügen von **drehe x Grad** in die **wiederhole**-Schleife kann das verdeutlichen:



Anregung: *Durch die Variation von* **groesse**, **drehung**, **bewegung** *und* **warten** *können Sie völlig unterschiedliche - häufig überraschende - Abläufe erreichen. So entstehen z.B. bei Wahl von* **drehung = 90** *auf der Stelle drehende Kreise. Versuchen Sie doch herauszufinden, wie die Gesamtfigur fortlaufend größer und kleiner werden kann („pulsiert").*

Das folgende Projekt ist eine kleine Abwandlung des vorigen Projekts. Als Vorlage zum Klonen werden Kreise genommen, die ihrerseits im Kreis angeordnet sind (also in einer **wiederhole**-Schleife gezeichnet werden). Das Ergebnis für 30 Kreise zeigt die folgende Abbildung links. Diese Anordnung wird nun mehrfach geklont. Bei 12 Klonen ergibt sich das zweite Bild von links. Im Skript **Wenn ich geklont werde** werden die Klone bewegt und gedreht, was u.a. die Bilder rechts ergibt.

Beendet wird das Programm mit dem -Schalter in der Werkzeugleiste, wodurch wieder alle Klone gelöscht werden und von der Bühne verschwinden.



Anregung: *Dieses einfache Programm eröffnet vielfältige Variationsmöglichkeiten. Nicht nur Anzahl und Größe der Kreise sowie Anzahl der Klone bietet sich dafür an. Auch die Bewegung und Drehung der Klone sollten Sie systematisch austesten!*

Das brauchen wir von **Snap!***:*





Mit **Taste Leertaste gedrückt?** wird geprüft, ob die Leertaste (oder eine andere vorgegebene Taste) gedrückt wurde. Dieser Reporter kann für logische Abfragen ausgewertet werden.



Analog kann mit **Maustaste gedrückt?** geprüft werden, ob die (rechte) Maustaste gedrückt wurde. Dieser Reporter kann für logische Abfragen ausgewertet werden.

TEIL III: KUNST PROGRAMMIEREN ?!

In den vorangegangen Kapiteln wurden bereits ausgewählte Beispiele markanter Vertreter der frühen Computerkunst und anderer Stilrichtungen gezeigt und programmtechnisch umgesetzt. In diesem Teil stelle ich eine Reihe weiterer typischer Einzelwerke der Computerkunst vor, sowie Beispiele aus anderen Kunstrichtungen.

Auf die algorithmische Umsetzung und Codierung wird bei diesen Beispielen nur noch in Ausnahmefällen eingegangen. Der Nachvollzug, das *Recoding,* sollte auf der Grundlage der bisher erarbeiteten Konzepte und Sprachelemente von Snap! ohne große Schwierigkeiten gelingen. Wie immer gilt, dass der Nachvollzug der Vorbilder zur eigenständigen Variation und Neugestaltung - dem *Remixing* - anregen soll.

22. NOCH MEHR COMPUTERKUNST ...

Zunächst soll noch einmal die Vielfalt und Bandbreite der Werke der frühen Computerkunst verdeutlicht werden. Die Vorstellung der achtzehn Beispiele erfolgt in alphabetischer Reihenfolge der Künstler.

22.1 ASCII-Art: Texte als Grafiken

Die frühen Computerkünstler beschränkten sich nicht auf die Erstellung von Grafiken, sondern einige von ihnen bewegten sich auch in den Kunstgattungen Film, Musik, Bildhauerei oder auch Lyrik. Manche haben das Arbeiten mit computergenerierten Texten mit deren grafischer Umsetztung verbunden. So hat z.B. Marc Adrian (1930 - 2008) seine Maschinentexte teilweise auch grafisch umgesetzt (Franke, 1971, S. 50). Ein vergleichbares Beispiel von Reiner Kallhardt (geb. 1933 in München) ist Vorbild für das nächste *Recoding* mit Bild 63.

Ausgangspunkt ist das Wort *Permutation*. Dieses Wort wird bei ihm in elf Reihen (entsprechend der elf Buchstaben des Wortes) mit zufällig unterschiedlich großen Buchstaben und jeweils unterschiedlich gewähltem Ausgangsbuchstaben dargestellt. Beim *Recoding* habe ich einerseits die Buchstabengröße variiert (links), andererseits zusätzlich die Buchstaben des Wortes pro Zeile zufällig durcheinandergewürfelt (rechts). Bei der Umsetzung können wir auf Bekanntes zurück greifen, denn die Buchstaben können einfach in eine *m*n-Matrix* eingetragen werden. Der vorgesehene Text kann als Buchstabenliste bereit gestellt werden 

, damit diese dann auch einzeln am Bildschirm darstellbar sind (wofür ein Befehl aus der Blockbibliothek nachgeladen werden kann: 

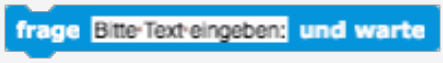
).

Anregung: *Es ist ein leichtes, solche Textgrafiken interaktiv zu gestalten. Dafür muss vom Betrachter ein Text abgefragt werden, z.B. mit* 

, und die 

entsprechend verwertet werden.

Das brauchen wir von **Snap!**:





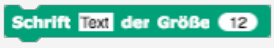
Bei **frage und warte** wird ein Textfenster auf der Bühne geöffnet, in das der Benutzer einen Text eingeben kann. Diese Eingabe ist mit der Enter-Taste abzuschließen:





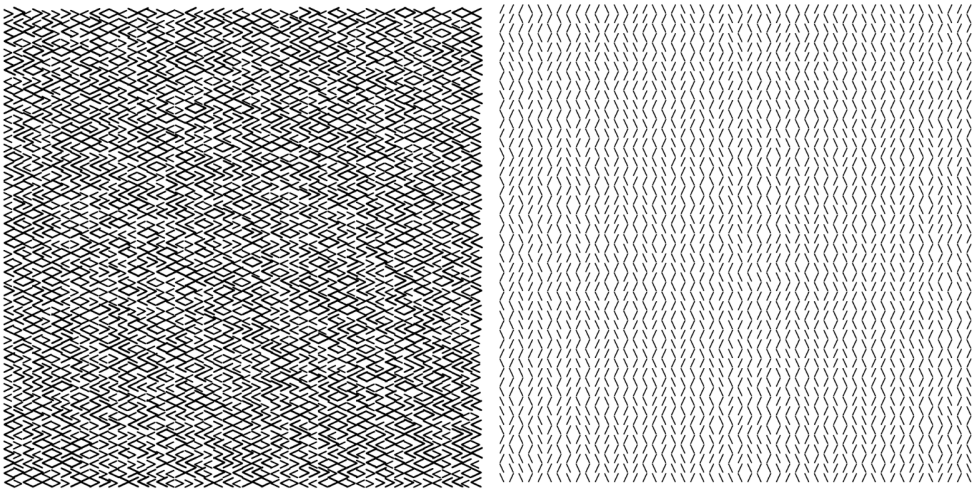
Der Reporter **Antwort** liefert die vom Benutzer eingegebene Zeichenkette, die dann weiter verarbeitet werden kann.

Hinweis: Der folgende Befehle gehört zur Blockbibliothek, die nachgeladen werden kann und in *Anhang C* beschrieben wird.



Mit **Schrift der Größe** wird **Text** an der aktuellen Position und mit der aktuellen Richtung der Schildkröte auf der Bühne ausgegeben.

Die Zusammensetzung kleinster Bildelemente zu Bildern haben als Mosaike eine lange Tradition. Werden dafür wie im obigen Beispiel Zeichen des ASCII-Zeichensatzes verwendet, wird auch von ASCII-Kunst (ASCII-Art) gesprochen. Als einer der Pioniere gilt Kenneth Knowlton. Dabei geht es weniger um die Erstellung von Piktogrammen, sondern eher um die Erzeugung grafischer Strukturen durch die wiederholte Verwendung weniger Zeichen. Die Ergebnisse ähneln dann den *Mustern aus Figuren* im Kapitel 11.



Im vorigen Beispiel (links) ist es die zufällige Anordnung der Zeichen **<** und **>**. Im zweiten Beispiel (rechts) ist es die Zeichenfolge **/** und ****, die sich gleichmäßig wiederholt. In der Bild 64 (oben) sind es die Zeichen **I** und **=**. In der Abbildung darunter ist in Anlehnung an die *Sehtexte* des Medienkünstlers Ferdinand Kriwet (geb. 1942 in Düsseldorf) ein fortlaufender Text kreisförmig und in Farbe angeordnet.

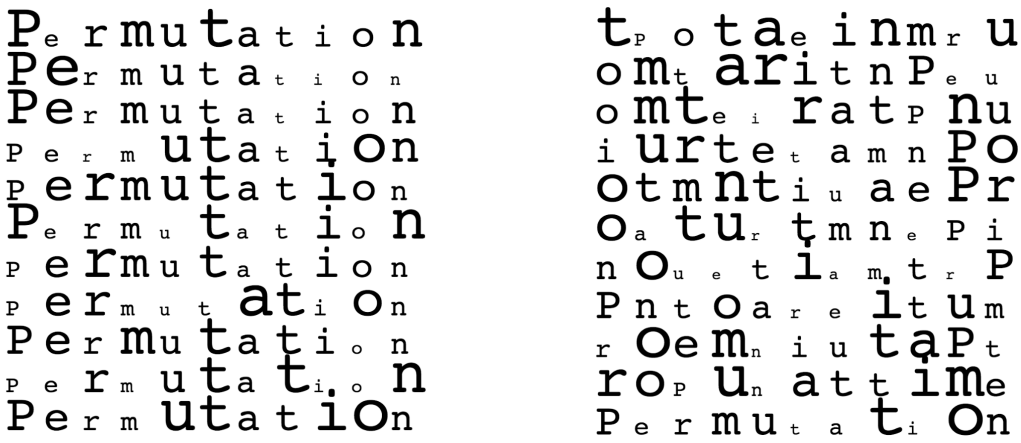


Bild 63: Hommage à Kallhardt: Permutation

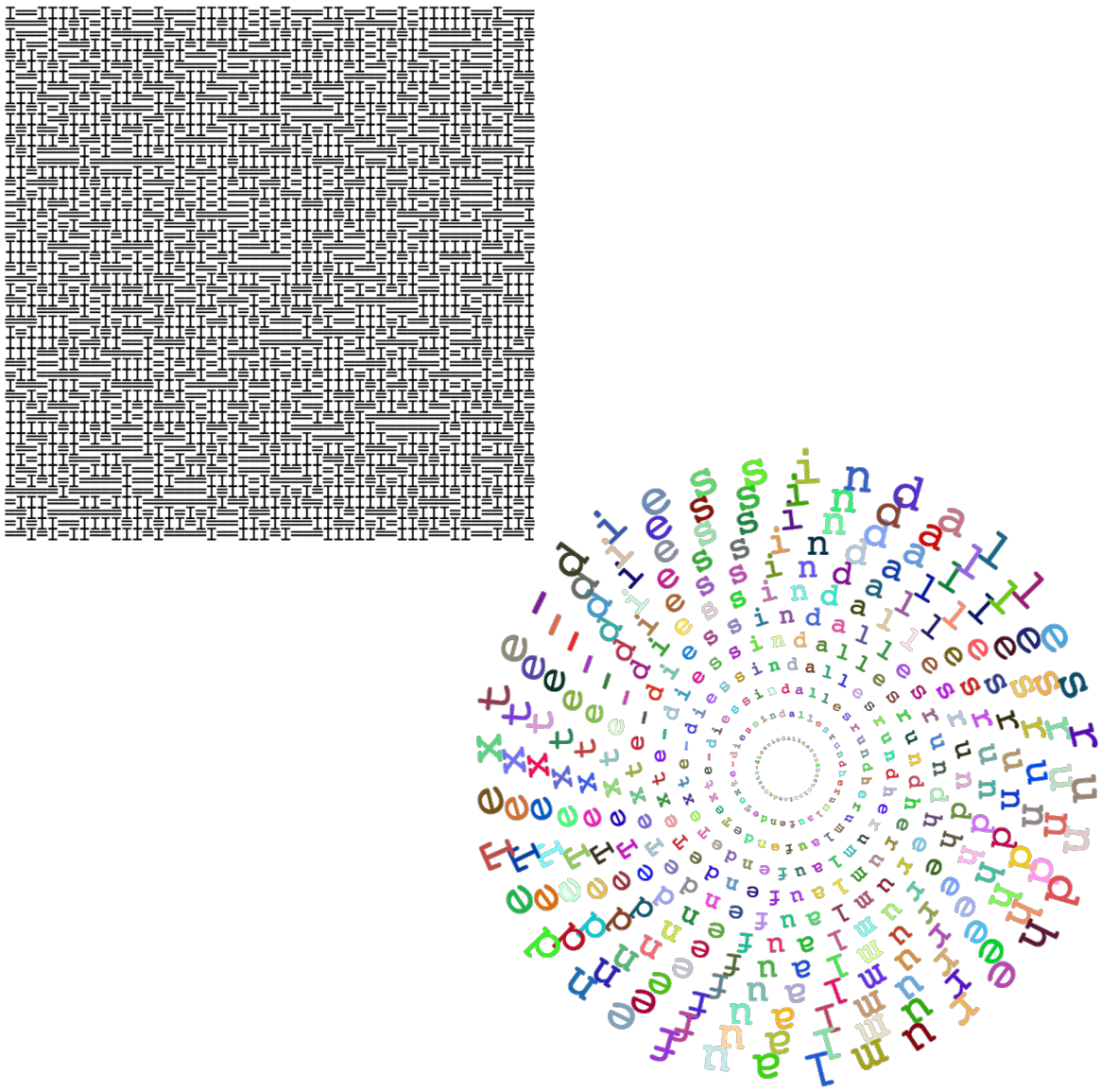
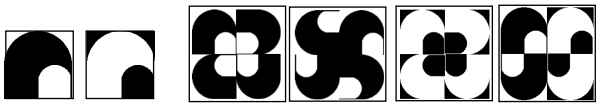


Bild 64: Hommage à Knowlton (oben): ASCII-Art; Hommage à Kriwet (unten): Sehtext

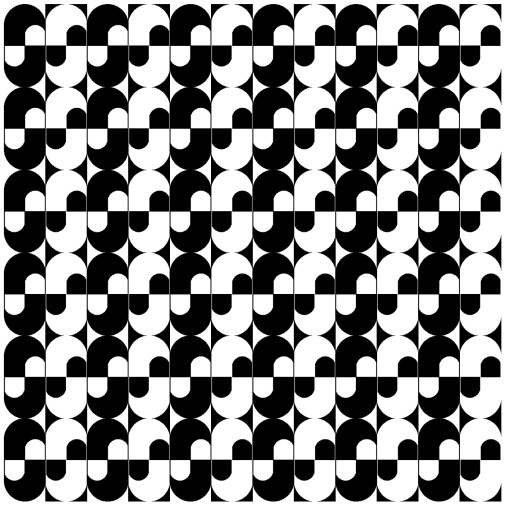
22.2 Hommage à Barbadillo: Bogenbilder

Der Spanier Manuel Barbadillo (1929 -2003) war Maler und Computerkünstler. Er selbst beschreibt in My way to cybernetics, dass ihn das Buch *Kybernetik und Gesellschaft - Mensch und Menschmaschine* von Norbert Wiener stark beeinflusst hat. Er beschäftigte sich mit dem Wechselspiel von Freiheit und Automatisierung und merkte, dass Wiener sich mit demselben Problem aus Sicht der Kybernetik befasste. Das führte ihn zu systematischen Untersuchungen der Variationen, die sich aus der Verwendung weniger modularer Elemente ergaben. Dabei half ihm der Computer, der ihm allerdings nur Vorlagen lieferte, die er dann mit Farbe in Gemälde übertrug.

Ein von Barbadillo gern benutzter Modul erinnert an romanische Fensterbögen. Er verwendet es in einer Ausgangsform und in seiner farblich invertierten Form (in der folgenden Abbildung links). Dieses Grundmodul besteht aus einem Quadrat mit weißen und schwarzen Teilbereichen. Deren Anordnung ist bewusst so gestaltet, dass sich bei der Aneinanderreihung farbliche Anschlüsse ergeben. Damit eröffnen sich entsprechend viele Variationsmöglichkeiten. Barbadillo arbeitete mit einem 4*4-Raster (in der Abbildung rechts), das als Vorbild für das *Recoding* dient. Innerhalb des Rasters können dem Grundmodul unterschiedliche Richtungen gegeben werden.

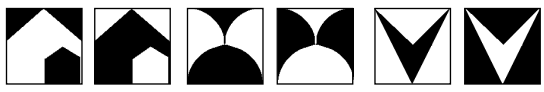


Die Anordnung vieler solcher Raster in einer m*n-Matrix (hier 12*12) ergibt die Gesamtmuster. Auf diese Art hat Barbadillo lange Bildserien produziert, aus denen er die Vorlagen für seine Ölbilder wählte.



Ein Vergleich der erzielten Ergebnisse mit den *Schwarz-Weiß-Strukturen* von Zdeněk Sýkora (vgl. Kapitel 11: *Muster aus Figuren*) zeigt deren große Ähnlichkeit. Für die programmtechnische Umsetzung der Bogenbilder von Barbadillo können tatsächlich die Prozeduren von Sýkora weitgehend übernommen und leicht angepasst werden.

Anregung: *Sie sind natürlich nicht an Barbadillos Entwürfe gebunden. Es ist ein Leichtes, andere Grundmodule zu definieren, wie die folgenden Beispiele verdeutlichen sollen.*



Barbadillo hat wohl überwiegend mit schwarzen unsd weißen Teilbereichen gearbeit, selten ergänzt um eine dritte Hintergrundfarbe. Eigene Farbgebungen und Farbverteilungen führen jedoch zu einer interessanten Erweiterung des möglichen Bilderspektrums.

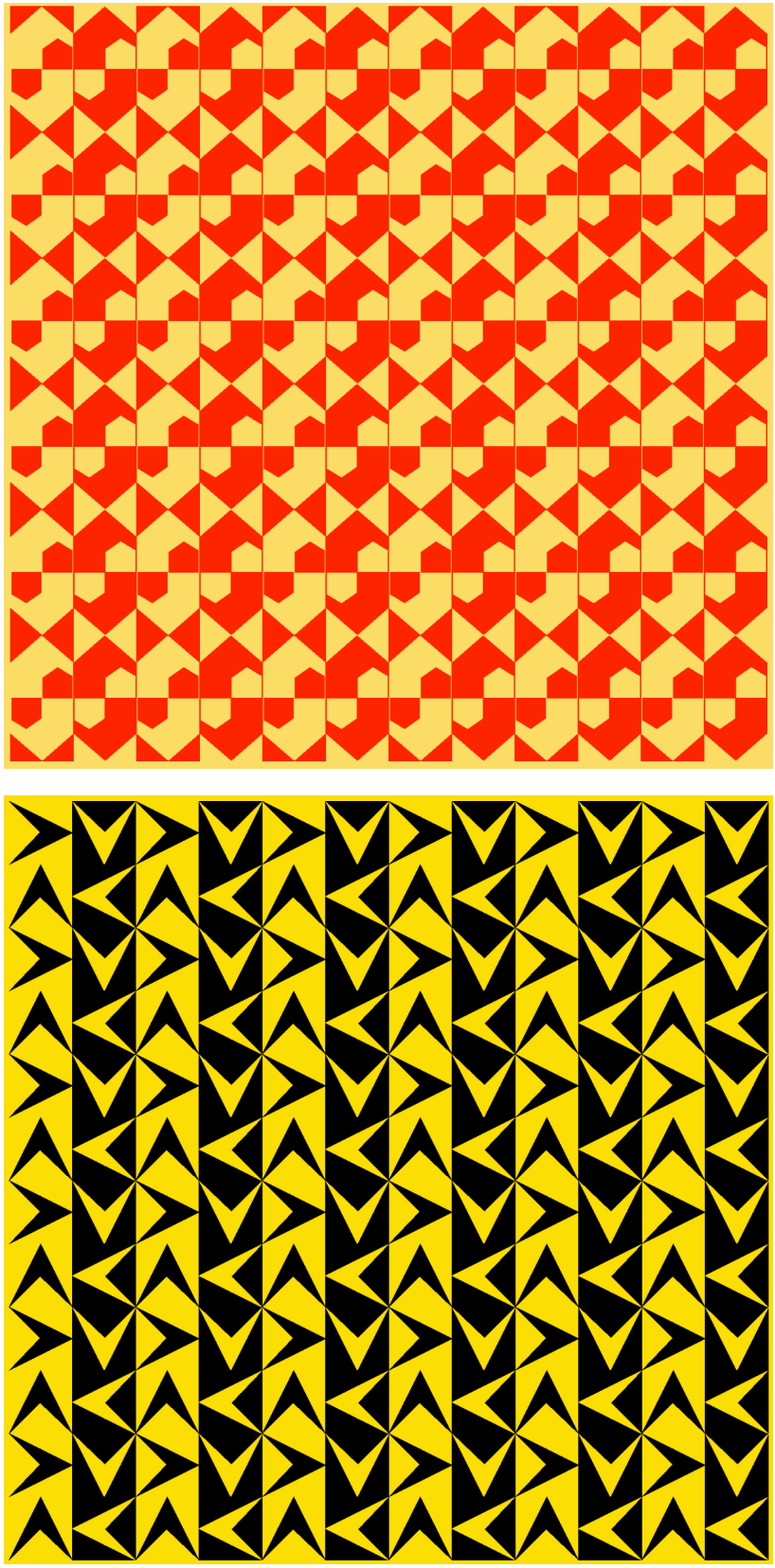


Bild 65: Hommage à Barbadillo: Remixing Bogenbilder

22.3 Hommage à Bartnig: Quadratentwicklung

Horst Bartnig (geb. 1936 in Militsch, Schlesien) ist ein Vertreter der Konkreten Kunst, der früh den Computer dazu nutzte, Serien und Variationen zu erstellen. Ein typisches Beispiel ist *Entwicklung zum Quadrat* (im Original Weiß auf Schwarz), bei dem in einer tabellarischen Anordnung Linien sowohl horizontal als auch vertikal vermehrt auftreten und einen Verdichtungseffekt erzeugen.

Die Erzeugung dieses und vergleichbarer Bilder ist relativ simpel mit geschachtelten Wiederholungsschleifen möglich. Dabei ist zu beachten, dass die hinzukommenden Linien immer von der Mitte des betreffenden Feldes aus platziert werden.

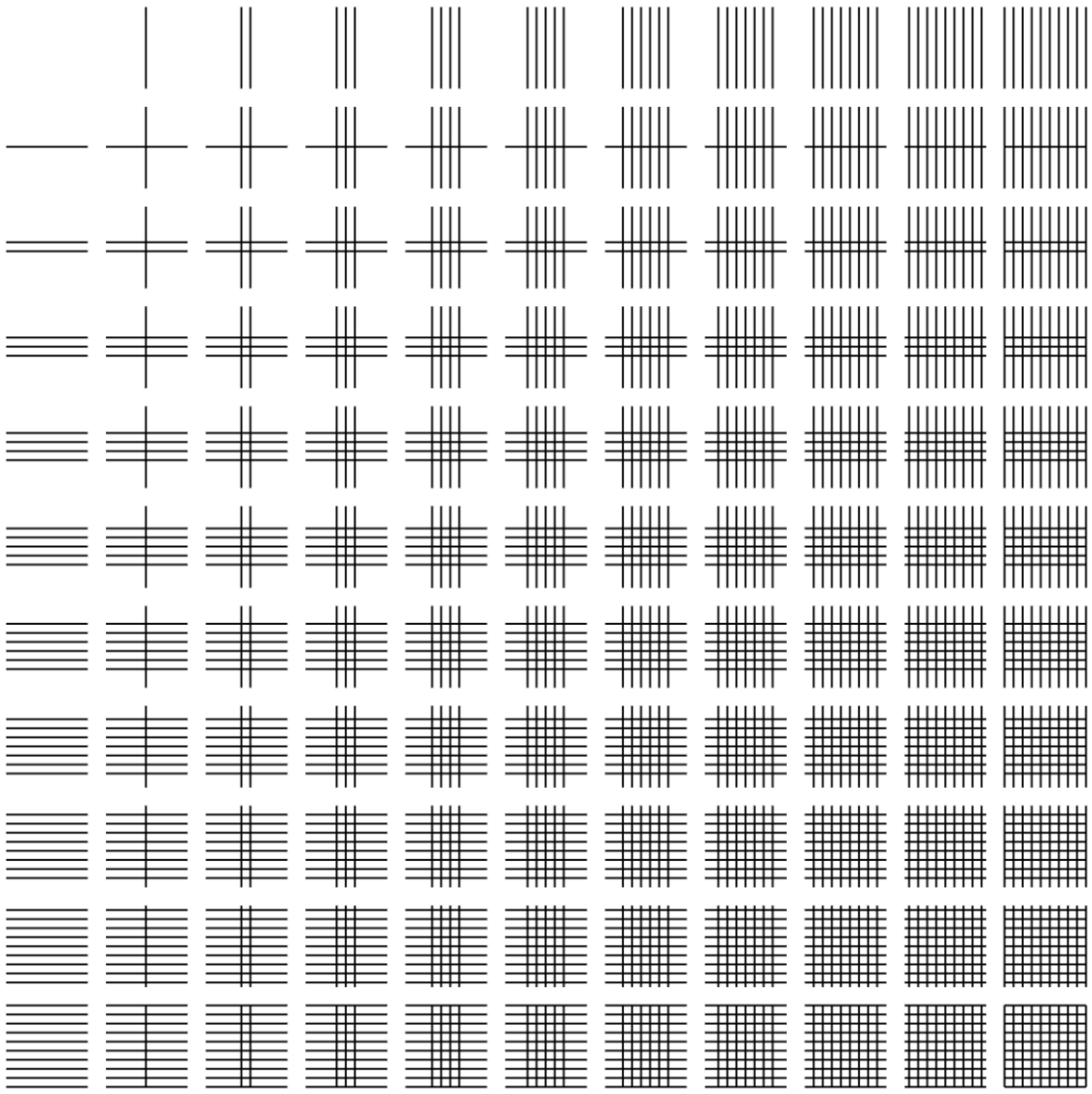


Bild 66: Hommage à Bartnig: Quadratentwicklung

22.4 Hommage à Beckmann: DONKO Generator

Der Maler und Bildhauer Otto Beckmann (1908 - 1997) machte vor seinem Kunststudium eine Ausbildung zum Elektromechaniker. Das war wohl auch eine Basis dafür, dass er ab 1966 versuchte, die künstlerische Dimension der Computertechnologie auszuloten und die Experimentalarbeitsgruppe *ars intermediale* gründete. Beckmann arbeitete mit der Verknüpfung von analogem und digitalem Computer, was ihm die interaktive Steuerung seiner Programme erlaubte. Seine genaue Vorgehensweise bei der Bilderzeugung ist im Einzelnen nicht dokumentiert. Typisch ist aber, dass seine frühen Computergrafiken - erzeugt mit einem speziellen Gerät, dem sogenannten DONKO-Generator - eine malerische Ästhetik entfalten. Diesen ist das folgende *Recoding* nachempfunden.

Die Umsetzung ist simpel; es reichen Wiederholungsschleifen. Eine äußere Schleife legt die Startpunkte fest, von denen aus senkrecht in der Dicke variierende Linien gezeichnet werden. In einer ersten inneren Schleife bis zur Bildmitte nimmt die Stiftdicke zufällig tendenziell zu, in einer zweiten inneren Schleife dagegen eher wieder ab.

Anregung: Bild 68 *orientiert sich stark an einigen Bildern Beckmanns, die er* Imaginäre Architektur *genannt hat (siehe Piehler, 2002, Abb. 8). Für die Erzeugung habe ich interaktive Elemente eingeführt, einerseits zur Festlegung der Startpunkte der senkrechten Linien, andererseits „Hindernisse", um die die Linien herumgeführt werden.*

22.5 Hommage à Beyls: Grid Based Systems

Der Maler, Musiker und Komponist Peter Beyls (geb. 1950 in Kortrijk, Belgien) gehört zu den produktivsten und vielseitigsten Computerkünstlern. Ein wesentlicher Bestandteil seiner Plotterzeichnungen sind Gitterstrukturen, in denen er unterschiedlichste Formen in zufälligen Variationen platziert. Die Formen reichen von quadratischen Farbfeldern über Strichanordnungen - wie unten gezeigt - bis zu figurativen Variationen.

Seine Werke entstehen durch generative Algorithmen, bei denen er häufig audiovisuelle Komponenten kombiniert und durch interaktive Elemente die Betrachter in das Geschehen einzubeziehen versucht.

Für das *Recoding* eines seiner Vorbilder (in Beyls, 2014, S. 39) kann hier erneut die m*n-Matrix genutzt werden (in Bild 69 z.B. 40*75). In jedem Feld ist dann zufällig zu bestimmen wieviel Linien (hier z.B. 1<n<4) in welcher Farbe (hier z.B. rot, grün oder blau) gezeichnet werden.



Bild 67: Hommage à Beckmann: Elektronische Computergrafik I

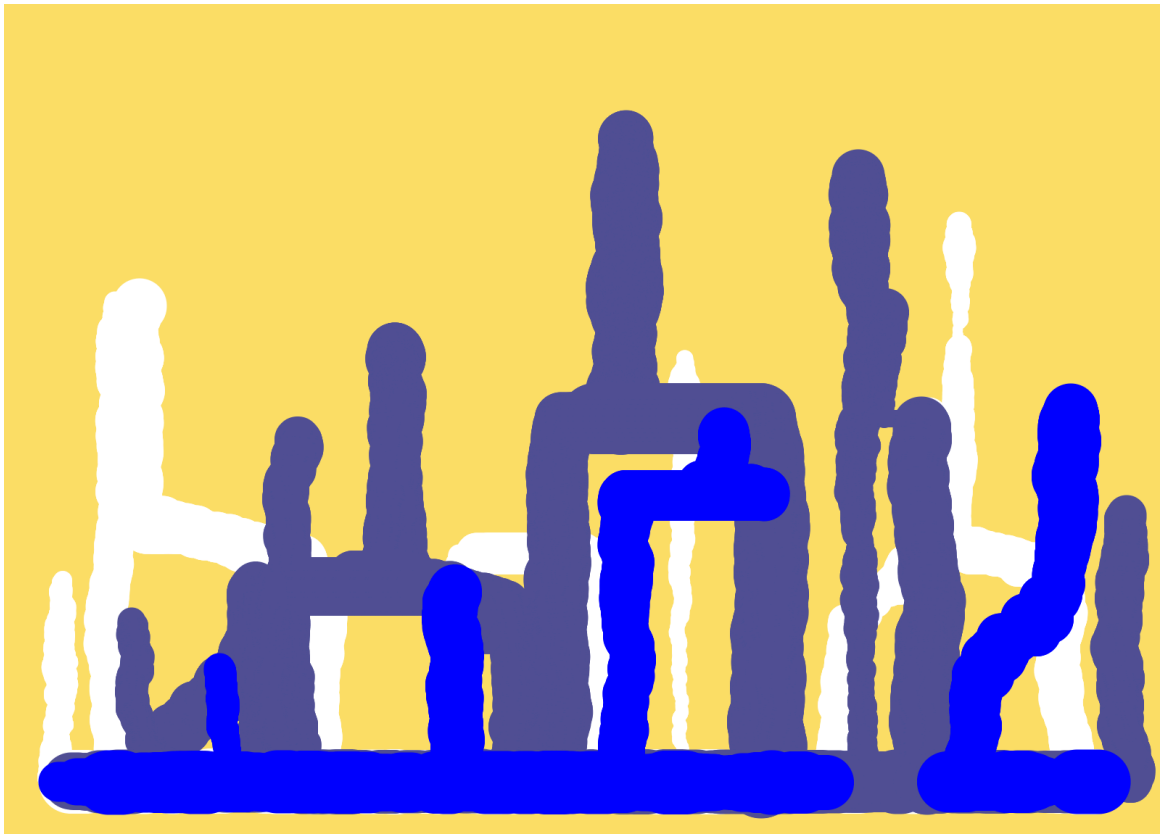


Bild 68: Hommage à Beckmann: Elektronische Computergrafik II

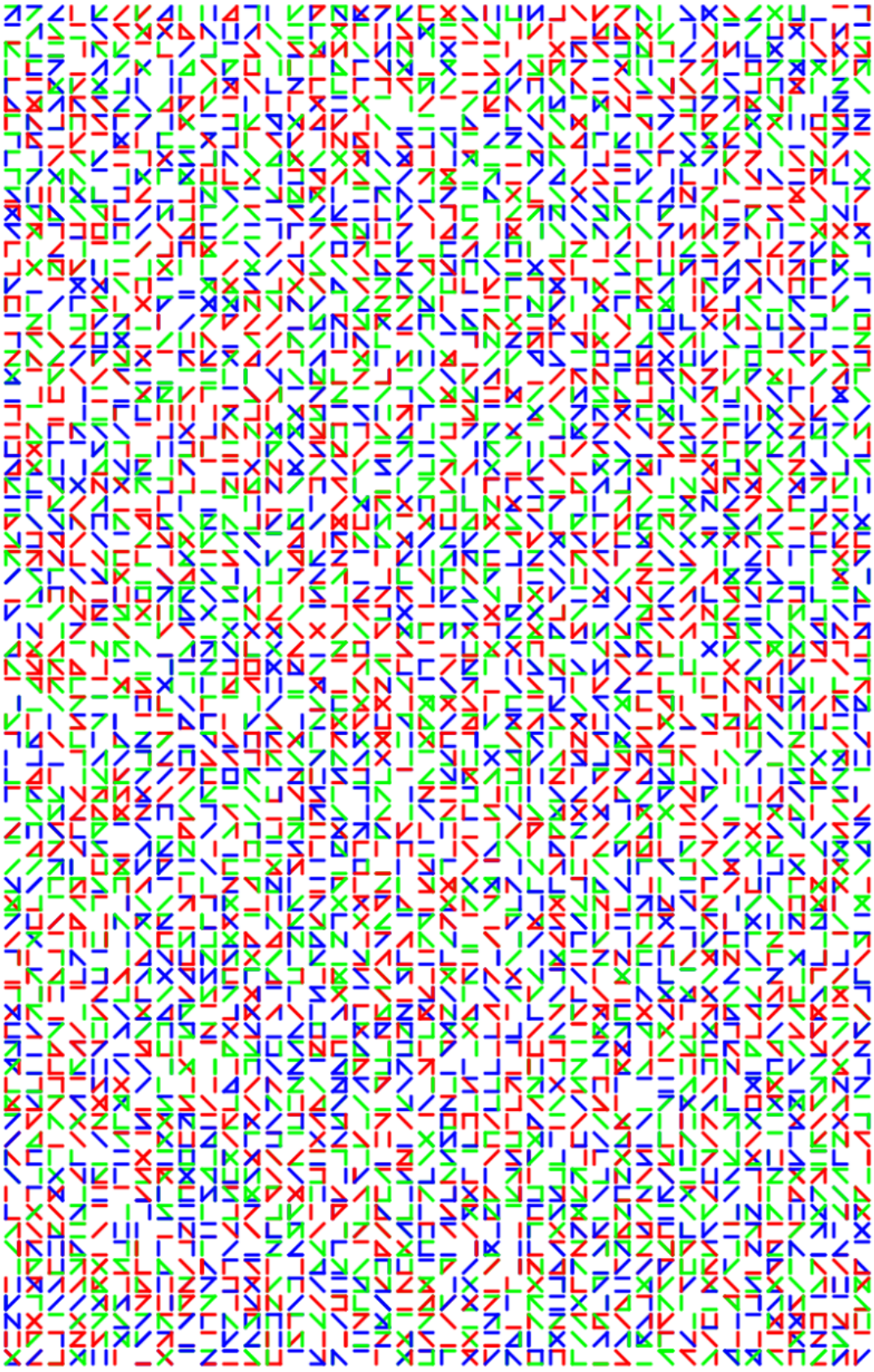


Bild 69: Recoding Beyls: Grid Based Systems

22.6 Hommage à Csuri & Schaffer: Feeding Time

Auch gegenständliche Grafiken haben in die Computerkunst Eingang gefunden. Die Ursprungsbilder sind manuell zu erstellen, können dann aber leicht in Größe und Ausrichtung variiert werden. Bei Charles Csuri und James Shaffer ist das Ursprungsbild die Zeichnung einer Fliege (siehe Glowski, 2006, S. 82 f.). Diese wird dann in konzentrischen Ringen um den Bildmittelpunkt mit zufälligen Abweichungen verteilt (das Original ist Schwarz-Weiß).

Für das *Recoding & Remixing* habe ich mit einem Zeichenprogramm das farbige Bild einer Fruchtfliege Drosophila melanogaster gezeichnet und als Sprite in Snap! importiert. Von diesem Sprite werden viele Klone erzeugt (im Beispiel 200), die dann vom Bildmittelpunkt aus sich davon zufallsgesteuert entfernen.

Anregung: *Diese Anordnung ruft geradezu danach, zu einer Animation ausgebaut zu werden, bei der sich der Fliegenschwarm fortlaufend über die Bühne bewegt.*

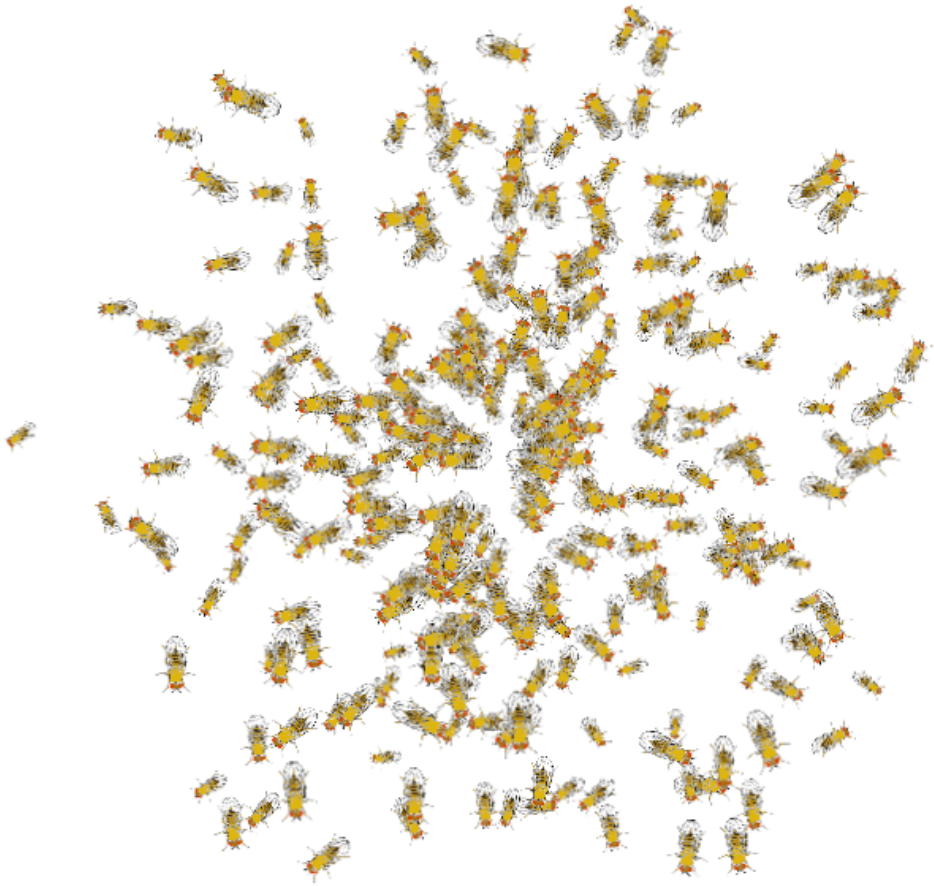


Bild 70: Recoding Csuri & Schaffer: Feeding Time

22.7 Hommage à CTG: Random Windows

Die Computer Technic Group (CTG) war ein loser Zusammenschluss mehrerer japanischer Computerkünstler. Entsprechend vielfältig und unterschiedlich sind ihre Werke ausgefallen. Manche ihrer Werke können einzelnen Mitgliedern zugeordnet werden (vgl. Bild 31 von Masao Komura). Andere, wie das folgende, werden der Gesamtgruppe zugeschrieben.

Bei *Collection of Random Windows* (nach Franke, 1971, S. 77) werden in einer 6*6-Matrix jeweils Quadrate eingezeichnet. In jedem Feld der Matrix wird die Drehrichtung der Quadrate, die Abnahme ihrer Seitenlänge und die schrittweise Änderung der Drehrichtung zufällig festgelegt. Bei der programtechnischen Umsetzung kann in der bekannten *m*n-Matrix* mit dem Block **n-eck um x y n (=4) radius** gearbeitet werden.

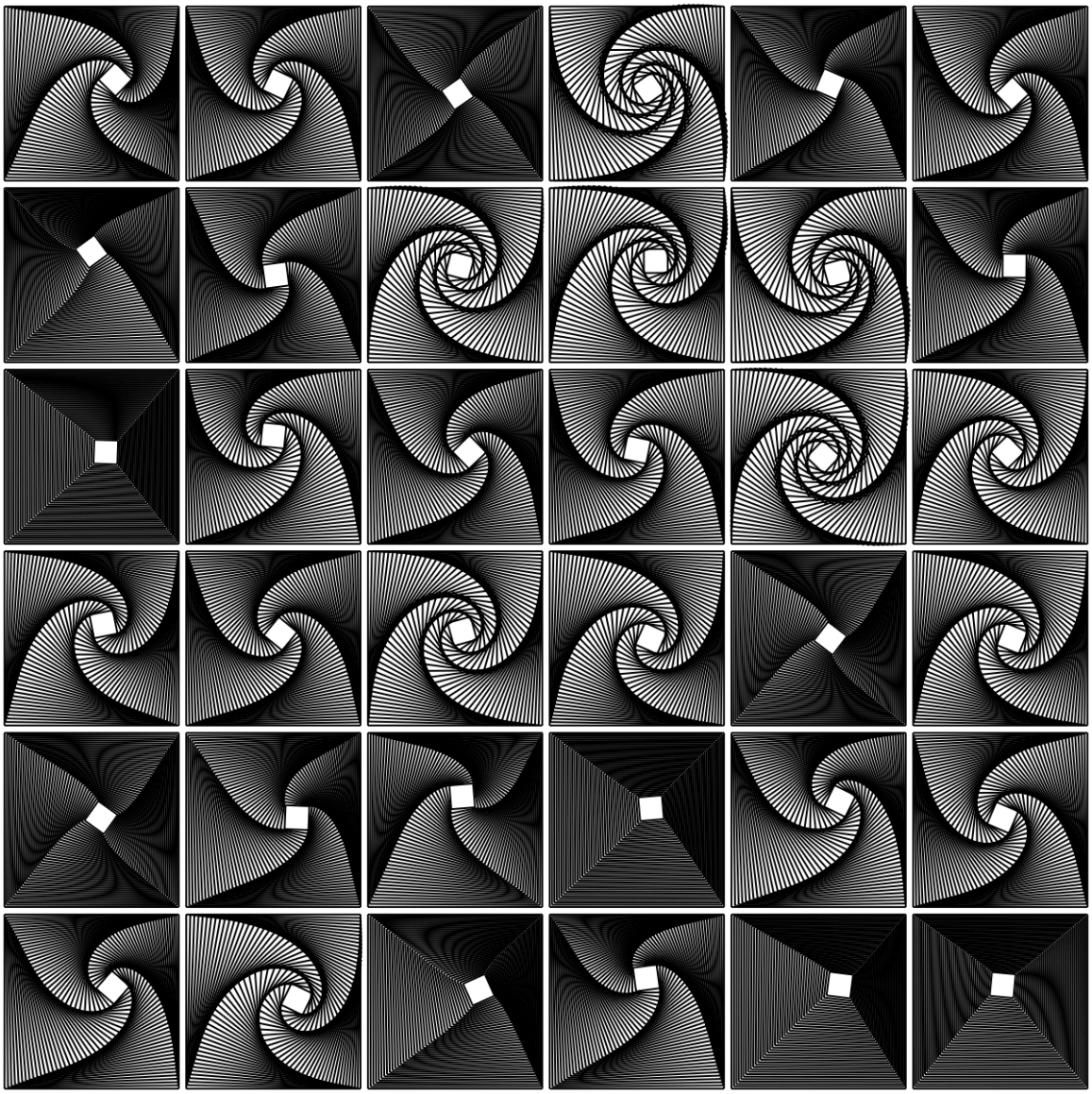


Bild 71: Hommage à CTG: Random Windows

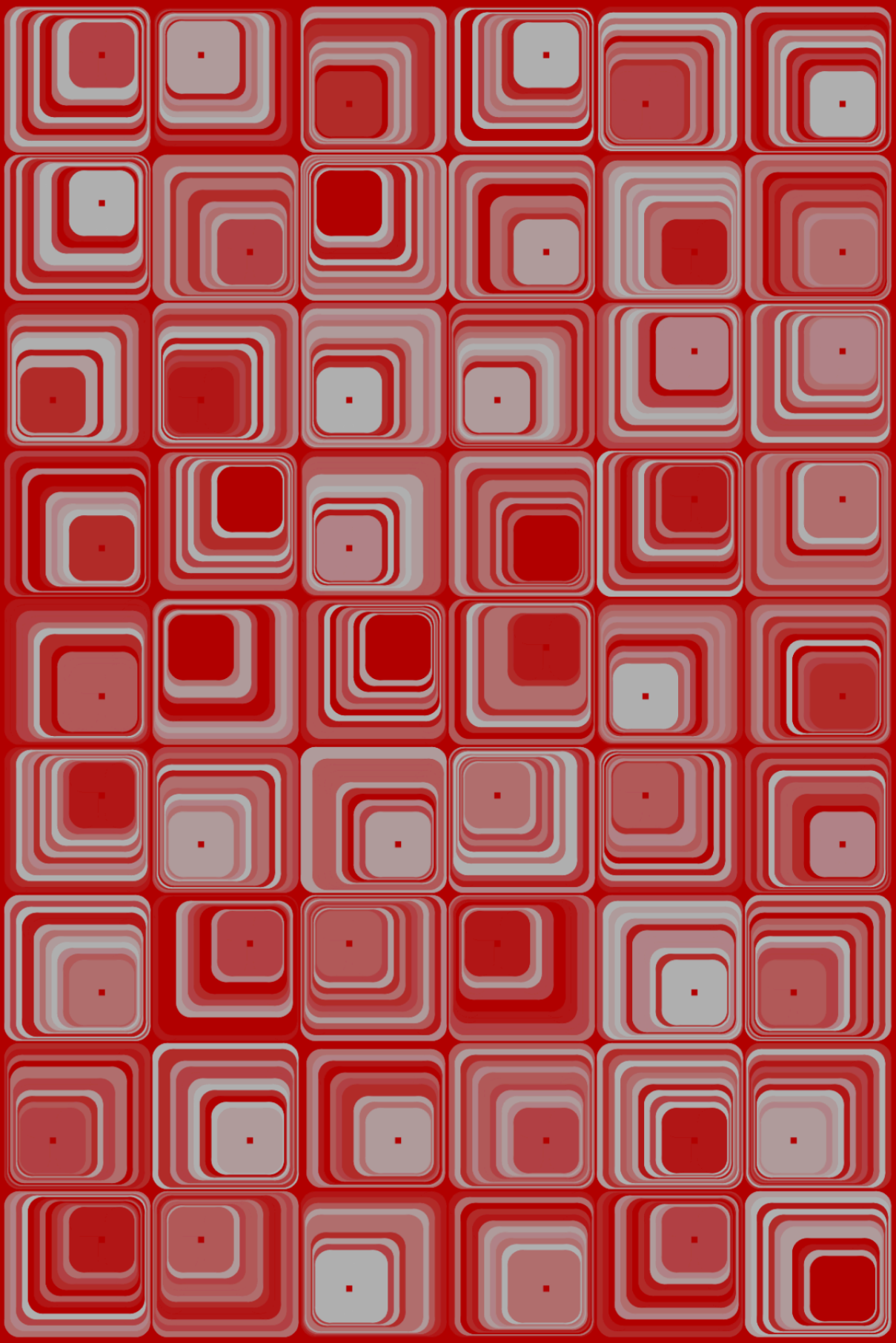


Bild 72: Remixing CTG Random Windows

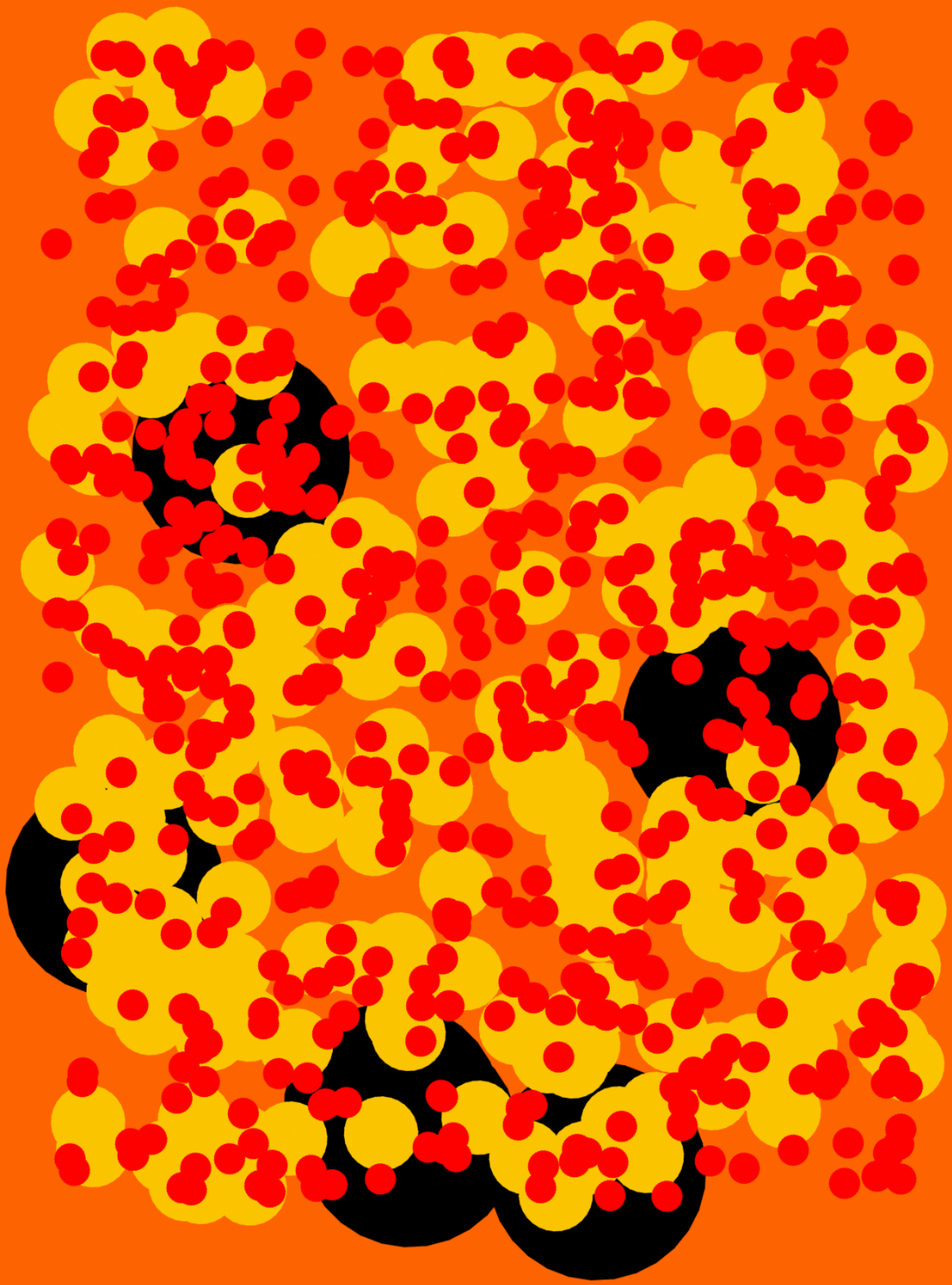


Bild 73: Remixing Franke: Kreise

22.8 Hommage à Franke: Quadrate und Kreise

Die Vorlage für das folgende Bild *Quadrate* stammt von Herbert W. Franke (geb. 1927 in Wien). Franke hat mit unterschiedlichsten Methoden, Geräten und Programmen gearbeitet. Er hat angefangen mit Pendeloszillogrammen, die (wie bei Laposky, Kapitel 10: *Vom Analogen zum Digitalen*) fotografisch reproduziert wurden, sich mit algebraischen Kurven höherer Ordnung beschäftigt, aber er hat auch bereits interaktive räumliche Projektionen entwickelt. Daneben hat er mehrere Bücher zur frühen Computerkunst herausgegeben (u.a. Franke 1971, 1984), denen ich viele Beispiele für das vorliegende Buch entnehmen konnte.

Sein Bild *Quadrate* findet sich in verschiedenen Quellen in unterschiedlichen (Farb-) Varianten. Für das *Recoding* verwende ich eine Version mit den Farben Orange, Rot und Schwarz. Die Erzeugung dieses und vergleichbarer Bilder ist einfach mit mehreren Wiederholungsschleifen für die jeweils gleich großen und gleichfarbigen Quadrate möglich. Dem Vorbild folgend sind es im konkreten Beispiel 400 kleine Quadrate (rot), 120 mittelgroße Quadrate (orange) und 4 große Quadrate (schwarz).

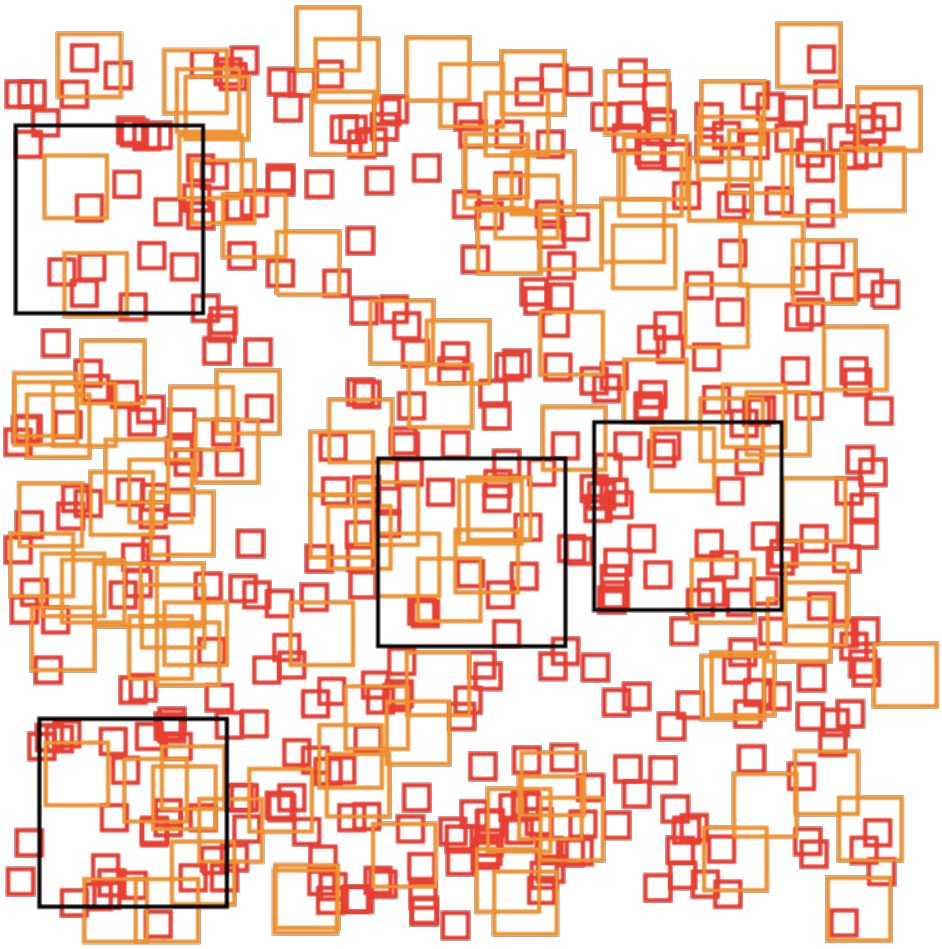


Bild 74: Hommage à Franke: Quadrate

Anregung: *Von Franke selbst gibt es strukturell gleichartige Bildversionen, bei denen die Quadrate durch Kreise ersetzt sind. Dieses Remixing seines eigenen Bildes kann erreicht werden, indem statt der Prozedur* **quadrat** *die allgemeinere Prozedur* **zeichne vieleck eckenzahl** *verwendet wird, oder, um das Remixing eine Stufe weiter zu treiben, indem gefüllte Vielecke gezeichnet werden. Ebenso leicht ist es dann, davon weitere Farbvarianten zu erstellen.*

22.9 Hommage à Gruppo N: Dynamic Visions

Nur wenige Jahre (1959 - 1966) existierte die italienische Künstlergruppe Gruppo N mit den Mitgliedern Alberto Biasi, Ennio Chiggio, Toni Costa, Edoardo Landi und Manfredo Massironi. Neben Kinetischer Kunst haben sie auch Programmierte Kunst erzeugt, die hier als Vorbild dienen soll. Obwohl die Gruppe eine individuelle Autorenschaft ablehnte, lassen sich die hier gewählten Beispiele individuell zuordnen.

Alle Gruppenmitglieder befassten sich in unterschiedlicher, aber ähnlicher Form mit Wiederholungsmustern und Strukturen. Sie wollten damit bewusst dynamische und wechselhafte Eindrücke hervorrufen. Die folgenden Beispiel bauen in diesem Sinne aufeinander auf.

Alberto Biasi (geb. 1937 in Padua, Italien) hat opto-dynamische Reliefs geschaffen. Grundlage bilden zweidimensionale Liniengrafiken. Bereits in der nichtdynamischen Form treten die für seine Reliefs charakteristischen Moiré-Muster auf (ausführlich dazu im Kapitel 23: *Moiré-Muster*). In seinem Bild Lot Nr. 639 werden Linien in regelmäßigem Abstand von der Quadratseite zum Mittelpunkt gezogen (in Bild 75 links). Wird der Zielpunkt gegenüber dem Mittelpunkt leicht versetzt (Bild 75 rechts), verstärkt sich der Effekt deutlich.

Das *Recoding* dieser Grafiken beschränkt sich darauf, in vier Schleifen (für jede Quadratseite eine) die gewünschte Linienanzahl vom Ausgangspunkt zum Mittelpunkt zu zeichnen. Wenn die Anzahl der Linien pro Seite und die Abweichungen vom Mittelpunkt variabel gestaltet werden, lassen sich unterschiedliche Effekte erzeugen.

Das gleiche Grundschema verwendet Ennio Chiggio (geb. 1938 in Neapel, Italien). Allerdings wählt er in einer Serie *Interferenze lineare* jeweils die vier Eckpunkte eines Quadrats als Zielpunkte. Die Linien werden wieder in regelmäßigem Abstand, diesmal von der gegenüberliegenden Quadratseite, zu dem entsprechenden Zielpunkt gezogen (das Schema wird in Bild 76 links deutlich). Je kleiner die Abstände der Ausgangspunkte werden, desto deutlicher tritt auch hier der typische Moiré-Effekt auf (Bild 76 rechts).

Das *Recoding* ist nahezu identisch wie im vorigen Beispiel, nur dass diesmal für jede Quadratseite der neue Zielpunkt festzulegen ist. Dieses Vorgehen wird dadurch erleichtert, dass Snap! einen Befehl **zeige auf ...** anbietet, der bewirkt, dass die Richtung der Schildkröte jederzeit auf den Zielpunkt festgelegt werden kann, egal wo sie sich gerade auf der Bühne befindet.

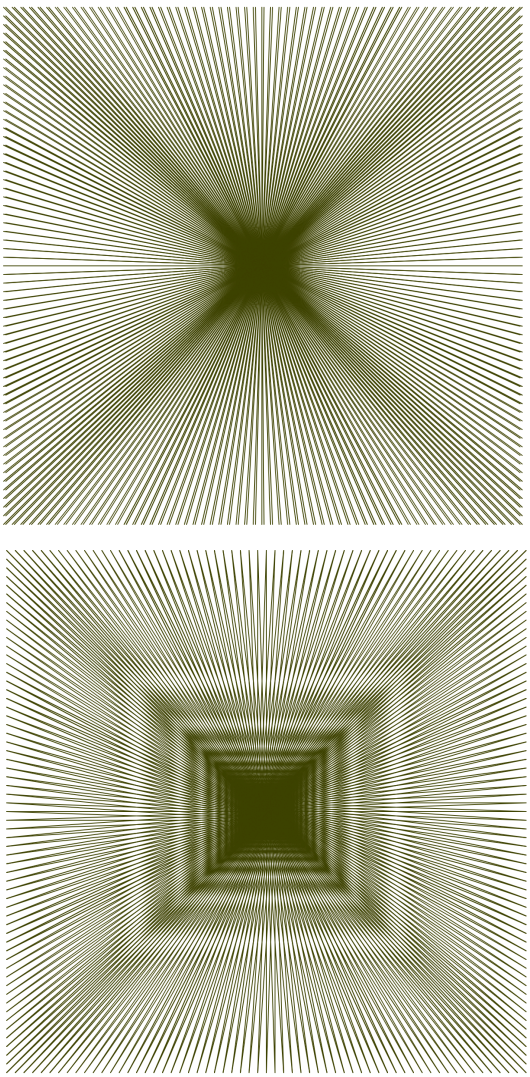


Bild 75: Hommage à Biasi: Lot Nr. 639

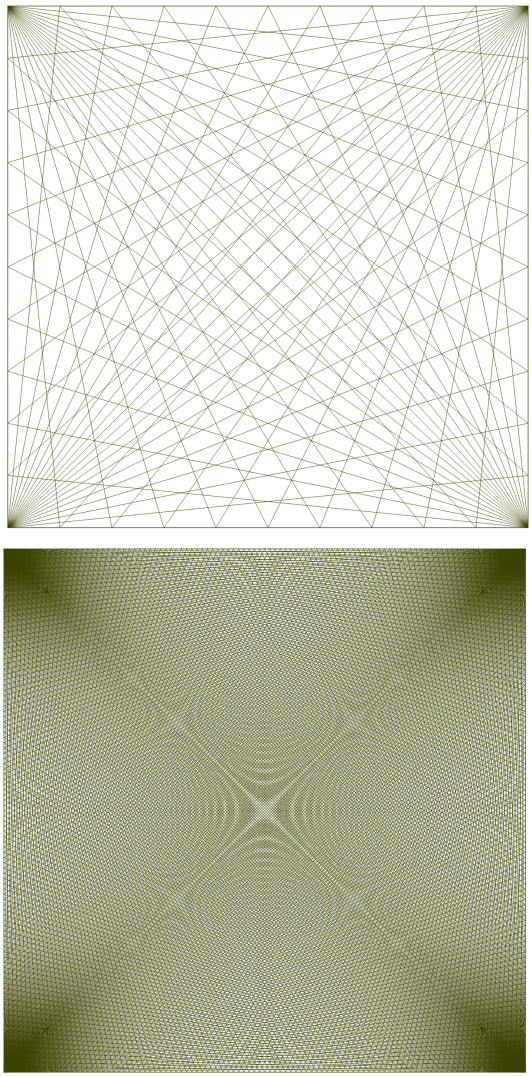


Bild 76: Hommage à Chiggio: Interferenze Lineare

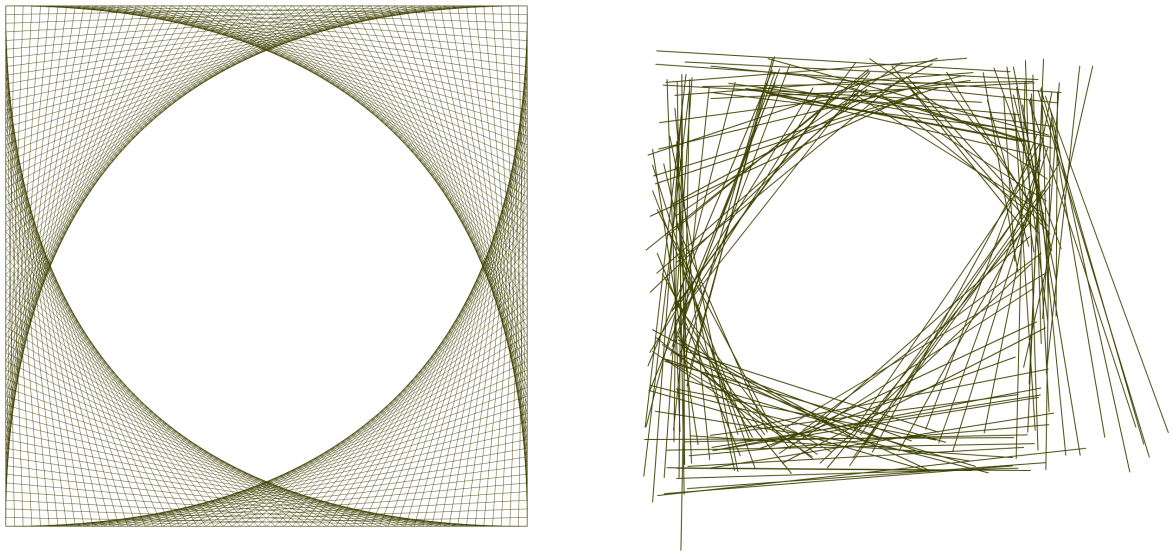


Bild 77: Hommage à Massironi: Sfera negativa

Eine daran anknüpfende Variante hat Manfredo Massironi (1937 - 2011) mit dem Bild *Sfera negativa* entwickelt (als Vorlage für eine Skulptur). Auch er unterteilt die Quadratseiten in regelmäßigem Abstand, aber bei ihm werden die Linien vom jeweiligen Ausgangspunkt zu Zielpunkten gezogen, die ebenfalls regelmäßigen Abstand auf der nachfolgenden Quadratseite aufweisen (das Schema wird in Bild 77 links deutlich). Sowohl Ausgangspunkte als auch Zielpunkte können innerhalb einer vorgegebenen Bandbreite zufällig verschoben werden. Ein Ergebnis zeigt Bild 77 rechts.

Das vierte Beispiel dieser Reihe stammt von Toni Costa (1935 - 2013). Bei seinem Bild *Dynamic Vision* wird der Ansatz von Alberto Biasi aufgenommen, in dessen Bild *Lot Nr. 639* auch Linien in regelmäßigem Abstand von der Quadratseite zum Mittelpunkt gezogen werden. Bei Costa nimmt allerdings die Strichdicke zu bis zum Erreichen der Mitte der Entfernung zum Zielpunkt, danach nimmt sie wieder ab. Auch hier zeigt Bild 78 links wieder das Schema, rechts eineVariante mit höherer Liniendichte (es gibt die gleiche Vorgehensweise mit Zu-und Abnahme der Liniendichte auch in einigen Bildern von Biasi).

Anregung: *In allen vier gezeigten Beispielen lohnt es sich, mit Farben und Strichstärken zu experimentieren. Die Moiré-Effekte verändern sich dadurch deutlich.*

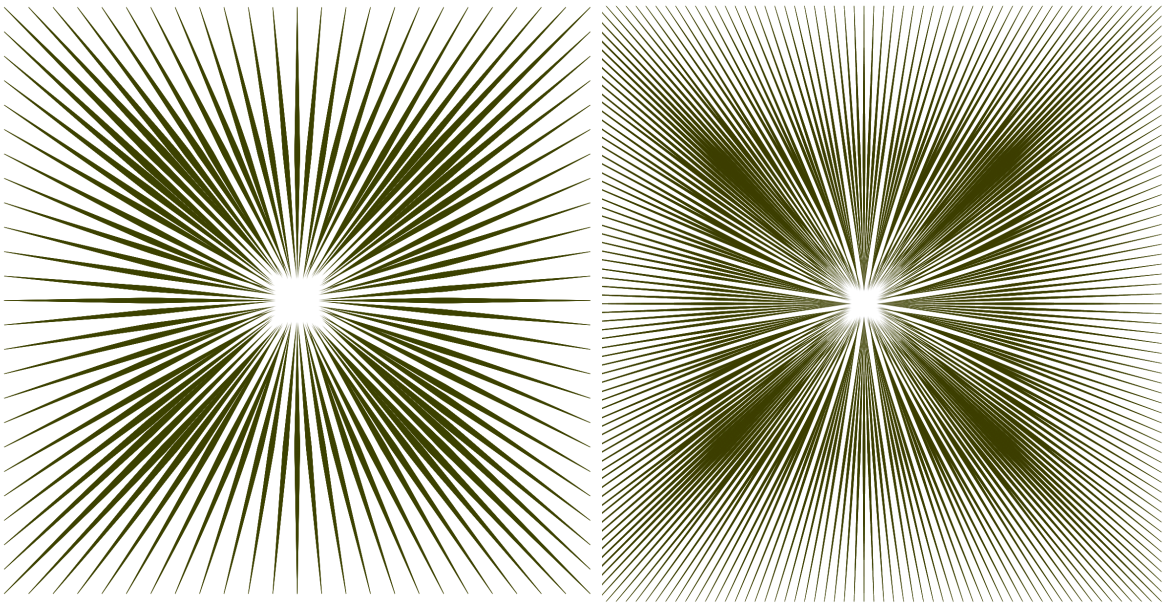


Bild 78: Hommage à Costa: Dynamic Vision

Das brauchen wir von **Snap!***:*

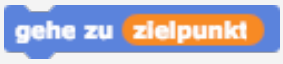


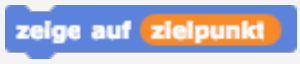
Mit **zeige auf ...** kann die Richtung der Schildkröte auf ein Ziel ausgerichtet werden. Als Ziel kann der Mauszeiger oder ein anderes Sprite dienen.

Soll ein bestimmter Punkt auf der Bühnen als Ziel definiert werden, können dessen Koordinaten in einer Liste gespeichert werden und diese Liste z.B. einer Variablen **zielpunkt** zugeordnet werden:



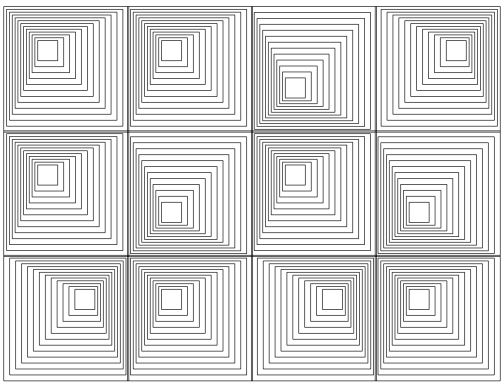
Nun kann die Schildkröte auf dieses Ziel ausgerichtet werden und dann schrittweise oder direkt zu diesem Zielpunkt geschickt werden:





22.10 Hommage à Korneder: Zentren

Beim ersten Wettbewerb für Computergrafik, 1979 veranstaltet von der Gesellschaft für Computergrafik und Computerkunst e.V., wurde eine Bildserie *Zentren* von Hans Korneder mit dem ersten Preis ausgezeichnet. Seine Arbeit entstand im Rahmen der Entwicklung des Grafiksystems SNE COMP ART (vgl. Kapitel 19: *Hommage à Schneeberger: SNE KAO*). Die Bilder dieser Serie bestehen aus einem Netz konzentrischer verschachtelter Rechtecke um einen Mittelpunkt.



Die Ausgangsform ähnelt stark sowohl der *Zerkiemung* von Ernst Schott (vgl. den Abschnitt *Hommage à Schott: Zerkiemung* in diesem Kapitel) als auch der *Collection of Random Windows* (vgl. den Abschnitt *Hommage à CTG: Random Windows* in diesem Kapitel). Für die programmtechnische Umsetzung könnten die dortigen Programme nahezu unverändert übernommen werden. Allerdings zeigen die Bilder der Serie *Zentren* nicht nur Verschiebungen der Mittelpunkte innerhalb rechtwinkliger Vierecke, sondern zusätzlich die zufallsgesteuerte Verschiebung der Eckpunkte in horizontaler und vertikaler Richtung (dies wiederum ähnlich wie bei *Gestörtes Gewebe*; vgl. Kapitel 15: *Hommage à Nees: Die Sprache G*).

Hinweis: *Es sind mehrere Wege denkbar, die inneren Lininen zu berechnen und zu zeichnen. Meine Lösung sieht vor, zunächst die Eckpunkte aller Vierecke in einer Liste zu speichern. Daraus wird für jedes Viereck im Raster der Mittelpunkt ermittelt und daran ausgerichtet die inneren Linien errechnet und gezeichnet.*



Bild 79: Hommage à Korneder: Zentren

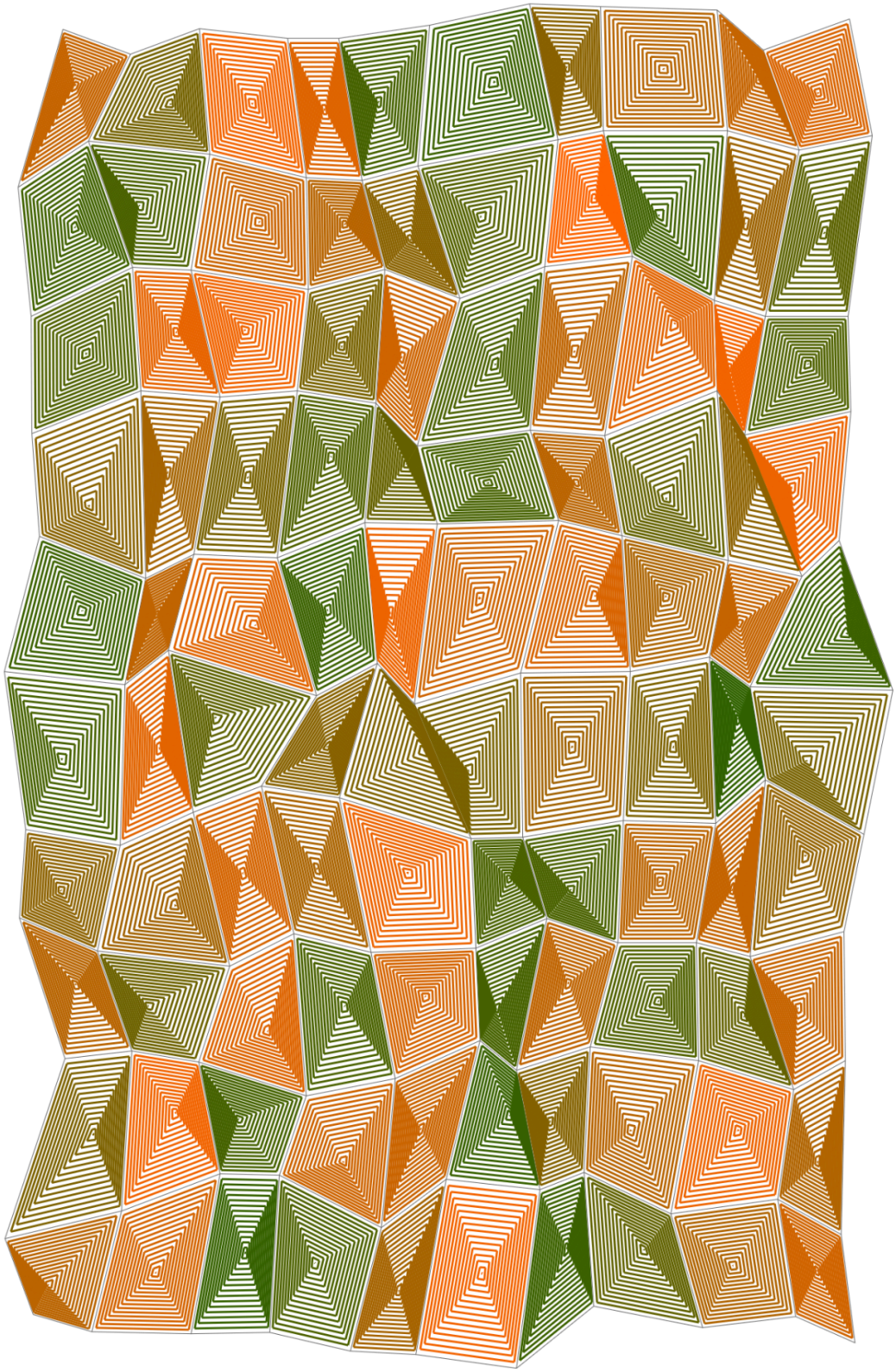


Bild 80: Hommage à Korneder: Remixing Zentren

22.11 Hommage à Land & Cohen: Flowers

Wie bereits mehrfach erwähnt, war anfangs die Ausgabe farbiger Computergrafiken auf Papier die Ausnahme. Auch elektronische Datensichtgeräte waren meist monochrom. Allerdings konnten bei diesen durch Farbfilter Farbeffekte hervorgerufen werden. Das folgende Beispiel geht zurück auf das Bild *Flowers* von Richard I. Land und Dan Cohen (in Franke, 1971, S. 45).

Das *Recoding* ist äußerst simpel, reicht es doch, auf der Bühne zufällig verteilt „Blumen“ zu zeichnen (vgl. z.B. das ganz ähnliche Grundelement *Stern* im Klecksbild, Kapitel 8: *Alles Zufall* ...). Das große Bild (oben links) ähnelt mit farbigen Blüten auf schwarzem Hintergrund stark der Vorlage von Land & Cohen.

Wenn wie üblich alle Kenngrößen (Zahl der Blumen, Zahl und Länge der Blütenblätter, Änderung der Blattlänge im Kreis, Strichdicke, Blüten- bzw. Blattfarbe und Hintergrund) variabel gehalten werden, lassen sich davon leicht viele ansprechende Varianten erzeugen.

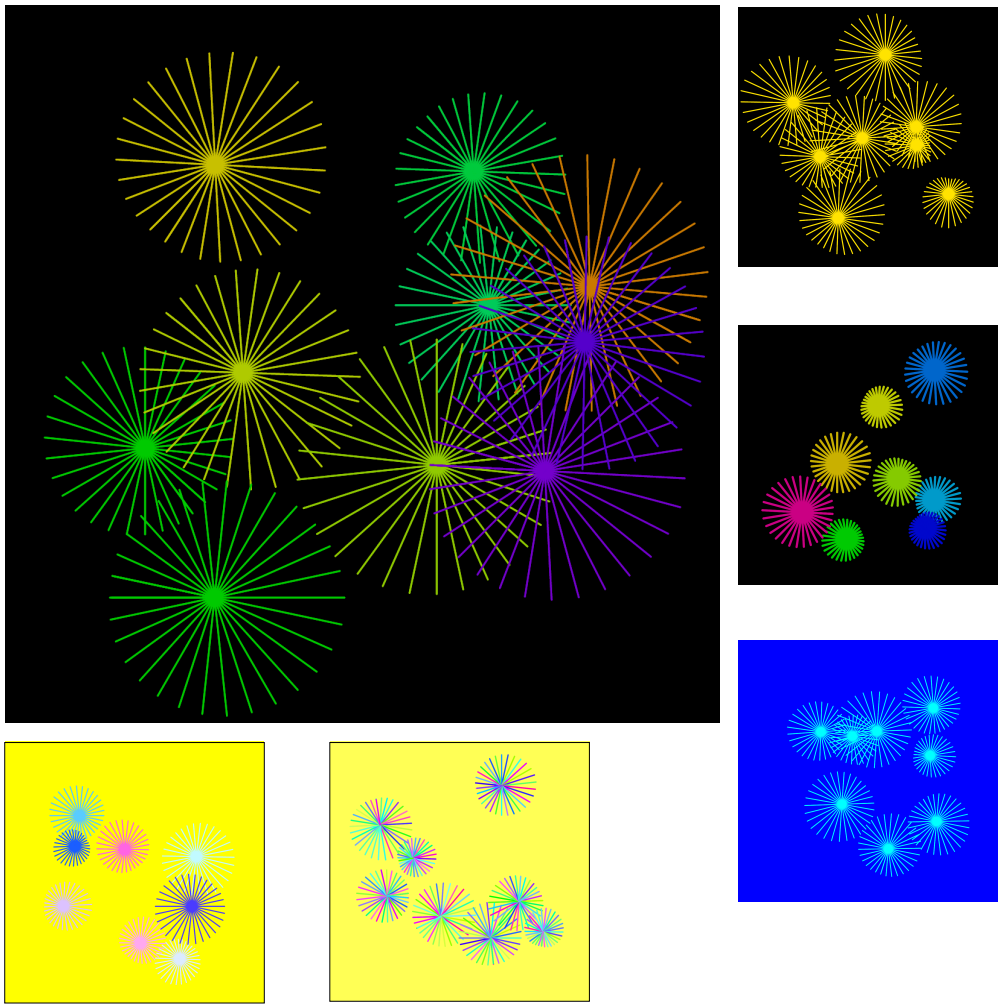


Bild 81: Recoding & Remixing Land Cohen: Flowers

22.12 Hommage à Nash: Triangle 9

Bei der ASCII-Art werden Bilder mit dem Zeichensatz von Schreibmaschinen bzw. Druckern erzeugt. Der dafür verwendete ASCII-Zeichensatz ist weltweit genormt. Besonders geeignet dafür sind nichtproportionale Schriften, d.h. solche, bei denen alle Buchstaben gleich breit sind (etwa Courier, die ursprünglich für elektrische Schreibmaschinen entwickelt wurde). Auch in der Computerkunst wurden Drucker verwendet. Als Beispiel dient das Bild *Triangle 9* (im Original im Querformat; in Franke, 1971, S. 25) von Katherine Nash (1910 - 1982).

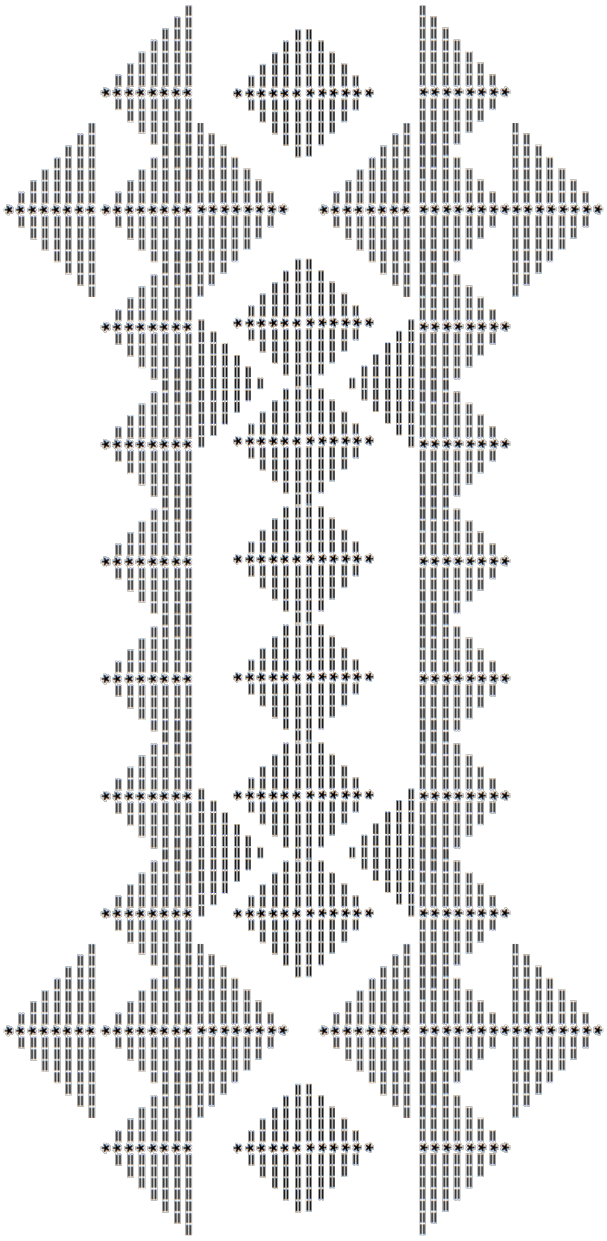
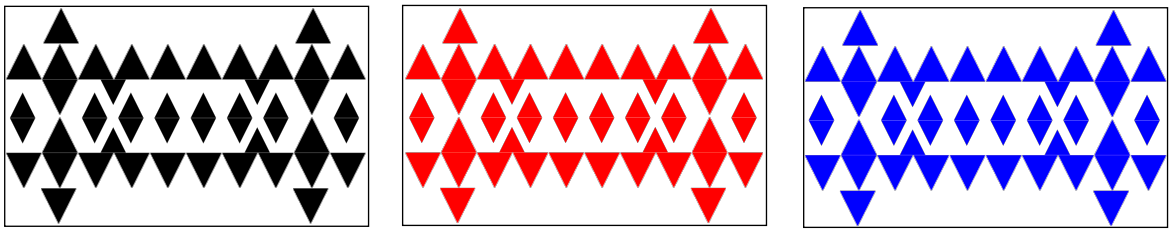


Bild 82: Hommage à Nash: Triangle 9

Dieses und vergleichbare Werke könnten natürlich „stilecht“ mit elektrischen Schreibmaschinen oder Zeilendruckern nachvollzogen werden. Da es hier aber vor allem um die grafischen Strukturen geht, habe ich einen anderen Weg gewählt. Die grafischen Grundelemente, also Dreiecke unterschiedlicher Höhe (acht bzw. sechs Zeilen), die aus dem „=“-Zeichen gebildet werden - jeweils auch mit einer Achse aus dem „*“-Zeichen - sind mit einem externen Texteditor erstellt und dann als Kostüme in Snap! importiert worden. Die Positionierung und Ausrichtung wird für jede „Grafikzeile“ angegeben und dann das entsprechende Kostüm mit dem Befehl **stemple** gesetzt.

Die Konzentration auf die grafische Struktur hat den Vorteil, daraus leicht einen *Remix* mit anderen Grafikelementen zu erhalten. Im Beispiel unten sind die ASCII-Dreiecke durch dreieckige Kostüme ersetzt (erstellt mit dem Block **dreieck** und dann mit **stift-spuren** als Kostüm gespeichert), deren unterschiedliche Größe jeweils über **setze Größe auf** festgelegt werden kann.



Ein einmal definiertes Sprite (z.B. für die Farbe Schwarz) mit den entsprechenden Positionsdaten kann einfach kopiert werden. In der Kopie ist dann die gewünschte Farbe festzulegen. Interaktiv wird dieses kleine Projekt, wenn die Farbvariante durch Tastendruck (also z.B. **s** für Schwarz, **r** für Rot usw.) gewählt werden kann.

22.13 Hommage à Roubaud: Kreissegmente

Die Firma Messerschmidt-Bölkow-Blohm unterstützte ein Projekt MBB Computer Graphics (1971 - 1972), um das Potenzial des Computers für die Kunst auszuloten. Sylvia Roubaud (geb. 1941 in München) war die einzige Malerin in der Gruppe. Sie widmete sich während dieser kurzen experimentellen Phase ihres Schaffens der Computerkunst.

Ihre Arbeiten waren bestimmt durch Prinzipien, die wir in den meisten unserer Beispiele ebenfalls vorgefunden und angewendet haben: Roubauds Arbeiten (die bereits im Abschnitt *Linien und Strecken* des Kapitels 9: *Figurenbaukasten* angesprochen wurden) nutzen die Möglichkeit, algorithmisch Ordnungen zu schaffen - nämlich durch Wiederholungen - um diese wieder gezielt zu durchbrechen - nämlich durch die zufällige Abwandlung von Grundformen - wodurch komplexere Strukturen und Muster entstehen.

Das Beispiel *Kreissegmente* (in Franke, 1971, S. 35) ist typisch dafür: Kreissegmente werden aneinander gezeichnet, wobei die Ausrichtung und der Winkel zufällig variiert werden. In Bild 83 links sind Kreissegmente bis zu 270 Grad erlaubt (dieses *Recoding* orien-

tiert sich am Vorbild von Roubaud), in Bild 83 Mitte sind sie auf maximal 180 Grad, in Bild 83 rechts auf maximal 90 Grad eingeschränkt. Diese Einschränkung führt zur Ausbildung deutlich erkennbarer Strukturen.

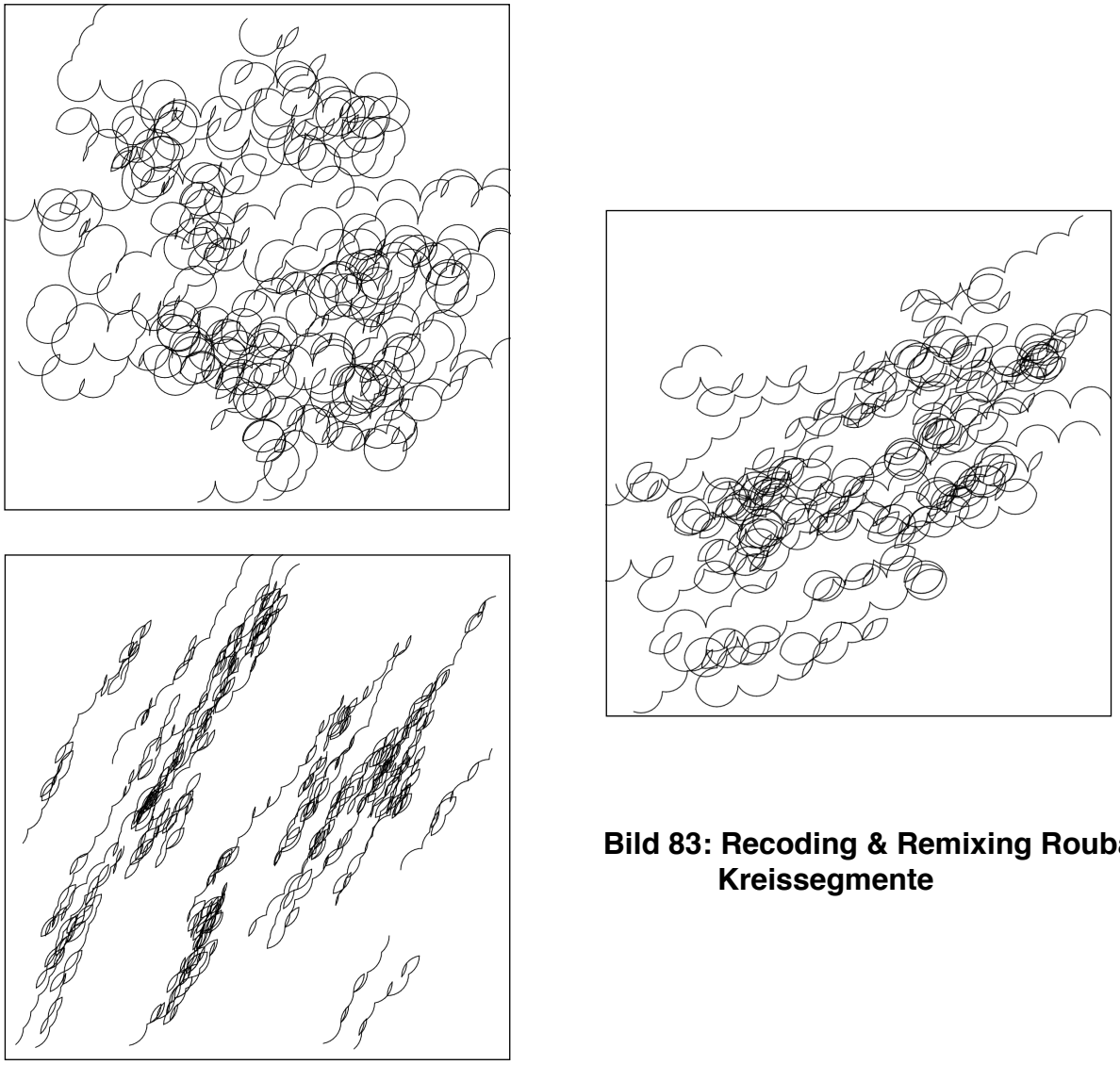


Bild 83: Recoding & Remixing Roubaud: Kreissegmente

Bei dem Bild 84 *Connection of points by arc sequences* ist das Vorgehen umgekehrt. Zu Beginn wird eine regelmäßige Struktur erzeugt (in der folgenden Abbildung oben) mit gleichmäßigen Radien der Kreissegmente. Beim *Recoding* mit Variation der Radien innerhalb vorgebener Grenzen entsteht die Abbildung unten.

Die programmtechnische Umsetzung ist insofern recht einfach, als auf die Prozedur **kreisbogen** und weitgehend auf das Vorgehen beim *Linien Malen* im Kapitel 20: *Hommage à Sykora* zurück gegriffen werden kann. Es sind nur minimale Anpassungen notwendig, damit die gleichmäßige Anordnung der Kreisbögen in untereinander folgenden Reihen erreicht wird.

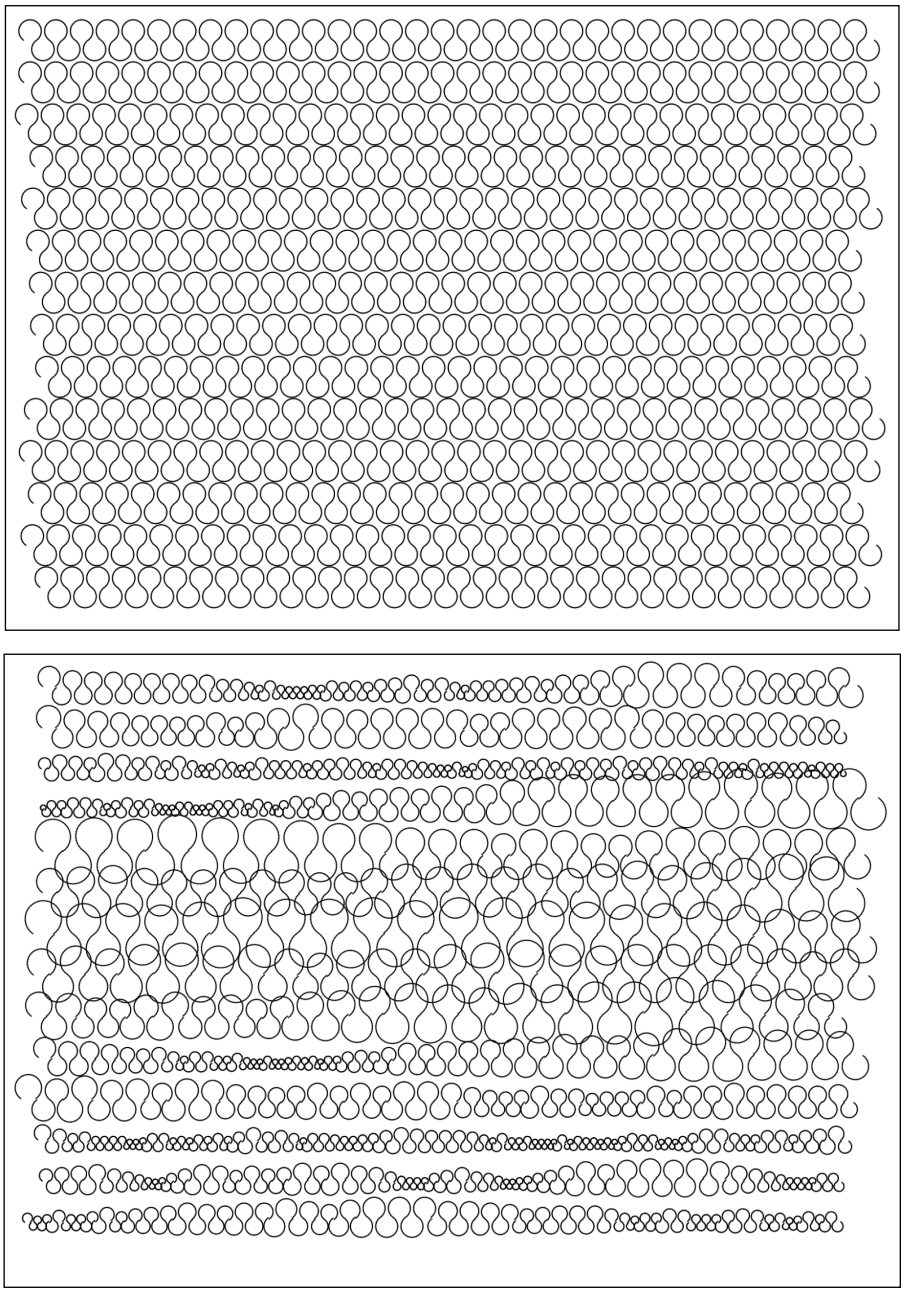


**Bild 84: Recoding & Remixing Roubaud:
Connection of points by arc sequences**

22.14 Hommage à Schott: Zerkiemung

Der Mathematiker und Computergrafiker Ernst Schott (geb. 1942 in Walpershofen) befasst sich intensiv mit Ordnungsstrukturen und Bildern im Stil der Op Art, die er mit Zufallsgeneratoren moduliert. Etliche seiner Bilder wurden von seiner Frau, der Grafikerin Milada Schott, handkoloriert. Das hier gezeigte *Recoding* bezieht sich auf sein Werk *Zerkiemung*, bei dem er eine solche Variation in einer Bilderserie dokumentiert hat (Herzogenrath & Nierhoff-Wielk, 2007, S. 282).

Das Ausgangsbild (in Bild 85 oben links) sind jeweils konzentrische Kreise (hier mit n = 12) in einer 6*6-Matrix. Der Kreismittelpunkt kann nun zufällig horizontal (oben rechts) oder vertikal (unten links) sowie in beiden Richtungen (unten rechts) verschoben werden. Bei entsprechend großen Werten kann es dabei zu Überlappungen kommen (wie in der Abbildung übernächste Seite oben).

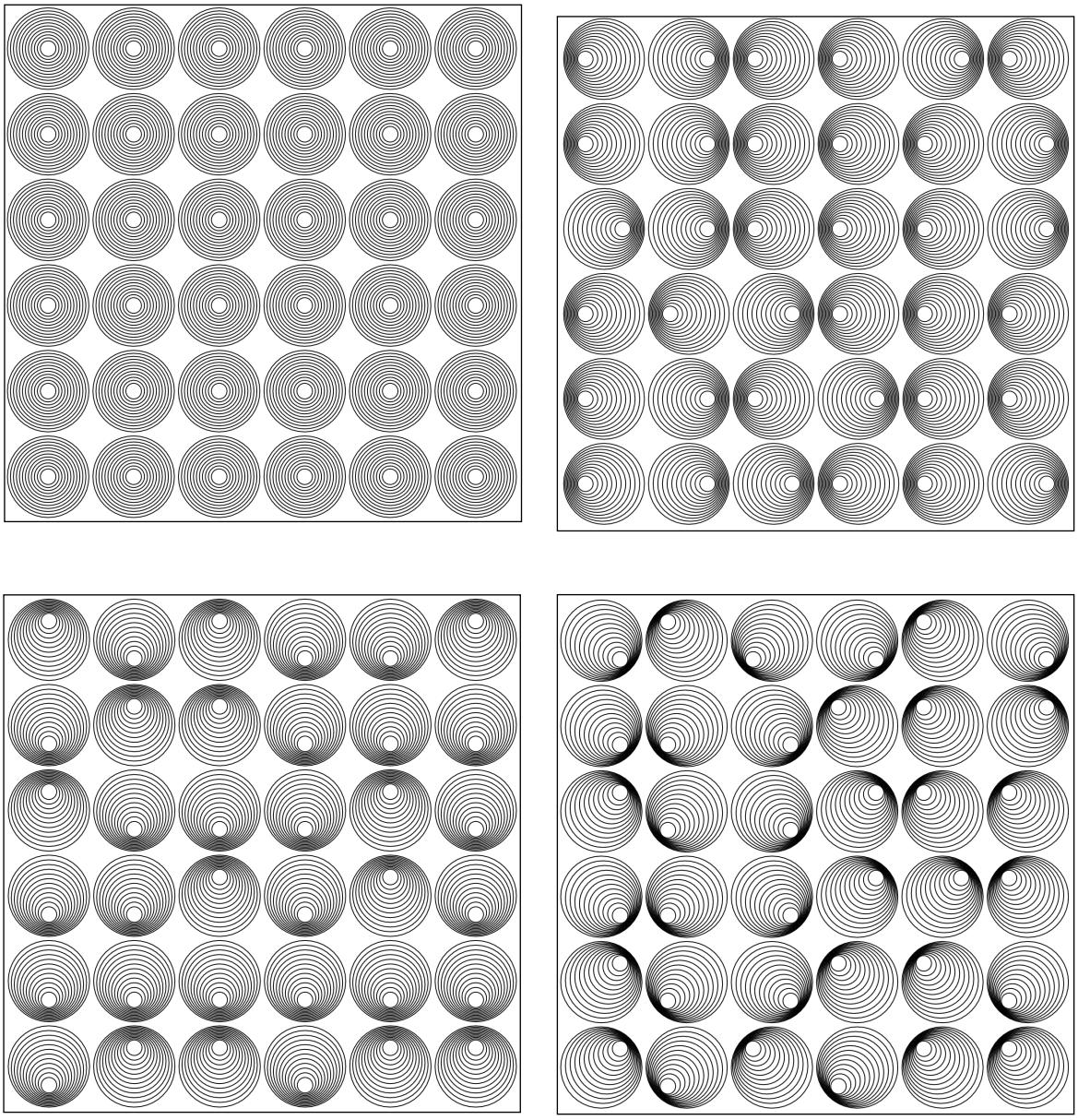


Bild 85: Hommage à Schott: Zerkiemung

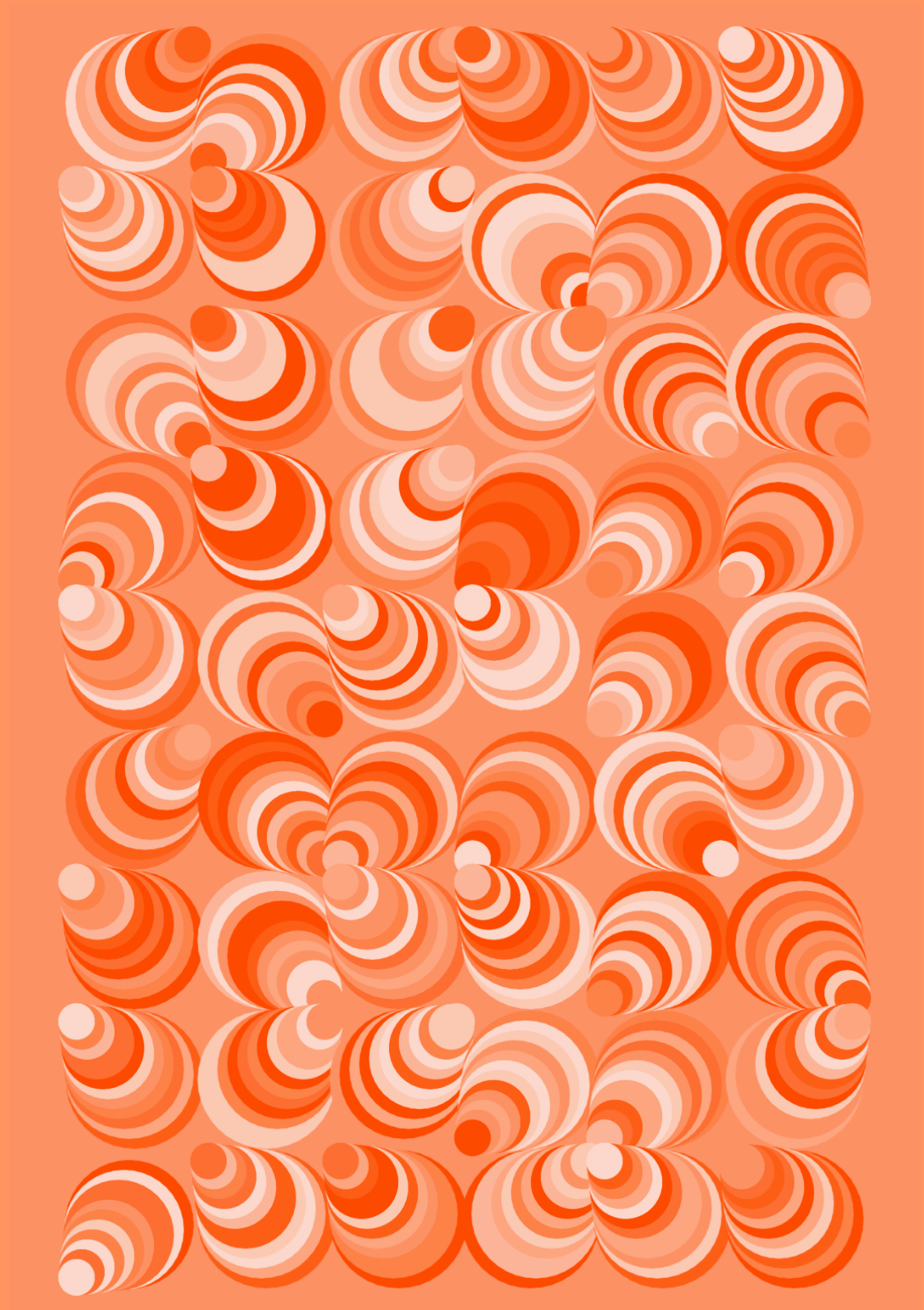
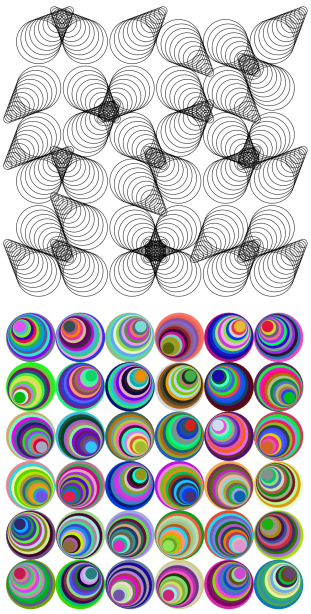


Bild 86: Remixing Schott Zerkiemung

Beim *Recoding* können auch farbige Kegel leicht durch die Wahl entsprechender Strichstärken erzeugt werden (Abbildung rechts unten).



Das *Recoding* in Bild 86 kombiniert Farben und Überlappung mit der Einfärbung des Hintergrundes mit der letzten Kegelfarbe.

Für die programmtechnische Umsetzung kann übrigens das Programm *Collection of Random Windows* der *Gruppe CTG* nahezu unverändert übernommen werden. Statt eines Vierecks sind hier Vielecke zu zeichnen (z.B. n = 30). Die Verschiebung der Mittelpunkte erfolgt zufällig, aber dann immer in der einmal festgelegten Richtung.

22.15 Hommage à Sonderegger: Schmetterling

Bei Franke (1971, S. 16) findet sich ein Bild von Bruno Sonderegger, das den Namen *Schmetterling* erhalten hat. Die Figur wurde ursprünglich als Testbild für eine Schweizer Zeichenanlage Coragraph entwickelt. Diese Entwicklung infolge technischer Entwicklungen und deren Test ist durchaus typisch für zahlreiche Beispiele der frühen Computerkunst.

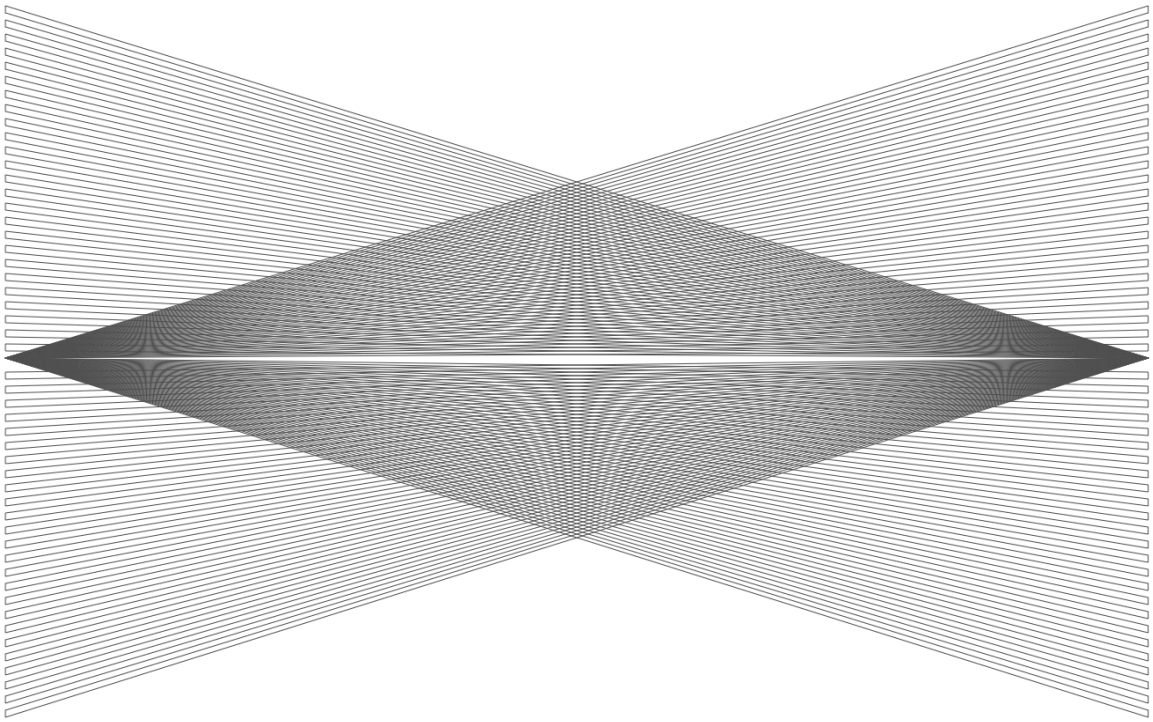


Bild 87: Hommage à Sonderegger: Schmetterling

Das Bild besteht aus vier gleichartigen Linienmustern (von rechts nach links bzw. umgekehrt und nach oben bzw. unten), den „Flügeln“. Entsprechend können alle Flügelteile jeweils mit der gleichen Prozedur gezeichnet werden, wenn diese mit Ausgangspunkt und Richtung der Strahlen versehen wird.

Die Überlagerung der Strahlenbündel erzeugen einen *Moiréeffekt*, auf den ich im Kapitel 23: *Moiré-Muster* noch ausführlich eingehen werde.

22.16 Hommage à Steller: Tektonik

Auf den ersten Blick scheint in der Computerkunst alles penibel festgelegt, resultiert doch die Entstehung eines Bildes aus dem Abarbeiten eines Algorithmus. Zufälligkeiten, die sich sonst in der Kunst kaum aussschließen lassen, müssen in der Computerkunst bewusst wieder eingeführt und zugelassen werden. Bei fast allen der bisherigen Projekte ist dies der Fall gewesen. Erwin Steller (1992) widmet in seinem Buch *Computer und Kunst* dem Zufall deshalb auch ein eigenes, längeres Kapitel. Bei seinen systema-



Bild 88: Hommage à Steller: Tektonik

tischen Untersuchungen des Zufalls entstehen gleichzeitig Grafiken mit eigenständigem Charakter.

Steller geht aus von Histogrammen, wie wir sie aus Geschäftsgrafiken kennen. In seiner Bildserie verwendet er keine statistischen Daten, sondern es können zufallsbedingt die Länge, Breite, Abstand und Farbe der Balken eines Histogramms festgelegt werden. Die Kontrolle besteht in der Bestimmung der Ober- und Untergrenzen für diese Werte. Die Grafiken werden dadurch bei ihm *„ästhetisch überhöht“*.

Für das Beispiel ist bewusst der Abstand der Stäbe kleiner als ihre Dicke gewählt. Durch die dabei möglichen Überlappungen ergibt sich ein ganz anderer Eindruck als bei herkömmlichen Histogrammen, den Steller (a.a.O., S. 166) als „tektonisch“ interpretiert.

22.17 Hommage à Strand u.a.: Drehflächen

Es ist ganz typisch, dass sich bei verschiedenen Vertretern der frühen Computerkunst zum Teil sehr ähnliche Bilder finden. Im Kapitel 1: *Computerkunst* wurden solche Beispiele schon vorgestellt. Einige Vertreter haben sich besonders mit der Konstruktion symmetrisch aufgebauter Grafiken befasst, bei denen der Zufall ausgeschlossen ist. Diese Bilder gewinnen ihre Vielfalt durch die systematische Variation der Kenngrößen.

Den folgenden Bildern, die viele ineinander geschachtelte Dreiecke bzw. Vierecke enthalten, liegen sogenannte Verfolgungskurven zugrunde. Solche Spuren zieht nämlich ein Verfolger nach sich, wenn er einem Zielobjekt nachstellt, wenn also z.B. ein Hund seinem Herrchen folgt. Wichtig ist für die Bildentstehung, dass sich das Zielobjekt und der Verfolger jeweils mit konstanter Geschwindigkeit bewegen. Der Verfolger bewegt

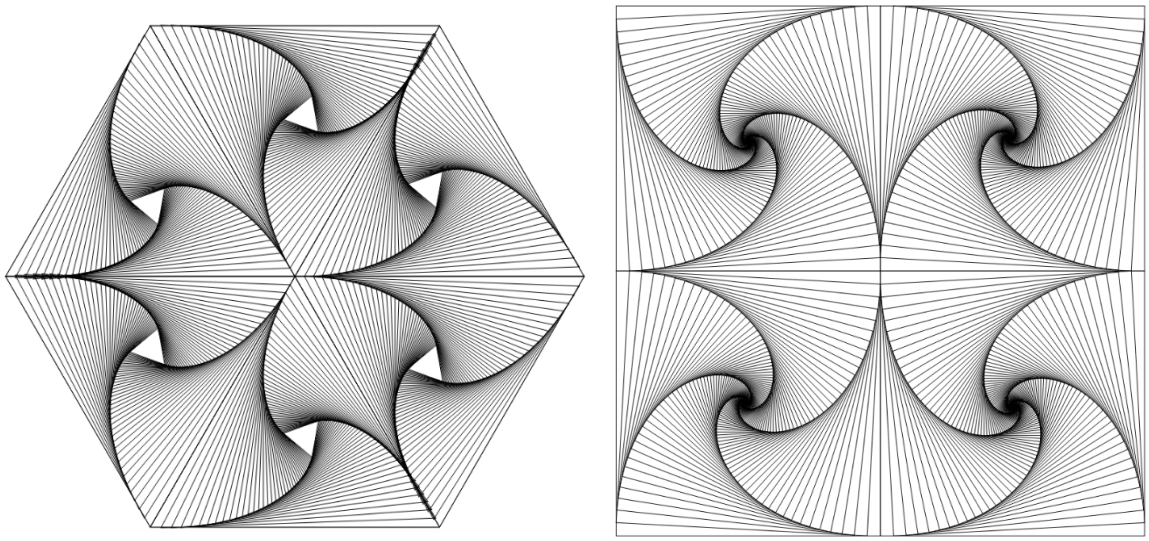


Bild 89: Hommage à Strand: ohne Titel (links);
Hommage à Böttger: Drehflächen III (rechts)

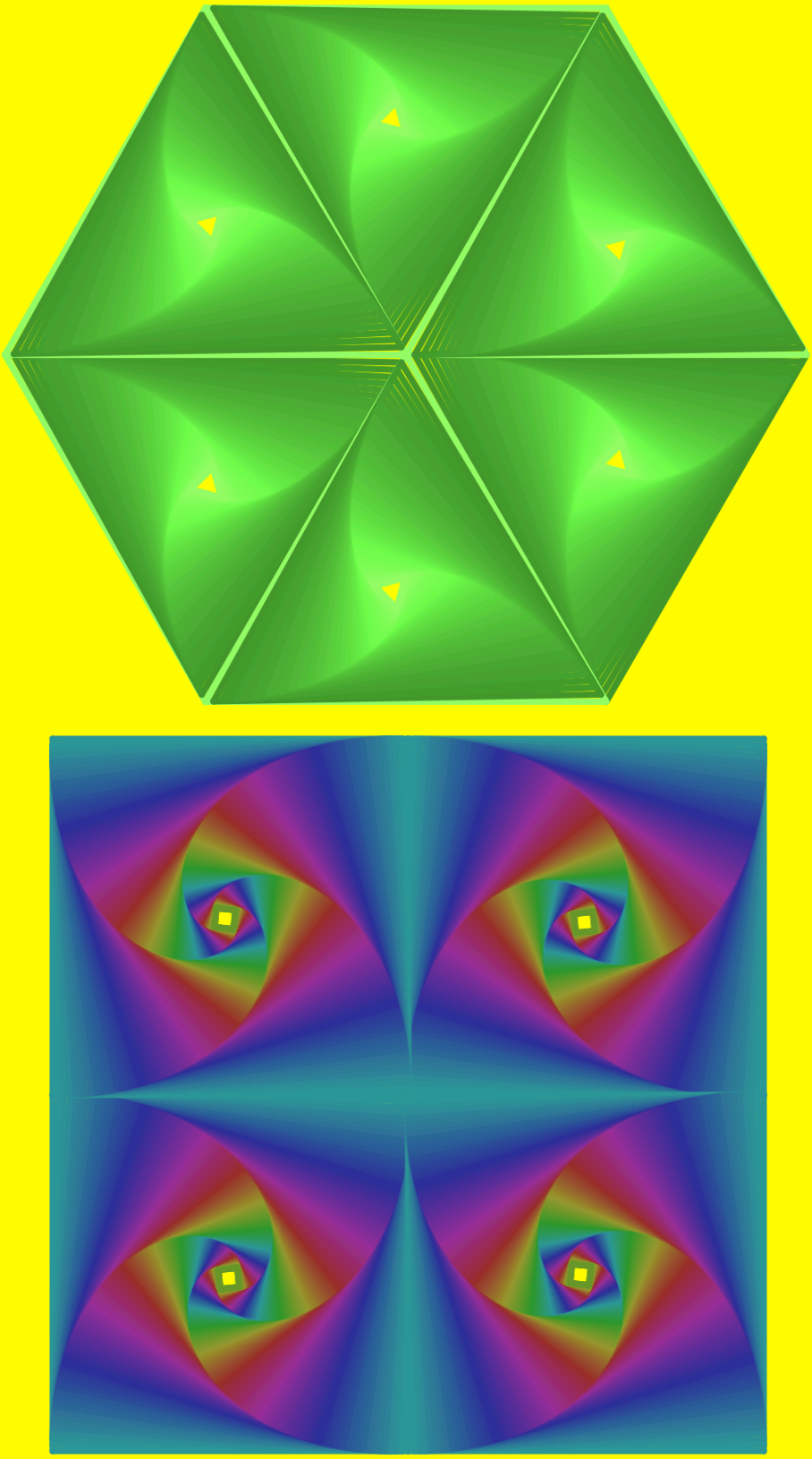


Bild 90: Remixing Strand (oben) & Böttger (unten)

sich also zum aktuellen Standort des Zielobjekts. Erreicht er diesen Punkt, hat sich das Zielobjekt ja bereits weiter bewegt und der Annäherungsvorgang wiederholt sich.

Das Bild 89 (links) orientiert sich an einem Vorbild *ohne Titel* von Kerry Strand (geb. 1940 in Emida, USA), der für einen Plotterhersteller tätig war. Seine Arbeiten wurden deshalb auch unter dem Künstlergruppennamen *CalComp Artist Group* ausgestellt.

Das Bild 89 (rechts) geht auf das Vorbild *Drehflächen III* von Frank Böttger (geb. 1946 in Speyer) zurück. Böttger war Ingenieur bei der Firma Messerschmidt-Bölkow-Blohm und als solcher beteiligt an der Ausstellung MBB Computer Graphics (1972). Sie zeichnet sich - wie alle Arbeiten von Böttger - durch ihren symmetrischen Aufbau aus und damit, wie Böttger es ausdrückt, durch „*die Verbindung, die sich zwischen mathematischem Formalismus und künstlerischem Ausdruck ergibt: Die Änderung eines Parameters bewirkt oft Veränderungen des Gesamtbilds, die der Betrachter - und sogar der Schöpfer - des Bildes nur mit Erstaunen zur Kenntnis nehmen kann.*“

Die Verfolgungsspuren lassen sich ohne Mathematik allein dadurch erzeugen, dass die jeweiligen Punkte eines Annäherungsschrittes errechnet und verbunden werden. Die gezeigten Beispiele sind noch etwas komplizierter, weil sie aus sechs solcher Dreiecke (links) bzw. vier solcher Quadrate (rechts) aneinandergesetzt werden.

Anregung: *Für das* Remixing *bietet sich die Ausgestaltung durch Farben und unterschiedliche Strichdicken an, wie in* Bild 90 *exemplarisch gezeigt.*

22.18 Hommage à von Graevenitz: Große horizontale Verteilung

Gerhard von Graevenitz (1934 - 1983) war ein bildender Künstler, der sich nach Arbeiten an monochromen Reliefs und kinetischen Objekten ab 1962 auch der Computergrafik zuwandte. Er beteiligte sich u.a. an den Gruppen Zero und Groupe de Recherche d’Art Visuel (GRAV) sowie der Bewegung der Nouvelle Tendence. Seinen Serien geometrisch geordneter Rasterstrukturen ist anzumerken, dass er Phänomenen der Wahrnehmung nachspürte. Die folgenden Beispiele sind daraus entnommen.

Die Umsetzung erfolgt in beiden Fällen, indem in Doppelschleifen das Kostüm einer Mondphase unterschiedlicher Größe (oben) bzw. Ausrichtung (unten) mit **stemple** gesetzt wird.

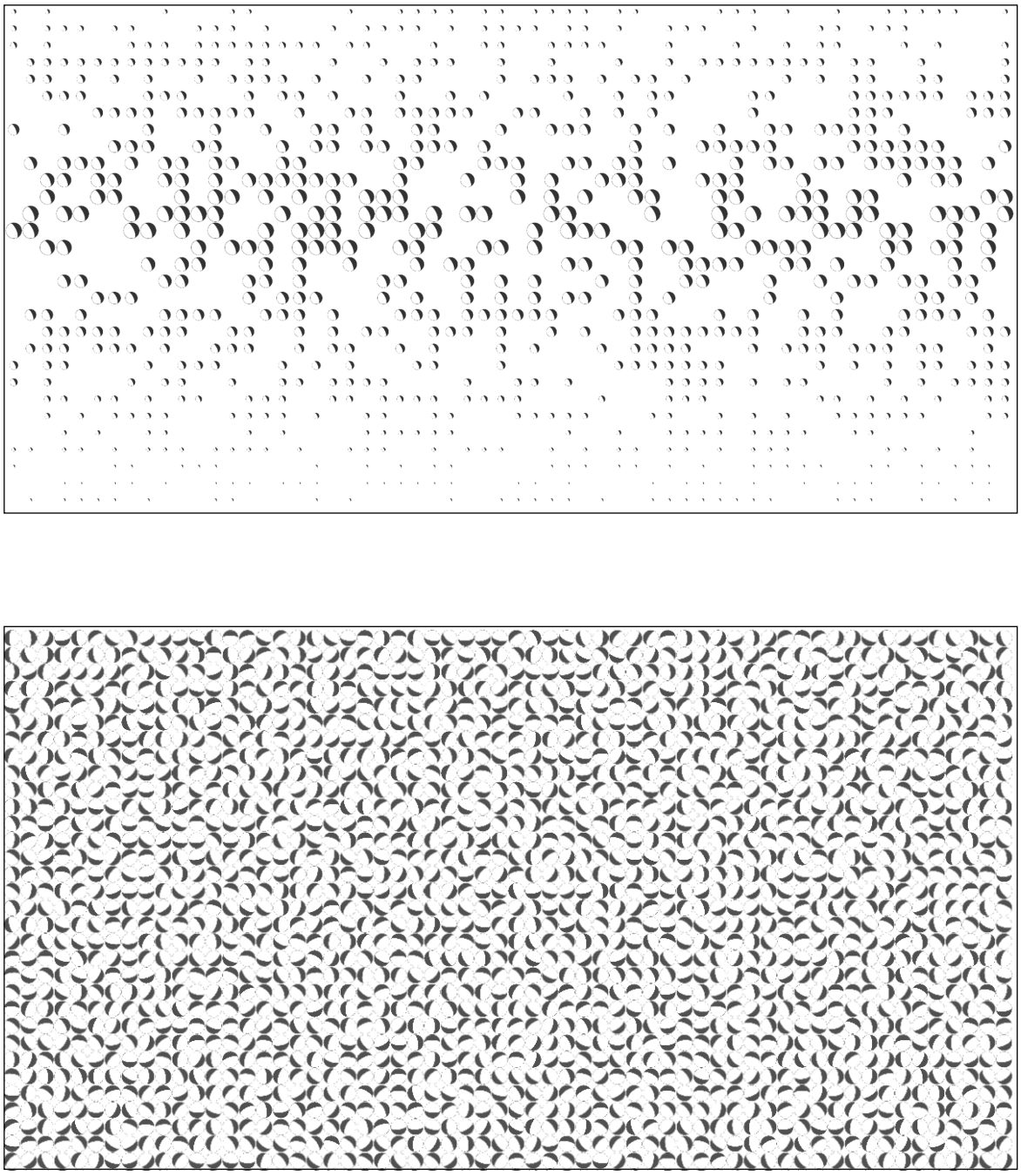


Bild 91: Hommage à von Graevenitz: Große horizontale Verteilung (oben); Homogene Struktur I (unten)

23. MOIRÉ-MUSTER

Mit musterähnlichen Wiederholungen geometrisch-abstrakter Motive (Punkte, Streifen, Linien, Rechtecke, Kreise, Spiralen) lassen sich überraschende optische Wirkungen erzeugen, wenn sie in verschiedenen Ebenen gegeneinander verschoben werden. Die Erzeugung solcher Effekte ist oft unerwartet einfach. In der Op-Art wurde deshalb viel mit geometrischer Abstraktion und optischen Illusionen gearbeitet (z.B. bei Vasarély und Riley) und die Effekte werden bis heute bildnerisch eingesetzt (etwa von Anoka Faruqee, Liz Deschenes, Reginald Neal u.a.). Auch bei einigen Vertetern der frühen Computerkunst sind Beispiele für den Einsatz des Moiré-Effekts zu finden.

Das Phänomen der Moiré-Muster ist außerdem ein Gegenstand der Forschung, da die Muster in ganz verschiedenen technischen Bereichen von Bedeutung sind. Ein Grundlagenwerk dazu stammt von Isaac Amidror, der zusätzlich Demonstrationen der Effekte online zugänglich gemacht hat. Mit dem Buch moiré index bewegt sich Carsten Nicolai an dieser Schnittstelle von Wissenschaft und Kunst, weil er darin systematische Transformationen von Grundelementen wie Punkten und Linien in Gittern vornimmt. Neben dem dabei entstandenen visuellen Kompendium entwickeln die erzeugten Muster eine eigenständige ästhetische Qualität.

In diesem Kapitel soll das Grundprinzip zur Erstellung von Moiré-Mustern an wenigen Beispielen demonstriert werden. Mit deren *Remixing* kann dann eine große Palette verschiedenster Muster und Phänomene erzeugt werden[69]. Für die Darstellung der Ergebnisse eignen sich bereits statische Bilder; besonderen Reiz gewinnen sie jedoch für die Betrachter, wenn sie als programmgesteuerte Animation ablaufen oder im Idealfall selbst interaktiv (durch händische Verschiebung der Muster) beeinflusst werden können.

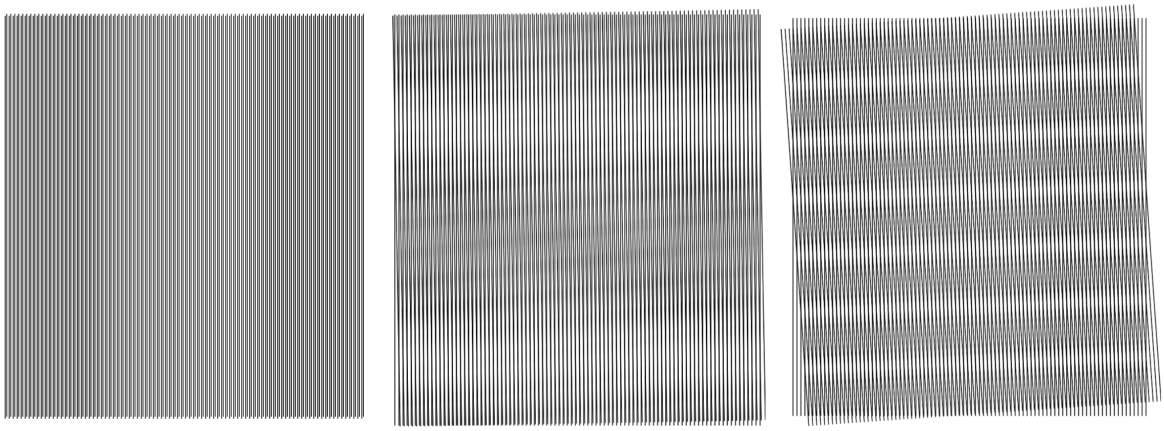
Das Vorgehen ist durchgängig dasselbe (auf den Programmcode wird an dieser Stelle verzichtet; die programmtechnische Umsetzung sollte aber mit dem erarbeiteten Kenntnisstand wenig Mühe bereiten):

1. Es wird das erste gewünschte Grundmuster erzeugt und mit dem Block **erstelle kostüm aus stiftspuren** als Kostüm abgespeichert (es kann natürlich auch extern mit einem Grafikprogramm erzeugt und dann importiert werden).
2. Je nach angestrebtem Bild und Effekt können eines oder mehrere weitere Muster erzeugt/importiert und hinzugefügt werden.
3. Das erste so entstandene Kostüm ist dann mit dem Block **stemple** zu platzieren.
4. Dasselbe (oder ein weiteres) Kostüm wird gewählt und mit **anzeigen** darüber oder bewusst leicht versetzt platziert.

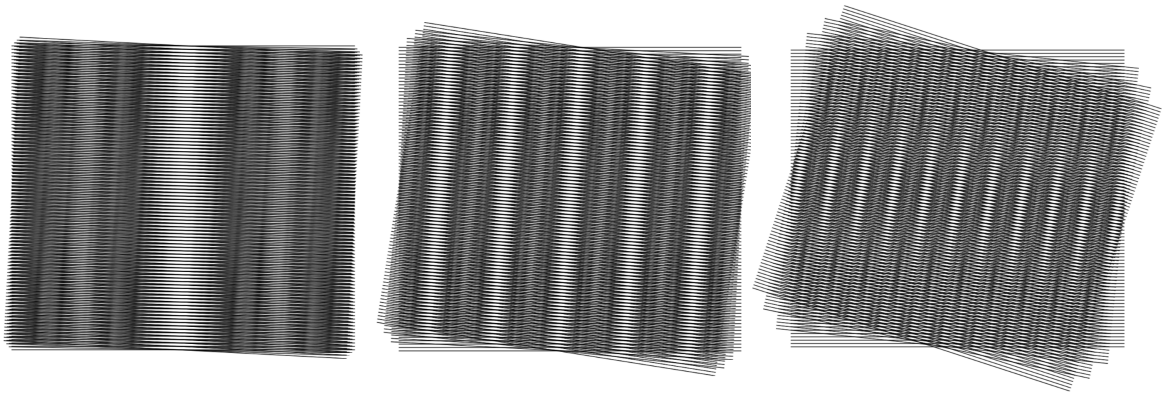
[69] Wer Lust verspürt, kann auf diese Weise so ziemlich das ganze Buch *moire index* recoden, denn dort wird genau nach dem Prinzip „Verwendung weniger Grundmuster und deren systematischer Variation“ vorgegangen.

5 Dieses zweite Kostüme wird außerdem über **zeige richtung ...** leicht verdreht.

In der folgenden Abbildung sind zwei identische Kostüme (aus je 90 parallelen Linien) leicht gegeneinander versetzt (vier Punkte in x-, fünf Punkte in y-Richtung) übereinander gelegt. Das Grundmuster ohne Verdrehung zeigt die Abbildung links. In der Mitte ist das zweite Kostüm um ein Grad verdreht. In der Abbildung rechts ist diese Verdrehung auf vier Grad erhöht.



Natürlich können mehr als zwei Muster überlagert werden. So sind in der folgenden Abbildung vier identische Kostüme (dieses Mal aus je 85 parallelen Linien) direkt übereinander gelegt. In der Abbildung links sind die drei über das Grundmuster gelagerten Kostüme um ein, zwei bzw. drei Grad verdreht. In der Mitte betragen die Verdrehungen drei, sechs und neun Grad. In der Abbildung rechts ist die Verdrehung auf sechs, zwölf und sechtzehn Grad erhöht.



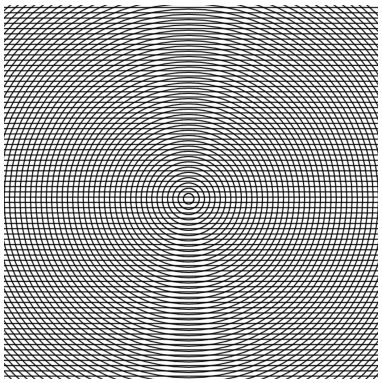
Es ist ein Leichtes, über die schlichte Wiederholung eines grafischen Elements - statt der Linie etwa Punkte, Kreise, Rechtecke usw. - weitere Grundmuster zu erzeugen. Diese können dann mehrfach identisch oder in Kombination miteinander übereinander gelegt werden. Mit allen sind dann mehr oder weniger deutliche Moiré-Effekte darstellbar.

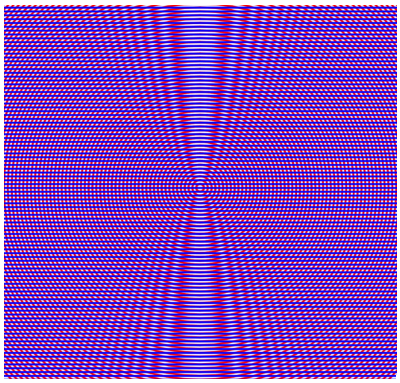
Bei der Kombination unterschiedlicher Grundmuster, etwa Punkte mit Linien oder Linien mit Kreisen, enstehen häufig noch interessantere Effekte. Auch diese Kombinati-

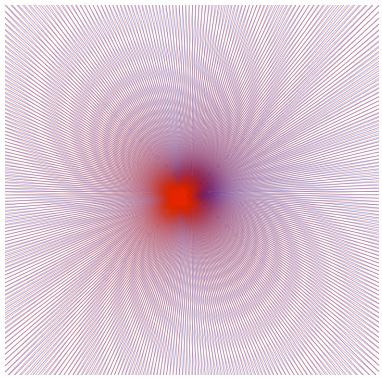
onen sind wie beschrieben leicht zu erzeugen. Einzelne Beispiele wurden bereits im letzten Kapitel vorgestellt mit *Dynamic Visions* der Gruppo N.

Hinweis: *Das Moiré-Phänomen ist sehr empfindlich gegenüber kleinsten Variationen oder Bewegungen in einer der beiden oder mehreren Schichten. Aus diesem Grund lohnt es sich, solange mit kleinen manuellen Änderungen zu arbeiten, bis interessante Effekte gefunden werden.*

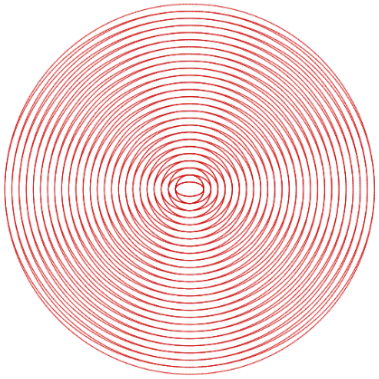
Anregung: *Neben den gezeigten Grundmustern können Sie viele weitere Elemente zugrunde legen (Anregungen finden sich u.a. in Grafton, 1976). Mit unterschiedlichen Liniendicken, Farben und Musterkombinationen ergeben sich nahezu unerschöpfliche Varianten. Oder - um hier noch einmal Georg Nees zu zitieren (Nees, 1995, S. 183): „Irgendwann wird man durch einen außergewöhnlichen Fang belohnt."*







Der niederländische Designer und Professor für Visuelle Kommunikation Ootje Oxenaar (1929 - 2017) war berühmt für seine Banknotenserien (Dutch Central Bank) und Sonderbriefmarken (PTT). Einigen der Briefmarken hat er Plottergrafiken von Moiré-Mustern zugrunde gelegt, so auch das nebenstehende Bild, entstanden durch Überlagerung gegeneinander verdrehter Ellipsen. Vorbild dafür war eine 45-Cent Sondermarke der Niederlande von 1970.



Das Bild 92 ist Ergebnis eines Moiré-Effekts überlagerter Ellipsen. Es orientiert sich an einer Vorlage von Linda Besemer und ist ein gutes Beipiel für ein farblich abgestimmtes statisches Bild, bei dem der Moiré-Effekt dezent für die Strukturierung des Gesamtbildes eingesetzt wird. In der programmtechnischen Realisierung werden die Ellipsen für den Hintergrund in einer Schleife erzeugt und mit **erstelle kostüm ...** abgespeichert. Dieses Kostüm wird mit **stemple** auf einem farbigen Hintergrund fixiert. Dabei kann über die Größe des Kostüms (mit Hilfe des Befehls **setze Größe auf ...**) der Bildcharakter beeinflusst werden. Darüber wird dann dasselbe Kostüm mit veränderter Transparenz angezeigt (mit Hilfe des Befehls **setze Durchsichtigkeit- Effekt auf ...**) - bei Bedarf ebenfalls mit einer veränderten Kostümgröße. Dieses vordere Kostüm kann nun beliebig manuell verschoben oder gedreht werden bis der Effekt in gewünschter Form auftritt.

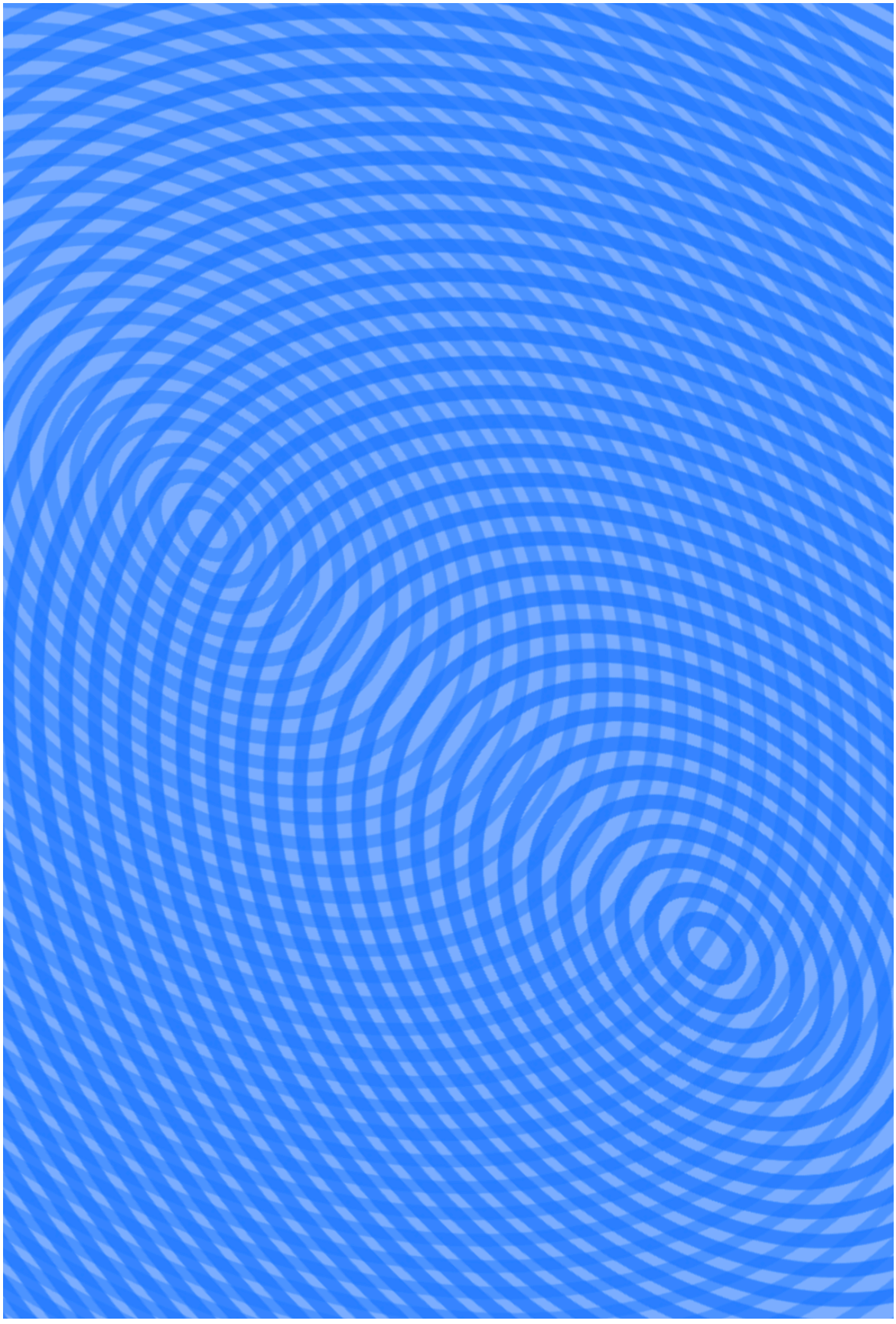


Bild 92: Hommage à Linda Besemer: Swoop Wave Bulge

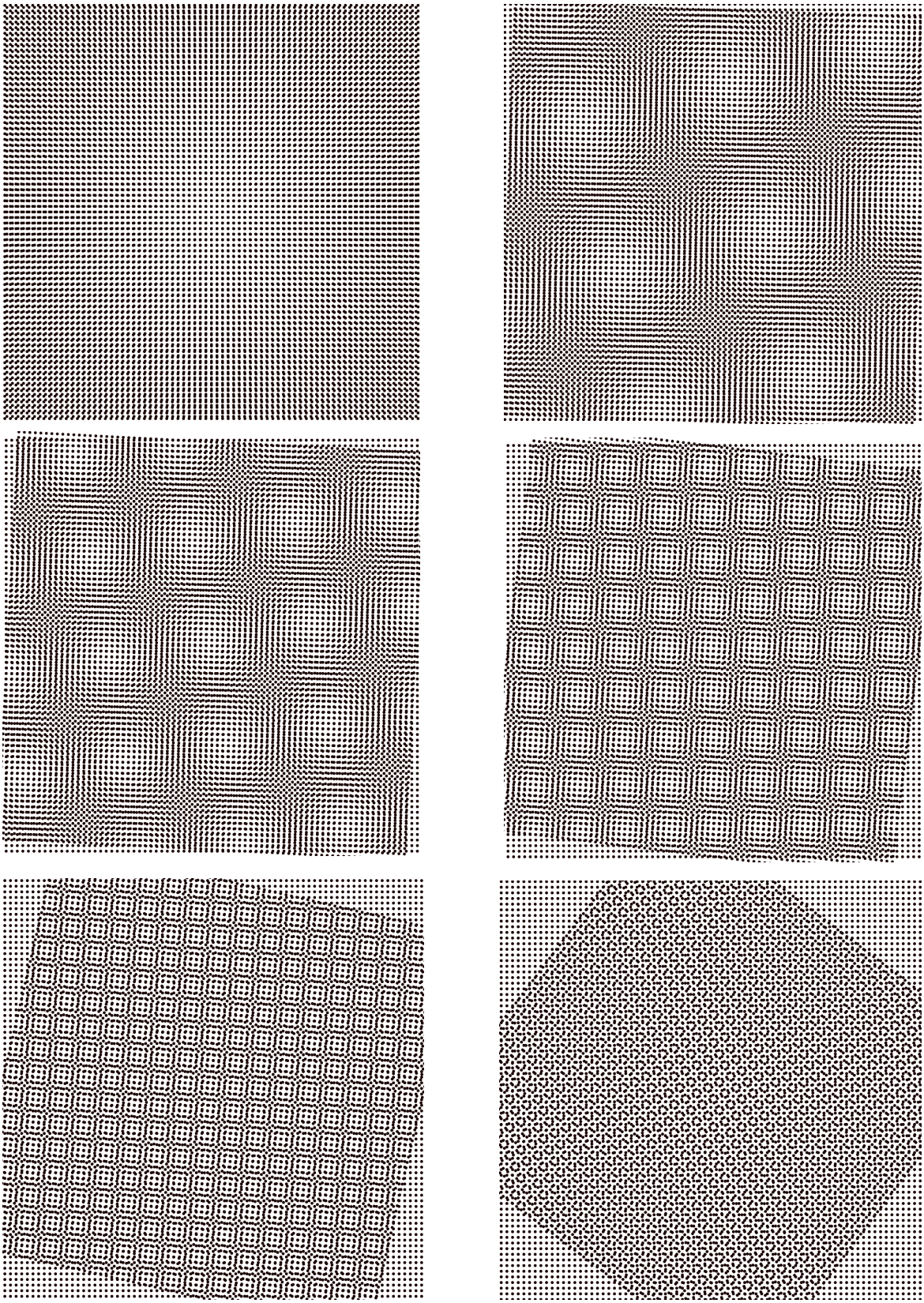


Bild 93: Hommage à Gabriele Devecchi: URMNT

Das Vorbild für Bild 93 findet sich bei dem Italiener Gabriele Devecchi (1938 - 2011), der sich 1962 an der Arte programmata. Arte kinetica. beteiligte. Bei seinem kinetischen Objekt URMNT bewegt sich eine durchlöcherte Scheibe vor einem strukturierten Hintergrund und erzeugt dadurch kontinuierlich Wellenbilder. Hier wird analog ein Punktmuster vor einem punktiertem Hintergrund bewegt.

24. KINETISCHE KUNST

Ende der 1950er-Jahre entstand mit der Kinetischen Kunst (kinetic art) eine neue Kunstrichtung. Ihr zentrales Element ist natürlich die Bewegung, egal ob dieser Eindruck durch die Bewegung des Betrachters, eine optische Illusion oder die tatsächliche Eigenbewegung des Objekts hervorgerufen wird. Häufig werden die kinetischen Kunstprodukte thermisch-mechanisch in Bewegung versetzt (als Mobiles), vielfach aber über elektromechanische Antriebe. In der Regel sind es dreidimensionale Objekte, die durch technische Konstruktionen in Bewegung gehalten werden.

Einen Höhepunkt erreichte die kinetische Kunst etwa zeitgleich mit der Computerkunst; oft werden die Objekte auch von Computern gesteuert. Der Ausgangspunkt ist wohl vergleichbar, nämlich der Versuch, Technik und Kunst miteinander zu verbinden. Bekannte Vertreter sind z.B. Alexander Calder mit seinen Großmobiles oder Jean Tinguely mit dadaistischen Maschinen. Für den Nachvollzug durch Animationen eignen sich zweidimensionale Objekte am besten. Da manche Vorbilder als Videos dokumentiert sind, empfiehlt es sich, dort die Bewegungsmuster zu studieren.

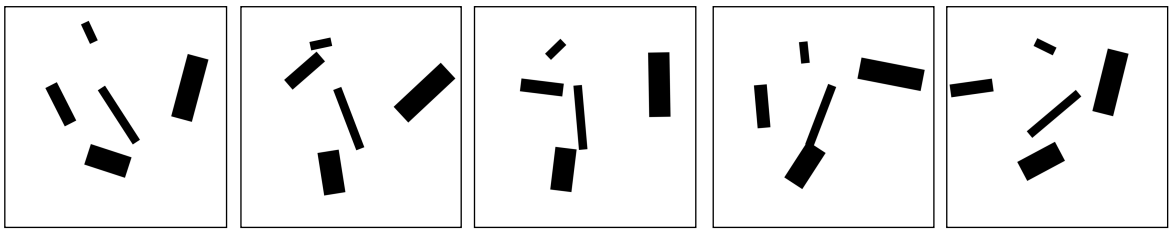
24.1 Hommage à Gerhard von Graevenitz

Neben Reliefs und Computergrafiken hat der Mitbegründer der Nouvelles Tendences Gerhard von Graevenitz (1934 - 1983) vor allem kinetische Objekte gestaltet. Seine Objekte zeigen oft eine limitierte Anzahl einfacher Elemente, wie runde oder ovale Scheiben oder einfarbige Streifen bzw. Rechtecke. Die Elemente bewegen sich langsam horizontal oder um Drehpunkte. Die sich dabei ständig ändernden Kombinationen und Bildeindrücke erschließen sich den Betrachtern meist erst beim längeren Zuschauen. Das gilt besonders für Zufallsereignisse mit Überraschungsmomenten.

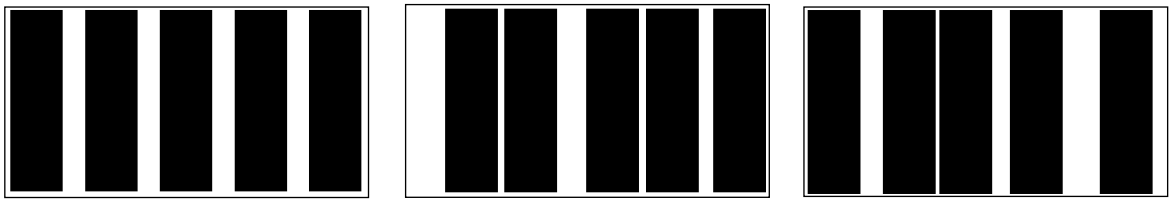
Drei solcher Objekte sollen hier Gegenstand des *Recoding & Remixing* werden. Alle drei sind programmtechnisch sehr einfach umzusetzen, so dass kurze Beschreibungen zum Ablauf ohne weitere Kenntnisse des Programmcodes genügen sollten.

Das erste Beispiel orientiert sich am Bild *Exzentrische Streifen*. Fünf schwarze Streifen unterschiedlicher Dicke und Länge befinden sich zufällig verteilt auf einer weißen Fläche. Alle Streifen drehen sich in Richtung und Geschwindigkeit unabhängig voneinander im Kreis. Eine mögliche Realisierung besteht darin, fünf Sprites zu erzeugen, in jedem Sprite einen schwarzen Streifen gewünschter Länge und Dicke zu zeichnen und diesen mit Hilfe von **stiftspuren** als Kostüm zu speichern. Das Sprite wird an der gewünschten Ausgangspostion platziert. Mit  oder einer definierten Taste können die Skripte aller Sprites gestartet werden, womit sie in einer Endlosschleife in Rotation versetzt werden. Die folgende Abbildung zeigt eine statische Abfolge sich daraus ergebender Bildeindrücke.

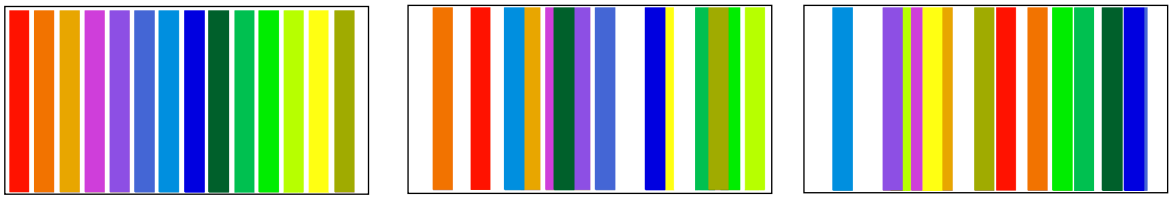
Das kinetische Objekt *5 schwarze Rechtecke auf Weiss* aus dem Jahr 1974 kenne ich nur von einem statischen Bild des Kröller-Müller Museum. Ich vermute, dass die schwar-



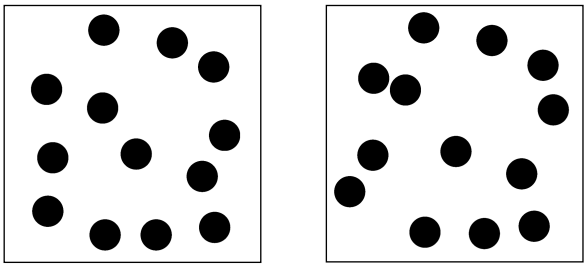
zen Rechtecke sich jeweils seitlich bewegen bis sie an ein benachbartes Rechteck anstossen und daraufhin ihre Bewegungsrichtung ändern. Die programmtechische Umsetzung ähnelt der vorigen mit den exzentrischen Streifen: In fünf Sprites werden jeweils schwarze Rechtecke gezeichnet, als Kostüm gespeichert und an der gewünschten Ausgangsposition platziert. Die Bewegungsrichtung nach rechts oder links wird zufällig ermittelt. Die Geschwindigkeit kann über eine Kenngröße **delta** gesteuert werden. Berührt ein Rechteck ein benachbartes Rechteck oder berühren die beiden äußeren Rechtecke die Bühnenkante, wird ihre Richtung umgedreht (abgefragt über **falls berühre kante ?**). Die folgende Abbildung zeigt eine statische Abfolge der sich daraus ergebenden Bildeindrücke.



Für ein farbiges *Remixing* ist ein bisschen Fleißarbeit nötig. Für die untere Abbildung werden diesmal 14 schmalere, farbige Rechtecke als Kostüme erzeugt. Deren Bewegung wird diesmal nur an den Bühnenkanten umgedreht. Dadurch gleiten die Rechtecke über (oder unter) den Nachbarn weiter. Insgesamt ergibt sich so ein abwechslungreicherer, aber auch unruhigerer Ablauf.



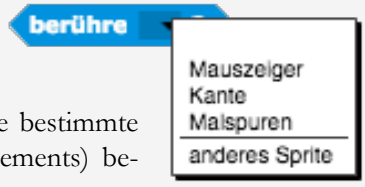
Das dritte Beispiel nach dem Vorbild von Graevenitz orientiert sich an *Schwarze Punkte auf Weiss*. Die auf einer weissen Fläche verteilten Punkte entfernen sich nach einer zufälligen Zeitspanne ruckartig in zufälliger Richtung von ihrem Platz und kehren mit einer Pendelbewegung langsam zu ihrem Ausgangspunkt zurück. Die Abbildung unten zeigt links die Ausgangsverteilung und rechts eine Situation, bei der sich gerade drei Punkte bewegt haben.



Das brauchen wir von **Snap!***:*



Mit **berühre ... ?** kann geprüft werden, ob das aktuelle Sprite das angegebene Element berührt. Dieses Element kann sowohl ein anderes Sprite sein, aber auch eine Kante der Bühne, der Mauszeiger oder eine Malspur auf der Bühne.



Analog kann mit **berühre eine Farbe ?** geprüft werden, ob eine bestimmte Farbe (eines anderen Sprite, einer Malspur oder eines Bühnenelements) berührt wird.

In Abhängigkeit des Resultats werden die darauf folgenden Skripte ausgeführt.



24.2 Hommage à van Weeghel: Dynamic Structures

Der Niederländer Willem van Weeghel gehört derzeit zu den bekanntesten und aktivsten kinetischen Künstlern. Er verwendet in seinen Werken stets wenige gleichartige Elemente mit identischen Bewegungsoptionen. Für ihn ist die Form der Elemente in der Regel weniger wichtig als ihre Bewegungen. Erst deren Koordination schafft aus bewusst einfachen Elementen komplexere Strukturen:

„Movement is the central means of expression in my work. The changing structures that appear move in the transitional area between chaos and order, between variability and uniformity, between volatility and consistency. As a reconciliation of opposites."

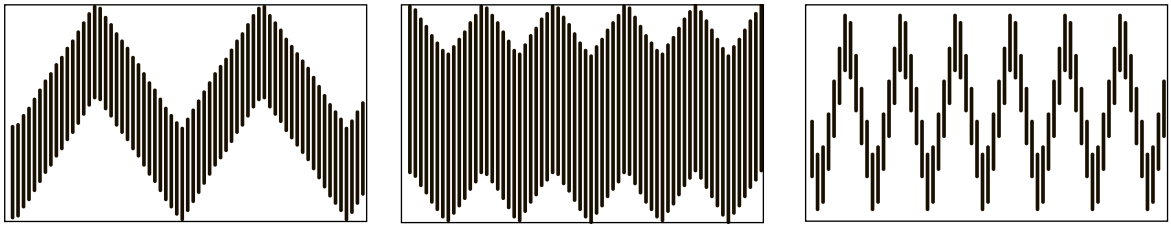
Für die Betrachter nicht sichtbar ist die Antriebstechnik und das intgrierte Computersystem, mit dem van Weeghel die Bewegungen steuert. Die sehr langsamen Bewegungen verlangen von den Betrachtern Geduld, denn erst bei längerer Betrachtung werden die sich immer wieder neu ergebenden Konstellationen sichtbar. Bei unserem *Recoding* kann die Geschwindigkeit leicht gesteuert werden; für Testzwecke ein schneller Ablauf, für eine Präsentation im Sinne van Weeghels entsprechend langsamer.

Das erste Projekt orientiert sich an *Dynamic Structure 1586*, bei dem senkrechte parallele Linien sich auf und ab bewegen und an den oberen und unteren Begrenzungen reflektiert werden. Bei der programmtechnischen Umsetzung wird eine gewünschte Anzahl schwarzer Streifen (als Klone einer entsprechenden Vorlage) im gleichen Abstand nebeneinander positioniert. Allen Klonen wird nach ihrer Positionierung eine Wartezeit **klone_delay** verordnet, bevor sie sich bewegen. Durch diese kleine (steuerbare) Zeit-



versetzung ensteht eine wellenartige Bewegung. Jeder Klon bewegt sich anschließend fortlaufend um eine (über eine Variable **tempo** regelbare) Schrittzahl und ändert bei **pralle vom rand** ab seine Richtung.

Im Folgenden sind unterschiedliche resultierende Muster zu sehen, deren Struktur sich jeweils aus dem Verhältnis **tempo** zu **klone_delay** ergibt. Die Abbildung links zeigt mittlere Striche bei **tempo** 15 und **klone_delay** 0.05, die Abbildung mittig längere Striche bei **tempo** 15 und **klone_delay** 0.05 sowie die Abbildung rechts kurze Striche mit **tempo** 60 und **klone_delay** 0.05.

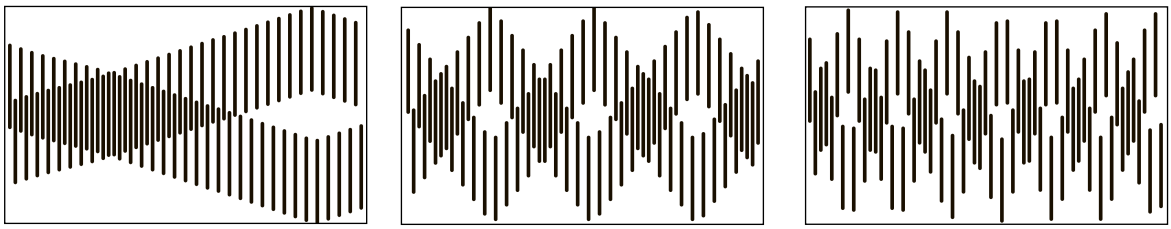


Ein einfaches *Remixing* besteht darin, die Klone zwar im gleichen Abstand nebeneinander, aber vertikal versetzt gegeneinander zu positionieren. Dazu reicht, im Block **klone Sprite** mittels einer bedingten Verzeigung die vertikale Position im Wechsel festzulegen.



Auch hier ist das Verhältnis **tempo** zu **klone_delay** entscheidend. Einige Ergebnisse zeigt die folgende Abbildung, links mit **tempo** 3 und **klone_delay** 0.05, mittig mit **tempo** 10 und **klone_delay** 0.075 und rechts mit **tempo** 20 und **klone_delay** 0.075.

Wie bei allen hier entwickelten animierten Beispielen sind die spezifischen Wirkungen nur bei Betrachten langsam ablaufender Animation zu beobachten.



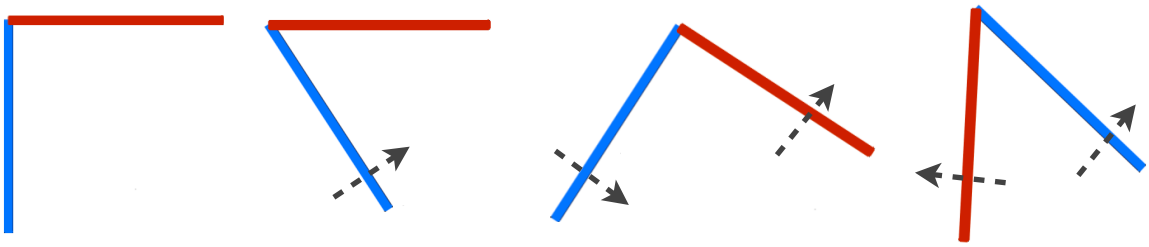
Mich haben weitere Vorbilder von van Weeghel zum *Recoding* gereizt. Dabei handelt es sich um *Dynamische Strukturen*, bei denen miteinander gekoppelte Stäbe sich nach einer genauen *Choreographie* bewegen (laut David C. Roy: *„moving lines that seem to be carefully choreographed“*).

Kopplung bedeutet dabei, dass zwei (oder mehrere) Linien an einem Gelenk mechanisch miteinander verbunden sind. Je nach Steuerung können sie gemeinsame Bewegungen durchführen (die angekoppelte Linie ist quasi „eingerastet) oder aber unabhängig voneinander (die angekoppelte Linie hängt zwar am Gelenk, kann sich aber unabhängig drehen). Durch diese Formen der Kopplung kommen Bewegungsüberlagerungen zustande. Je nach Geschwindigkeitsunterschieden können diese zu wiederkehrenden Mustern führen.

24.3 Exkurs: Gekoppelte Objekte

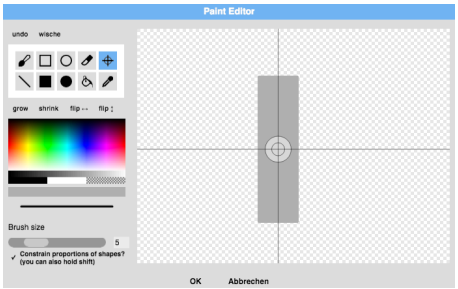
Erfreulicherweise bietet Snap! genau solche Möglichkeiten, Objekte (Sprites) zu koppeln (sogenannte *nested sprites*). Ein Objekt, an das andere Objekte angekoppelt wird, bildet den *Anker*. Ein angekoppeltes Objekt bildet einen *Teil* und kann bei Bedarf selbst wieder den Anker für weitere Objekte bilden.

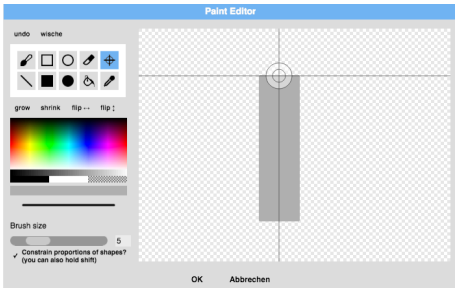
Im folgenden Beispiel wird eine rote Linie im rechten Winkel an eine blaue Linie gekoppelt. Diese Ausgangssituation zeigt die folgende Abbildung ganz links. Für das resultierende Verhalten bei Rotationsbewegungen gibt es drei Möglichkeiten:



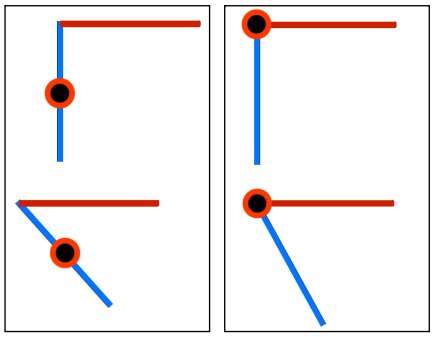
- Bei Rotationsbewegung der blauen Linie kann die rote Linie in ihrer Ausgangsposition verharren (zweite Abbildung von links).
- Bei Rotationsbewegung der blauen Linie wird die rote Linie (hier im rechten Winkel) mitgeführt (zweite Abbildung von rechts).
- Die Rotationsbewegungen der blauen und roten Linie laufen unabhängig voneinander, also entweder mit unterschiedlichem Tempo in gleicher Richtung oder in gegenläufiger Richtung (Abbildung ganz rechts).

Hinweis: *Es ist wichtig, den Rotationspunkt sowohl von* Anker *als auch* Teil *vorher zu bestimmen. Er ist jeweils auf den Mittelpunkt des Sprite voreingestellt, kann im* Paint Editor *aber beliebig verschoben werden.*





Der Unterschied wird im Beispiel rechts deutlich. Ist der Rotationspunkt der blauen Linie mittig und die rote Linie am oberen Ende fixiert, ergibt sich bei Rotation der blauen Linie die linke Konfigration. Ist der Rotationspunkt der blauen Linie dagegen am oberen Ende, an dem auch die rote Linie andockt, ergibt sich die rechte Konfigration.

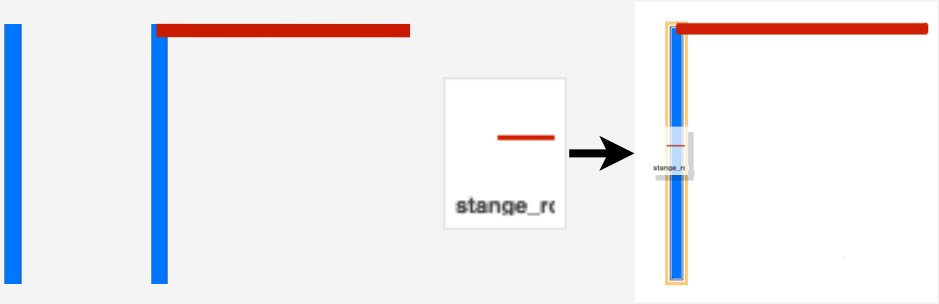


Das brauchen wir von **Snap!**:



Zum Aufbau von Nested Sprites:

- Zuerst wird der *Anker* an der gewünschten Stelle auf der Bühne platziert.
- Danach wird das *Teil* an der gewünschten Stelle des *Ankers* mit der gewünschten Richtung platziert.
- Danach ist das Icon des *Teils* aus dem Objektbereich mit gedrückter Maustaste über den *Anker* zu führen. Sobald sich die Maus über dem *Anker* Befindet, wird dieser als Zielbereich gelb umrandet.



- Wird das Icon des *Teils* losgelassen, springt es zurück in den Objektbereich, ist aber nun Teil des *Ankers*. Symbolisiert wird das ab jetzt durch eine kleine rote Linie innerhalb des *Anker*-Icons.
- Im *Teil*-Icon wird die Kopplung ab jetzt durch eine kleine rote Linie angezeigt.
- Ob das Teil starr oder eigenbeweglich ist, kann über einen kleinen Schalter oben rechts im Icon des Teils festgelegt werden.



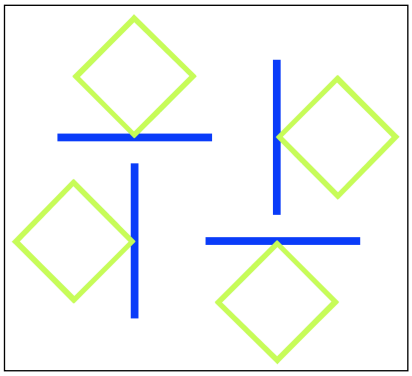


Ab jetzt kann das *Teil* zusammen mit dem *Anker* bewegt, rotiert, skaliert usw. werden.

Teile können natürlich auch wieder vom *Anker* gelöst werden (im Kontextmenü des Sprite mit **Abtrennen von Anker**).

Mit so gekoppelten Teilen ist es möglich, weitere von van Weeghels dynamischen Strukturen für das *Recoding* heranzuziehen.

Bei *dynamische struktuur 13* sind vier gerade Stäbe jeweils mit einem Quadrat aus anderfarbigen Stäben mittig gekoppelt. Die nächste Abbildung zeigt eine mögliche Ausgangssituation. Die Kopplung kann starr oder flexibel sein. Der Ablauf der Animation



soll flexibel steuerbar sein, um den *Choreographien* van Weeghels möglichst nahe zu kommen.

Zunächst sind die Stäbe und Quadrate im *Paint Editor* zu erstellen und mit den gewünschten Rotationspunkten zu versehen (hier z.B. beim Stab mittig und beim Quadrat als Eckpunkt).

Insgesamt ergeben die vier Stäbe und die vier Quadrate also acht Objekte. Für Ablauf und Eindruck der Animation ist deren Positionierung auf der Bühne sowie die Drehrichtung und Drehgeschwindigkeit der Obkjekte entscheidend. Damit die Kennwerte an einem zentralen Ort festgelegt werden können und möglichst einfach zu ändern sind, können in einem Skript für die Bühne zwei Listen definiert werden. Eine Liste für die **kenngroessen_staebe** enthält für die vier Stäbe jeweils **x-Koordinate** und **y-Koordinate** für die Positionierung auf der Bühne sowie **Winkel** und **delta-Winkel** für den Ausgangswinkel und die Änderung des Winkels bei jeder Iteration:



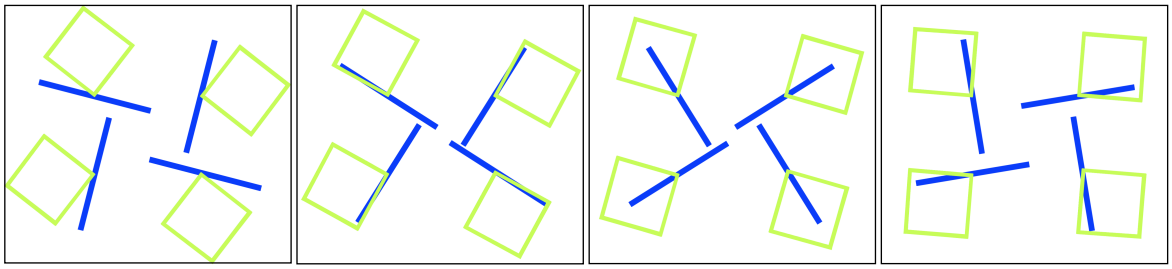
In gleicher Weise sind die **kenngroessen_quadrate** festzulegen, also z.B.



Alle Objekte enthalten ihrerseits dann dieselben Skripte. Wenn nun z.B. durch Drücken der **Taste n** das Programm neu gestartet werden soll, werden im Skript eines Objekts die ersten drei Listenelemente abgerufen und das jeweilige Objekt entsprechend positioniert. Bei Drücken z.B. der *Leertaste* wird die Animation mit dem Änderungswinkel (dem vierten Listenelement) ausgeführt.

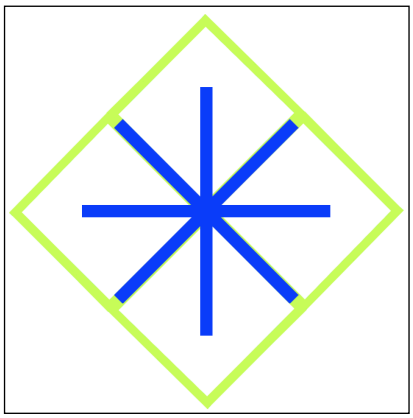


Die folgende Abbildung zeigt eine statische Abfolge sich daraus ergebender Konstellationen.

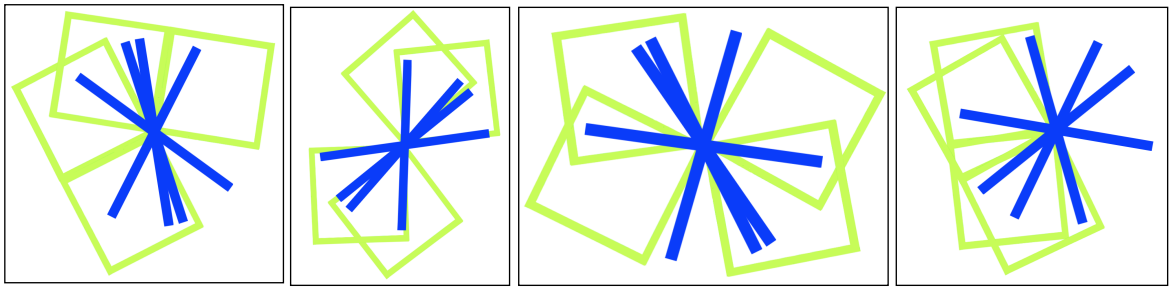


Hinweis: *Bei van Weeghel ergeben sich im Laufe der Bewegungen immer wieder auch die Ausgangskonstellationen, gewissermaßen als wiedererkennbare Muster im sonst sich ständig ändernden Ablauf. Im Programm ist das zu erreichen, wenn die Winkeländerungen* **delta-Winkel** *der Objekte untereinander immer ein Vielfaches bilden.*

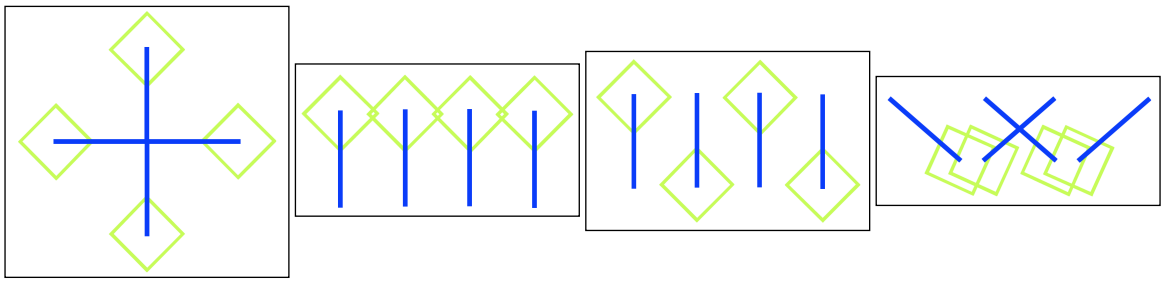
Durch den zentralen Zugriff auf die Kenngrößen im Skript der Bühne lassen sich sehr einfach neue Choreographien definieren. Im folgenden Beispiel z.B. sind alle Stäbe und Quadrate mittig zentriert (wobei sich alle Stäbe oberhalb der Quadrate drehen, diese also teilweise verdecken) und alle Objekte haben unterschiedliche Winkeländerungen.





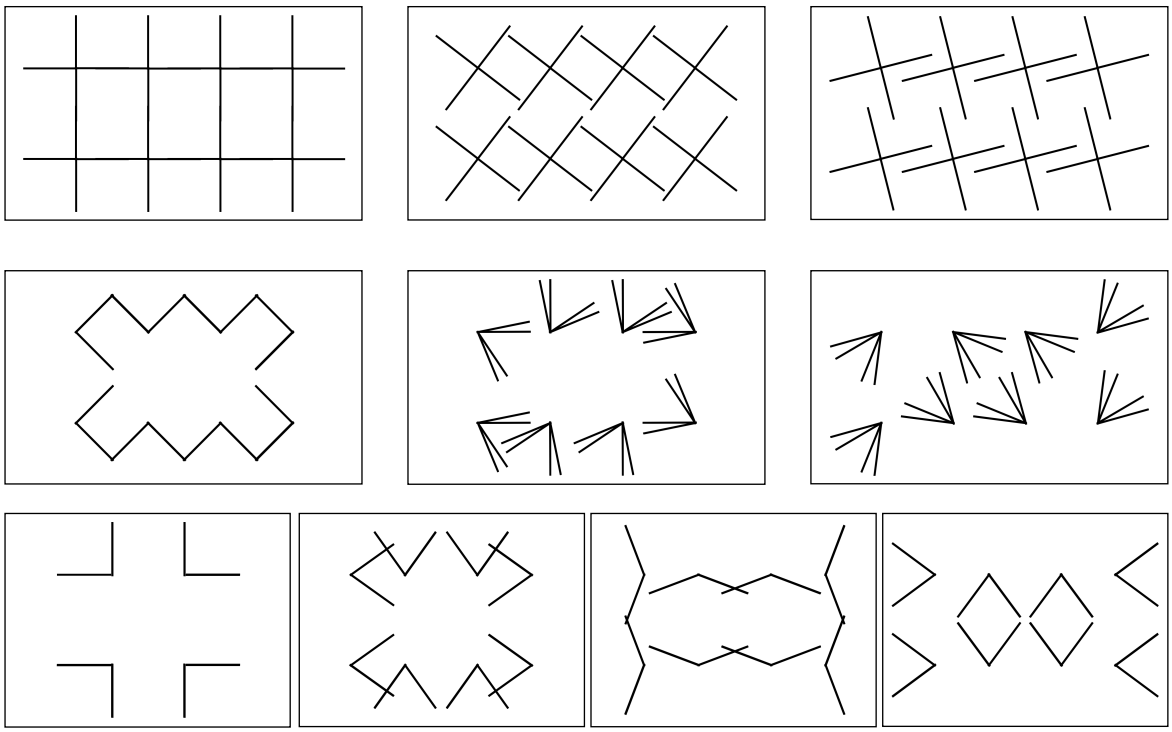


Anregung: *Auch die Rotationspunkte der Stäbe können verlegt werden, z.B. an deren oberes oder unteres Ende. Außerdem kann die Kopplung der Quadrate an anderen Punkten der Stäbe erfolgen, z.B. auch an deren oberen oder unteren Ende. Damit lassen sich dann weitere vielfältige neue Ausgangskonfigurationen definieren, wie in der folgenden Abbildung.*



Das Prinzip der zentralen Steuerung vieler Objekte kann noch weiter getrieben werden. In der Installation *dynamische struktuur 29117* sind es immerhin 32 Stäbe, die es zu steuern gilt. Dabei gibt es acht *Anker* mit jeweils drei angehängten *Teilen*. Die acht Anker sind in zwei Reihen mit je vier Ankern im gleichmäßigem Abstand angeordnet. Ihre Ausgangswinkel und Drehrichtungen sollen ebenso individuell festgelegt werden können, wie die der angehängten Teile. Damit bei 24 angehängten *Teilen* die Listen nicht zu unübersichtlich werden, sollten sie z.B. als eine Liste **kenngroessen_teile** von acht Listen (den acht *Ankern* zugeordnet) mit je drei Listen (der jeweils drei an einen *Anker* angehängten *Teile*) geschachtelt werden. Die Belegung mit Parametern erfordert dennoch Sorgfalt.

Mögliche Ausgangskonstellationen mit daraus folgenden Zwischenstadien zeigen die folgenden Abbildungen:



Anregung: *Die bisherigen Möglichkeiten, Anfangskonfigurationen und Drehbewegungen zu definieren sind schon äußerst vielfältig. Für Installationen wäre es natürlich wünschenswert, längere Abfolgen mit wechselnden Anfangskonfigurationen zu definieren. Das erfordert den Zusatzaufwand, bei*

den Ankern (eventuell auch bei den anghängten Teilen) über Zeitabfragen oder über Positionsabfragen der Anker einen Bewegungsablauf zu stoppen und eine neue Anfangskonfiguration aufzurufen.

Soweit zum *Recoding* der kinetischen Kunst von Van Weeghel. Zum *Remixing* könnte es gehören, noch größere Arrangements zu definieren, z.B. mit vier Reihen zu je sechs Ankern. Zur Wirkung kommen diese Animationen aber erst, wenn sie mit einem Beamer auf eine sehr große Fläche projiziert werden können. Für solche Großprojektionen lohnt es sich zudem, statt einfacher Linien für *Anker* und *Teile*, mit komplexeren Figuren zu arbeiten. Die bieten dann unterschiedlichste Möglichkeiten der gegenseitigen Verankerung und eröffen wieder ein nahezu unerschöpfliches Experimentierfeld.

Auch wenn bei van Weeghel die technische Realisierung mit konkreten Materialien einen hohen Stellenwert im Entstehungsprozess einnimmt, so gilt doch auch beim *Recoding & Remixing*, was bei ihm am Anfang im Vordergrund steht (im Interview mit Alison Copley): „*An important aspect is the challenge to create in reality what is in my mind. Although I start with an artistic idea of what should be seen and what could be experienced, the technical realisation is very important.*“

24.4 Hommage à Talman: k25

Der Schweizer Paul Talman (1932 - 1987) war neben seiner Tätigkeit als Produktdesigner und Werbegestalter auch freier Künstler. Als solcher war er u.a. Mitglied der Groupe de Recherche d’Art Visuel GRAV (wie auch François Morellet, siehe Kapitel 25: *Codierte Kunst*). Bekannt wurde er mit seinen kinetischen Kugelbildern, die er auch bei der Ausstellung *Neue Tendenzen* (Nouvelles Tendances bzw. Nove Tendencije) in Zagreb zeigen konnte.

Die Gruppe GRAV führte u.a. ein kooperatives Forschungsprogramm ein, um ihre visuellen Ziele wissenschaftlich zu untermauern. Sie formulierte in einem Manifest, dass mit den Mitteln der Kinetischen Kunst die Herstellung einer neuen visuellen Beziehung zwischen dem Objekt und dem Auge des Beschauers geschaffen werden soll, in der die Bedeutung und die Intervention des Künstlers auf ein Minimum beschränkt sind.

Das passt bestens auf Talmans *interaktive Kugelbilder*, bei denen beweglich montierte, zweifarbige Kugeln sich in unendlich viele Positionen drehen lassen. Das Publikum bestimmt innerhalb der vom Künstler gegebenen Vorgaben das Bild mit und definiert die Komposition durch seinen Eingriff. Das Kunstwerk ist nicht länger ein statisches Bild, sondern ein Konzept, in dessen Rahmen sich eine unendliche Anzahl unterschiedlicher Bilder realisieren lassen.



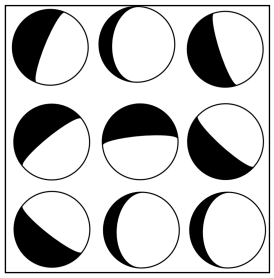
Paul Talman: k25 (1961)
Quelle: Rosen, 2011, S. 75)

Talman hat unterschiedliche Varianten der Kugelbilder erstellt, mit 3*3, 5*5 bzw. 7*7 Kugeln unterschiedlicher Färbung, z.B. grün-rot, wie hier gezeigt, aber auch schwarz-weiß, weiß-grün oder schwarz-rot.

In Anlehnung an sein Exponat *k25* entstand die folgende Animation. Basis dafür ist die Darstellung der Kugelbewegung mit 48 Zustandselementen (ähnlich aussehend wie Mondphasen)[70], die in rascher Abfolge dargestellt werden können:



Um dem Vorbild nahezukommen, macht es Sinn, gerade diese Animation interaktiv zu gestalten. Für die Steuerung können z.B. folgende Tasten vorgesehen werden: **n** um mit **Neustart** das Ausgangsbild herzustellen (wobei die Drehrichtung der Kugel zufällig erfolgt), **s** um mit **Stopp** das Programm zu beenden, sowie **d** um die **Drehrichtung** für alle Elemente zusätzlich kontinuierlich zu verändern.



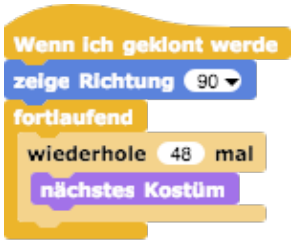
Das erste Beispiel soll aus neun solcher Elemente bestehen, angeordnet in einer 3*3-Matrix. Es wäre nun sehr umständlich und unflexibel dafür neun Sprites anzulegen (umso mehr, falls noch mehr Sprites benötigt werden). Eine größere Flexibilität kann dadurch erreicht werden,

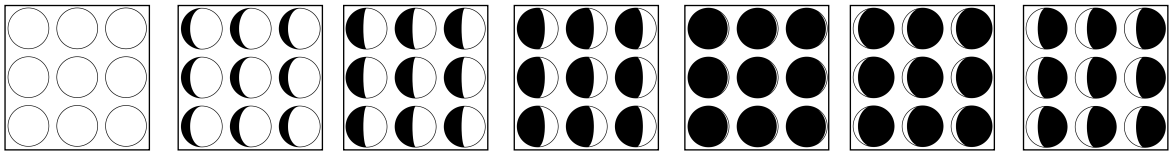
1. dass in einer m*n-Matrix
2. Klone des Ausgangssprites mittig in den Feldern platziert werden.

Die Klone „erben“ dabei alle Kostüme und alle Skripte für die Interaktion. Bei Bedarf ist die Größe der Kostüme mit **setze Größe auf x %** an die Feldgröße innerhalb der Matrix anzupassen.

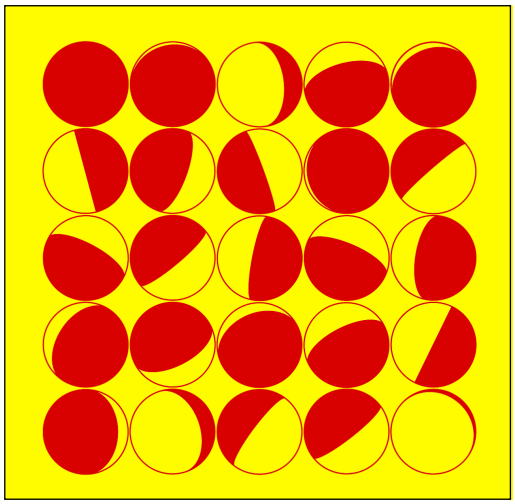


Die folgende Abbildung zeigt eine statische Abfolge einiger Zwischenstadien, die sich aus dieser Ausgangskonfiguration (hier mit gleicher Drehrichtung aller Kugeln) ergeben:





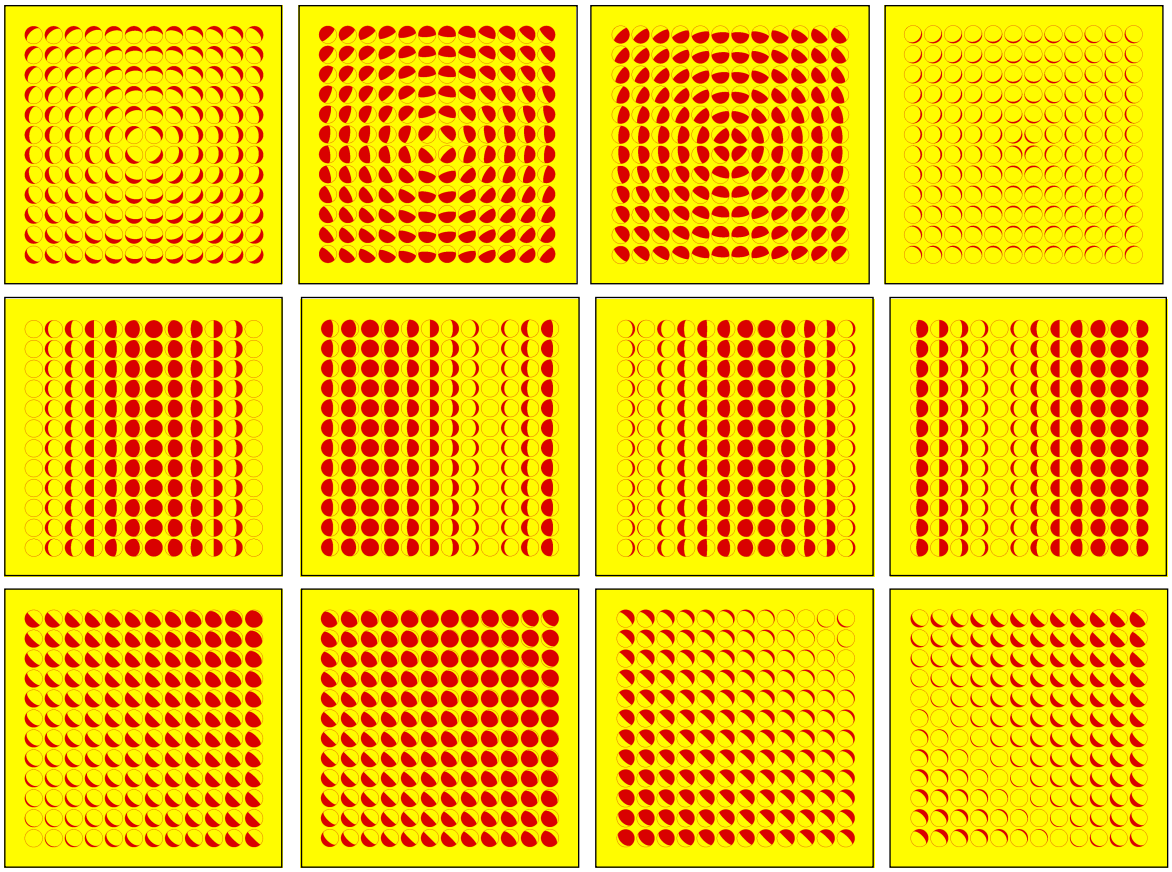
[70] Die Erstellung dieser Elemente wird sehr erleichtert durch die Verwendung eines externen Grafikeditors.



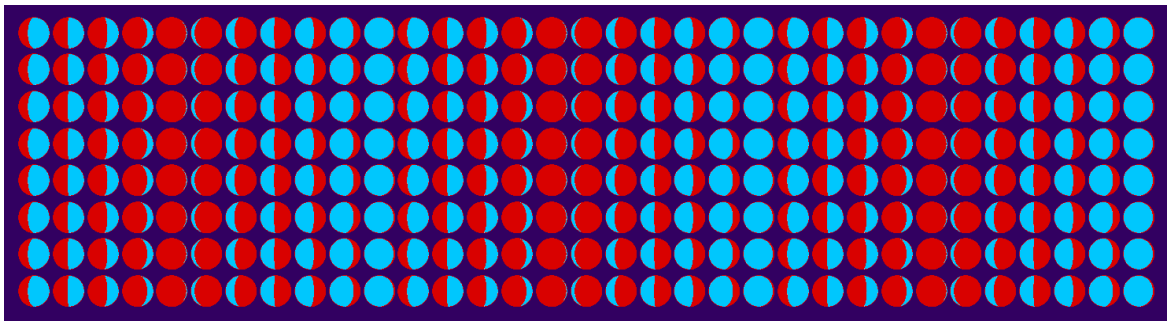
Eine 5*5-Matrix mit farbigem Hintergrund und zufällig bestimmter Drehrichtung der Kugeln kommt dem Vorbild *k25* von Talman sehr nahe:

Da die Klon-Eigenschaften flexibel festgelegt werden können, kommen beim *Remixing* der Kugelbilder neue Aspekte zum Tragen. So führt z.B. eine anfänglich gemeinsame Ausrichtung aller Kugeln in der Animation zu rhythmischen Veränderungen. Entsprechend sind in der folgenden Abbildungsreihe oben anfangs alle Kugeln mit demselben Anfangskostüm auf das Zentrum ausgerichtet: **zeige auf Liste 0 0**.

In der Reihe darunter erhalten die Kugeln anfangs von links nach rechts zunehmend höhere Kostümnummern zugeordnet und alle die gleiche Ausrichtung nach rechts. In der unteren Reihe erhalten die Kugeln anfangs von links unten nach rechts oben zunehmend höhere Kostümnummern zugeordnet und alle die gleiche Ausrichtung nach oben rechts. Die statischen Bildfolgen können die entstehenden Rhythmen wieder nur andeuten.



Anregung: *Es ist zwar ein gewisser Aufwand, die 48 Kostüme zweifarbig zu gestalten, aber die Mühe lohnt sich. Es werden dann vielfältige Kombinationen mit Hintergrundfarben möglich. Für die großformatige Präsentation mit Beamer bietet sich eine m*n-Matrix im Querformat an:*



25. CODIERTE KUNST

Die bisher ausgewählten Projekte orientierten sich alle - bis auf die Ausnahmen Bild 17 *Binary Rhythm* nach James Hugonin; Bild 22 *Bands of Color* nach Sol LeWitt, die *Kinetische Kunst* und die *Moiré-Muster* - an Vorlagen der frühen Computerkunst. Weil die Algorithmen der Vorbilder meist nicht zugänglich sind, war für das *Recoding & Remixing* in der Regel ein vierstufiges Vorgehen naheliegend und notwendig:

1. Am Anfang stand die *Analyse*, welche Muster und Strukturen charakteristisch für die Bilder sind, ob und welche wiederkehrenden Muster erkennbar sind, sowie die Analyse von zufälligen Abweichungen der Formen, Anzahl der Elemente und der Anordnung von Objekten.
2. Es folgte die *Abstraktion*; das wurde mit der Beschreibung der typischen Grafikelemente der frühen Computerkunst geleistet (vgl. das Kapitel 2: *Recoding & Remixing)*. Ergebnis war u.a. die Zusammenstellung eines Kompendiums der verwendeten grafischen Elemente im *Figurenbaukasten*.
3. Für das *Recoding & Remixing* der Beispiele mussten dann entsprechende *Algorithmen* formuliert werden. Auch dabei fanden sich typische Konzepte.
4. Im vierten Schritt erfolgte schließlich die Umsetzung der Algorithmen in *Programme*, mit denen am Ende die gewünschten Grafiken erzeugt wurden.

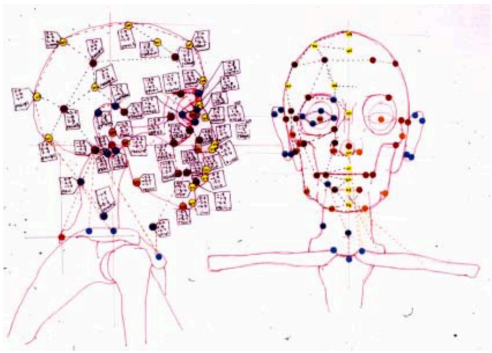
Dieser analytische Ansatz hat inzwischen zugegebenermaßen meinen Blick auf die Werke der modernen Kunst insgesamt verändert: In den stilistischen Richtungen der Geometrischen Abstraktion, der Konkreten Kunst, der Konzeptkunst, des Konstruktivismus oder der Op Art werden sehr oft wiederkehrende Muster und Strukturen - bestehend aus geometrischen Objekten - gefunden, die (mich) dann dazu reizen, diese auch algorithmisch zu rekonstruieren.

Die Computerkunst hat ein gewisses Alleinstellungsmerkmal durch die alleinige Erzeugung und Ausgabe der Bilder mit Hilfe des Computers. Dennoch hat sie enge Bezüge zu den gerade genannten Kunstrichtungen. Diese Bezüge sollen in diesem Kapitel an einigen Beispielen verdeutlicht werden. Beim *Recoding & Remixing* dieser Beispiele geht es mir insbesondere darum zu zeigen, dass auch in den genannten Kunstrichtungen vergleichbare Vorgehensweisen zu finden sind. Meines Erachtens sind das Belege dafür, dass sich die Computerkunst eben nicht auf die Verwendung des Werkzeugs reduzieren lässt, sondern dass sie sich in die vielen Bestrebungen des letzten Jahrhunderts einordnen lässt, mit den Mitteln der Abstraktion, der Reduktion und der Serialität die Ausdrucksformen der bildenden Kunst zu erweitern.

Georg Nees hat sich früh und intensiv mit theoretischen Fragen zur Rolle der Computerkunst befasst (Nees, 1969). Er wurde bei einer Ausstellung seiner computergenerierten Grafiken von einem anwesenden bildenden Künstler gefragt, ob denn der Computer am Ende ihn - den Künstler - ersetzen könne (Nees, 2005, S. VIII). Nees antwortete, ja, wenn er - der Künstler - dem Computer sagen könne, wie er denn male. Das bedeutet also, wenn der Künstler in der Lage wäre, dem Computer seinen Malstil beizu-

bringen, ihm diesen Malstil einzuprogrammieren (was dann über die Beispiele für das *Recoding & Remixing* in diesem Buch deutlich hinausgehen würde).

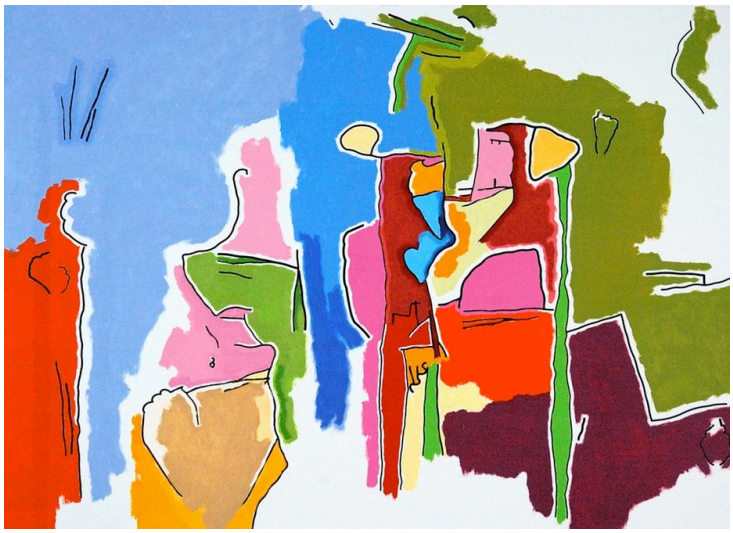
Tatsächlich begann der 2016 verstorbene Amerikaner Harold Cohen, Professor und anerkannter Maler und Grafiker mit internationaler Reputation, während eines Gastaufenthalts am Labor für Künstliche Intelligenz der Stanford University damit, ein Computerprogramm zu schreiben (das er AARON nannte), mit dem er seinen persönlichen Malstil umsetzen wollte. Er führte dazu detaillierte Beschreibungen von Bildelementen ein, anfangs beschränkt auf Kreise und Kästen, später mit Regeln über Perspektiven, die menschliche Anatomie (wie nebenstehend in der Studie eines Kopfes skizziert; Cohen, 1994) oder den Aufbau von Pflanzen, etwa dass Äste mit zunehmender Länge immer dünner werden.



Anfangs ließ er vom Programm AARON Steine, dann Pflanzen, schließlich Personen zeichnen. Er begann mit S/W-Bildern auf Plottern, ergänzte dies später aber um Farben und Pinsel. Seine späteren Werke wurden dabei immer abstrakter.



Aaron, with Decorative Panel, 1992



In Zana's Room, painting by AARON from 2012

Harold Cohen betrachtete sein Programm AARON als nicht kreativ. *"Ich denke, Kreativität muss Selbstveränderung in dem Sinne einbeziehen, dass der Schöpfer den kreativen Akt mit einem anderen Weltmodell verlassen muss, als er es vorher hatte"*, argumentiert er. AARON sei zwar in gewisser Weise intelligent, aber *"ein Gehirn zu haben und ein Leben zu haben, sind zwei verschiedene Dinge."*

Frieder Nake (1995) drückt es so aus: *„Cohen, der Maler, zwingt eine Maschine dazu, das zu tun, was er will. Er erreicht dies, indem er einen Teil seines Zeichnens und Malens veräußert. Genauer gesagt, veräußert er einen Teil seines Wissens vom Zeichnen und Malen. Einen Teil seines Denkens also, einen solchen Teil, dem er algorithmische Form geben kann. Harold Cohen hat es auf der Welt am weitesten darin gebracht, bestimmte Formgebungen und Farbaufträge in einem Regelwerk zu fixieren. Diese Regeln sind so genau (einmalig) und enthalten soviel an Offenheit (beliebig), daß der Computer ihnen folgen kann und der Anschein entsteht, die Maschine sei kreativ."*

Es kann und soll im Folgenden daher gar nicht erst der Versuch unternommen werden, vergleichbare *Regelwerke* für das Vorgehen bestimmter Künstler zu entwickeln. Es soll reichen, für ausgewählte Beispiele vergleichbare Vorgehensweisen wie bei der Computerkunst aufzuzeigen. Die Vorstellung erfolgt in alphabetischer Reihenfolge der Künstler.

25.1 Hommage à Josef Albers: Interaction of Color

Der Deutsch-Amerikaner Josef Albers (1888 - 1976) ist nicht nur als Künstler weltbekannt geworden. Mindestens gleich wichtig, wenn nicht noch wichtiger ist seine Rolle als Lehrer. Zuerst am Bauhaus in Dessau, nach seiner Emigration nach Amerika am Black Mountain College und später für kurze Zeit (1954-1955) auch an der Hochschule für Gestaltung in Ulm, beeinflusste er ganze Generationen von Künstlern.

Seine Arbeit war geprägt von genauer Beobachtung und Experimentieren. Besonders interessierte ihn die Wirkung von Farben in einer Gesamtkomposition. Dafür beschäftigte er sich mit wahrnehmungsrelevanten Fragen, der Subjektivität optischer Wahrnehmungen und optischen Täuschungen.

Die geometrische Grundform des Quadrats spielt eine zentrale Rolle in seinem Werk. 1950 beginnt er mit der Serie Hommage to the Square. In dieser Bilderserie legt er immer drei oder vier verschiedenfarbige Quadrate ineinander. Die Farben verwendet er „tubenrein", d.h. sie werden nie gemischt, sondern direkt aufgetragen; die Artikel-Nummern der Farben dokumentiert er auf der Rückseite der Bilder. Seine Frage ist: Wie interagieren die Farben miteinander, wie beeinflussen sie sich wechselseitig? Über 26 Jahre hinweg hat Albers hunderte von Variationen davon gemalt, immer bestehend aus den drei oder vier konzentrischen Quadraten. Viele dieser Bilder finden sich weltweit in renommierten Sammlungen und Museen und gehören zum Kanon der Klassischen Moderne.

Frieder Nake, dem wir in mehreren Kapiteln bereits begegnet sind, diskutiert die kulturelle Bedeutung des Computers als Werkzeug und Medium (Nake, 1995). Er setzt die Computerkunst an konkreten Beispielen in Bezug zur Pop Art, der konkreten Kunst und dem Konstruktivismus. Eines seiner Beispiele ist die genannte Bildserie *Homage to the Squar*e. Für Nake ist es das vielleicht einfachste Beispiel der Kunstgeschichte, das in seiner Systematik mit einem kombinatorischen Programm vollständig ersetzt werden

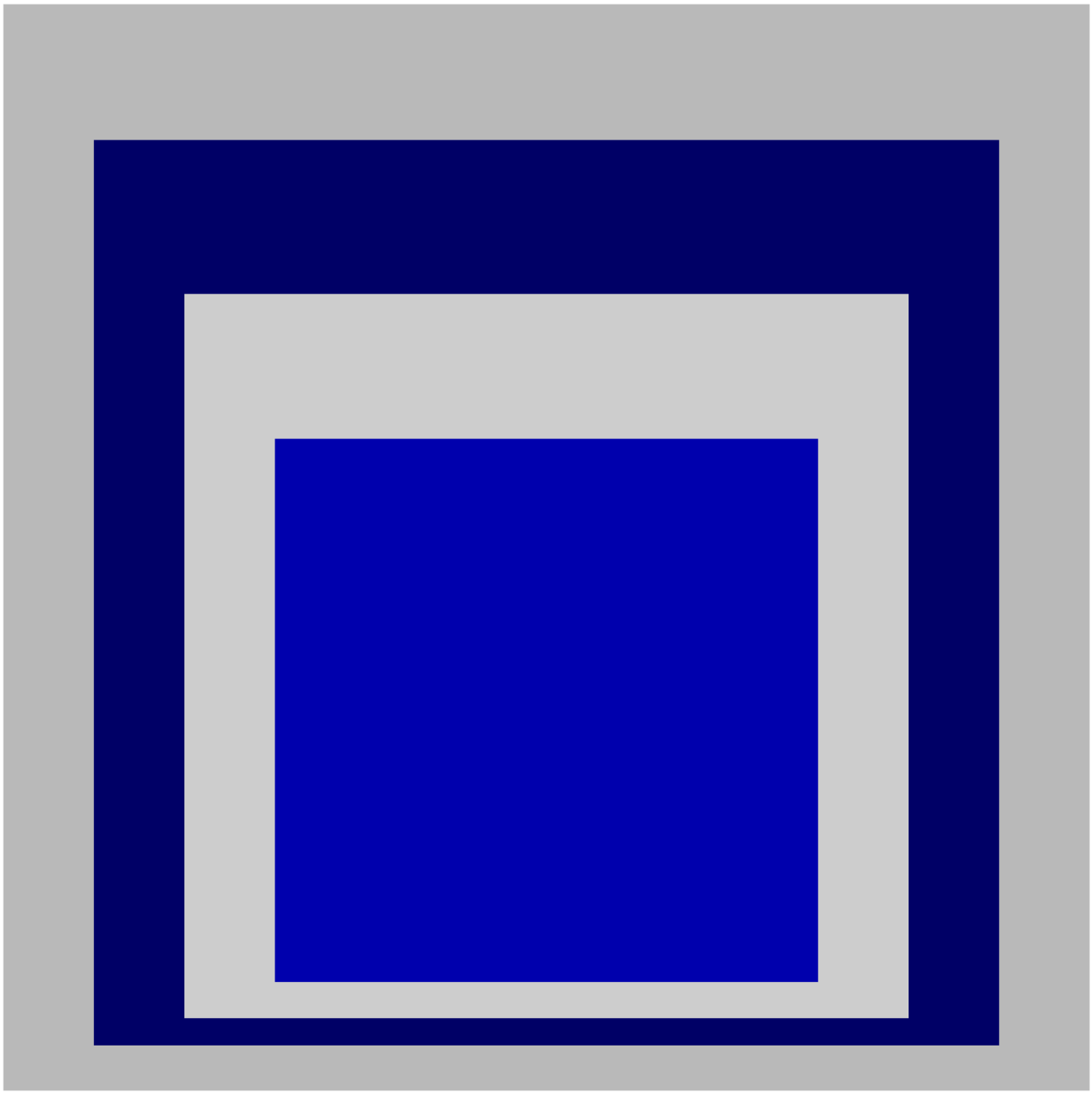


Bild 94: Hommage à Albers: Homage to the Square

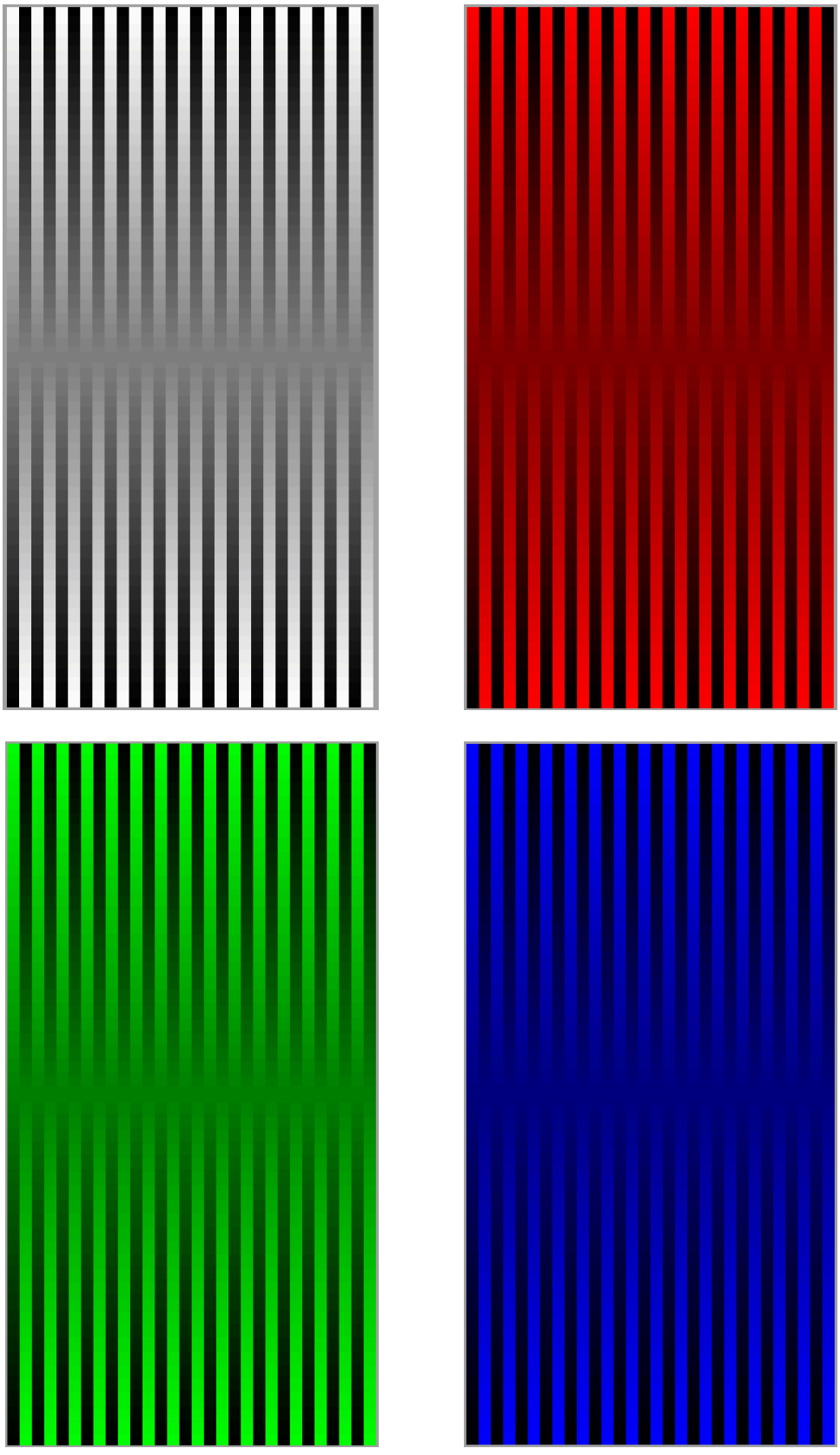


Bild 95: Hommage à Albers: Interaction of Colors

kann. Sein ironisches Fazit (das m.E. wohl für viele Werke moderner Kunstrichtungen gelten dürfte) ist, dass

„[...] die Kunst an den Albersschen Quadraten nicht im Schema liegt. Sie muss in seiner speziellen Wahl der Farben liegen. Ansonsten aber ist diese pädagogische Übung "Kunst" nur dadurch, dass die Kunstgeschichte diese Bilder in die Museen hängt und in die Kataloge aufnimmt. [...] Der soziale Prozess der Rezeption macht ein Werk zum Kunstwerk, nicht der Schaffensprozess des Künstlers."

Nakes Vorschlag eines kombinatorischen Programms („*ein Schema für den Anfangsunterricht in Informatik*"; Nake, a.a.O.) ist tatsächlich äußerst einfach umzusetzen (siehe Bild 94). Es reicht die Position der konzentrischen Quadrate festzulegen und dann die Quadrate durch Linien entsprechender Strichdicke zu zeichnen (oder alternativ mit **n-eck gefuellt um x y n=4 radius**). Wenn Sie - wie Albers - die Farbwirkungen systematisch untersuchen wollen, sollten Sie die Farben allerdings nicht vom Programm zufällig festlegen lassen, sondern für die einzelnen Quadrate gezielt auswählen!

Albers veröffentlichte 1963 als Quintessenz seiner Experimente in *Interaction of Colors* seine eigene Farbtheorie. Das Buch gilt bis heute als klassisches Lehrbuch für Kunststudenten (und ist inzwischen sogar als interaktives Programm für iPad-Tablets erschienen). Viele seiner Experimente sind auch bestens geeignet für das *Recoding*, wie Bild 95 zeigt. Dort wird der Verlauf in Graustufen von Hell nach Dunkel und umgekehrt (oben links) im RGB-Farbmodell einfach in Schleifen durch gleichartige schrittweise Änderung der drei Farbkomponenten erreicht. Das *Remixing* besteht in der Anwendung des gleichen Vorgehens für jeweils eine einzelne Farbkomponente (Rot, Grün bzw. Blau). In allen vier Fällen verschwimmt die Aufteilung in ansonsten deutlich unterscheidbare Linien in einem diffusen Mittelbereich.

25.2 Hommage à Morellet: Zufall

Der Franzose François Morellet (geb. 1926 in Cholet, Frankreich)) ist ein Vertreter der Konkreten Kunst, der für seine geometrisch-abstrakten Bilder berühmt geworden ist. Er war Mitbegründer der Groupe de Recherche d'Art Visuel (GRAV). Morellet entwickelte Formgebungssysteme aus denen Serien mit Linien, Dreiecken, Quadraten und Rastern hervorgingen (die er später auch in Neon-Installationen umgesetzt hat).

Seine erste Serie entstand etwa 1953 zur Zahl π, bei der er die ersten 24 Ziffern durch weiße (ungerade Zahl) bzw. schwarze Felder (gerade Zahl) darstellte. Sein wohl berühmtestes Beispiel ist Répartition aléatoire de 40.000 carrés suivant les chiffres pairs et impairs d'un annuaire de téléphone. Morellet schildert seine Vorgehensweise dabei wie folgt (Morellet in Weber, 2002. S. 58):

1. „*Ich vermeide es, auch nur das geringste Interesse an der Form oder an der Struktur zu wecken,*
2. *verwende nur zwei Farben,*
3. *und zwar in einem Verhältnis von 50 % zu 50 % gleichmäßig verteilt über die gesamte Bildfläche*
4. *und setze auf eine zufällige Verteilung aller Details.*

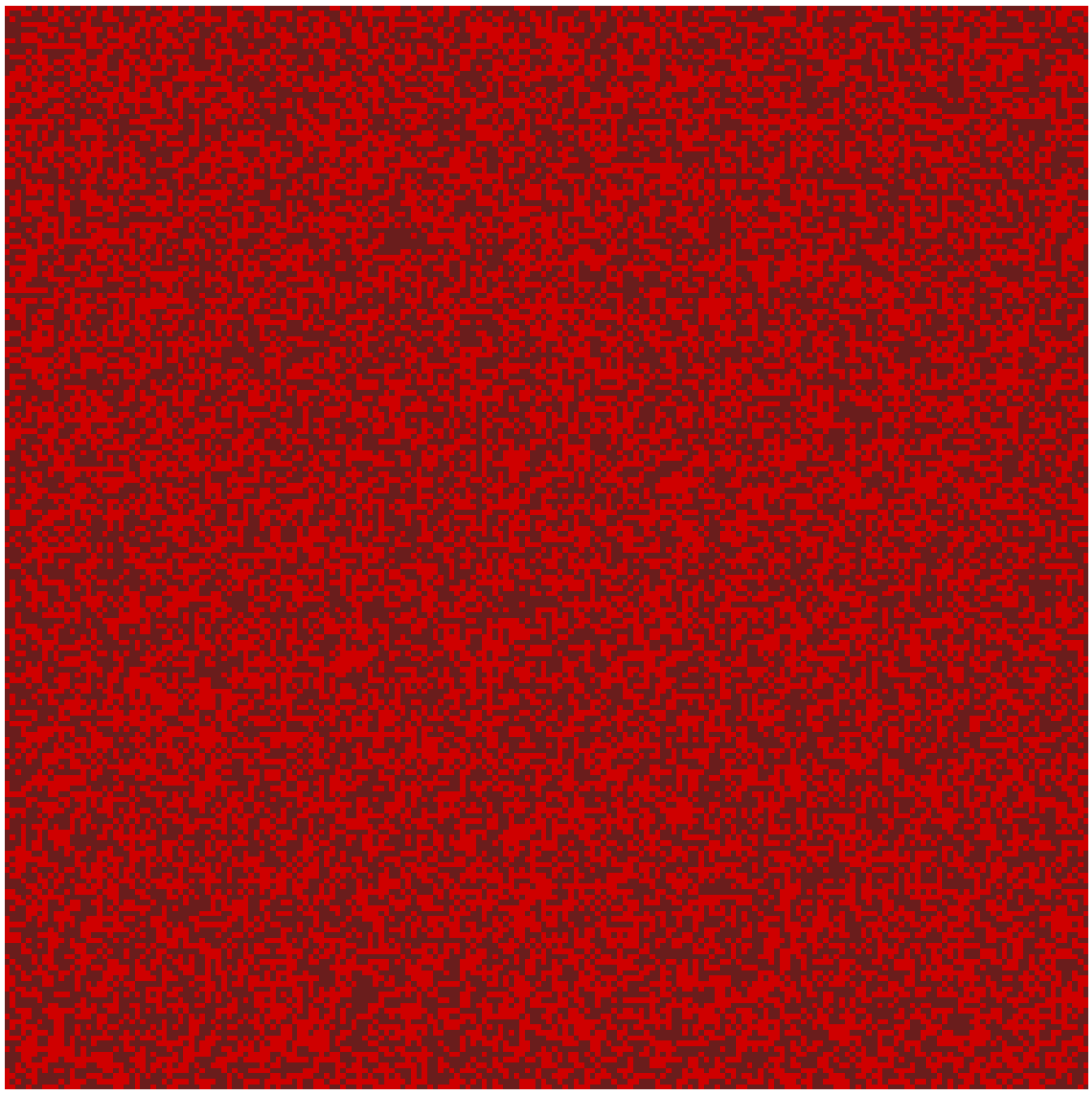


Bild 96: Hommage à Morellet: Répartition aléatoire de 40.000 carrés suivant les chiffres pairs et impairs d'un annuaire de téléphone, 50% noir, 50% rouge

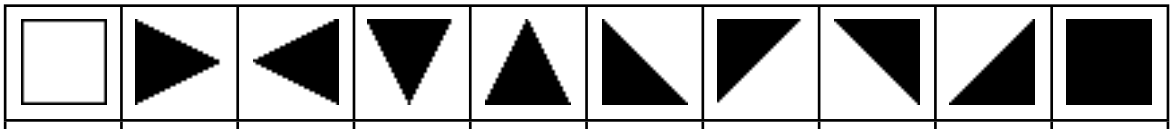
Auf einer Bildfläche von 1x1 m habe ich 200 horizontale und 200 vertikale Linien gezeichnet und damit 40.000 Quadrate von je 5 mm Seitenlänge gebildet.

Ich hatte mich für eine Zahlenreihe aus einem beliebigen Telefonbuch entschlossen und bat meine Frau und meine Kinder, sie mir vorzulesen. Jedes Quadrat bekam eine Zahl. Bei einer geraden Zahlen machte ich ein Kreuz, bei einer ungeraden ließ ich das Feld frei. Nach Beendigung dieser Arbeit hatte ich ca. 20.000 Quadrate ohne Kreuz. Jetzt brauchte ich nur noch die Quadrate mit Kreuz mit einer Farbe (blau) und die ohne Kreuz mit einer anderen Farbe (rot) anzumalen. Diese Arbeit erstreckte sich beinahe auf ein ganzes Jahr.

Und auch andere Farbreaktionen beobachten zu können, fotografierte ich diese Bilder und machte systematisch Serigrafien mit hunderterlei unterschiedlichen Erfahrungswerten. Ich setzte vor allem Farben mit einem sehr ähnlichen Farbwert ein, um eine optisch wirksamere Mischung zu erreichen."

Für das *Recoding* in Bild 96 reicht es, in einer Doppelschleife in den Spalten und Reihen nacheinander die Quadrate zu zeichnen. Die Farbverteilung (bei Morellet immer 50%) kann mit **Zufallszahl von 1 bis 100 > Grenzwert** genau gesteuert werden. Unterhalb des Grenzwerts wird die erste Farbe, oberhalb des Grenzwerts die zweite Farbe verwendet. Statt Fotografien der Bilder zu bearbeiten, können auf diese Weise ebenfalls - wie von Morellet praktiziert - systematisch Serigrafien erzeugt werden.

In einer Variante dieser Vorgehensweise hat Morellet nach dem gleichen Zufallsprinzip erzeugte gerade bzw. ungerade Zahlen durch schwarze bzw. weiße Dreiecke (jeweils als Bestandteile eines Quadrats) dargestellt. Bei diesem Vorgehen sind also den Zahlen 0 bis 9 entsprechende Dreiecks-Kostüme zuzuordnen:



0	1	2	3	4	5	6	7	8	9

Bild 97 ist auf diese Weise entstanden. Die zugrunde liegende Ziffernfolge kann zu Beginn vom Betrachter mit **frage Deine Ziffernfolge? und warte** erfragt werden. Diese Ziffernfolge muss für die weitere Verarbeitung zunächst in eine Liste der Ziffern umgewandelt werden mit **setze ziffernliste auf (trenne ziffernfolge nach Buchstabe)**. In den Schleifen wird dann jedem Element der Liste (die nun aus den einzelnen eingegebenen Ziffern besteht) das entsprechende Kostüm mit **ziehe Kostüm ((Element i von zeichenliste) + 1) an** zugeordnet und mit **stemple** gezeichnet.

Anregung: *Für ein Exponat bei dem Generate! 2015 Festival habe ich eine erweiterte Fassung dieses Programms verwendet. Ziel war es, die Besucher einen beliebigen Text mit einer Länge von 49 Zeichen eingeben zu lassen. Neben Ziffern waren auch alle Buchstaben des Alphabets mitsamt Umlauten und einigen Sonderzeichen erlaubt, deshalb insgesamt 52 Zeichen. Entsprechend viele Kostüme mussten dafür erstellt werden. Die Zuordnung der Zeichen zu den Kostümen ist hierbei aufwändiger, da für jedes einzelne Zeichen der Liste des Eingabetextes das zugehörige Kostüm zu ermitteln und in einer weiteren Liste zu speichern ist. Der Zeichenvorgang erfolgt dann wie bereits beschrieben.*

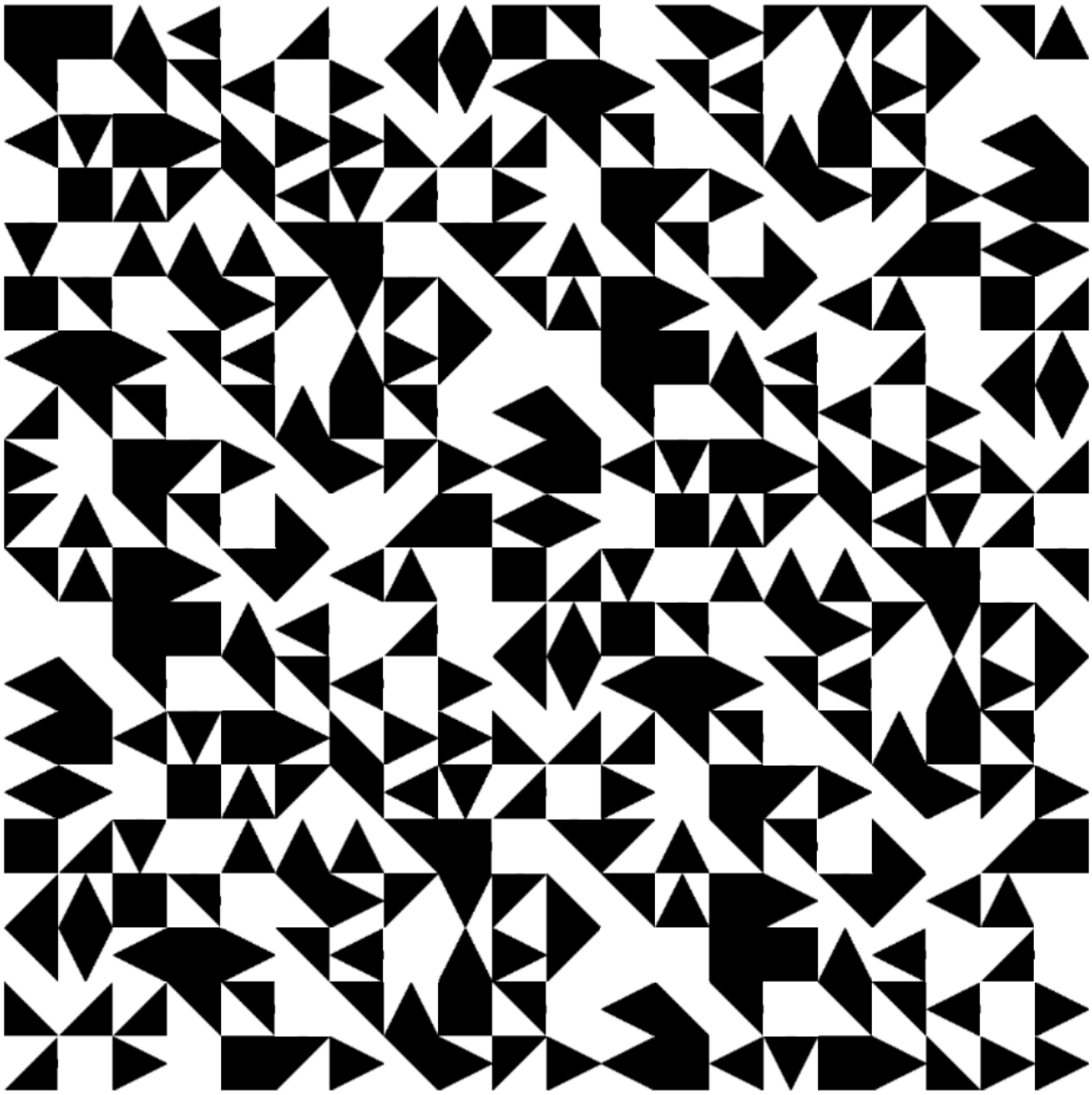
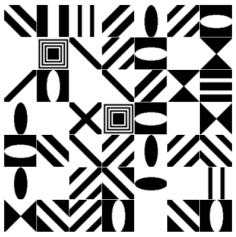


Bild 97: Hommage à Morellet: Répartition aléatoire de triangles suivant les chiffres pairs et impairs d'un annuaire de téléphone

Das Ergebnis wurde mit einem Beamer groß auf Leinwand projiziert. Die Besucher konnten dieses Bild mit Smartphone aufnehmen und so ihre persönliche Hommage à Morellet dokumentieren. Die Abbildung rechts zeigt als Beispiel die Umcodierung des Textes Dies ist ein Probetext für das Buch zur Computerkunst. *Größere Bilder (z.B. mit Texten der Länge 100) benötigen ein wenig Geduld wegen der notwendigen Umcodierungen.*



25.3 Ein kleiner Exkurs: Malen mit Maus und Finger

Anders als bei fast allen bisherigen Projekten wird beim gleich folgenden Beispiel des *Drip-Painting* die Erstellung des Bildes nicht alleine durch Algorithmen gesteuert, sondern erfolgt interaktiv, indem Kleckse nutzergesteuert gesetzt werden können. Die Annäherung an dieses Verfahren soll in zwei Schritten erfolgen. Zunächst werden die Mausbewegungen direkt für die Steuerung und Ausgabe grafischer Komponenten ausgewertet. Im zweiten Schritt wird dies dann mit der Erzeugung der typischen Kleckse in Anlehnung an das Drip-Painting kombiniert.

Für das Malen mit der Maus reicht der Befehl **gehe zu x: y:** bereits aus, wenn wir als gewünschte Position jeweils die durch unsere Bewegung der Maus erzeugte x- bzw. y-Position angeben. Da in der Palette **Fühlen** die Reporter **Maus x-Position** und **Maus y-Position** bereit gestellt werden, reicht bereits ein Zwei-Zeiler zum Malen auf der Bühne. Der Hut-Block **Wenn Taste s gedrückt** dient dabei dazu, den Vorgang kontrolliert zu starten, also z.B. erst dann, wenn sich die Maus am gewünschten Startort auf der Bühne befindet (rechts ein so gezeichnetes Bild):



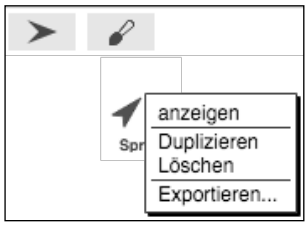


Hinweis: *Mir erscheint es dem normalen Zeichnen eher zu entsprechen, mit gedrückter Maustaste zu zeichnen und bei nicht gedrückter Maustaste die Maus ohne Spur weiter zu bewegen. Zu erreichen ist das, indem entsprechende Abfragen eingeschoben werden, ob die* **Maustaste gedrückt?** *ist - dann* **Stift runter** *- oder ob* **nicht Maustaste gedrückt?** *ist - dann* **Stift hoch**. *Unser Programm wird dadurch nur wenig länger. Damit ist auch ein Unterbrechen des Zeichnens und die Fortsetzung an einer anderern Stelle möglich.*

Dieser kleine Exkurs hat nur mittelbar mit der Computerkunst zu tun: Eine beliebte Erweiterung dieses Malens am Bildschirm ist es, Spiegelbilder am Bildschirm zu erzeugen, wie wir sie von Kaleidoskopen kennen[71]. Sie benötigen dazu mehrere Schildkrö-

[71] Auf der Website http://www.myoats.com/create.aspx können Kaleidoskope mit unterschiedlichsten Effekten interaktiv erzeugt werden. Sie können sich davon zu Erweiterungen des Grundgerüsts inspirieren lassen.

ten, die sich gleichzeitig bewegen und zwar gegenläufig zu Ihren Mausspuren. Dazu muss das **Schildkröten-Sprite** vervielfacht werden; für das Kaleidoskop genau drei mal.





Mit rechtem Mausklick auf das Sprite-Symbol im Objektbereich erhalten Sie die Option **Duplizieren**. Da dabei den so erzeugten Duplikaten **Sprite(2)** usw. der vorhandene Programmcode mitvererbt wird, brauchen Sie in den neuen Objekten jeweils nur beim **gehe zu x: y:** die Reaktion auf Ihre Mausposition verändern, also hier z.B. bei den drei neuen Sprites:

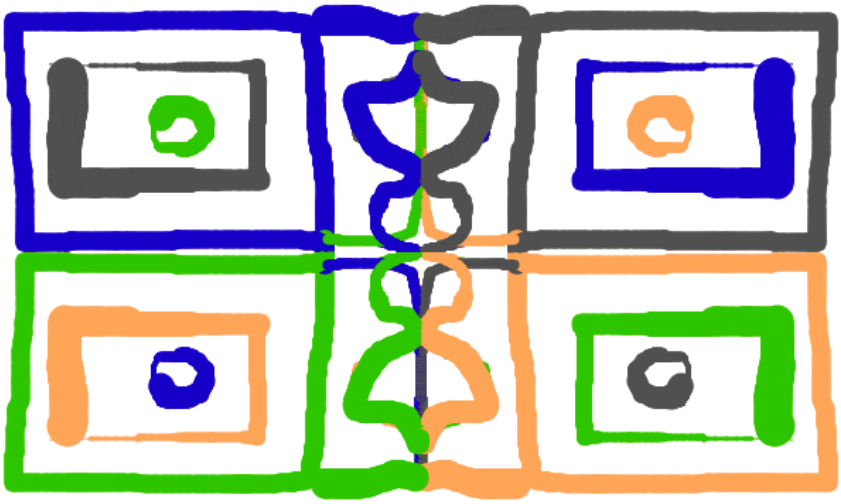
Sprite(2): 

Sprite(3): 

Sprite(4): 

Wenn Sie nun noch mit der Farbe und der Stiftdicke der vier Objekte experimentieren, können Sie vielfältige Kaleidoskopbilder erzeugen:

Wie kann dann, wie im vorigen Bildbeispiel zu sehen, z.B. die Stiftdicke während des Programmablaufs für alle Sprites gleichzeitig geändert werden? Wenn z.B. mit der Taste **d** die Stiftdicke erhöht werden soll (oder andere Ereignisse, die durch Tastendruck ausgelöst werden), kann dies mit **sende Nachricht an alle** verteilt werden, und wenn in den adressierten Sprites dann jeweils das Gegenstück **wenn ich Nachricht empfange** vorhanden ist, dem die dann auszuführende Befehlsfolge angefügt ist.



25.4 Hommage à Jackson Pollock: Drip Painting

Ein dankbares Beispiel für das *Recoding* von Bildern anderer Kunstrichtungen als der Computerkunst sind Bilder à la Jackson Pollock (1912 - 1956), der als Begründer einer eigenen Stilrichtung - des Action Painting - bekannt geworden ist: Er fertigte großformatige Werke im Drip-Painting-Verfahren an, d.h. er trug Farben auf am Boden liegende Leinwände mit großen Pinseln schleudernd oder direkt aus Farbtöpfen tropfend auf[72]. Dieses expressive und unmittelbare Arbeiten war prägend für viele Weggefährten und Nachfolger (wie Alex Katz, Allan Kaprow u.a.).

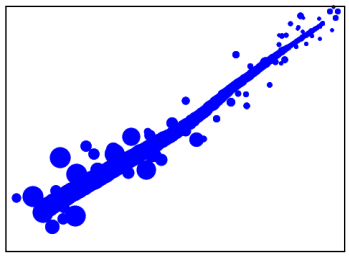
Pollocks Technik, das Spritzen, Tropfen, Schütten und Spachteln, sieht auf den ersten Blick kinderleicht aus. Wohl deshalb gibt es eine Vielzahl von Unterrichtseinheiten für Schulen, in denen Pollocks Bilder als Ausgangspunkt für das Experimentieren der Schülerinnen und Schüler mit Zufallstechniken genommen werden. Aber: *„Noch heute sagen viele: ‚Das kann mein Kind auch' [...] Aber haben Sie es mal probiert? Einen ‚Pollock' zu machen ist wahnsinnig schwer. Es ist unglaublich, wie er die Farben aufgetragen und komponiert hat. Aber erst, wenn man sich in die Bilder, diese Struktur vertieft, erfährt man ihre ganze Komplexität."* (Davidson, 2012).

Seine Bilder sind weit entfernt von „zufälliger Kleckserei", wie zeitgenössische Kritiker meinten, sondern die kontrollierte Anwendung seiner Maltechnik: *„Wenn ich anfange zu malen, habe ich eine Vorstellung von dem, wohin ich gelangen will. Es gibt keinen Zufall."* Der kommt erst durch die Eigenschaften von Farben und Farbträger ins Spiel. Immerhin, dieses Zusammenspiel kann mit einfachen programmtechnischen Mitteln erfahren werden. Da bei Pollock weniger das Endprodukt als der Entstehungsprozess von Bedeutung ist, halte ich hierbei eine interaktive Umsetzung für angemessen, bei der die Entscheidungs- und Gestaltungsmöglichkeiten nicht vollständig an das Programm abgegeben werden.

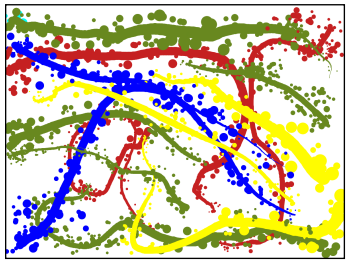
Der Exkurs *Malen mit Maus und Finger* hat die Grundlagen geliefert, wie die Annäherung an das *Drip Painting* erfolgen kann: Der Malvorgang beginnt, wenn die **Maustaste gedrückt?** ist und kann jederzeit unterbrochen werden, wenn **nicht Maustaste gedrückt?** ist. Damit die Strichdicke während des Malvorgangs veränderbar wird, kann eine Variable **strichdicke** eingeführt werden, die durch Tastendruck **wenn Taste ... gedrückt** erhöht oder verringert werden kann (es vereinfacht das Malen, wenn dafür nebeneinander liegende Tasten - z.B. **a** und **s** - verwendet werden). Auf dieselbe Weise kann die **stiftfarbe** gewechselt werden; auch in diesem Fall der Einfachheit halber über nebeneinander liegende Tasten (z.B **1 - 6**). Damit ist das Gerüst gegeben und liefert Ergebnisse wie in nebenstehender Abbildung.



[72] Diese Technik brachte ihm auch den Spitznamen „Jack the Dripper" ein. Auf der Website http://jacksonpollock.org kann interaktiv in seinem Stil mit der Maus gemalt werden.



Es fehlen nur noch die „Spritzer“ neben den Linien. Eine mögliche Lösung dafür ist z.B. das Anlegen eines zweiten Sprites, bei dem bei gedrückter Maustaste Punkte um die aktuelle Mausposition herum gesetzt werden (etwa mit **punktscheibe-rekursivgefuellt um x y radius**). Wenn der **radius** um die aktuelle **stiftdicke** schwankt, kann der Eindruck zunehmender oder abnehmender Farbmengen erzeugt werden.



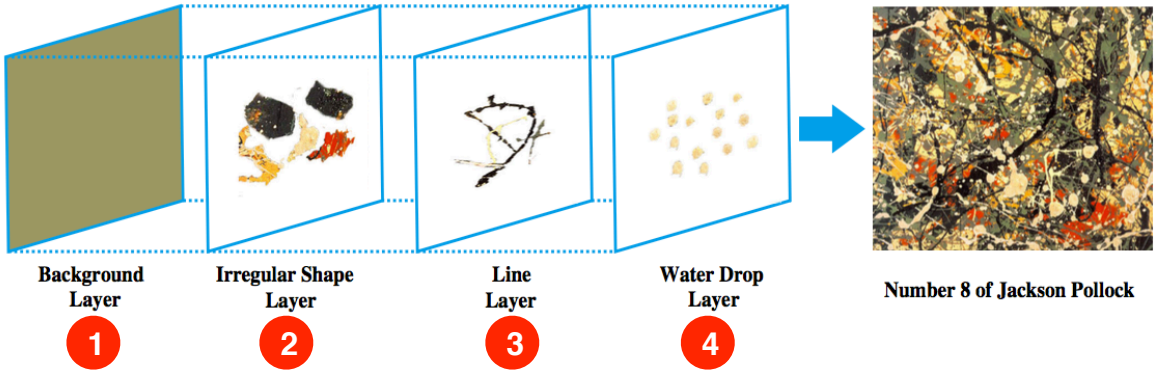
Diese Interaktion erlaubt nun Linien unterschiedlicher Farbe und unterschiedlicher Dicke durch Mausbewegungen zu erzeugen. Mit ein wenig Übung (zum Ändern der Farbe oder Strichdicke kann der Vorgang jederzeit durch Loslassen der Maustaste unterbrochen werden) lassen sich Bilder erzeugen, die an das Vorbild *Wallpaper* von Jackson Pollock erinnern, wie Bild 98. Die Bilder von Jackson Pollock sind also gute Beispiele für die bewusste und gesteuerte Verwendung von Zufallselementen.

Anregung: *Pollock hat auch mit Farbeimern gearbeitet, die mehrere Löcher aufwiesen und so mehrere Farblinien gleichzeitig erzeugten. Auch das kann mit mehreren Sprites, die gegeneinander versetzte Linien zeichnen, nachvollzogen werden. Eine genaue Steuerung, wo welche Linien und Kleckse platziert werden, kann durch die Kombination von Mausbewegung und Tastenkommandos präzise erfolgen (siehe Bild 98, bei dem jeweils zwei Linien unterschiedlicher Farbe nebeneinander versetzt gzeichnet wurden).*

Es gibt allerdings auch Versuche, die Vorgehensweise von Pollock vollständig zu simulieren und vergleichbare Bilder ganz maschinell zu erstellen. Dazu wurden seine Bilder und seine Malweise mehrfach einer genaueren Analyse unterworfen. Dabei wollen Taylor et al. (siehe Taylor, 2002) fraktale Strukturen in Pollocks Bildern entdeckt haben und sprechen von *fraktalem Expressionismus*. Für das *Recoding & Remixing* hat mir dieser Ansatz aber nicht weiter geholfen.

Bei Zheng et al. (2015) bin ich dagegen fündig geworden. Sie verfolgen einen anderen analytischen Ansatz, der zu einem Modell geführt hat, das sich programmtechnisch gut umsetzen lässt. Um den Entstehungsprozess der Bilder bei Pollock zu verstehen und nachzuvollziehen, unterteilen sie ihn in vier Phasen, in denen vier Schichten überlagert werden:

1. Der Hintergrund, einfarbig, verfeinert durch Farbverlauf und Farbspritzer,
2. darüber unregelmäßige, flächige Formen,
3. darüber Linien unterschiedlicher Längen und Dicken, und
4. darüber Punkte und Spritzer unterschiedlicher Größen.



Schichtenmodell zur Modellierung des Drip Painting von Jackson Pollock (aus Zheng et al., 2015)

Die vier Schichten enthalten demnach jeweils unterschiedliche grafische Elemente. Sie können unabhängig voneinander erstellt und mit zufällig angeordneten Komponenten gefüllt werden. Das eigentliche Bild entsteht am Ende durch die Kombination und das Zusammenfügen der vier Schichten[73]. Programmtechnisch bietet dieser Ansatz Vorteile:

- Jede Schicht kann gesondert programmiert und getestet werden.
- Alle Elemente einer Schicht können durch Parameter festgelegt und bei Bedarf leicht angepasst werden.
- Für jede Schicht können mehrere Versionen erzeugt und abgespeichert werden.
- Die Versionen der vier Schichten lassen sich am Ende beliebig kombinieren.

Und so kann die konkrete Umsetzung aussehen:

1. Für den *Hintergrund*, einfarbig, mit leichtem Intensitätsverlauf, reicht eine Schleife, in der waagrechte Linien über die ganze Bühnenbreite gezogen werden und von Zeile zu Zeile die Farbkomponenten (z.B. im RGB-Modell) leicht verändert werden. In einer weiteren Schleife können zufällig „Spritzer“ auf der Fläche verteilt werden (z.B. mit dem Block **Pixel bei x y**). So erzeugte Hintergründe können mit **erstelle Kostuem hintergrund aus stiftspuren** abgespeichert und so eine „Bibliothek“ angelegt werden.

2. Für die nächste Schicht der *Formen* kann der Code für das Klecksbild aus Kapitel 8: *Alles Zufall* ... übernommen werden. Wieder können „Spritzer“ für weitere Unregelmäßigkeiten sorgen. Auch diese Formen können in der Bibliothek angelegt werden.

3. Bei der *Linien*-Schicht können die einzelnen Linien mit einer Schleife erzeugt werden, in der die Richtung zufällig variiert und die Stiftdicke sukzessive erhöht (oder erniedrigt wird). Die Ergebnisse werden wieder in der Bibliothek abgelegt.

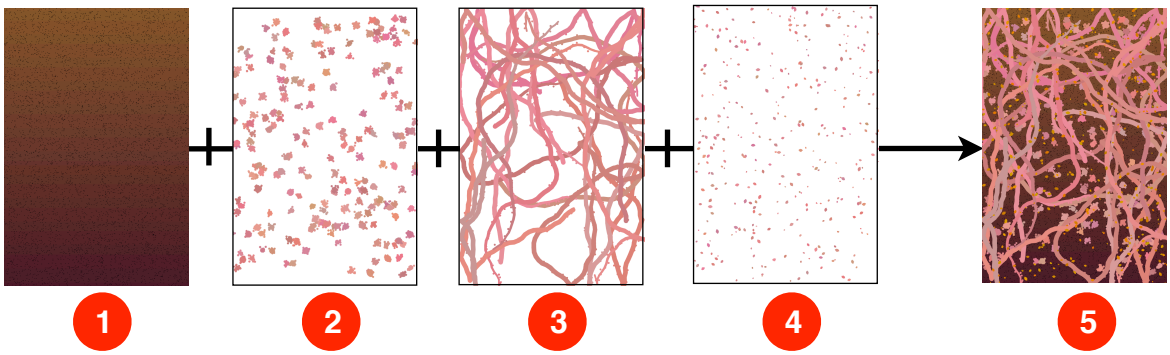
[73] Wie Zheng et al. (2015) betonen, ist ein solches Vorgehen ganz allgemein für das Erstellen abstrakter Bilder verschiedener Stile und Künstler geeignet, weil die einzelnen Algorithmen relativ leicht an verschiedene Stile angepasst werden können.



Bild 98: Hommage à Pollock: No. 4

4 Bei der letzten Schicht können einzelne Pixel oder Kleckse gesetzt und die Ergebnisse in der Bibliothek abgelegt werden.

5 Das eigentliche Bild entsteht dadurch, dass aus der Bibliothek pro Schicht jeweils ein Kostüm ausgewählt und mit **stemple** überlagert wird.



Ein mögliches Ergebnis dieser Vorgehensweise zeigt Bild 99.



Bild 99: Hommage à Pollock: No. 8

Der Ausflug von der Computerkunst in andere stilistische Richtungen soll an dieser Stelle beendet werden. Er sollte zeigen, mit welch teilweise sehr einfachen Mitteln eine Annäherung an die sehr unterschiedlichen Vorgehensweisen und Ergebnisse erreicht werden kann.

Auch mit dem *Recoding & Remixing* der Computerkunst sind wir hiermit an einem vorläufigen Endpunkt angelangt. Weitere Projekte sollten Sie mit dem erarbeiteten Repertoire an Analyse- und Vorgehensweisen sowie dem Befehlsumfang von Snap! ohne größere Schwierigkeiten umsetzen können.

Das folgende Abschlusskapitel dient deshalb einem *Blick über den Tellerrand*, d.h. in ihm sollen einige Ausformungen der aktuellen Medienkunst skizziert werden. Ich hoffe, mit dem dann noch einmal erweiterten Repertoire an aufgezeigten Möglichkeiten steigt Ihre Motivation, eigene Ausdrucksformen zu entwickeln und auszuprobieren.

26. BLICK ÜBER DEN TELLERRAND

Gemessen an den Beispielen der frühen Computerkunst haben Sie in den vorangegangenen Kapiteln bereits ein deutlich breiteres Spektrum an Darstellungsformen kennen gelernt. Die Ergänzungen des Repertoires um Farben, Formen, Animationen und Interaktionen sind von mir immer mit Bezug auf die Vorbilder eingeführt worden. Der Stellenwert der Computerkunst ist heute aber auch - wie bereits im Kapitel 1: *Computerkunst* herausgestellt - in ihrer Rolle als Vorläufer der Medienkunst einzuordnen. Wobei sich der Begriff Medienkunst gleich wieder in zahlreiche aktuelle Richtungen und Ausdifferenzierungen auffächern lässt. Der Hinweis auf einige davon soll in diesem Kapitel genügen, um die weitere Beschäftigung mit spannenden Entwicklungen anzuregen, zumal unser Werkzeug Snap! durchaus geeignet ist, auch in diesen Feldern erste Gehversuche zu unternehmen.

26.1 Mathematik und Kunst

In den Projekten der vorangegangenen Kapitel konnte weitestgehend auf mathematische Formeln verzichtet werden. Es gibt aber auch eine lange Tradition der programmtechnischen Umsetzung mathematischer Modelle und Prinzipien, um auf diese Weise ästhetische Objekte zu erzeugen. Damit gelingt eine fruchtbare Synthese von Wissenschaft und Kunst. Einige solcher mathematischen Phänomene habe ich in dem Büchlein *Programmierung interaktiver Grafiken* (Wedekind, 2014) beschrieben und umgesetzt: Polygone, Spirolaterale, Rekursive Grafiken und L-Systeme.

Das bekannteste Beispiel sind vermutlich die Fraktale[74]. Das sind natürliche und künstliche Gebilde, die selbstähnliche geometrische Muster aufweisen. Es gibt davon unterschiedliche Ausprägungsformen wie die Mandelbrot-Mengen (deren Visualisierung üblicherweise als *Apfelmännchen* bezeichnet wird), Julia-Mengen, Lindenmayer-Systeme und einige mehr. In zahllosen Büchern (mit einem Höhepunkt Ende der 80-er, Anfang der 90er-Jahre) wurde für praktisch alle gängigen Programmiersprachen gezeigt, wie die faszinierenden charakteristischen Bilder erzeugt werden können. Wer die Grundlagen dahinter verstehen möchte, dem kann ich vor allem die beiden Bücher von Peitgen, Jürgens & Saupe empfehlen: *Bausteine des Chaos: Fraktale* (1992) und *Chaos: Bausteine der Ordnung* (1994). Zu vielen der typischen Bildbeispiele liefern die Autoren den Pseudocode mit, teilweise auch die Umsetzung in BASIC-Programmen. Das Autorenteam hat zusätzlich mit Wanderausstellungen ihrer Bilder viel zur Popularisierung beigetragen. Manche bezeichnen das auch als Fraktalkunst.

Natürlich lassen sich vergleichbare Bilder auch mit Snap! erzeugen. Als Anregung und Beispiel kann das *Mira-Gumowski-Fraktal* dienen, das sich mit zwei iterativen Gleichun-

[74] Den Begriff fraktal hat der Mathematiker Benoit Mandelbrot (1924 - 2010) in seinem Kultbuch *Die fraktale Geometrie der Natur* geprägt und bezeichnet damit gebrochene, irreguläre Strukturen, wie wir sie vielfach in der Natur finden: *„Wolken sind keine Kugeln, Berge keine Kegel, Küstenlinien keine Kreise. Die Rinde ist nicht glatt – und auch der Blitz bahnt sich seinen Weg nicht gerade."*

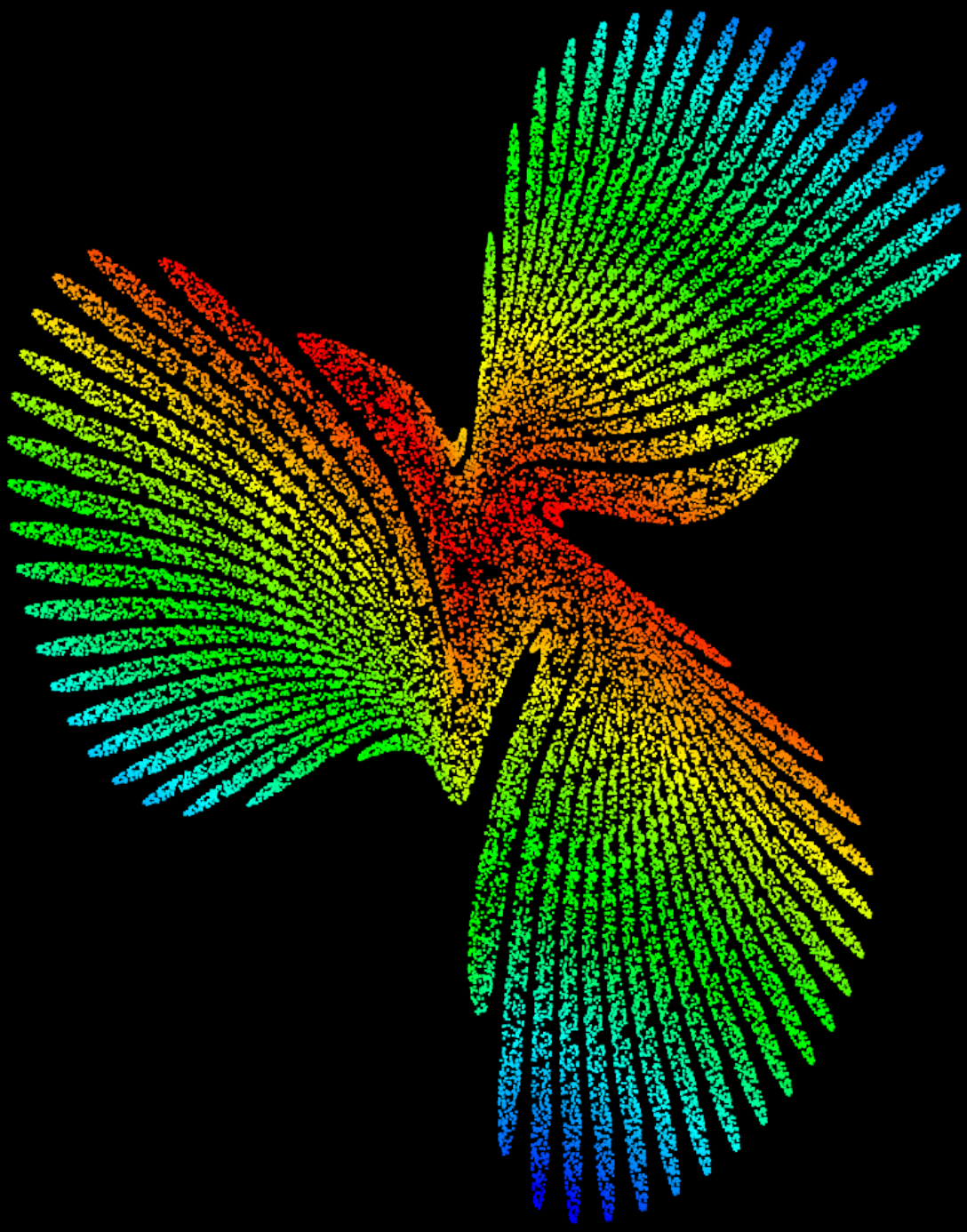


Bild 100: Mira-Gumowski Fraktal

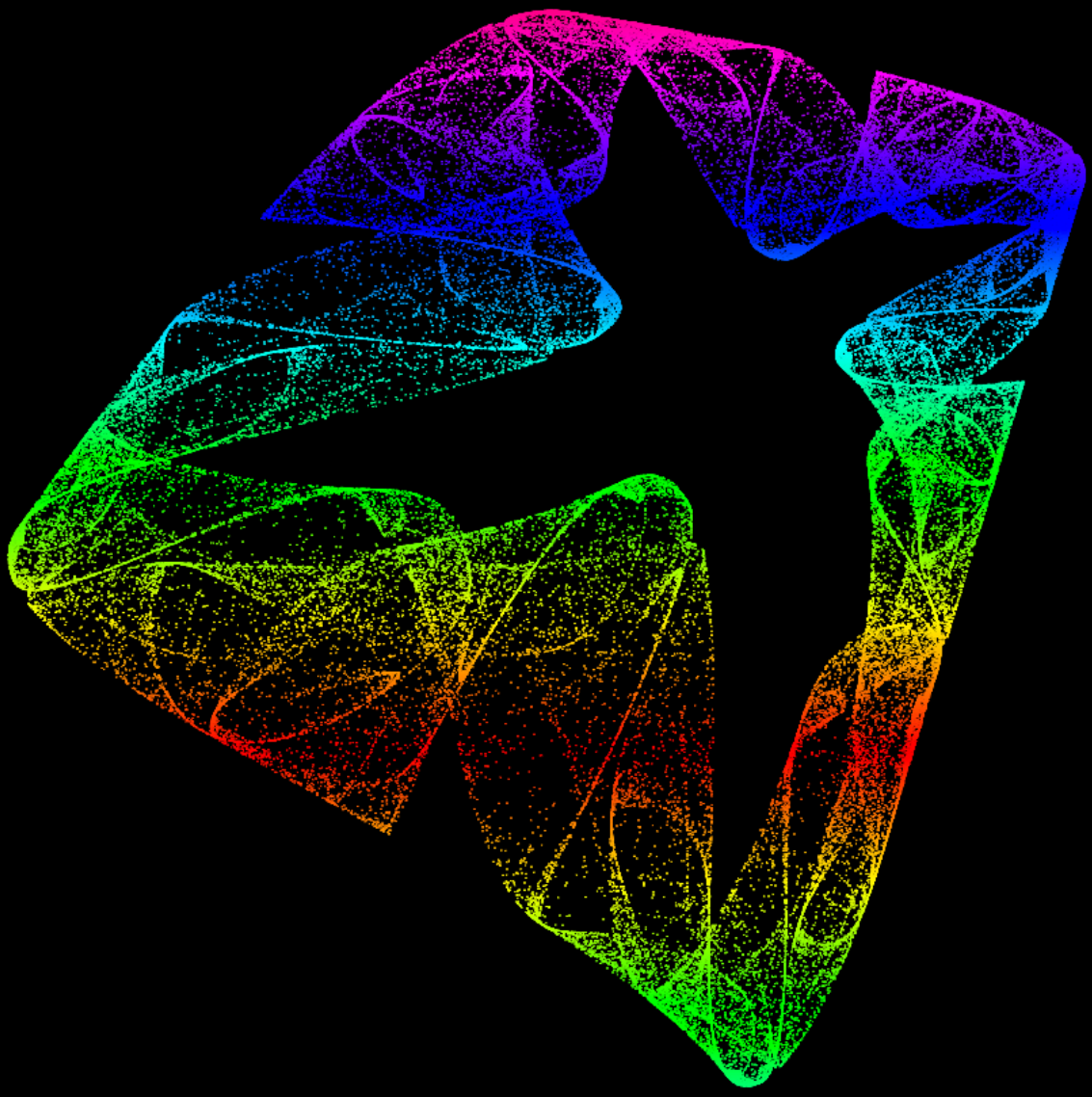
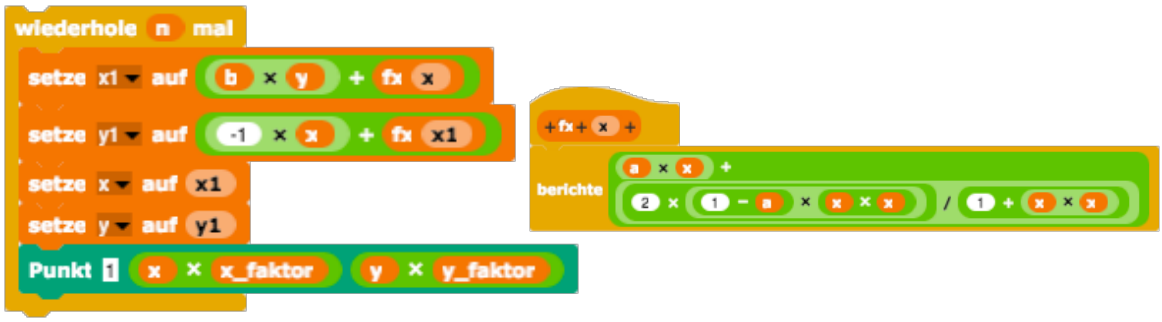


Bild 101: Kaneko Fraktal

gen darstellen lässt (Bild 100). Es ist damit typisch für die rechnerische Behandlung dynamischer Systeme, bei denen die errechneten Werte als Ausgangswerte des jeweils nächsten Rechenschrittes genommen werden:

$$x_{n+1} = by_n + F(x_n)$$
$$y_{n+1} = -x_n + F(x_{n+1})$$
$$F(x) = ax + 2(1 - a)\, x2/(1 + x2)$$



Das ist ein Fall für eine simple Wiederholungsschleife, in der die Werte für x und y jeweils mit Aufruf von **fx** errechnet werden. Durch leichte Variation der Parameter **a** und **b** lassen sich eine verblüffende Zahl sehr unterschiedlicher Muster erzeugen; die Bildserien in Otsubo et al. (2000) zeigen das eindrücklich. Bild 100 zeigt ein typisches Beispiel mit einer federförmigen Struktur.

In ganz ähnlicher Weise lassen sich eine Vielzahl *seltsamer Attraktoren*[75] berechnen und darstellen. Zu den bekanntesten gehören u.a. der *Hénon*-Attraktor, der *Gingerbread*-Attraktor oder der *Pickover*-Attraktor[76]. Bild 101 zeigt z.B. den *Kaneko*-Attraktor.

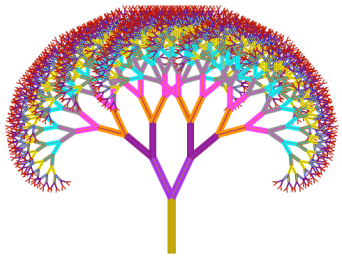
Zwar sind Fraktale prominente Vertreter der Verbindung von Mathematik und Kunst, aber beileibe nicht die Einzigen. Ob es um das Simulieren dynamischer Systeme, das Lösen von Parkettierungsproblemen oder die Visualisierung komplexer Daten geht, häufig ergeben sich überraschende und ansprechende Bilder. Einen guten Einstieg in die Verbindung von Kunst und Mathematik bilden die Tagungsbände der Bridges Konferenzen, die für die Jahre 1998 bis 2017 online zugänglich sind, sowie die Zeitschrift von ISAMA HYPERSEEING.

[75] Julien Sprott hat den seltsamen Attraktoren ein ganzes Buch gewidmet: *Strange Attractors: Creating patterns in Chaos*.

[76] Auf der Website von Jürgen Meier finden sich Tutorials mit der Beschreibung dieser und vieler weiterer Attraktoren und zugehörige Bildbeispiele: http://www.3d-meier.de

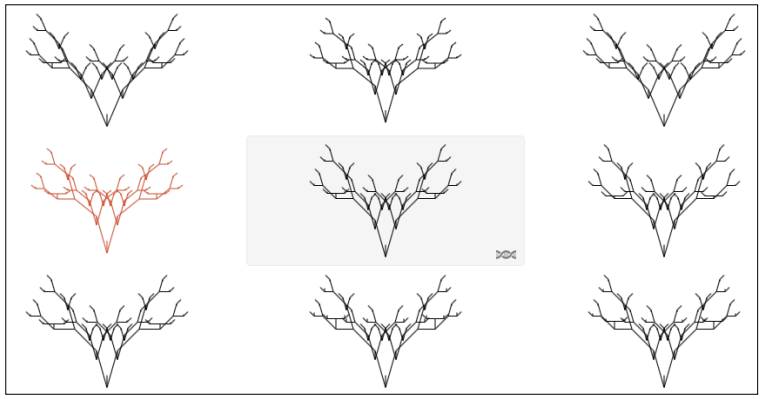
26.2 Biologie und Kunst

Auch aus der Verbindung der Mathematik mit der Biologie sind ästhetische Objekte und eigene Kunstanwendungen entstanden. Ihre Ergebnisse kennen wir u.a. aus vielen Filmen, in denen uns synthetische Pflanzen und ganze virtuelle Landschaften gezeigt werden, oft kaum noch unterscheidbar von Realbildern. Zugrunde liegen unterschiedliche Modelle. Begonnen hat es mit regelbasierten Ersetzungssystemen von Aristid Lindenmayer (1990). Sie sind geeignet, botanische Verzweigungsstrukturen zu beschreiben. Konkret werden dabei Zeichenketten abgearbeitet, die aus einfachen Befehlen zusammengesetzt sind, z.B.: V = Vorwärtsbewegung, P = Drehung nach links, M = Drehung nach rechts. Da ihre Interpretation mit der Schildkrötengrafik kompatibel ist, kann das Regelsystem sehr einfach umgesetzt werden. So kann etwa ein *Initiator* V (also eine einfache Vorwärtsbewegung) durch einen *Generator* VPVMMVPV (einen „Umweg“ mit einer Dreiecksspitze) ersetzt werden. Dies erzeugt dann die Koch-Kurve (dieses Beispiel wurde bereits im Kapitel 13: *Ein Block ist ein Block* ... gezeigt).



Interessanterweise können mit DOL-Systemen (sogenannten *deterministisch kontextfreien L-Systemen;* Näheres dazu in Deussen, 2003, S. 70) die gleichen busch- und baumartigen Figuren erzeugt werden, wie sie typischerweise auch als fraktale Objekte erhalten werden.

Bei der evolutionären Kunst (Romero & Machado, 2008) werden die ästhetischen Objekte mit Algorithmen erzeugt, die Prinzipien der natürlichen Evolution nachahmen. Der Ansatz geht auf den Evolutionsbiologen Richard Dawkins zurück, der in seinem Buch *Der blinde Uhrmacher* (1987) ein Programm beschrieben hat, mit dem er *Biomorphe*, d.h. biologisch anmutende Formen, erzeugen konnte. Er wollte damit die Leistungsfähigkeit der Darwinschen Evolutionstheorie veranschaulichen, denn in seinem Programm wirken nur die zufällige Veränderung von Genen (Mutation) und kumulative Selektion. Er konnte so aus sehr einfachen Grundformen in erstaunlich wenigen Durchläufen komplexe Gebilde „züchten“.

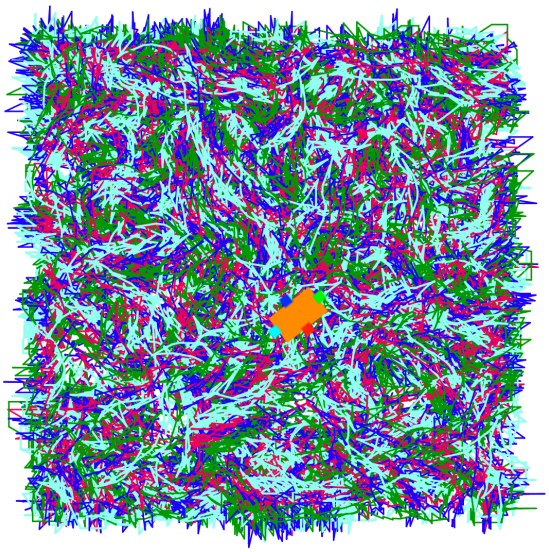


Es gibt inzwischen eine Reihe aktueller Programmversionen, manche laufen direkt im Webbrowser, sogar in einer von Dawkins autorisierten aktuellen Fassung. Alle diese Programmversionen sind interaktiv, denn die Betrachter müssen von Generation zu Generation die Selektion des für die weitere Vermehrung vorgesehenen Biomorphs selbst vornehmen[77].

Der amerikanische Künstler und Informatiker Karl Sims liess 1993 in seiner Installation *Genetic Images* diesen Selektionsvorgang durch die Besucher dadurch ausführen, dass sie vor das von ihnen gewählte Bild traten und durch Sensoren dieses zur Reproduktion der nächsten Bildfolge bestimmt wurde[78].

26.3 Malmaschinen und Malroboter

Malmaschinen - ohne Computersteuerung - zur automatischen Erzeugung von Kunstwerken haben schon eine längere Tradition[79]. Die *Méta-Matic*-Automaten des Schweizer Malers und Bildhauers Jean Tinguely (1925-1991) gehören zu den Klassikern dieser Maschinen (siehe dazu Stahlhut u.a., 2008). Sie variieren in phantasievollen Arrangements das Zeichnen überlagerter Kurven, wie wir das von den Lissajous-Figuren kennen (siehe dazu Kapitel 10.1: *Hommage à Laposky*). Seine damit erzeugten Bilder passen in die abstrakt-expressive Malerei seiner Zeit.



Eine lustige, ohne Programmierarbeit funktionierende Vorstufe solcher Malmaschinen sind die *Vibrobots* (Bauanleitung z.B bei Schön, Ebner & Narr, 2016, S. 120), also kleine Maschinen, die durch Vibrationsmotoren (genauer: mit Unwuchtmotoren, wie man sie z.B. in Handys oder Zahnbürsten findet) in Bewegung versetzt werden. Werden sie zusätzlich mit Farbstiften versehen, entstehen Zufallsbilder, die ein wenig an die Bilder der Méta-Matic-Automaten von Jean Tinguely erinnern (die nebenstehende Abbildung ist aus einer Vibrobot-Simulation mit Snap!).

[77] Das Beispiel in der Abbildung wurde erzeugt bei http://www.emergentmind.com/biomorphs.

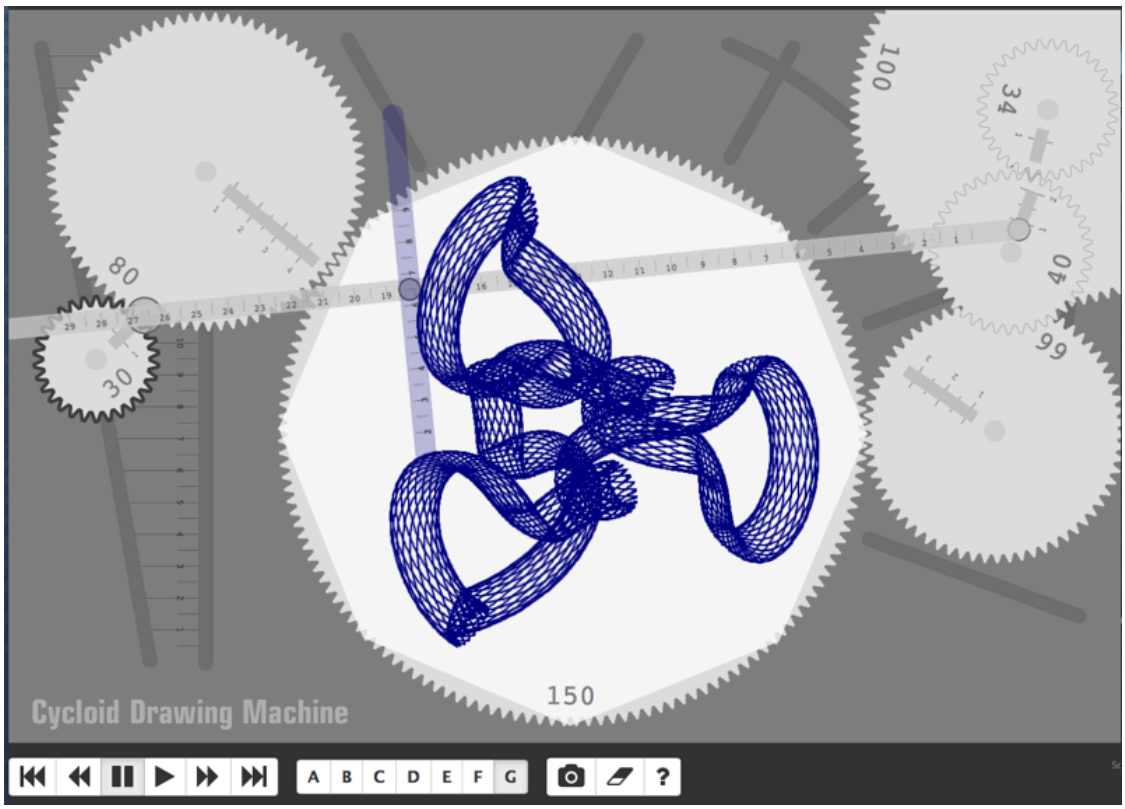
[78] Die Installation wurde mehrfach gezeigt: Centre Georges Pompidou in Paris, Ars Electronica in Linz, und beim Interactive Media Festvial in Los Angeles, beschrieben unter: http://www.karlsims.com/genetic-images.html

[79] einige Beispiele - manche davon ziemlich skurril - finden sich unter http://cyberneticzoo.com/tag/painting-machine/ bzw. http://cyberneticzoo.com/category/robots-in-art/

Wenn gezielter grafische Strukturen erzeugt werden sollen, steigt natürlich der Aufwand. Vom dänischen Künstler Eske Rex gibt es ziemlich monumentale Malmaschinen, zwar mit wenigen Einstellmöglichkeiten (es handelt sich einfach um große Doppelpendel), die dafür aber großdimensionierte Bilder produzieren können[80]. Das reizt eventuell zum Selbermachen im Kleinen; machbar ist sowas etwa mit zwei gekoppelten Plattenspielern.

Der (Buch)Designer Joe Freedman, der auch optische Spielzeuge und Aufklapp- Bilderbücher gestaltete, hat wunderschöne Malmaschinen aus Holz gefertigt. Das reicht vom klassischen Harmonographen bis zur komplexesten, der *Cycloid Drawing Machine*. Leider sind diese über *kickstarter* finanzierten Produkte inzwischen alle vergriffen. Wayne Schmidt hat einen begeisterten Video-Review der Maschine verfasst. Er demonstriert sowohl die mechanische und designerische Qualität als auch die enorme Brandbreite an Variations- und damit Gestaltungsmöglichkeiten.

Diese Maschine bietet nun wieder einen konkreten Anknüpfungspunkt zu unseren Programmierprojekten, denn Jim Bumgardner hat eine Simulation der Cycloid Drawing Machine mit Hilfe von *Processing* programmiert. Da er das mit Kenntnis und Unterstützung von Joe Freedman gemacht hat, ist eine frappierend realistische Umsetzung gelungen. Am besten sollten Sie das selber im Webbrowser ausprobieren[81].



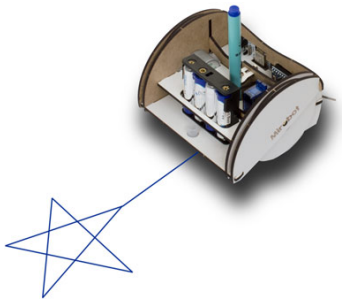
[80] Eske Rex präsentiert zwei Versionen: Drawing machine (http://www.eskerex.com/?p=464) und Drawing Machine #2 (http://www.eskerex.com/?p=1497). Das Beispiel der gekoppelten Plattenspieler zeigt Robert Howsare: http://roberthowsare.com/rational-aesthetics/drawing-apparatus/

[81] http://wheelof.com/sketch/

Die nächste Stufe, nämlich *programmgesteuerte Malroboter*, gibt es auch schon lange - für die Programmiersprache Logo in Form von ansteuerbaren Bodenturtles (siehe dazu im Kapitel 1: *Computerkunst* den Abschnitt 1.3: *Vom Plotter zum Zeichenroboter*). Ein aktueller Nachfolger ist Terrapins Pro-Bot. Er kann mit Stiften bestückt werden und damit seine Spuren zeichnen.

Man kann natürlich auch versuchen, eine Bodenturtle selber zu bauen. Josh Burker hat das auf der Basis eines Einplatinencomputer gezeigt[82].

Weniger Bastelarbeit erfordern Bausätze, die nur einen mechanischen Aufbau benötigen. Mit dem (nicht ganz billigen) *mDrawbot Kit* können gleich vier unterschiedliche Zeichenroboter gebaut werden. Programmierbar sind sie mit dem Scratch-Abkömmling mBlock. Eine preisgünstigere Alternative und mein persönlicher Favorit ist der *Mirobot*. Den mag ich besonders, weil er relativ leicht zusammen zu bauen ist (auch wenn die Zeichengenauigkeit für unsere Zwecke nicht immer optimal ist) und neben Blockly, Javascript, Python oder Scratch auch mit Snap! programmierbar ist.



26.4 Software-Werkzeuge für die Kunstproduktion

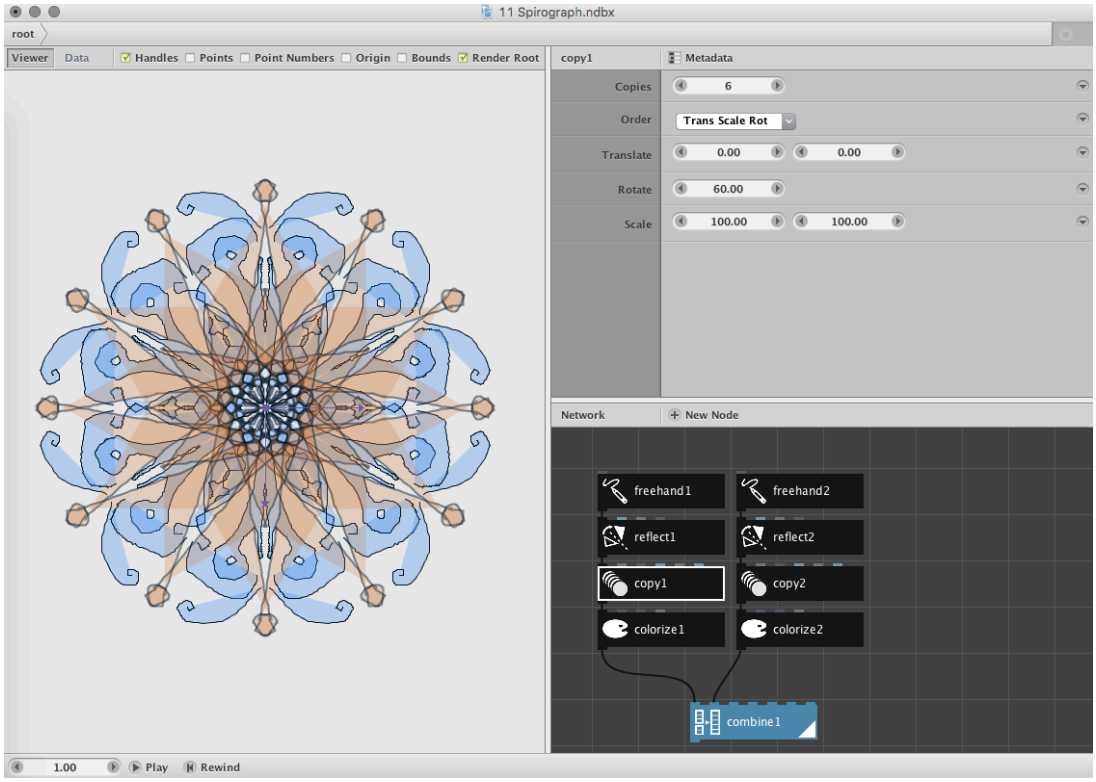
Die Gründe für die Wahl von Snap! als das Werkzeug für *Recoding & Remixing* habe ich im Kapitel 1: *Computerkunst* genannt. Ich möchte an dieser Stelle aber darauf hinweisen, dass es eine ganze Reihe weiterer leistungsfähiger Werkzeuge gibt, die sich besonders an Designer und Künstler richten. Sie zeichnen sich dadurch aus, dass sie spezielle Befehle zur Erzeugung geometrischer Objekte anbieten (z.B. für Polygone, Ellipsen, Freihandzeichnen u.ä.), die wir in Snap! großteils erst selbst erzeugen mussten. Auch die Behandlung von Farben, Zufall und Ebenen bieten oft weitergehende Unterstützung bei der Erzeugung Generativer Kunst und Animationen. Ich beschränke mich im Folgenden auf die exemplarische Nennung einiger solcher Systeme, die frei (kostenlos) zugänglich sind.

Für Einsteiger gut geeignet ist *NodeBox*, denn es bietet visuelle Programmierelemente (sog. *Nodes*). Die Nodes können grafische Grundelemente generieren wie z.B. Linien, Rechtecke oder Füllfarben. Eigene Erweiterungen, d.h. nutzerspezifische Nodes, können mit textuellen Ergänzungen in der Programmiersprache Python hinzugefügt werden. NodeBox eignet sich dadurch sowohl für schnelle Grafiktests als auch für komplexere Anwendungen, insbesondere Datenvisualisierungen und Animationen. Einen ganz ähnlichen Ansatz verfolgen die Entwickler von *vvvv*[83]. Auch in dieser Program-

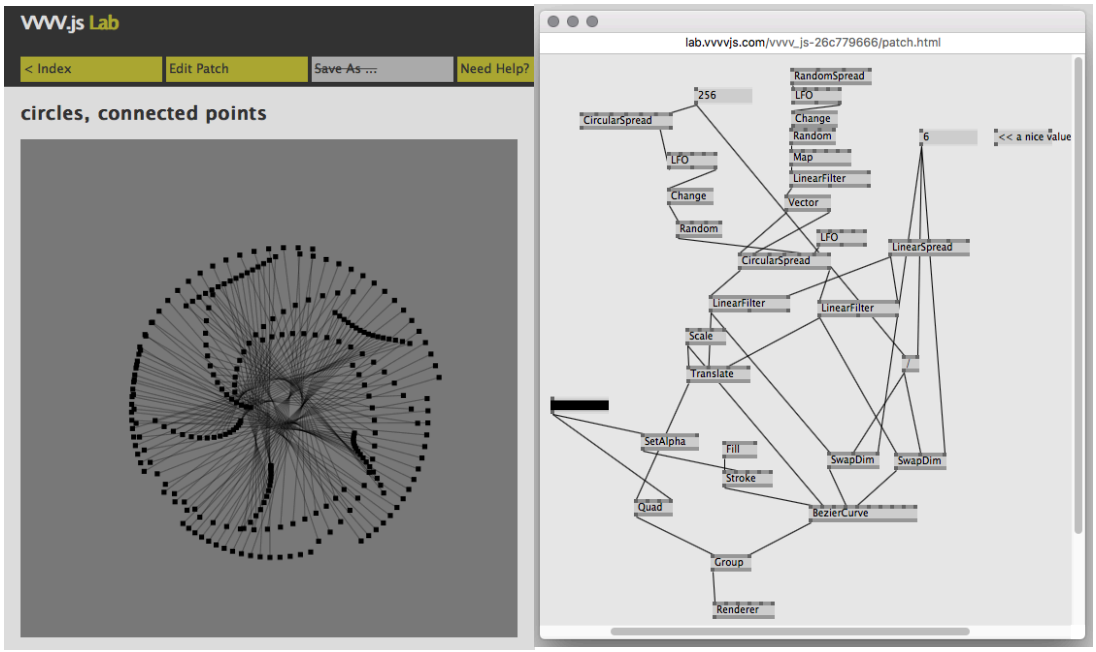
[82] Josh Burker gibt dafür eine detaillierte Bauanleitung: http://archive.monograph.io/joshburker/logoturtle

[83] Das Programm setzt das Betriebssystem Windows voraus: https://vvvv.org/. Es gibt aber auch eine Javascript-basierte Version (noch im Teststadium), die im Browser läuft: http://lab.vvvvjs.com

mierumgebung kann kombiniert visuell (mit ebenfalls *Nodes* genannten grafischen Objekten) und textuell codiert werden.



NodeBox: Generierung eines Spirograph-Musters



VVVV.JS: Erzeugung von Mustern gekoppelter Punkte

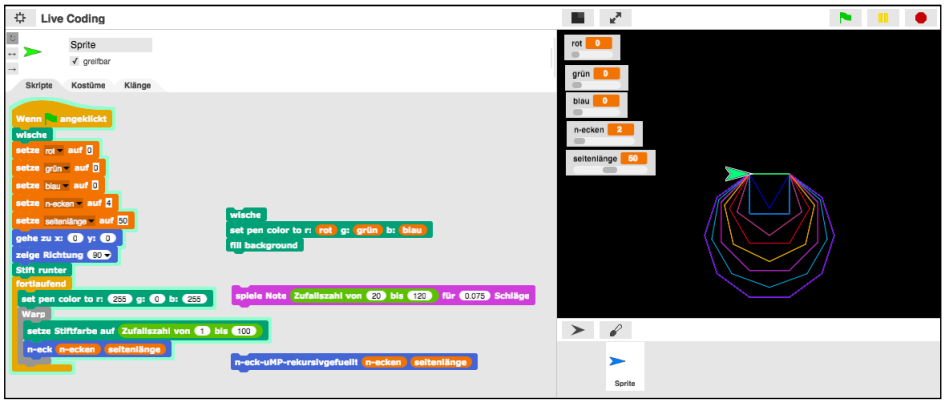
Als Schwerpunkt wird das Arbeiten mit Audio- und Videosignalen und die Verarbeitung von Sensordaten genannt. Da vvvv immer in Echtzeit läuft, d.h. direkt auf alle

Programmeingaben reagiert, ist es für *Live-Coding* geeignet: Dabei werden durch die Entwicklung bzw. Veränderung des Programmcodes live grafische und musikalische Ereignisse erzeugt bzw. beeinflusst.

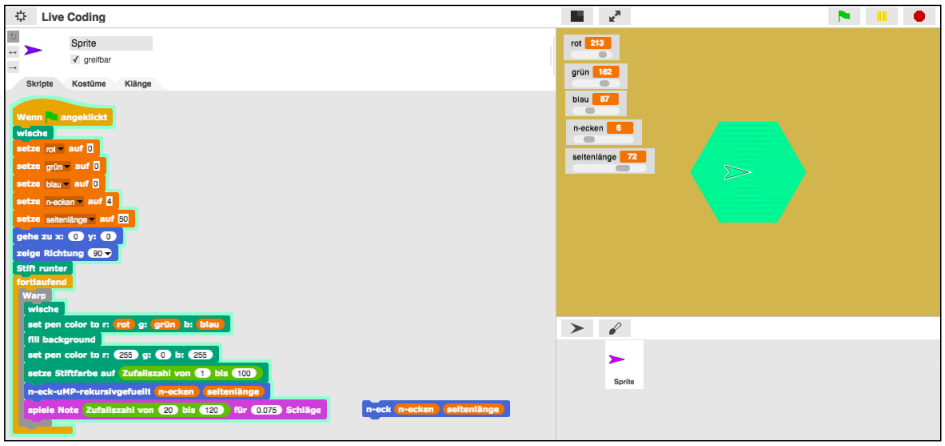
26.5 Exkurs: Live-Coding mit Snap!

Vielleicht sind Sie ja schon zufällig darüber gestolpert, dass Live-Coding auch in Snap! möglich ist, z.B. wenn Sie zufällig bei laufendem Programm einen Zahlenwert verändert haben - und dann bemerkten, dass das Programm eine entsprechend veränderte Reaktion zeigte. Es ist tatsächlich so, dass Snap! unmittelbar auf alle Eingaben reagieren kann. Nützlich ist dafür die Änderung der Werte über Schieberegler: Sie können bei den auf der Bühne eingeblendeten Variablen mit der rechten Maustaste ein Kontextmenü öffnen, dort den Regler aktivieren und zulässige Minimal- und Maximalwerte festlegen.

Live-Änderungen sind aber nicht auf Zahlenwerte beschränkt. Sie können auch im laufenden Programm zusätzliche Befehle einsetzen oder herausnehmen und damit die Funktionalität verändern. Im folgenden Beispiel zeigt die obere Abbildung, wie die Variable **n-ecken** über ihren Schieberegler verändert wird, was zu den unterschiedlichen Polygonen führt. Die untere Abbildung zeigt, wie weitere Befehle im laufenden Programm ausgetauscht bzw. eingefügt wurden und nun zu unterschiedlich großen gefüllten Polygonen führen, wobei nun zusätzlich auch noch Noten abgespielt werden:



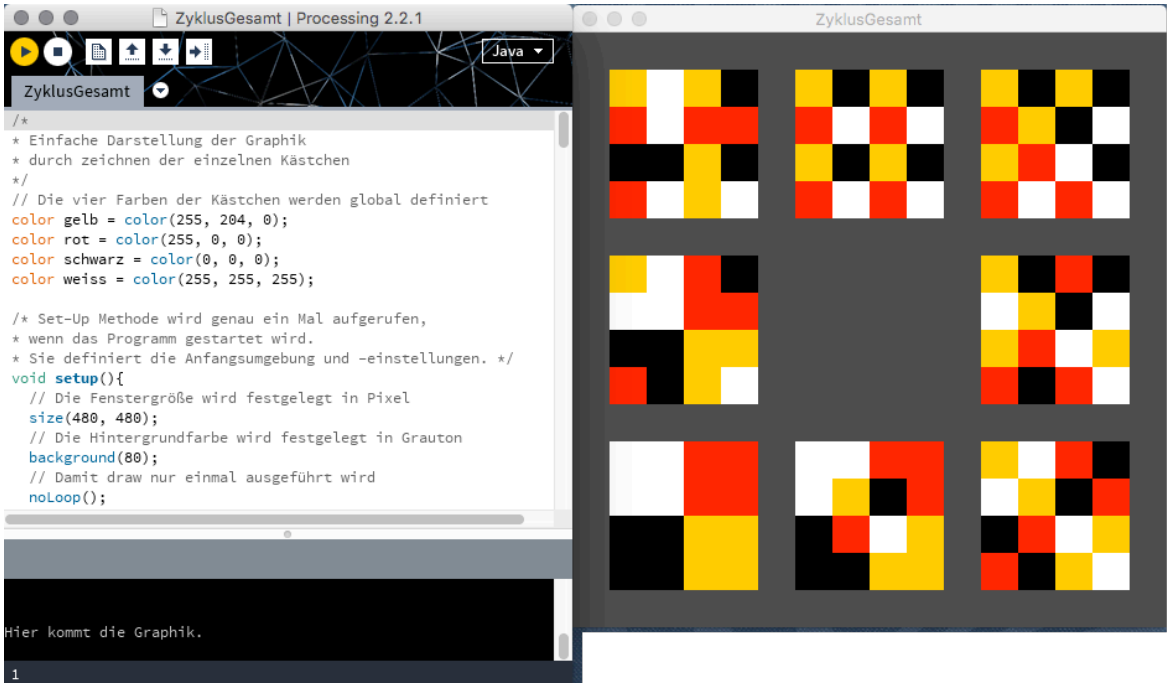
Live-Coding mit Snap!: Änderungen über Schieberegler



Live-Coding mit Snap!: Befehlsergänzungen im laufenden Programm

Die dritte hier zu nennnende Programmierumgebung *Processing* ist eigentlich die wichtigste: Diese (textuelle) Programmiersprache[84] wurde 2001 von Casey Reas und Ben Fry entwickelt - damals Studenten von John Maeda - und war maßgeblich beeinflusst von dessen Sprache *Design by Numbers* (DBN). Ihr Sprachentwurf richtete sich gezielt an Designer und Künstler, weshalb sie sich besonders auf Sprachelemente für Grafik und Interaktion konzentrierten. Die Zielgruppe wird folgerichtig ausdrücklich in mehreren Buchpublikationen zu Processing angesprochen: *Processing: A Programming Handbook for Visual Designers* (Reas & Fry, 2014), *Learning Processing: A Beginner's Guide to Programming Images, Animation, and Interaction* (Shiffman, 2015), *Processing for Visual Artists: How to Create Expressive Images and Interactive Art* (Glassner, 2010), *Generative Art: A Practical Guide Using Processing* (Pearson, 2011) oder *Generative Gestaltung* (Bohnacker et al., 2009).

Da Processing auf der Programmiersprache Java aufsetzt, läuft es auf allen gängigen Plattformen[85]. Es bietet eine reduzierte Entwicklungsumgebung und ein Grafikfenster, in dem das Programm (dort *Sketch* genannt) abläuft. Das entscheidende Merkmal sind die vielen mächtigen Befehle für 2D- und 3D-Elemente, Typografie, Farben und Lichteffekte.



Processing: Karl Hermanns „Zyklus nach Max Bill" (Programm von Kerstin Bub, 2014)

[84] Alle Informationen zu Processing (Downloads, Sprachreferenz, Tutorials und Ergänzungen) sind zu finden unter: https://processing.org/

[85] Inzwischen gibt es auch zwei Implementierungen mit Javascript, die direkt im Browser ablaufen: p5js.org bzw. http://processingjs.org/. Die Unterschiede dieser konkurrierenden Versionen liegen im Detail und werden hier beschrieben: https://www.sitepoint.com/processing-js-vs-p5-js-whats-difference/

Zu Processing hat sich mittlerweile eine riesige, weltweite Nutzergemeinschaft gebildet, die ihre Programmbeispiele und Programmcodes auf diversen Plattformen zugänglich macht. Einen Einstieg bietet die Sammlung bei *processing.org* (Menüpunkt *Exhibition*). Eine zweite Anlaufstelle ist *OpenProcessing*, eine Plattform, die sich ganz der Sammlung und Diskussion von Processing-Sketches gewidmet hat. Damit bietet sich Anfängern (aber durchaus auch Fortgeschrittenen) ein fast unerschöpfliches Reservoir an Beispielen, von denen gelernt werden kann und die Ausgangspunkt für das eigene *Remixen* werden können.

27. FAZIT UND AUSBLICK

In den vorangegangenen Kapiteln wurden viele Beispiele der frühen Computerkunst (und anderer Kunstrichtungen) vorgestellt und durch Recoding & Remixing nachvollzogen und weiter entwickelt. Die Beispiele stellen natürlich meine subjektive Auswahl dar. In der Literatur sind viele weitere interessante Projekte zu finden, insbesondere bei Franke (1971, 1984), IBM (1978), Steller (1992) oder Rosen (2011). Dort und inzwischen auch in vielen Online-Quellen können Sie herausfordernde Anregungen finden.

Für das Recoding & Remixing wurden einige grundlegende informatische Konzepte vorgestellt und mit entsprechenden Sprachelementen von Snap! bei der Realisierung der grafischen Beispiele konkret eingesetzt, angefangen bei der Wiederholung, Modularisierung und Erweiterbarkeit, bis hin zu Prozeduren als Daten.

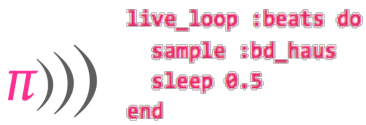
Damit haben im Rahmen dieser Einführung allerdings nur solche Konzepte und Sprachelemente Eingang gefunden, die für die Realisierung der ausgewählten Projekte sinnvollerweise eingesetzt werden können. Snap! bietet etliche weitere leistungsfähige Konstrukte, die im Snap!-Manual knapp skizziert werden. Ihr adäquater Einsatz erfordert einen vertieften Einstieg, um die sich damit ergebenden Möglichkeiten zu verstehen und zu nutzen. Leider gibt es noch kein entsprechendes Lehrbuch. Ich empfehle deshalb den ausgezeichneten und frei zugänglichen Online-Kurs BJC - The Beauty and Joy of Computing. Erfreulicherweise finden sich dort an vielen Stellen auch Beispiele zu Digital Art und Computerkunst.

Wer sich bis zu diesem Abschlusskapitel durchgearbeitet hat, stellt sich vielleicht die Frage, wie weiter? Doch dafür bietet sich ein weites Feld, die Medienkunst!

27.1 Medienkunst: Interaktion & Multimedia & Vernetzung

Wenn ich im Eingangskapitel die frühe Computerkunst als Vorläufer der Medienkunst bezeichnet hatte, dann muss an dieser Stelle eine Präzisierung folgen, was denn darunter zu verstehen ist. Medienkunst bezeichnet alles künstlerische Arbeiten, das sich der Medien bedient. Damit ist der Begriff genauso breit wie der Medienbegriff selbst, beinhaltet also das künstlerische Arbeiten mit Fotografien, Filmen und Videos genauso wie die Verwendung digitaler Medien. Da uns besonders der letzte Aspekt und damit der Einfluss digitaler Technologien auf die Produktion und Rezeption von Kunst interessiert, ist es naheliegend wie Bruce Wands (2006) von *Kunst im digitalen Zeitalter* (Art of the Digital Age) zu sprechen.

Die damit verbundenen neuen Entwicklungen sind nicht zu trennen von den - verglichen mit den 60er-Jahren - drastischen Verbesserungen der inzwischen verfügbaren Werkzeuge und Ausgabemedien. Das erlaubt etwa ganz selbstverständlich auch farbige Darstellungen in hoher Auflösung. Somit sind deutliche Varianten und Erweiterungen der frühen Computerkunst und anderer Werke realisierbar:

Farben	
Flächen	
Ausgabe Bildschirm und Alternativen	
Animation	
Interaktion (Eingabe, Steuerung)	
Sensoren	
Zeichenroboter	
Live Coding	

Mit dem Einbezug von *Farben* (Kapitel 7: *Alles schön bunt hier* ...), der Bearbeitung *flächiger Elemente* und auch mit dem Übergang zur *Animation* (Kapitel 21: *Alles in Bewegung*) werden neue Darstellungsformen erschlossen. Die Ausgabe der Programmergebnisse ist nicht mehr auf Papier beschränkt. Große Monitore erlauben höchstaufgelöste, farbige statische und animierte Darstellungen. Als Ausgabemedium können sogar ganze Häuserfronten dienen, sei es als Projektionsflächen oder mit programmgesteuerten Farbfenstern, wie es das Projekt Arbolet frontage eindrucksvoll zeigen konnte.

Die *interaktive Änderung* von Kenngrößen durch die Betrachter sowie die Auswertung von *Sensoren* und die direkte Reaktion der Programme darauf verwischen zunehmend die Trennung zwischen der technischen Abarbeitung von Algorithmen und der individuellen Gestaltung eines Werkes durch den Künstler und/oder Betrachter. Mit erschwinglichen Zeichenrobotern ist sogar die Bodenturtle zurückgekehrt, die heute eine neue Rolle in interaktiven Installationen übernehmen kann.

Besonders offensichtlich werden die neuen Interaktionsformen beim *Live Coding*, bei dem Änderungen am Programmcode direkte Auswirkungen zeitigen.

Es ist deshalb abzusehen, dass

- durch den niedrigschwelligen Zugang zu ihrer Programmierung,

- durch die Erweiterung der Darstellungsmöglichkeiten und
- durch die Kopplung mit *Sensoren* und *Zeichenrobotern*

der Übergang von bildbezogenen Computerkunst-Programmen zur *Medienkunst für jedermann* möglich wird.

Auch Frieder Nake - der Pionier der Computerkunst - sieht die Zukunft der algorithmischen Kunst darin (Nake, 2014):

„In jedem Fall muss das Bild in Bewegung sein: Ob als dynamische Installation oder durch Interaktion mit der Umwelt, das ist egal. Das statische Bild ist nicht auf der Höhe des Mediums algorithmische Kunst. [...] In der weiteren Erforschung des Urproblems – Form und Farbe, die nur zusammen aufreten können – da glaube ich, wird die Zukunft liegen."

Für mich ist das *Recoding & Remixing* der Computerkunst und die Entwicklung multimedialer Installationen heute nicht mehr nur spielerisches Experiment, auch wenn der Einstieg mit niederschwelligen Werkzeugen wie Snap! zunächst einen solchen Eindruck provozieren mag. Meine eigenen Erfahrungen haben gezeigt, dass damit auch qualitativ hochwertige Werke möglich werden, die ihren berechtigten Platz in Ausstellungen und auf (Multimedia-) Festivals finden können.

Für Ihr eigenes Engagement in diesem spannenden und dynamischen Feld wünsche ich Ihnen viel Freude und Erfolg!

LITERATUR

Abelson, H. (1983). Einführung in Logo. München: IWT-Verlag.

Arnone, W. (2014). Geometrie für Dummies. Weinheim: Wiley-VCH.

Böcker, H.-D., Fischer, G. & Plehnert, M. (1986). Interaktives Probemlösen mit LOGO Band 1: Einführung in das interaktive Programmieren. München: IWT-Verlag.

Böcker, H.-D., Fischer, G. & Schollwöck, U. (1987). Interaktives Probemlösen mit LOGO Band 2: Praktische Projekte (Mathematik, Informatik, Künstliche Intelligenz und Sprache, Spiele). München: IWT-Verlag.

Beyls, P. (2014). Simple Thoughts. Gent: AsaMER.

Bohnacker, H., Groß, B., Laub, J. & Lazzeroni, C. (Hg.) (2009). Generative Gestaltung. Entwerfen, Programmieren, Visualisieren. Mainz: Verlag Hermann Schmidt.

Boytchev, P. (2014). Logo Tree Project. Rev. 2.09. Download unter http://www.elica.net/download/papers/logotreeproject.pdf

Breen, D. (2016). Erste Schritte mit Scratch für Dummies Junior. Weinheim: Wiley-VCH Verlag.

Brill, B. (2002). The Endless Wave. in Sarhangi, R. (ed.). Bridges: Mathematical Connections in Art, Music, and Science. pp. 55-66. Bridges Conference. Download unter http://archive.bridgesmathart.org/2002/bridges2002-55.html

Burmeister, A. (2013). Frieder Nake - ein Pionier der Computerkunst. In: HPImgzn, Ausgabe 15, S. 11-13. Potsdam: HPI. Download unter https://hpi.de/fileadmin/user_upload/hpi/dokumente/hpi_mgzn/HPImgzn_Ausgabe15.pdf

Chakraborty, A., Graebner, R. & Stocky, T. (1999). Logo - a Project History. Download unter http://web.mit.edu/6.933/www/LogoFinalPaper.pdf

Clayson, J. (1988). Visual Modeling with Logo. Cambridge: MIT Press.

Cohen, H. (1994). The further exploits of AARON, Painter. Stanford Humanities Review. Download unter http://www.aaronshome.com/aaron/aaron/publications/furtherexploits.pdf

Csuri, C. (1978). Real Time Picture Animation. In: IBM Deutschland Gmbh (Hrsg.): Computerkunst. Stuttgart: IBM Deutschland GmbH.

Davidson, S. (2012). Jackson Pollock. Zeitlos wie Picasso. Monopol-Magazin. http://www.monopol-magazin.de/zeitlos-wie-picasso

Dewdney, K.A. (1988). Computer-Kurzweil. Spektrum der Wissenschaft, Juli 1988, S. 11

Dohm, K., Stahlhut, H., Hollein, M. & Magnaguagno, G. (2008). Kunstmaschinen Maschinenkunst. Heidelberg: Kehrer Verlag.

Dorling, M. & White, D. (2015). Scratch: A Way to Logo and Python. SIGCSE '15 Konferenz, Kansas City. Download unter http://ispython.com/wp/wp-content/uploads/2014/11/Asigsce16_titled.pdf

Franke, H. W. (1971). Computergraphik Computerkunst. München: Bruckmann.

Franke, H. W. (1984). Computergrafik-Galerie: Bilder nach Programm. Köln: DuMont Buchverlag.

Feurzeig, W. (2010). Toward a Culture of Creativity: A Personal Perspective on Logo's Early Years and Ongoing Potential. International Journal of Computers for Mathematical Learning, Vol. 15, 3, pp. 257-265.

Glassner, A. (2010). Processing for Visual Artists. Natick, MA: A K Peters, Ltd.

Glowski, J. M. (ed.) (2006). Charles A. Csuri. Beyond Boundaries, 1963 – present. Columbus: College of the Arts, The Ohio State University.

Grafton, C. B. (1976). Optical Designs in Motion with Moiré Overlays. New York: Dover Publications.

Groos, U. & Froitzheim, E.-M. (2017). [un]erwartet. Die Kunst des Zufalls. Köln: Wienand Verlag.

Harvey, B. & Mönig, J. (2017). Snap! Reference Manual 4.0.10. Download unter http://snap.berkeley.edu/SnapManual.pdf

Herzogenrath, W. & Nierhoff-Wielk, B. (2007). Ex Machina - Frühe Computergrafik bis 1979. München: Deutscher Kunstverlag.

Hoppe, H.-U. (1984). LOGO im Mathematikunterricht. München: IWT-Verlag.

IBM (1978). Computerkunst. Stuttgart: IBM Deutschland.

Immler, C. (2015). Der kleine Hacker: Programmieren für Einsteiger. Mit Scratch schnell und effektiv programmieren lernen. Haar: Franzis Verlag.

Klütsch, C. (2007). Computergrafik. Ästhetische Experimente zwischen zwei Kulturen. Die Anfänge der Computerkunst in den 1960er Jahren. Wien: Springer-Verlag.

Kolomyjec, W.J. (1975). The Appeal of Computer Graphics. In: R. Leavitt (Ed.): Artist and Computer. pp. 45–51, New York: Harmony Books. Download unter http://www.atariarchives.org/artist/sec15.php

Krawczyk, R.J. (2002). A Shattered Perfection: Crafting a Virtual Sculpture. In: Proceedings Sixth International Conference on Information Visualisation, pp. 771-776. Washington: The Printing House. Download unter http://mypages.iit.edu/~krawczyk/rjkiv02.pdf

Kyriakou, H. & Nickerson, J. (2014). Collective Innovation in Open Source Hardware. In: Proceedings Collective Intelligence 2014. Boston: MIT. Download unter http://arxiv.org/pdf/1404.1799v1.pdf

Laposky, B. (1953). Electronic Abstractions. Cherokee, Iowa: Laposky. Download unter http://www.vasulka.org/archive/Artists3/Laposky,BenF/ElectronicAbstractions.pdf

Lewandowsky, P. & Zeischegg, F. (2002). Visuelles Gestalten mit dem Computer. Hamburg: Rowohlt.

Lieser, W. (2009). Digital Art. Potsdam: Tandem Verlag.

Limbeck, L. & Schneeberger, R. (1979). Computergrafik. München: Ernst Reinhardt Verlag.

Lindner, A. (2001). Zufallszahlen. Bad Ischl: Manuskript. Download unter http://teacher.eduhi.at/alindner/Sites/skripten/zufzahlen.pdf

McKenna, D. (2011). From Lissajous to Pas de Deux to Tattoo: The Graphic Life of a Beautiful Loop. In Sarhangi, R. & Carlo H. Séquin (eds.): Proceedings of Bridges 2011: Mathematics, Music, Art, Architecture, Culture, pp. 295-302. Phoenix: Tessellations Publishing. Download unter http://archive.bridgesmathart.org/2011/bridges2011-295.html

Mihich, V. (2007). Vasa. Los Angeles: Vasa Studio. Download unter http://vasastudio.com/vasa_book.pdf

Mihich, V. (2015). VASA Renderings for Paintings 2007-2015. Los Angeles: Vasa Studio. Download unter http://vasastudio.com/vasa_book_2.pdf

Modrow, G. (2013). Informatik mit BYOB / Snap!. Download unter https://de.scribd.com/doc/210283773/BYOB-Snap-Scratch-Uni-Gottingen

Mohr, M. (1971). Computer Graphics. Une esthétique programmée. Paris: A-R-C Musée d'art Moderne. Download unter http://www.emohr.com/ww4_out.html

Molnar, V. (1975). Vera Molnar. In: R. Leavitt (Ed.): Artist and Computer. New York: Harmony Books. Download unter http://www.atariarchives.org/artist/sec11.php

Molnar, V. (1990). Lignes, Formes, Couleurs. Ausstellungskatalog. Budapest: Vasarely Múzeum.

Nake, F., Mathews, M.V., Deitschman, B. & Stickel, G. (1966). Herstellung von zeichnerischen Darstellungen, Tonfolgen und Texten mit elektronischen Rechenanlagen. Programm-Information P1-21. Darmstadt: Deutsches Rechenzentrum. Download unter http://dada.compart-bremen.de/docUploads/ProgrammInformation21_PI21.pdf

Nake, F. (1971). There should be no computer art. PAGE 18 - Bulletin of the Computer Arts Society. Download unter http://www.bbk.ac.uk/hosted/cache/archive/PAGE/PAGE18.pdf

Nake, F. (1974). Ästhetik als Informationsverarbeitung. Grundlagen und Anwendungen der Informatik im Bereich ästhetischer Produktion und Kritik. Wien, New York: Springer.

Nake, F. (1995). Künstliche Kunst im Strom der Zeit. Vortrag Berliner Sommer-Uni `95. Download unter http://radicalart.info/AlgorithmicArt/Nake95.html

Nake, F. (2010). Paragraphs on Computer Art, Past and Present. CAT 2010 London Conference. Download unter https://pdfs.semanticscholar.org/d536/d22b09c4f6207b8fb78192e551c20fef2282.pdf

Nake, F. (2014). „No Message Whatsoever" (Interview). HPImgzn Ausgabe 15, S. 11-13. Potsdam: HPI. Download unter: https://hpi.de/fileadmin/user_upload/hpi/dokumente/hpi_mgzn/HPImgzn_Ausgabe15.pdf

Nake, F. (2016). The Disappearing Masterpiece Digital Image & Algorithmic Revolution. In: M. Verdicchio et al. (Ed.): xCoAx 2016 Proceedings, pp. 12-27. Bergamo. Download unter http://2016.xcoax.org/xcoax2016.pdf

Nees, G. (1965). Computer-Grafik. In M. Bense & E. Walther (Hrsg.): edition rot 19. Stuttgart: Mayer Druck.

Nees, G. (1969). Generative Computergraphik. Ein Nachdruck mit einleitenden Texten von von Herrmann und Nees (2006) ist als eTexT erhältlich.

Nees, G. (1995). Formel, Farbe, Form. Computerästhetik für Medien und Design. Heidelberg: Springer.

Nees, G. (2005). Künstliche Kunst. Die Anfänge. Ausstellungsbroschüre (unpaginiert). Bremen: Kunsthalle.

Noll, A.M. (1966). Human or Machine: A Subjective comparison of Piet Mondrians »Composition with Lines« (1917) and a computer- generated Picture. Psychological Record, Vol. XVI, pp. 1–10. Download unter http://noll.uscannenberg.org/Art%20Papers/Mondrian.pdf

Noll, A. M. (1967). Computers and the Visual Arts. In: M. Krampen & P. Seitz (Eds.). Design and Planning 2: Computers in Design and Communication, pp. 65-79. New York: Hastings House Publishers. Download unter http://noll.uscannenberg.org/Art%20Papers/Computers%20Visual%20Arts.pdf

Otsubo, K., Washida, M. Itoh, T., Katsuura, K. & Hayashi, M. (2000). Computer Simulation on the Gumowski-Mira Transformation. Forma, Vol. 15, pp. 121-126. Download unter: http://www.scipress.org/journals/forma/pdf/1502/15020121.pdf

Papert, S. (1982). Mindstorms: Kinder, Computer und Neues Lernen. Stuttgart: Birkhäuser.

Papert, S. & Minsky, M. (1969, 1988[2]). Perceptrons. Cambridge, Mass.: The MIT Press

Peitgen, H.-O., Jürgens, H. & Saupe, D. (1992). Bausteine des Chaos: Fraktale. Berlin: Springer.

Peitgen, H.-O., Jürgens, H. & Saupe, D. (1994). Chaos: Bausteine der OrdnungBerlin: Springer.

Perron, O. (1962). Nichteuklidische Elementargeometrie der Ebene. Stuttgart: Teubner.

Piehler, H. M. (2001). Die Anfänge der Computerkunst. Frankfurt: dot-Verlag

Pickover, C.A. (1991). Computers and the Imagination. New York: St. Martin´s Press.

Price, T.W. & Barnes, T. (2015). Comparing Textual and Block Interfaces in a Novice Programming Environment. ICER '15 Konferenz, Omaha. Download unter http://www4.ncsu.edu/~twprice/website/files/ICER%202015.pdf

Rauch, H. & Wedekind, J. (1989). Schildkrötengrafik. Einfaches Programmieren von Grafiken. Lehren und Lernen mit dem Computer 13. Tübingen: DIFF.

Prusinkiewicz, P. & Lindenmayer, A. (1990). The algorithmic beauty of plants. New York: Springer.

Resnick, M. & Siegel, D. (2015). A Different Approch to Coding. Bright: What‘s New in Education. Download unter https://medium.com/bright/a-different-approach-to-coding-d679b06d83a#.4tfmfe8x3

Riley, B. (2009). At the End of My Pencil. London Review of Books Vol. 31, No. 19, pp. 20-21. Download unter https://www.lrb.co.uk/v31/n19/bridget-riley/at-the-end-of-my-pencil

Riley, B. (2015). The Curve Paintings 1961-2014. London: Ridinghaouse.

Romero, J. & Machado, P. (2008). The Art of Artificial Evolution: A Handbook on Evolutionary Art and Music. Berlin: Springer

Rosen, M. (2011). A Little-Known Story about a Movement, a Magazine, and the Computer´s Arrival in Art: NewTendencies and Bit International, 1961-1973. Karlsruhe: ZKM.

Rosen, M. (Hg.) (2013). Der Algorithmus des Manfred Mohr. Texte 1963-1979. Karlsruhe: ZKM.

Schiffer, S. (1996). Visuelle Programmierung – Potential und Grenzen. In: Mayr, H. C. (Hrsg.): Beherrschung von Informationssystemen. Schriftenreihe der Österreichischen Computergesellschaft, Band 88, S. 267–286. München: Oldenbourg. Download unter http://www.schiffer.at/publications/se-96-19/se-96-19.pdf

Schneeberger, R. (2013). Programmierte Fälschungen Digitaler Kunst als Museumsprojekt? Telepolis (Online-Journal). Haar: Heise Medien. Download unter http://www.heise.de/tp/artikel/38/38864/1.html

Schön, S., Ebner, M. & Narr, K. (Hrsg.) (2016): Making-Aktivitäten mit Kindern und Jugendlichen. Handbuch zum kreativen digitalen Gestalten. Norderstedt: Books on Demand. Auch zum Download unter http://www.bimsev.de/n/userfiles/downloads/making_handbuch_online_final.pdf

Schwill, A. (1999). Programmierstile. Vorlesungsmanuskript. Download unter http://ddi.cs.uni-potsdam.de/Lehre/UebersichtInfoSST/Programmierstile.pdf

Seitz, W.C. (1965). The Responsive Eye. New York: MoMA. Download unter https://monoskop.org/images/1/11/Seitz_William_C_The_Responsive_Eye_1965.pdf

Skasa-Weiß, R. (1965). Künstliche Kunst. In: Stuttgarter Zeitung Nr. 262, 11.November 1965, S. 31.

Sprott, J. (1993). Strange Attractors: Craeting patterns in Chaos. NewYork: M&T Books. Download unter http://sprott.physics.wisc.edu/fractals/booktext/sabook.pdf

Stalder, F. (2009). Neun Thesen zur Remix-Kultur. Download unter https://irights.info/wp-content/uploads/fileadmin/texte/material/Stalder_Remixing.pdf

Stein, H. (1984). Logo - Grafik, Sprache, Mathematik. München: Markt&Technik Verlag.

Steller, E. (1992). Computer und Kunst. Mannheim: BI-Wissenschaftsverlag.

Taylor, G. D. (2014). When the Machine made Art. The Troubled History of Computer Art. New York: Bloomsbury Academic.

Taylor, R.P. (2002). Order in Pollock's Chaos. Scientific American, Vol. 287, pp. 116-121. Download unter http://authenticationinart.org/pdf/literature/Richard-P.-TaylorOrder-in-Pollocks-Chaos.pdf

von Herrman, H.-C. (2010). Künstliche Kunst. Abstraktion als Mimesis. In: T. Bothe & R. Suter (Hg.). Prekäre Bilder, S. 225-245. München: Wilhelm Fink.

Warnke, M. (2006). Kunst aus der Maschine – Informationsästhetik, Virtualität und Interaktivität, Digital Communities. In: J. Sieck & M.A. Herzog (Hg.). Kultur und Informatik: Entwickler, Architekten und Gestalter der Informationsgesellschaft, S. 19-33. Frankfur: Lang Verlag.

Weber, C.S. (2002). Morellet. Katalog zur Ausstellung Museum Würth. Künzelsau: Swiridoff Verlag.

Wedekind, J. (2014). Programmierung interaktiver Grafiken. Eine Einführung mit ACSLogo. Band 1: Polygone, Spirolaterale, Rekursive Grafiken, L-Systeme. Download unter http://programmieren.joachim-wedekind.de/logo/logo-buch/

Weintrop, D. & Wilensky, U. (2015). Using Commutative Assessments to Compare Conceptual Understanding in Blocks-based and Text-based Programs. ICER '15 Konferenz, Omaha. Download unter http://dweintrop.github.io/papers/Weintrop_Wilensky_ICER_2015.pdf

Welsch, N. & Liebmann, C.C. (2003). Farben: Natur, Technik, Kunst. Heidelberg: Spektrum.

Werler, K.-H. (1991). Programmierte Phantasie. Berlin: AkademieVerlag.

Wheeler, S. (2016). Digital literacies in the age of remix. Blogbeitrag unter http://www.steve-wheeler.co.uk/2016/06/digital-literacies-in-age-of-remix.html

Wilensky, U. & Rand, W. (2015). An Introduction to Agent-Based Modeling. Cambridge: MIT Press.

Wilson, M. (1985). Drawing with Computers. The Artist's Guide to Computer Graphics. New York: Perigee Books. Download unter http://mgwilson.com/Drawing%20with%20Computers.pdf

Zheng, Y., Nie, X., Meng, Z., Feng, W. & Zhang, K. (2015). Layered modeling and generation of Pollock's drip style. Visual Computer, Vol. 31, Issue 5, pp. 589-600.

Ziegenbalg, J. (1986). Informatik. Logo Lern- und Arbeitsbuch. Braunschweig: Westermann. Download unter http://www.ziegenbalg.ph-karlsruhe.de/materialien-homepage-jzbg/mybooks-scans/Logo-Lern-und-Arbeitsbuch.pdf

Ziegenbalg, J. (2005). Algorithmik als Schnittstelle zwischen Kunst und Mathematik. In: B. Könches & P. Weibel (Hg.). inSICHTBARes. Kunst_Wissenschaft, S. 174-199. Bern: Benteli Verlags AG.

BILDERNACHWEISE

Zuse Graphomat (S. 22): Photo Inge Wedekind, Ausstellung DAM Galerie, Berlin

Bodenturtle (S. 22):
http://cyberneticzoo.com/cyberneticanimals/1969-the-logo-turtle-seymour-papert-marvin-minsky-et-al-american/

Seymour Papert (S. 30): Transformative Learning Technologies Lab
https://tltl.stanford.edu/papert_tribute

MIT Logo für den Apple II (S. 31): Photo Joachim Wedekind

Piet Mondrian: Compositie in lijn (S. 204). The Kröller-Müller Museum
(http://krollermuller.nl/en/piet-mondriaan-composition-in-line-second-state-1)

A. Michael Noll: Computer Composition with Lines (S. 204).
(http://noll.uscannenberg.org/Art%20Papers/Mondrian.pdf)

Aaron, with Decorative Panel (S. 292):
http://aaronshome.com/aaron/aaron/publications/furtherexploits.pdf

In Zana's Room, painting by AARON (S. 292):
https://newatlas.com/creative-ai-algorithmic-art-painting-fool-aaron/36106/#p317522

Schichtenmodell zur Modellierung des Drip Painting von Jackson Pollock (S. 310): aus Zheng u.a., 2015, S. 3.

Mirobot (S. 322): Foto von Ben Pirt (https://mime.co.uk/products/mirobot/)

Alle anderen Bilder sind Screenshots aus den jeweiligen Programmen bzw. damit erzeugter Ergebnisgrafiken.

ANHANG A: TIPPS UND TRICKS FÜR SNAP!

Wenn Sie alle Kapitel des Buches bis hierher durchgearbeitet haben, dann kennen Sie Snap! schon ziemlich gut. Für die vorgestellten Projekte haben wir aber nicht alle verfügbaren Programmkomponenten benötigt. Etliche mehr davon werden in diesem Anhang vorgestellt; allerdings kann das kein Ersatz für das Handbuch sein (das derzeit in der Version 4.1 - leider nur auf Englisch - vorliegt). Selbst dort werden nicht alle Blöcke im Detail erklärt. Auch die Zusammenstellung des Befehlssatzes von Snap! in *Anhang B* soll dafür kein Ersatz sein.

Hilfen und Arbeitserleichterungen in Snap!

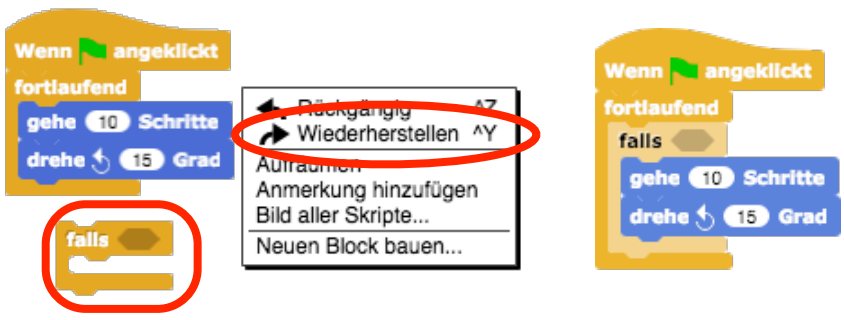
Die Autoren Brian Harvey und Jens Mönig gehen davon aus, dass Sie durch Experimentieren und durch Lesen der **Hilfe**-Bildschirme mehr über Snap! lernen. Den **Hilfe**-Bildschirm eines Blocks erreichen Sie durch Rechtsklick oder Ctrl-Klick auf einen Block. Es erscheint ein Kontext-Menü, in dem Sie **Hilfe ...** auswählen können. Falls Sie zunächst gezielt einen Block suchen wollen, können Sie mit **CTRL-F** ein Suchfeld öffen und dort einen Namen(steil) eingeben. Dieser erscheint dann im Palettenbereich.

Bei Blöcken, die über **Module...** importiert wurden, fehlt die Hilfe-Funktion. Dafür ist bei ihnen über **Bearbeiten...** der (JavaScript-) Code einsehbar. Daraus lässt sich oft auch ohne JavaScript-Kenntnisse einiges über die Funktionalität eines Blocks lernen.

undo/redo: Beim Zusammenstellen der Blöcke zu einem Skript können immer wieder Fehler passieren. Für schrittweise Korrekturen gibt es im Programmbereich oben links die Symbole für **Rückgängig** bzw. **Wiederherstellen** (undo/redo). Diese Optionen sind auch als Kontextmenü über Klick der rechten Maustaste zugänglich:



Ausgangscode *Block einfügen* *Aktion rückgängig machen*

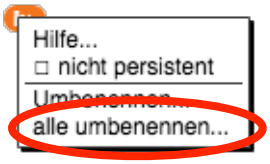


Wiederherstellen *Ergebnis*

Auto-Wrapping: Neben der visuellen Hilfe beim Andocken eines Blocks an vorhandene Blöcke gibt es als weitere Erleichterung beim Zusammenstellen der Programmblöcke das „automatische Einwickeln“ (auto-wrapping). Dabei werden beim Überfahren vorhandener Blöcke mit einem C-förmigen Block die Blöcke markiert, die vom C-Block eingeschlossen werden sollen:

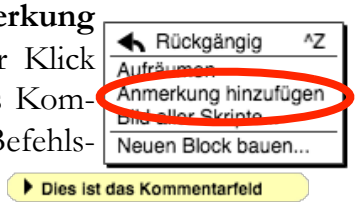


Variablen umbenennen: Wenn Variablen (auch Skriptvariablen) nachträglich umbenannt werden sollen, kann im Kontextmenü über **alle umbenennen...** festgelegt werden, dass alle Aufrufe dieser Variablen ebenfalls mit geändert werden.



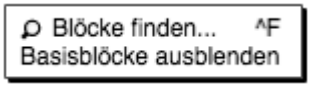
Programmausführung verfolgen: Zur Analyse und dem Verstehen eines Programms ist es oft hilfreich, den Programmablauf in einzelnen Schritten zu verfolgen, was normalerweise aufgrund der Ausführungsgeschwindigkeit nicht möglich ist. Diese Option kann in der Werkzeugleiste unter dem **Werkzeug-Smbol** eingeschaltet werden oder direkt in der Leiste mit dem -Symbol. Jeder einzelne aktuell ausgeführte Schritt wird dann farblich unterlegt. Das gilt auch für geöffnete Blöcke! Die Ablaufgeschwindigkeit kann mit dem daneben befindlichen Schieberegler eingestellt werden.

Kommentare: Wenn sich der Sinn von Programmcode uns auch nach längerer Zeit noch erschließen soll, dann sollten wesentliche Programmschritte im Code selbst kommentiert werden. In Snap! ist dafür die Option **Anmerkung hinzufügen** im Kontextmenü des Programmbereichs über Klick der rechten Maustaste zugänglich. Nach Texteingabe ist das Kommentarfeld frei zu platzieren oder kann gezielt an einen Befehlsblock angedockt werden.



Aufräumen: Die Befehlsgruppen können - insbesondere wenn Elemente aus den Blockbibliotheken nachgeladen wurden - lang und unübersichtlich werden. Die Option **Ungebrauchte Blöcke...** unter dem Datei-Symbol in der Werkzeugleiste erlaubt nachgeladene Blöcke nach Bedarf wieder zu entfernen. Auch die Grundausstattung der Paletten kann um ungenutzte Befehle bereinigt werden; Anklicken eines Befehls mit der rechten Maustaste öffnet das Kontextmenü mit der Option **verstecken**.





Mit rechtem Mausklick im Palettenbereich kann ein Kontextmenü geöffnet werden mit der Option **Basisblöcke ausblenden**, womit die gesamte Palette geleert wird.

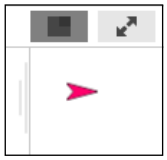
Bei Vorführungen oder bei Kursen für Anfänger bietet es sich an, so bereinigte Paletten zu verwenden, die dann natürlich sukzessive wieder erweitert werden können.



Skripte bearbeiten: Häufig werden Befehle oder Befehlsfolgen mehrfach gebraucht. Dann ist es hilfreich, diese zu duplizieren und an gewünschter Stelle einzufügen. Das entsprechende Kontextmenü bietet diese Optionen. Bei **Duplizieren** wird die ganze Befehlsfolge kopiert. Bei Anwahl des Blocksymbols darunter wird nur der Block unter der aktuellen Mausposition kopiert.

Mit **Skriptbild...** wird ein angewählter Block oder Skript als Bild im PNG-Format abgespeichert. Das ist sehr nützlich für Dokumentationen. Für dieses Buch wurde davon intensiv Gebrauch gemacht.

Bühnengröße: In der Werkzeugleiste von Snap! finden Sie unter dem Werkzeugsymbol die Option **Bühnengröße...**, mit der Sie die **Bühnenbreite** und **Bühnenhöhe** nach Bedarf bzw. den Möglichkeiten Ihres Monitors einstellen können. Das ist vor allem dann hilfreich, wenn Sie großformatige Ausdrucke Ihrer Grafiken anstreben.



Aber auch die Darstellung am Bildschirm lässt sich - unabhängig von der Bühnengröße - anpassen. Die drei hellen Linien am oberen linken Bühnenrand können mit der Maus gefasst und verschoben werden, bis die gewünschte Größe erreicht ist. Mit dem -Schalter in der Werkzeugleiste kann die Bühne zwischen dieser und der Originalgröße hin- und hergeschaltet werden. Mit dem -Schalter kann zwischen einer Vollbild-Darstellung (bei der die Bühne das ganze Browser-Fenster ausfüllt) und der Originalgröße hin- und hergeschaltet werden.

Speichern und Laden

Wenn Sie die Produkte Ihrer Arbeit für die weitere Nuzung speichern wollen, so ist das lokal im Browser, als XML-Export und in der „Cloud“ möglich, alles Optionen, die Sie in der Werkzeugleiste unter dem **Datei-Smbol** finden.



Browser: Die Option **Sichern als...** bietet **Browser** als Speicherort an, was bedeutet, dass Ihr Projekt in einer speziellen Datei gespeichert wird, die nur auf demselben Computer, mit demselben Browser und mit der Snap!-Webseite wieder gelesen werden kann. Sie sehen sie nicht

in einer Liste Ihrer Dateien außerhalb von Snap!. Das ist der Schutzmechanismus von Javascript.

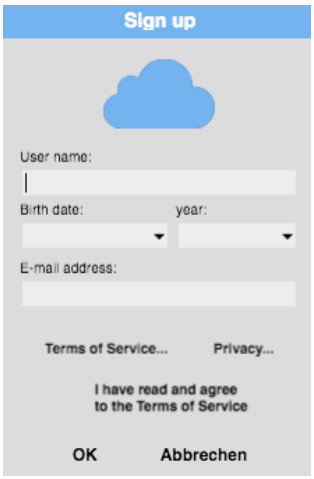
Eine Einschränkung besteht dabei in der begrenzten Gesamtmenge des verfügbaren Speichers Ihres Browsers (dieses Limit kann in den Einstellungen Ihres Browsers geändert werden). Diese Art des Speicherns ist daher auf wenige Projekte beschränkt.

XML-Export: Die zweite Möglichkeit, ein Projekt lokal auf Ihrem Computer zu speichern, sind Dateien auf Ihrem Computer. Diese können mit jedem Browser wieder geöffnet werden und unterliegen keiner Größenbeschränkung. Sie wählen dazu die Option **Projekt exportieren...** Sofern Ihr Projekt noch keinen Namen hat, werden Sie zu dessen Eingabe aufgefordert.



Je nach Browser passieren dann unterschiedliche Dinge. In einfachsten Fall wird Ihr Projekt in eine Datei in Ihr Download-Verzeichnis heruntergeladen. In anderen Fällen wird das Snap!-Fenster verlassen und ein zweites Fenster in einem weiteren Tabulator des Browsers oder ein neues Browser-Fenster geöffnet. Darin wird das Projekt als Textdatei mit XML-Notation angezeigt (für Sie vermutlich ziemlich unverständlicher Zeichensalat). Verwenden Sie nun den Befehl **Speichern** Ihres Browsers (im Menü **Datei**). Sie können einen Dateinamen dafür eingeben und die Datei wird in Ihrem normalen Download-Ordner gespeichert.

Cloud: Die dritte Speichermöglichkeit ist in der „Cloud“, d.h. über die Snap!-Webseite. Dafür müssen Sie zuerst ein Benutzerkonto anlegen. Unter dem -Symbol in der Werkzeugleiste finden Sie die Option **Benutzerkonto einrichten...**, die Sie durch den Anmeldeprozess führt. Nach Eingabe von Nutzername, Geburtsmonat, Geburtsjahr sowie E-Mail-Adresse erhalten Sie eine Mail an diese Adresse mit einem anfänglichen Kennwort für Ihr Konto. Die E-Mail-Adresse bleibt gespeichert, sodass Ihnen, sollten Sie Ihr Passwort vergessen haben, ein Link zum Zurücksetzen des Passworts gesendet werden kann.



Sobald Ihr Konto erstellt ist, können Sie sich mit der Option **Anmelden...** in der Cloud anmelden; verwenden Sie dazu den Benutzernamen und das Kennwort, das Sie zuvor angegeben haben. Wenn Sie das Feld **Angemeldet bleiben** aktivieren, werden Sie beim nächsten Start von Snap! automatisch angemeldet.

Sobald Sie angemeldet sind, können Sie im Dialogfeld **Sichern als...** die Option **Cloud** auswählen. Geben Sie dann einen Projektnamen und optional Projektnotizen dazu ein. Ihr Projekt wird online gespeichert und kann mit Netzzugang überall geladen werden.

Projekte laden: Natürlich möchten Sie gespeicherte Projekte wieder in Snap! laden können. Dafür gibt es zwei Möglichkeiten:

Wenn Sie das Projekt im lokalen Speicher des Browsers oder in der Cloud unter Ihrem Benutzerkonto abgespeichert haben, wählen Sie die Option **Öffnen...** im Menü **Datei**. Wählen Sie zuerst die Schaltfläche **Browser** oder **Cloud** aus. Es erscheint die Liste Ihrer abgespeicherten Dateien. Wählen Sie das gewünschte Projekt aus (es erscheinen dann im rechten Bereich evtl. das letzte Bühnenbild und die Projektnotizen) und klicken Sie auf **Open**.

Hinweis: *Eine dritte Schaltfläche* **Examples** *ermöglicht die Auswahl von Beispielprojekten, die von den Snap!-Entwicklern zur Verfügung gestellt werden. Bei Anklicken können Sie - bevor Sie ein Beispiel starten - die Projektnotizen sehen.*

Haben Sie Ihr Projekt als XML-Datei auf Ihrem Computer gespeichert, wählen Sie **Importieren...** aus dem Menü **Datei**. Dadurch erhalten Sie ein Auswahlfenster, in dem Sie wie gewohnt in Ihrem Dateisystem zu der gesuchten Datei navigieren können. Alternativ können Sie die XML-Datei außerhalb des Browsers in Ihrem Dateisystem lokalisieren und dann einfach über das Snap! Fenster ziehen.

Auf die gezeigten zwei Arten können Sie auch Medien (Kostüme und Sounds) in ein Projekt importieren.

Weitere Informationsquellen

Dokumentationen:

https://snap.berkeley.edu HomePage von Snap! mit aktuellen Infos und Hinweisen sowie Verweisen auf weitere Informatiosnquellen.
http://snap.berkeley.edu/SnapManual.pdf Download des jeweils aktuellen Snap!-Handbuchs.

Hilfsprogramme:

http://snapp.citilab.eu Erlaubt die Erstellung lauffähiger Programme aus Ihrem Snap!-Programm.
http://djdolphin.github.io/Snapin8r2/ Erlaubt den Import von Scratch 2.0-Programmen, die direkt in Snap!-Programme gewandelt werden.

Erweiterungen:

http://www.snap-apps.org Entwicklungswerkzeuge für den Informatik-Unterricht und andere Fächer, entwickelt an der Monash University, Australien, darunter:

- *Scribble:* Snap!-Erweiterung zur Erzeugung von Generative Art (der Ansatz ähnelt dem von mir in diesem Buch verfolgten Konzept).
- *Cellular:* Snap!-Erweiterung für agentenbasierte Simulationen.
- *Edgy:* Snap!-Erweiterung für Netzwerk-Algorithmen und deren Visualisierung.

http://beetleblocks.com Erlaubt die Erstellung von 3D-Modellen und deren Vorbereitung für 3D-Drucke.
https://netsblox.org Erlaubt die Erstellung vernetzter Anwendungen.
http://www.turtlestitch.org Erlaubt die Ausgabe von mit Snap! erzeugten Grafiken mit programmierbaren Strickmaschinen.
http://snap4arduino.rocks Erlaubt die nahtlose Interaktion mit fast allen Versionen des Arduino-Microcontrollerboards.

Programmbeispiele:

Leider gibt es aktuell keine zentralen Programmsammlungen zu Snap!, aus denen Sie Beispiele laden könnten. Die Snap!-Entwickler arbeiten aber daran, eine vergleichbare Website aufzubauen, wie sie für Scratch mit der Übersicht Entdecke vorhanden ist. Dort finden Sie unzählige Einzelprojekte, von denen viel für die Umsetzung in Snap! gelernt werden kann. Besonders nützlich sind die *Studios*, in denen Projekte thematisch zusammen gefasst sind. So gibt es z.B. auch einige Studios zum Thema Kunst.

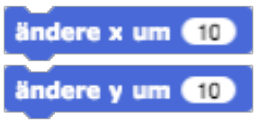
Da Snap! viele Eigenschaften von Scratch geerbt hat und die Programmoberfläche viele Ähnlichkeiten aufweist, lohnt es sich, interssante Scratch-Projekte herunter zu laden.

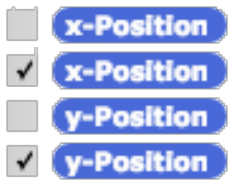
Mit Hilfe von Snapin8r2 lassen sich Projekte von Scratch 2.0 in Snap!-Projekte konvertieren und häufig ohne weitere Änderungen weiter verwenden:

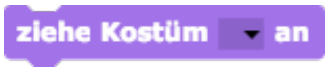
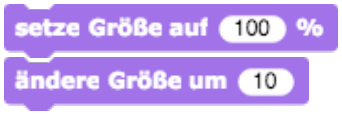


ANHANG B: DIE BEFEHLSBLÖCKE VON SNAP!

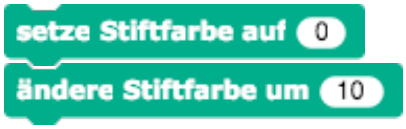
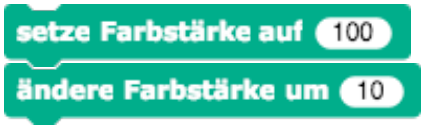
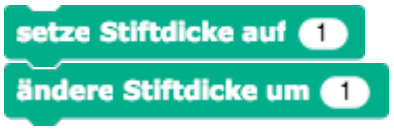
Im Folgenden finden Sie eine Zusammenstellung eines Großteils der Befehlsblöcke von Snap! mit der Erläuterung ihrer Bedeutung. Die Zusammenstellung enthält auch etliche Befehlsblöcke, die bei den Projekten nicht genutzt wurden, die aber beim weiteren *Recoding* und *Remixing* von Nutzen sein können.

Kategorie Bewegung	Befehl
	Bewegen der Schildkröte um x Bildpunkte.
	Drehen der Schildkröte um x Grad im Uhrzeigersinn.
	Drehen der Schildkröte um x Grad entgegen dem Uhrzeigersinn.
	Drehen der Schildkröte in die durch x angegebene Richtung bzw. Drehung der Schildkröte in Richtung eines ausgewählten Objekts (z.B. Mauszeiger oder Sprite) bzw. eines Punkt auf der Bühne.
	Bewegen der Schildkröte an die durch x und y angegebene Position bzw. an die Stelle eines ausgewählten Objekts (z.B. Mauszeiger oder Sprite) bzw. Punkt der Bühne.
	Bewegt die Schildkröte in einer vorgegebenen Zeit an die durch x und y angegebene Position.
	Relative Verschiebung der Schildkröte um den bei x angegebenen Wert in horizontaler bzw. bei y in vertikaler Richtung. Die zugehörige vertikale bzw. horizontale Position wird jeweils beibehalten.
	Bewegt die Schildkröte an die durch x angegebene horizontale Position bzw. an die durch y angegebene vertikale Position. Die zugehörige vertikale bzw. horizontale Position wird jeweils beibehalten.
	Prüft, ob die Schildkröte den Rand der Bühne erreicht hat. Ist dies der Fall, dreht sie sich weg vom Rand.

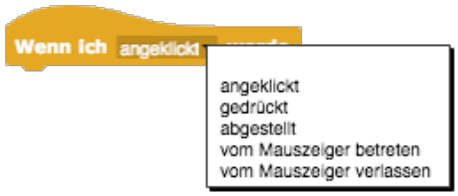
Kategorie Bewegung	Befehl
	Reporter, die bei x die aktuelle horizontale Position bzw. bei y die aktuelle vertikale Position der Schildkröte liefern. Bei Klicken auf das Kontrollkästchen wird die Position auf der Bühne angezeigt.
	Reporter, der die aktuelle Blickrichtung der Schildkröte liefert.

Kategorie Aussehen	Befehl
	Das aktuelle Kostüm der Schildkröte wird durch das ausgewählte Kostüm ersetzt.
	Das aktuelle Kostüm durch das jeweils nächste Kostüm aus der Kostümliste ersetzt.
	Reporter, der die Nummer des aktuell verwendeten Kostüms liefert.
	Setzt die Durchsichtigkeit (Transparenz) eines Objekts auf die gewählte Zahl in % bzw. ändert die aktuelle Durchsichtigkeit um die gewählten %. 0% bedeutet dabei keine Transparenz, 100% völlige Transparenz, d.h. das Objekt wird unsichtbar.
	Schaltet alle Grafikeffekte für das gewählte Objekt aus.
	Setzt die Größe eines Objekts auf die gewählte Zahl in % bzw. ändert die aktuelle Größe um die gewählten %.
	Reporter, der die Größe des aktuell verwendeten Sprite in Bildpunkten liefert.
	Mit diesem Befehlspaar kann die Schildkröte unsichtbar bzw. wieder sichtbar gemacht werden.
	Bei übereinanderliegenden Sprites kann das aktuelle Sprite nach vorne geholt werden.
	Bei übereinanderliegenden Sprites kann das aktuelle Sprite eine oder mehere Ebenen zurück gestellt werden.

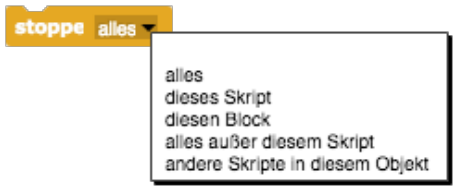
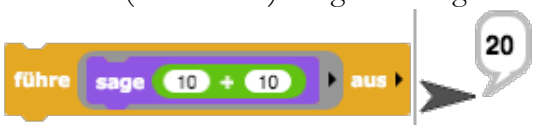
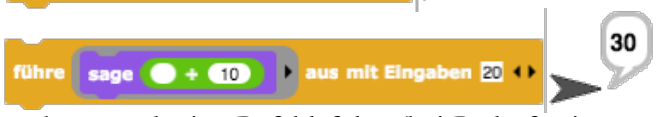
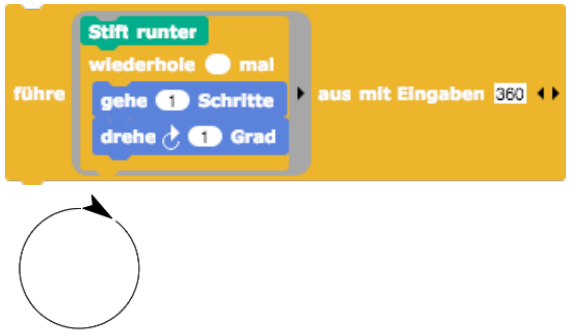
Kategorie Klang	Befehl
	Spielt einen Klang und geht direkt zum nächsten Befehlsblock über.
	Spielt einen Klang vollständig aus und geht erst dann zum nächsten Befehlsblock über.

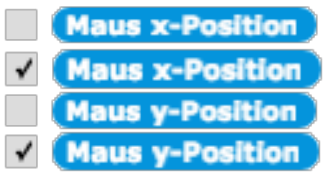
Kategorie Stift	Befehl
	Löscht alle Malspuren und gestempelten Objekte von der Bühne.
	Senkt den Malstift der Schildkröte, die ab dann bei Bewegungen Malspuren hinterlässt bzw. hebt den Malstift und ab dann werden bei Bewegungen keine Malspuren mehr hinterlassen.
	Setzt die Stiftfarbe auf die aus der Farbpalette durch die Mausposition ausgewählte Farbe. Diese Farbe wird dauerhaft beibehalten, bis sie durch eine neue erneute Wahl geändert wird. Als aktuelle Mausposition kann auch jede Position im Snap!-Fenster ausgewertet werden, d.h. auch in den Befehlspaletten, dem Programmbereich oder der Bühne!
	Setzt die Stiftfarbe auf den zahlenmäßig zwischen 0 und 100 definierten Wert. Diese Farbe wird dauerhaft beibehalten, bis sie durch eine neue erneute Wahl geändert wird. Alternativ kann die Stiftfarbe auch um einen gewünschten Wert (innerhalb der Grenzen 0 und 100) erhöht oder verringert werden.
	Setzt die Farbintensität auf den zahlenmäßig zwischen 0 und 100 definierten Wert (in %). Diese Farbintensität wird dauerhaft beibehalten, bis sie durch eine neue erneute Wahl geändert wird. Alternativ kann die Farbintensität auch um einen gewünschten Wert erhöht oder verringert werden.
	Setzt die Stiftdicke auf den zahlenmäßig zwischen 1 und 1000 definierten Wert. Diese Stiftdicke wird dauerhaft beibehalten, bis sie durch eine neue erneute Wahl geändert wird. Alternativ kann die Stiftdicke auch um einen gewünschten Wert erhöht oder verringert werden.

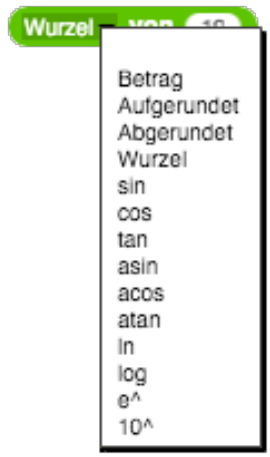
Kategorie Stift	Befehl
	Das Kostüm des aktuellen Sprite wird an der Position der Schildkröte „gestempelt“. Dieses Kostüm wird dann dauerhaft angezeigt bis zum nächsten **wische**-Befehl, auch wenn die Schildkröte weiter bewegt wird.
	Es wird die Fläche von der aktuellen Schildkrötenposition aus bis zur nächsten Linienbegrenzung mit der aktuellen Farbe ausgefüllt.
	Liefert alle auf der Bühne gezeichneten Linien als Kostüm, d.h. auch die Linien von mehreren Sprites werden zusammen gefasst. Die Sprites selbst und der Bühnenhintergrund sind dabei nicht enthalten!

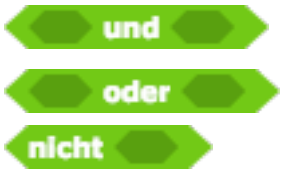
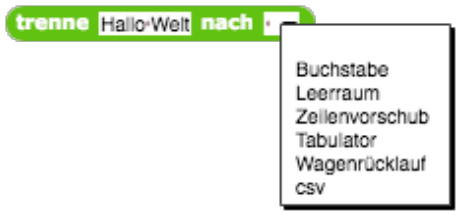
Kategorie Steuerung	Befehl
	Der grüne Hut-Block bewirkt, dass bei Anklicken der grünen Fahne in der Werkzeugleiste die nachfolgende Befehlsfolge ausgeführt wird.
	Der Hut-Block bewirkt, dass bei Anklicken der gewählten Taste die nachfolgende Befehlsfolge ausgeführt wird.
	Die nachfolgende Befehlsfolge wird jeweils ausgeführt, wenn • das Sprite angeklickt und die Maustaste wieder losgelassen wird, • das Sprite angeklickt wird, die Maustaste aber noch gedrückt bleibt, • das Sprite angeklickt, verschoben und dann die Maustaste losgelassen wird, • die Maus den Sprite-Bereich betritt (ohne Mausklick), • die Maus nach Betreten des Sprite-Bereichs diesen wieder verlässt (ohne Mausklick).
	Wenn der Test im Hut-Block den Wert wahr liefert, wird die nachfolgende Befehlsfolge ausgeführt.
	Sendet eine bestimmte Nachricht an alle anderen Sprites (und die Bühne), worauf diese reagieren können.

Kategorie Steuerung	Befehl
	Sendet eine bestimmte Nachricht an alle anderen Sprites (und die Bühne), worauf diese reagieren können, und wartet bis alle reagiert haben.
	Wenn eine bestimmte Nachricht empfangen wird, wird die nachfolgende Befehlsfolge ausgeführt.
	Reporter, der die aktuell versendete Nachricht liefert.
	Beschleunigt die Grafikausgabe, indem alle von Warp eingeschlossenen Blöcke abgearbeitet und keine Zwischenergebnisse ausgegeben werden. Auch andere Blöcke laufen erst ab, wenn Warp abgeschlossen ist.
	Für das aktuelle Sprite werden für die festgelegte Zeitspanne keine Aktionen ausgeführt.
	Für das aktuelle Sprite werden bis zum Eintreffen eines Ereignisses keine Aktionen ausgeführt.
	Die eingeschlossene Befehlsfolge wird solange wiederholt, bis durch ein äußeres Ereignis oder z.B. bei Abfrage eines Grenzwertes die Wiederholung abgebrochen wird.
	Die eingeschlossene Befehlsfolge wird entsprechend der angegebenen Anzahl wiederholt.
	Die eingeschlossene Befehlsfolge wird bis zum Eintreffen eines Ereignisses wiederholt.
	Wenn eine formulierte Bedingung zutrifft, wird die eingeschlossene Befehlsfolge ausgeführt, ansonsten wird die Befehlsfolge ignoriert.
	Wenn eine formulierte Bedingung zutrifft, wird die erste Befehlsfolge ausgeführt, wenn nicht, wird die zweite Befehlsfolge ausgeführt.
	Es wird das Ergebnis einer Berechnung bzw. einer Befehlsabfolge zurück gegeben.

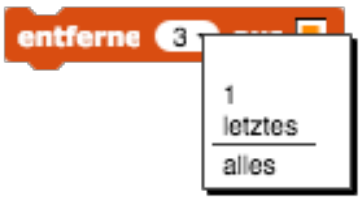
Kategorie Steuerung	Befehl
	Das laufende Programm kann hiermit beendet werden. Bei Auswahl einer der Alternativen im Menü können gezielt einzelne Skripte oder Blöcke beendet werden. Der restliche Programmablauf bleibt davon unberührt.
	Führt einen umringten Befehl aus. Diesem kann auch eine (berechnete) Eingabe übergeben werden.   Es kann auch eine Befehlsfolge (bei Bedarf mit Eingaben) ausgeführt werden. 
	Das aktuelle Sprite (selbst) oder das angegebene Sprite (Objekt) wird vervielfacht.
	Mit diesem Hut-Block wird ein Skript eingeleitet, dass nach dem Erzeugen eines Klons ausgeführt wird. Es können auch mehrere solche Skripte parallel das Verhalten eines Klons bestimmen.
	Der aktuelle Klon wird gelöscht, ist also von keinem Skript mehr ansprechbar und sein Bild wird von der Bühne entfernt.
	Alle laufenden Skripte werden angehalten (entspricht dem Pause-Schalter in der Werkzeugleiste).

Kategorie Fühlen	Befehl
	Es wird geprüft, ob das aktuelle Sprite das angegebene Element berührt. Dieses Element kann sowohl ein anderes Sprite sein, aber auch eine Kante der Bühne, der Mauszeiger oder eine Malspur auf der Bühne.
	Es wird geprüft, ob eine bestimmte Farbe (eines anderen Sprite, einer Malspur oder eines Bühnenelements) berührt wird. In Abhängigkeit des Resultats werden die darauf folgenden Skripten ausgeführt.
	Es wird ein Textfenster auf der Bühne geöffnet, in das der Benutzer einen Text eingeben kann. Diese Eingabe ist mit der Enter-Taste abzuschließen: 
	Reporter, der die vom Benutzer eingegebene Zeichenkette liefert, die dann weiter verarbeitet werden kann. Bei Klicken auf das Kontrollkästchen wird die Antwort auf der Bühne angezeigt.
	Reporter, die bei x die aktuelle horizontale bzw. bei y die aktuelle vertikale Position Position der Maus liefern. Bei Klicken auf das Kontrollkästchen wird die Position auf der Bühne angezeigt.
	Es wird geprüft, ob die (rechte) Maustaste gedrückt wurde. Dieser Reporter kann für logische Abfragen ausgewertet werden.
	Es wird geprüft, ob die Leertaste (oder eine andere vorgegebene Taste) gedrückt wurde. Dieser Reporter kann für logische Abfragen ausgewertet werden.
	Es wird die Entfernung zwischen der aktuellen Position der Schildkröte und dem Mauszeiger, einem Sprite oder einem definierten Ziel ermittelt und ausgegeben.

Kategorie Operatoren	Befehl
	Die *Ringe* sind Behälter für Blöcke, Reporter oder Prädikate. Das bedeutet, dass bei ihrem Aufruf der Block, der Reporter oder das Prädikat zurückgegeben wird, nicht deren Ergebnis. So ergibt der Operator im Beispiel die Summe zurück, der Ring um den Operator dagegen den Operator selbst:  Zunächst wird der gewünschte Block in den Ring platziert. Dieser Ring kann dann als Element einer Liste oder entsprechenden Befehlen übergeben werden. Bei Bedarf können dem umringten Block auch Parameter übergeben werden.
	Dieser Reporter liefern die Ergebnisse der arithmetischen Grundoperationen. Außer Zahlen können auch Parameter oder verschachtelte Ausdrücke eingegeben werden.
	Dieser Reporter liefert den Rest einer Division. So ergibt z.B. 20 modulo 3 die Zahl 2, denn 20 geteilt durch 3 ergibt 6 und den Rest 2.
	Dieser Reporter liefert den gerundeten Wert einer Zahl, d.h eine Dezimalzahl wird in die nächstgelegene ganze Zahl verwandelt (im Gegensatz zu Aufgerundet bzw. Abgerundet!). So liefert 5.1 den Wert 5, 5.6 dagegen den Wert 6.
	Dieser Reporter liefert den Funktionswert, der sich mit der Ausführung der gewählten mathematischen Funktion für den übergebenen Wert ergibt.
	Dieser Reporter liefert gleichverteilte Zufallszahlen als ganze Zahlen im Bereich, der durch die Übergabe der unteren und oberen Grenze gesteuert wird.

Kategorie Operatoren	Befehl
	Mit den Vergleichsoperatoren **kleiner**, **gleich** und **größer** werden die Wahrheitswerte **wahr** oder **falsch** für den jeweiligen Vergleich geliefert.
	Mit den logischen Verknüpfungen **und** (Konjunktion), **oder** (Disjunktion) bzw. **nicht** (Negation) können komplexere logische Ausdrücke gebildet werden.
	Der Reporter übergibt den eingestellten Wahrheitswert.
	Der Reporter trennt einen gegebenen Text • nach jedem Buchstaben • nach jedem Leerzeichen • nach jedem Zeilenende • nach jedem Tabulator • nach jedem Zeilenanfang und liefert eine Liste der so entstandenen Textbausteine.
	Dieser Reporter liefert das erste Zeichen des übergebenen Textes.
	Dieser Reporter liefert die Anzahl der Zeichen des übergebenen Textes.

Kategorie Variablen	Befehl
	Mit **Neue Variable** können *globale* Variablen erzeugt werden, die in allen Prozeduren bekannt und gültig sind. Sind Variablen bereits angelegt, können sie bei Bedarf mit **Variable löschen** wieder entfernt werden.
	Durch diese Befehle können einer vorher definierten globalen bzw. lolalen Variablen ein gewünschter Wert zugewiesen bzw. systematisch um einen Wert geändert werden.
	Mit diesen Befehlen können die entsprechenden Variablenwerte programmgesteuert in einem Monitorfenster auf der Bühne ein- bzw. ausgeblendet werden.

Kategorie Variablen	Befehl
	Mit dem Befehl **Skriptvariablen** können *lokale* Variablen erzeugt werden, die nur temporär innerhalb der aktuellen Befehlsfolge oder innerhalb eines Blocks gültig sind.
	Der Reporter **Liste** zeigt den Inhalt einer Liste an. Mit den Pfeilen **<>** kann die Zahl der Elemente verändert werden. **Liste >** erzeugt eine leere Liste ohne Elemente.
	Der Reporter liefert eine Liste, die um ein neues Anfangselement erweitert wurde.
	Der Reporter liefert das Element an der ersten, letzten oder n-ten Stelle in der Liste. *beliebig* liefert ein zufällig ausgewähltes Element der Liste.
	Der Reporter liefert alle Element der aktuellen Liste, mit Ausnahme des Ersten.
	Der Reporter liefert die Länge (Anzahl der Elemente) der aktuellen Liste.
	Der Reporter liefert den Wahrheitswert **wahr**, wenn *etwas* in der Liste enthalten ist, bzw. **falsch**, wenn es nicht enthalten ist.
	Durch den ersten Befehl wird ein neues Element am Ende der Liste hinzugefügt. Mit dem zweiten Befehl wird ein neues Element an der ersten, letzten oder n-ten Stelle in die Liste eingefügt. Mit *beliebig* wird Element an einer zufälligen Stelle in die Liste eingefügt. Die weiteren Elemente werden entsprechend nach hinten verschoben.
	Es wird das erste, letzte oder n-te Element aus der aktuellen Liste entfernt. Die folgenden Elemente werden entsprechend nach vorne verschoben. Mit *alles* kann die Liste auch vollständig geleert werden.
	Es wird das erste, letzte oder n-te Element der aktuellen Liste durch *etwas* ersetzt. Mit *beliebig* wird ein Element an einer zufälligen Stelle der Liste ersetzt.

Kategorie Variablen	Befehl
Neuer Block	Bei **Neuer Block** öffnet sich ein Dialogfenster, in dem ein neuer Block mit Namen und Eigenschaften definiert werden kann.

ANHANG C: BLOCKBIBLIOTHEK

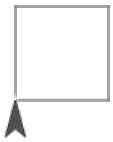
Für Snap! gibt es zahlreiche Befehlserweiterungen, die unter dem Datei-Symbol mit der Option **Module ...** nachgeladen werden können. Die Module sind thematisch zusammen gefasst:

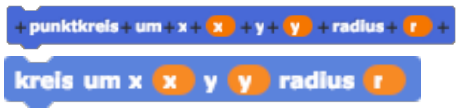
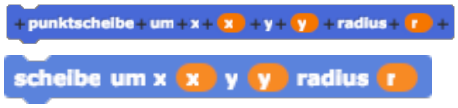
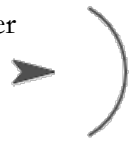
- Tools
- Iteration, composition
- List utilities
- Streams (lazy lists)
- Variadic reporters
- Web service access (https)
- Words, sentence
- Multi-branched conditional (switch)
- LEAP motion controller
- Set RGB or HSV pen color
- Catch errors in a script
- Allow multi-line text input to a block
- Provide getters and setters for all GUI-controlled global settings
- Infinite precision integers, exact rationals, complex
- Animation
- Pixels
- Audio Comp

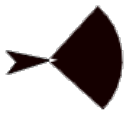
Die dort enthaltenen Befehlsblöcke werden auch im Snap!-Manual nicht alle und wenn, dann nur knapp erläutert. Mehr Informationen sind über die **Hilfe...**-Funktion des jeweiligen Blocks zu erhalten. Außerdem ist bei den meisten über **Bearbeiten...** der Code zugänglich, aus dem einiges über die Funktionalität gelernt werden kann. Leider sind diese Befehlsblöcke und die Hilfen in Englisch und bisher nicht ins Deutsche übersetzt.

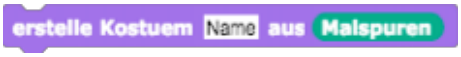
Bei etlichen Projekten zur Computerkunst wurden selbst erstellte Blöcke eingeführt[86]. Sie sind in der folgenden Tabelle zusammen gefasst. Der Code dieser Blöcke ist jeweils einsehbar und kann bei Bedarf an die eigenen Notwendigkeiten angepasst werden.

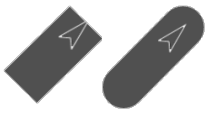
[86] Alle hier genannten Programmblöcke befinden sich auch in der ZIP-Datei mit den Programmen zur Computerkunst SnapProgrammeCompKunst.zip, die heruntergeladen, entpackt und dann mit Snap! aufgerufen und ausgeführt werden können.

Kategorie Bewegung	Befehl
	Zeichnet ein Vieleck mit **n** Ecken um den durch **x** und **y** festgelegten Mittelpunkt mit dem angegebenen **radius**. Die Schildkröte befindet sich nach Zeichnen des Kreises wieder am Ausgangspunkt mit ihrer ursprünglichen Richtung. Bei Bedarf ist die Ausrichtung des Vielecks neu festzulegen. 
	Der Befehl ist identisch mit dem vorigen Befehl, allerdings wird hier das Vieleck mit der aktuellen Stiftfarbe gefüllt. Der Befehl verwendet intern den Befehl **male aus**. 
	Der Befehl ist identisch mit dem Befehl **n-eck gefuellt um ...** Wenn **male aus** wegen Überschneidungen von Linien versagt, werden bei diesem Befehl Vielecke mit schrumpfendem **radius** um **x y** gezeichnet und so die Farbfüllung sicher gestellt. Der Befehl verwendet intern den Block **n-eck**.
	Zeichnet ein Vieleck der Seitenlänge **seite** mit **n** Ecken, ausgehend von der aktuellen Position und Richtung der Schildkröte. Sie befindet sich nach Zeichnen des Vielecks wieder am Ausgangspunkt mit ihrer ursprünglichen Richtung. 
	Die Schildkröte wird zur Position bewegt, die durch **x** und **y** festgelegt wird. Der Zeichenstift wird abgesenkt und zeichnet eine Linie der Länge 1. Danach wird die Schildkröte an den Ausgangspunkt zurückbewegt und der Zeichenstift wieder angehoben.
	Die Schildkröte wird zur Position bewegt, die durch **x** und **y** festgelegt wird. Der Zeichenstift wird abgesenkt und zeichnet mit der Stiftdicke **a** eine Linie der Länge **a** mittig um die Position. Die Schildkröte befindet sich danach am Ausgangspunkt, die Stiftdicke ist auf 1 zurückgesetzt und der Zeichenstift wieder angehoben.

Kategorie Bewegung	Befehl
	Beide Befehle zeichnen einen Kreis um den durch **x** und **y** festgelegten Punkt mit dem angegebenen **radius**. Die Schildkröte befindet sich nach Zeichnen des Kreises wieder am Ausgangspunkt mit ihrer ursprünglichen Richtung. Der Kreis wird von einem Vieleck mit 36 Ecken angenähert. Je größer der Radius, desto gröber wird deshalb der Kreisumriss. Falls eine bessere Auflösung benötigt wird, kann das im Block geändert werden. 
	Beide Befehle sind identisch mit dem Befehl **punktkreis**; allerdings wird hier der Kreis mit der aktuellen Stiftfarbe gefüllt. Der Befehl verwendet intern den Befehl **male aus**. 
	Der Befehl ist identisch mit dem Befehl **punktscheibe**, aber bei diesem Befehl werden Kreise mit schrumpfendem **radius** um **x** **y** gezeichnet und so die vollständige Farbfüllung sicher gestellt. Der Befehl verwendet intern den Block **n-eck**.
	Der Befehl zeichnet einen Kreisbogen von dem mit **x** und **y** gegebenen Mittelpunkt mit Radius **r** und dem Winkel **w** in Uhrzeigerrichtung. Die Schildkröte befindet sich nach Zeichnen des Kreisbogens wieder am Ausgangspunkt mit ihrer ursprünglichen Richtung. 
	Der Befehl zeichnet einen Kreisbogen über dem mit **x** und **y** gegebenen Mittelpunkt mit Radius **r** und dem Winkel **w** in Uhrzeigerrichtung. Er unterscheidet sich vom Kreisbogen dadurch, dass die Schildkröte vor und nach dem Zeichnen zum Mittelpunkt des gezeichneten Kreisbogens zeigt. 
	Zeichnet einen Kreisbogen von dem mit **x** und **y** gegebenen Punkt mit Radius **r** und dem Winkel **w** in Uhrzeigerrichtung, der an beiden Seiten von einer Linie mit der aktuellen Farbe begrenzt bzw. eingeschlossen wird. Die eingeschlossene Fläche des Kreisbogens wird mit der aktuellen Farbe gefüllt. Die Schildkröte befindet sich nach Zeichnen des Kreisbogens wieder am Ausgangspunkt mit ihrer ursprünglichen Richtung. 

Kategorie Bewegung	Befehl
	Zeichnet einen Kreisbogen über dem mit **x** und **y** gegebenen Punkt mit Radius **r** und dem Winkel **w** in Uhrzeigerrichtung, der an beiden Seiten von einer Linie mit der aktuellen Farbe begrenzt bzw. eingeschlossen wird. Die eingeschlossene Fläche des Kreisbogens wird mit der aktuellen Farbe gefüllt. Die Schildkröte zeigt vor und nach dem Zeichnen zum Mittelpunkt des gezeichneten Kreisbogens. 
	Bewegt die Schildkröte von ihrer aktuellen Position zu der Position, die durch **x** und **y** festgelegt wird. Dabei ist der Zeichenstift abgesenkt zum Zeichnen der Linie. Danach wird der Zeichenstift angehoben.
	Bewegt die Schildkröte von ihrer aktuellen Position mit angehobenem Zeichenstift zu der Position **A**. Dort wird der Zeichenstift abgesenkt und die Schildkröte zu der Position **B** bewegt, wodurch die Linie gezeichnet wird. Danach wird der Zeichenstift wieder angehoben.
	Bewegt die Schildkröte mit abgesenktem Zeichenstift von ihrer aktuellen Position in der vorgegebenen Richtung **r** um die Strecke **l**. Danach wird der Zeichenstift wieder angehoben.
	Bewegt die Schildkröte von ihrer aktuellen Position mit angehobenem Zeichenstift zu der Position **A**. Dort wird der Zeichenstift abgesenkt und die Schildkröte in der vorgegebenen Richtung **r** um die Strecke **l** bewegt. Danach wird der Zeichenstift wieder angehoben.
	Bewegt die Schildkröte von ihrer aktuellen Position zu der Position, die durch **x** und **y** festgelegt wird. Dabei ist der Zeichenstift angehoben und es wird keine Linie gezeichnet.

Kategorie Aussehen	Befehl
	Hiermit kann die gesamte aktuell auf der Bühne befindliche grafische Ausgabe (die **Malspuren**) als Kostüm gesichert werden. Dieses ist dann unter dem angegebenen Namen verfügbar.

Kategorie Stift	Befehl
	Mit dem Befehl wird die Stiftfarbe nach dem RGB-Modell bestimmt. Dabei sind für r, g und b Werte zwischen 0 und 255 erlaubt. (0, 0, 0) ergibt Schwarz, (255, 255, 255) ergibt Weiß. Gleiche Werte für alle drei Farbkomponenten ergibt Graustufen.
	Mit dem Befehl wird die Stiftfarbe nach dem HSV-Modell bestimmt. Dabei sind für h, s und v Werte zwischen 0 und 1 erlaubt.
	Mit dem Befehl wird die gesamte Bühne mit der aktuell eingestellten Farbe eingefärbt. Alle sonstigen auf der Bühne befindlichen Objekte und Linien bleiben davon unberührt.
	Mit diesem Befehl wird das Ende einer Linie entweder abgerundet (round) oder abgeflacht (flat). Bei round wird die Abrundung halbkreisartig der Linienlänge hinzugefügt. Das führt bei großer Stiftdicke zu Linienverlängerungen. 
	Der angegebene Text wird in der gewählten Größe an der aktuellen Position und mit der aktuellen Richtung der Schildkröte auf der Bühne ausgegeben.

Bei der Verwendung von Farben ist oft eine präzise Festlegung erwünscht, was sich am besten mit Zahlenwerten im RGB- bzw. HSV-Modell erreichen lässt. Entsprechende Zuordnungen der Zahlenwerte zu Farbbeispielen in tabellarischer Form sind im Web zu finden. Die folgende Tabelle ist ein Ausschnitt aus einer ausführlichen RGB-Tabelle des Karlsruher KIT, der die Werte für die gezeigten Farbe zu entnehmen sind, die dann direkt in Snap! übernommen werden können[87].

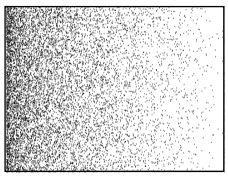
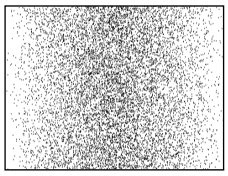
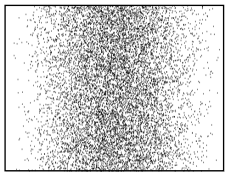
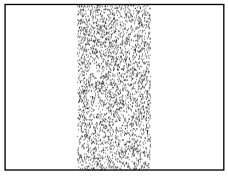
[87] Eine ähnliche Tabelle, bei der die Schattierungen nach Farben geordnet sind, findet sich bei http://www.farb-tabelle.de/

238	221	130	EEDD82
218	165	032	DAA520
184	134	011	B8860B
188	143	143	BC8F8F
205	092	092	CD5C5C
160	082	045	A0522D
222	184	135	DEB887
245	245	220	F5F5DC
245	222	179	F5DEB3
244	164	096	F4A460
210	180	140	D2B48C
210	105	030	D2691E
178	034	034	B22222
165	042	042	A52A2A
233	150	122	E9967A

Hinweis: *Der Snap!-Befehl setze Farbstärke auf erlaubt Werte zwischen 0 und 99. Die folgende Abbildung soll eine genaue Zuordnung der Zahlenwerte zu Farben erlauben:*

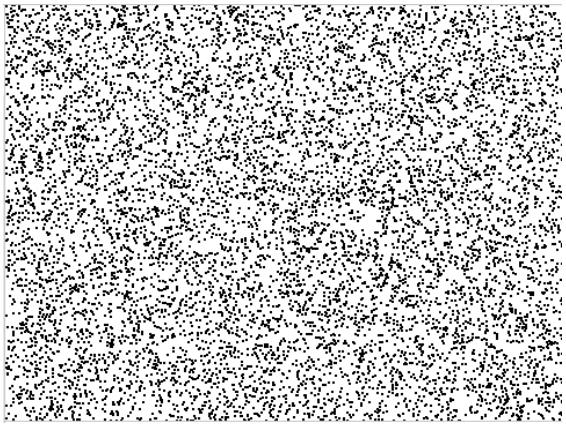
0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89
90	91	92	93	94	95	96	97	98	99

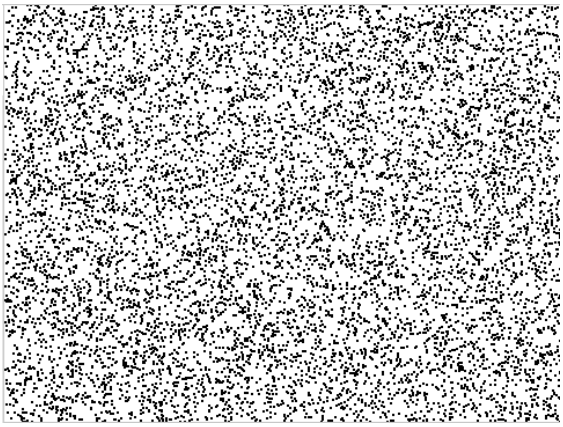
Steuerung	Befehl
	Es wird eine Abfolge von Befehlen zusammengefasst, die solange wiederholt wird, wie durch die Variable **i** angegeben. In jedem Durchlauf wird **i** um den Wert von **schritt** erhöht, bis der Endwert erreicht wird.

Kategorie Operatoren	Befehl
PI	Dieser Reporter liefert die Zahl **Pi** mit einer Genauigkeit von 5 Dezimalstellen.
potenzierung a hoch n	Dieser Reporter liefert die Potenz der Basis **a** für den Exponenten **n**.
Fakultät n	Dieser Reporter liefert die Fakultät der Zahl **n**, d.h. das Produkt aller ganzen Zahlen kleiner und gleich **n**.
zufallszahl startwert a	Dieser Reporter liefert reproduzierbar gleichverteilte Zufallszahlen im Intervall [0,1]. Für die Initialisierung des Zufallszahlengenerators wird ein Startwert aus dem Intervall [0,1 sowie ein konstanter Wert (z.B. die Kreiszahl **Pi**) übergeben. Intern wird der Reporter **Potenzierung** verwendet.
zufallszahl gerade	Dieser Reporter liefert *gewichtete* Zufallszahlen im Intervall [0,1,] **linear abnehmend** von links nach rechts. 
zufallszahl dachform	Dieser Reporter liefert *gewichtete* Zufallszahlen im Intervall [0,1] in **Dachform**, d.h. **symmetrisch** mit Zunahme von Null bis Eins bis zur Intervallmitte und von dort wieder abnehmend auf Null. 
zufallszahl glockentyp	Dieser Reporter liefert *gewichtete* Zufallszahlen im Intervall [0,1] in **Glockenform** (Normalverteilung), d.h. **symmetrisch** mit Zunahme von links bis zur Intervallmitte und von dort wieder abnehmend. 
zufallszahl stufe 0.33 0.66	Dieser Reporter liefert *gewichtete* Zufallszahlen im Intervall [0,1] in **Stufenform**, d.h. Null unter- und oberhalb vorgegebener Grenzwerte und gleichverteilt im Restintervall. 

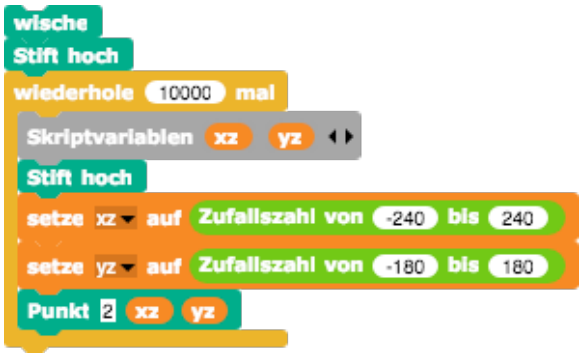
Hintergrund Zufallszahlen: Für die Reproduzierbarkeit von Zufallsgrafiken wurde ein eigener Zufallsgenerator eingeführt (siehe dazu im Kapitel 6: *Gibt's nicht? Gibt's nicht!*). Es ist an dieser Stelle nachzutragen, dass damit qualitativ gleichwertige Zufalls-

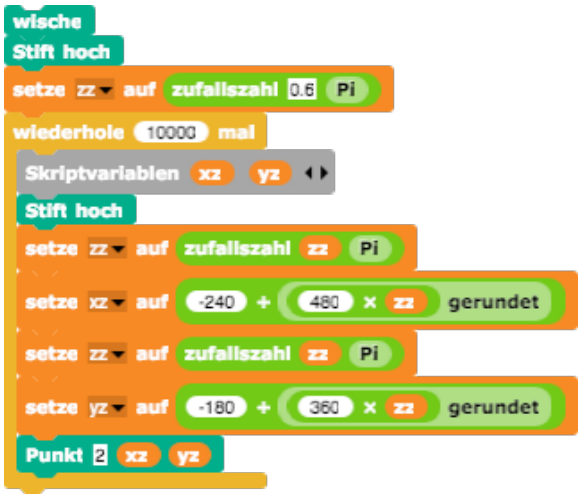
zahlen zum Snap!-Befehl **zufallszahl von ... bis** geliefert werden. Für unsere Zwecke soll ein grafischer Vergleich von jeweils 10.000 Punkten, die mit dem eingebauten Snap!-Generator (links) bzw. mit dem eigenen Generator (rechts) erzeugt wurden, ausreichen.





Wesentliche Unterschiede hinsichtlich der Punkteverteilung sind dabei nicht zu entdecken (ein kleines Testprogramm mit den Mittelwerten nach 10.000 Ziehungen bestätigt dies in beiden Fällen mit Abweichungen < 1% vom theoretischen Mittelwert). Die Verteilung rechts kann mit **zufallszahl 0.6 PI** jederzeit genau reproduziert werden!

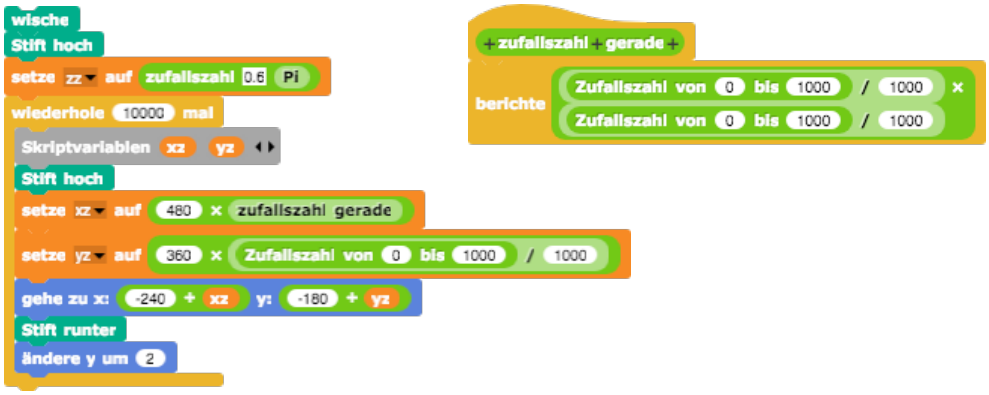




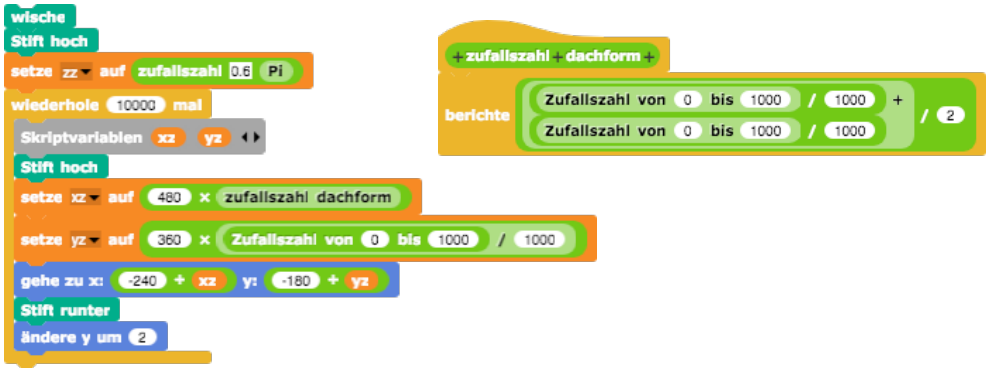
Die Zufallsgeneratoren, sowohl der von Snap! bereit gestellte als auch die reproduzierbare Version, erzeugen gleichverteilte Zahlen im Intervall [0,1]. Oft werden aber spezielle Verteilungen gebraucht (wie z.B. für die *Komputerstrukturen* von Peter Struycken im Kapitel 9: *Figurenbaukasten*). Das leisten die Varianten **zufallszahl gerade**, **zufallszahl dachform**, **zufallszahl glockentyp** und **zufallszahl stufe**.

Zum besseren Verständnis der Funktionsweise ist im Folgenden jeweils der Code des entsprechenden Blocks dargestellt (rechts). Als Beispiel für die Verwendung der Blöcke ist (links) der Code für die Erzeugung der Verteilungsbilder in obiger Tabelle dargestellt.

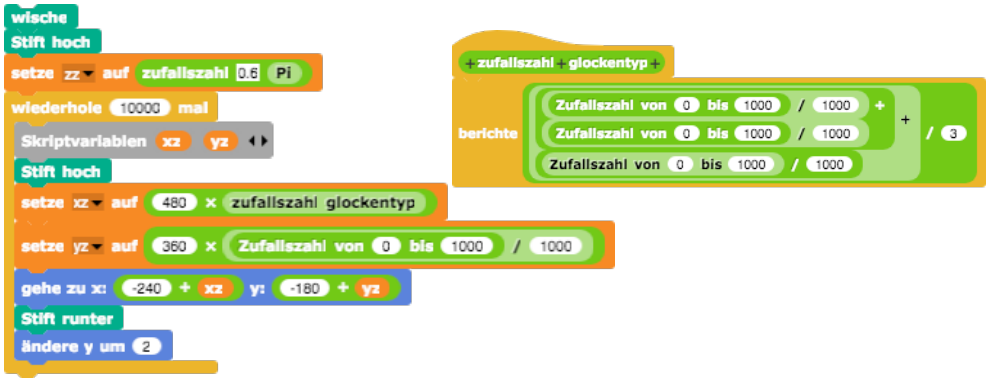
zufallszahl gerade:



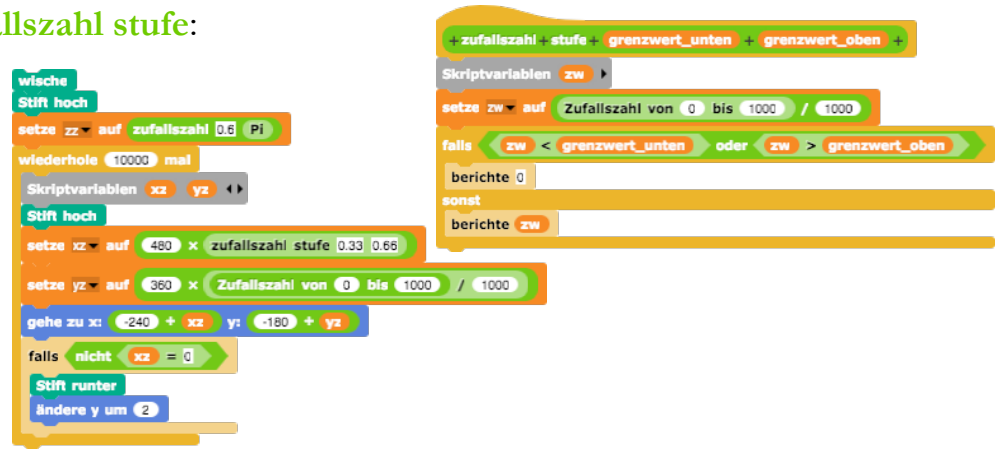
zufallszahl dachform:



zufallszahl glockentyp (Normalverteilung):



zufallszahl stufe:



Kategorie Variablen	Befehl
	Dieser Befehl **LINE (X,Y)** aus der Sprache G ist in der Wirkung identisch mit dem Befehl **gehe nach x y** bei abgesenktem Stift.
	Dieser Befehl **LEER (X,Y)** aus der Sprache G ist in der Wirkung identisch mit dem Befehl **springe nach x y**.

Zur Sprache G:

Für das Kapitel 15: *Hommage à Nees: Die Sprache G* wurden nur die beiden Befehle LINE undf LEER gezeigt und verwendet. Wer alle Sprachelemente der Sprache G von Georg Nees implementiern will (es reichen OPEN, ZIRK, SERIE, REC, ELINR, ELLIR), findet die Beschreibung in Nees (1969). Die Umsetzung in Snap!-Befehlsblöcke ist nicht sehr aufwändig.

PERSONENREGISTER

Adrian 245
Albers 200 299 ff.
Amidror 278
Barbadillo 248 f.
Bartnig 133 f., 250
Beckmann 114, 251 f.
Bense 11, 15, 23, 332
Besemer 280 f.
Beyls 251, 253, 330
Biasi 258 ff.
Böttger 274 ff.
Bohnacker 88, 114, 124, 325, 330
Brill 122 f., 330
Calder 284
Chiggio 259 f.
Cohen, D. 265
Cohen, H. 20, 298 f., 330
Coqart 5, 145, 147
Costa 259, 261
Csuri 223, 240, 254, 330, 331
CTG 130, 236 ff., 255 f., 262, 272
Dawkins 319 f.
Deschenes 278
Devecchi 282 f.
Dewdney 119, 330
Eggeling 223
Ernst 80
Fischinger 223
Faruqee 278
Franke 14, 15, 18, 19, 114, 119, 127, 133, 192, 219, 236, 242, 245, 255, 257 ff., 256, 257, 272, 327, 330 f.
GRAV 209, 276, 293, 302
Gruppo N 259 ff., 280
Harvey 29, 30, 331, 337
Hugonin 85, 87, 297
Kallhardt 245, 247
Kandinsky 84
Kaprow 308
Katz 308
Knowlton 223, 246, 247
Kolomyjec 7, 148, 168 ff. 188, 331
Komura 130 ff., 255
Korneder 262 ff.
Krawczyk 158, 168, 331
Kriwet 246 f.
Land 265
Laposky 7, 14, 113 ff., 258, 320, 331
Lecci 13, 14, 166
LeWitt 21, 98, 100, 297
Marcus 128 ff., 246Massironi 259 ff.
MBB Computer Group 97, 267, 276
McKenna 122, 332
Mihich 75 ff., 79, 98, 332
Mönig 2, 29, 227, 331, 337
Mohr 20, 135 ff., 332, 334
Molnar 7, 15, 18 ff., 53, 92, 95, 108 ff., 145, 155, 209 ff., 332
Mondrian 168, 202, 204 ff., 333, 336
Morellet 8, 30, 200, 209, 293, 302 ff., 334
Müller 88 f.
Nake 2, 7, 11, 12, 15, 20 ff., 80, 189 ff., 299, 302, 328, 330, 332
Nash 266
Neal 278
Nees 4, 5, 7, 11, 15, 16, 22, 25, 66, 96, 99, 105 ff., 119, 149 ff., 168, 177 ff., 192, 200, 231, 262, 280, 297, 332, 333, 363
Nicolai 278
Noll 7, 9, 11, 13ff., 15, 102 ff., 168, 200 ff., 333, 336
Oxenaar 280
Papert 30, 32, 38, 333, 336
Pickover 122, 318, 333
Pollock 8, 80, 308 ff., 330, 334, 335, 336
Rauch 4
Resch 242 f.
Resnick 6, 34, 333
Riley 14, 61 ff., 200, 202, 203, 278, 333
Roubaud 97, 267 f.
Ruttmann 223
Schaffer 254
Schneeberger 2, 7, 24, 25, 66, 214 ff., 262, 331, 334
Schott 262, 269 ff.
Schwartz 223
Sims 320
Sonderegger 272
Steller 11, 24, 27, 45 f., 124, 209, 273 f., 274, 327, 334

Strand 274 ff.
Struycken 88 f., 94, 101, 193, 362
Sýkora 7, 140 ff., 145, 219 ff., 248, 268
Talman 293 ff.
Tinguely 320
Vasarely 21, 200, 278, 332
Vilder 55 ff.
von Graevenitz 276 f., 284 f.
von Weeghel 286 f., 289 f., 293
Wedekind 4, 149, 315, 333, 335
Whitney 223
Wilson 24, 335
Zheng 315 f., 335, 336

SACHREGISTER

AARON 298, 330, 336
additive Farbmischung 70
Algorithmische Kunst 20
Algorithmus 20, 21, 33, 42, 49, 67, 99, 135, 149, 165, 166, 204, 273, 334
Algoristen 24
Animation 1, 7, 8, 55, 124, 195, 211, 212, 223 ff., 234, 236, 240 ff., 254, 278, 284, 287, 289 f., 293 ff., 315, 322, 328, 354
ASCII-Kunst 245 ff., 266
Attraktor 318
bedingte Anweisung 59 f., 136
bedingte Verzweigung 59 f., 142, 172, 176, 188
Befehlsgruppe 36, 50, 338
Biomorphe 319
BJC Beauty and Joy of Computing 34, 36, 327
Bodenturtle 22, 322, 328, 336
Bühne 36 ff., 40 ff., 68, 72, 74, 85, 98, 108, 115, 124, 140, 167, 201, 224 ff., 230, 233 ff., 246, 254, 262, 286, 289 ff., 306, 324, 339
Chaos 176, 286, 315, 318, 333, 334
Computeranimation 223
Computerkunst 1, 4 ff., 11 ff., 24 ff., 32, 34, 37, 45, 66, 69, 70, 80, 84, 96, 102, 108, 127, 128, 133, 135, 139, 149, 161, 168, 177, 189, 195, 209, 214, 223, 236, 245, 254, 258, 266, 273, 274, 278, 297, 299, 306, 308, 314, 315, 327 f., 354
Compiler 40
Computer Minimal Art 214
DBN Design by Numbers 325
Deutsch Limit 33
Digital Art 4, 189, 327,331
Digitale Kunst 19, 21, 25, 177, 233, 334
Direktmodus 40, 43
Drip Painting 8, 80, 306, 308, 336
Erweiterbarkeit 5, 6, 40, 49, 327
Evolutionäre Kunst 319
Expressionismus 84
Felder 5, 182
Flächen 6, 21, 85, 93, 108, 124, 127, 140, 160, 274, 328
Fraktale 309, 315, 318 f., 333
Fraktaler Expressionismus 309
Fraktalkunst 315
gekoppelte Objekte 288 ff.
Generative Kunst 19, 322
G-Sprache 5, 7, 177, 180 ff., 192, 262, 363
grafische Grundelemente 27, 52, 127, 135, 267, 322
Graphomat Z64 22, 177, 189, 336
Harmonograph 321
HSV-Modell 72 f., 112, 224, 358
Igelgrafik 2, 31
Interaktivität 20, 234
Interpreter 40
kartesische Koordinaten 39, 115, 153
Kinetische Kunst 5, 259, 276, 284 ff., 293
Klone 5, 227, 231 ff., 235, 242 f., 254, 286 f., 294
Koch-Kurve 161 f., 319
Konkrete Kunst 5, 8, 21, 209, 250, 297, 299, 302
Konstruktionismus 32 f.
Konstruktivismus 5, 32, 299
Konzeptkunst 8, 21, 75, 98, 127, 297
Koordinatensystem 37, 39, 153
Kontrollstrukturen 5, 40, 59 ff., 75
Kostüm 37, 69, 138 f., 142 f., 195, 201 f., 227, 231 ff., 240, 267, 276, 278 ff., 284 ff., 304, 312, 341
Kreiszahl Pi 66 f., 115
krumme Linien 6, 105 ff.
Linien 6, 11, 14, 15, 19, 21, 25, 27 f., 52, 61, 76, 84 ff., 96 ff., 105, 127, 130, 135, 138, 145, 192 ff., 208, 219 ff., 250 f., 259 ff., 278 f., 288, 293, 302, 309 ff., 322
LISP 30, 31
Lissajous 7, 130 ff., 201, 320, 332
Listen 5, 30, 40, 69, 146, 182 ff., 236, 292 f.
Live Coding 324, 328
Logo 1, 2, 4, 9, 22, 29, 30 ff., 34, 43, 66, 149, 322, 330, 331, 334
L-Systeme 319, 335
Malmaschine 320 ff.
Malroboter 320 ff.
Medienkunst 8, 11, 19, 23, 314 f., 327 f.
Mindstorms 32, 333
Minimal Art 5, 98, 214

Mirobot 332, 336
Modularisierung 5, 6, 40, 49 f., 55, 75, 172
Moiré 7, 200, 259 ff., 273, 278 ff., 297, 331
Morphing 236
Muster 7, 25 f., 45, 61, 85, 108, 127 ff., 131, 135, 148, 166, 180 f., 219, 246, 259, 267, 272, 278 ff., 287, 291, 297, 315
m*n-Raster 108 ff., 128, 131, 133, 136, 142, 145, 148, 155, 168, 172, 180, 183, 184, 248
natürliche Geometrie 37
nested sprites 288 f.
NodeBox 322 f.
Nouvelle Tendences 284
Objektbereich 36, 227, 289, 307
objektorientierte Programmierung 30, 227
Objektorientierung 40, 215, 227
Op Art 5, 7, 8, 14, 21, 61, 108, 200, 269, 278, 297
Oscillons 14, 113
Parameter 7, 21, 24, 53, 84, 114, 127, 146, 166, 169, 214, 218, 276, 292, 310
Parameterkunst 7, 214, 218
Parametrisierung 53 f., 165, 166, 168
Pixel 27, 43, 74, 84, 85, 124, 231, 312, 354
Plotter 14, 15, 19, 20, 21 f., 24, 70, 124, 177, 251, 272, 276, 280, 298, 322
Polygone 39, 55, 99 f., 208, 315, 322, 324, 335
Processing 25, 154, 321, 325, 326, 331
Programmbereich 36, 42, 43, 51, 139, 227, 337, 338
Programmierkonzepte 5, 6, 7, 27 f., 40
Prozedur 5, 7, 26, 33, 50, 54, 53, 56, 66, 146 f., 166, 227, 248, 327
Prozeduren als Daten 5, 7, 146 ff., 327
Pseudozufallszahlen 67
Punkte 6, 21, 25, 28, 38, 39, 52, 61, 84 ff., 96, 87, 101, 108, 122, 127, 138, 155, 180, 182, 236 ff., 276, 279, 285, 309, 361
recodeArt 25
ReCode Project 25
Recoding 1, 4, 8, 24 ff., 32, 55, 61, 102, 113 ff., 135, 151, 153, 160, 168, 189, 200, 202, 209, 212, 236, 245, 284, 297, 308, 327
191, 194, 199, 201, 207, 281, 296, 308, 311
Rekursion 5, 7, 153, 161 ff., 168, 319
Remixing 1, 4, 6 f., 24 f., 32, 64, 70, 113, 119, 130, 142, 154, 158, 168, 188, 195, 200, 209, 212, 231, 245, 278, 293, 297, 302, 314, 327, 334
RGB-Modell 70, 72, 73, 158, 192, 195, 302, 310, 354
Schildkrötengrafik 1, 4, 5, 22, 31, 32, 37 ff., 40, 115, 149, 177, 319, 333
Scratch 5, 6, 27, 29, 30, 34, 214, 322, 330, 331, 341, 342
Simultankontrast 225
Skriptvariable 83, 155, 167, 241, 338
Snap! 1, 2, 4 ff., 29 ff., 35 ff., 40 ff., 66 ff., 84, 101, 108, 112, 116, 119, 124, 139, 146, 162, 177, 180, 182, 201, 227, 231, 240, 254, 259, 288, 314, 315, 322, 324, 327, 331, 332, 337
SNE COMP ART 2, 7, 24, 25, 214 ff., 262
Sprache G 5, 7, 177 ff., 192, 262, 363
Sprite 5, 36, 69, 138, 227 ff., 240, 242, 254, 267, 284 ff., 294, 307, 309
Strecke 39, 96 ff., 99, 108, 161, 180 f.
Streckenzüge 99 ff., 135, 181
Turtle 1, 4, 22, 31, 322
UCB Logo 29, 66
Variable 28, 50, 53, 54, 56, 59, 166, 182, 324, 338
lokale Variable 54, 83, 166, 167
globale Variable 54, 56, 115, 166, 167, 168
Verallgemeinerung 5, 6, 40, 45, 48, 53 ff., 75, 165, 166, 168
Vereinfachung 54
Vergleichsoperatoren 59, 351
Vibrobot 320
visuelles Programmieren 1, 29, 32 ff., 322, 334
vvvv 322 f.
Wall Drawings 21, 98
Wiederholung 5, 6, 15, 25, 28, 40, 45ff., 53, 92, 98, 127, 161, 181, 201, 209, 278, 327
Wiederholungsmuster 25, 259
Zeichenroboter 8, 21, 98, 124, 322, 328
Zufallszahlen 66, 89, 156 f., 185, 193, 229, 331, 361 f.
Zufallszahlengenerator 25, 66 f., 80, 101 f.

BEFEHLSREGISTER

Antwort 246
anzeigen 126
ändere Farbstärke um 74
ändere Größe um x 138
ändere Stiftdicke um 74
ändere Stiftfarbe um x 72
ändere Variablenname um Wert 167
ändere x um 49
ändere y um 49
arithmetische Operationen + - x / 55
Aufgerundet bzw. Abgerundet 68
berichte 68
drehe links x Grad 42
drehe rechts x Grad 42
Element n von Liste 182
entferne diesen Klon 235
entferne Element aus strichliste 148
erstelle Kostuem aus stiftspuren 202
falls wahr ... 59
falls wahr ... sonst 59
färbe hintergrund 74
fortlaufend 48
frage und warte 246
füge Element zu Liste hinzu 182
füge Element als n. in Liste ein 182
füge ... zu liste hinzu 148
führe ... aus 148
für .. schritt .. bis 188
gehe x Schritte 42
gehe zu x: y: 49
klone mich/Objekt 235
Kostüm Nr. 138
kreis um x y radius 105
kreisbogen von x y radius winkel 105
Liste 182
logische Verknüpfungen und, oder, nicht 59
male aus 93
mathematische Funktionen 68
Maustaste gedrückt? 244
Maus x-Position 306
Maus y-Position 306
nächstes Kostüm 138
n-eck gefuellt um x y n radius 160
n-eck-rekursivgefuellt um x y n radius 160
n-eck um x y n radius 160
Nested Sprites 289
Neues Sprite hinzufügen 230
Neues Sprite zeichnen 230
Neue Variable 56
Pixel bei x y 84
pralle vom Rand ab 241
Punkt der Ausdehnung a bei x y 84
punktkreis um x y radius 93
punktscheibe um x y radius 93
punktscheibe-rekursivgefuellt um x y radius 93
Ringe 146
rundbogen um x y radius winkel 105
sende ... an alle 230
sende nachricht an alle 230
setze Farbstärke auf 74
setze Größe auf x % 138
setze linienende auf 74
setze name_liste auf Liste 182
setze Stiftdicke auf 74
setze Stiftfarbe auf ... 72
setze stiftfarbe auf h: s: v: 72
setze stiftfarbe auf r: g: b: 72
setze Stiftfarbe auf x 72
setze Variable auf Wert 56
setze Variablenname auf Wert 167
Skriptvariablen 83, 167
springe nach x y 192
stemple 138
Stift hoch 44
Stift runter 44
stiftspuren 202
stoppe alles 153
strecke laenge richtung 96
strecke nach xp yp 96
strecke von A laenge richtung 96
strecke von A nach B 96
Taste Leertaste gedrückt? 244
Text der Größe 246
Vergleichsoperatoren <, =, > 59
verstecken 126
verstecke Variable167
wenn angeklickt, 44
Wenn ich angeklickt werde 226
Wenn ich geklont werde 235

Wenn ich nachricht empfange 230
Wenn Taste gedrückt 226
wiederhole bis 48
wiederhole x mal 48
wische 44
x-Position 49
y-Position 49
zeige auf ... 262
zeige Richtung 49
zeige Variable 167
ziehe Kostüm an 138
Zufallszahl von x bis y 83

Über dieses Buch:

Programmieren lernen und dabei von Anfang an ästhetische grafische Objekte erzeugen? Dieses Buch will zeigen, dass und wie es funktioniert. Die visuelle Programmierumgebung *Snap!* bietet mit der Schildkrötengrafik einen idealen Einstieg für das Erzeugen ansprechender Grafiken. Leistungsfähige Konzepte und anwendungsorientierte Erweiterungen werden jeweils dann eingeführt, wenn sie für das *Recoding* interessanter Beispiele der frühen Computerkunst benötigt werden. Das Erlernte kann dann auf das *Remixing*, d.h. die Erweiterung und Variation dieser Beispiele sowie die Entwicklung eigener Werke angewandt werden. Herausgekommen ist so eine Mischung aus Bildband und Programmierleitfaden.

Über den Autor:

Joachim Wedekind ist zwar ein Silver Surfer, aber auch ein Digital Native, der als Anwender und Gelegenheitsprogrammierer die Entwicklung der digitalen Medien von der Zeit der Personalcomputer (wie APPLE II) über die Einführung des Internet, des WWW bis zu Smartphones und Sozialen Medien begleitet hat. Als Unterrichtstechnologe und Mediendidaktiker lernte er früh die Programmiersprache *Logo* und den *Konstruktionismus* von *Seymour Papert* kennen. Das ist die Basis, auf der er sein Interesse an geometrisch-abstrakten Kunstformen mit der Freude am Programmieren koppeln konnte.