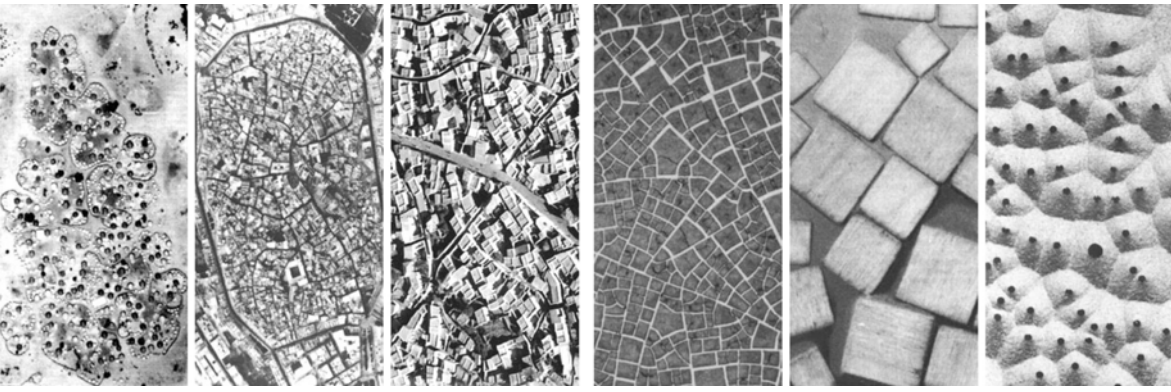
# Computergenerierte Strukturformen v.01


Reinhard König und Christian Bauriedel



# Inhalt

Seite

# Einleitung

Mit der Entwicklung des Internets und der Programmiersprache Java hat sich eine breite Gemeinschaft zusammengefunden, die sich auf experimenteller Weise mit den Möglichkeiten auseinandersetzt, die sich durch die relativ einfache Handhabung der auf Java basierenden Scriptsprachen eröffnen. In der vorliegenden Untersuchung werden hauptsächlich die beiden verwandten Sprachen Actionscript welches Bestandteil von Macromedia Flash ist, und Processing, dass am MIT als Weiterentwicklung der Schulungsumgebung dbn -design by numbers- entstanden ist.
In diesem Umfeld wachsen die unterschiedlichsten Versuche heran, neue Ausdrucksmöglichkeiten und Gestaltungsformen auszuprobieren, die sich im Spannungsfeld zwischen mathematischen Gesetzmäßigkeiten und den einfach zugänglichen, grafischen Möglichkeiten der Scriptsprachen bewegen. Betrachtet man die Ergebnisse dieser vielfältigen Prozesses mit den Augen eines Architekten entdeckt man immer wieder zahlreiche Analogien zu architektonischen und städtebaulichen Themen, die bereits seit längerer Zeit Gegenstand verschiedener Forschungsbereiche sind. Besonders erwähnenswert erscheinen uns hier die Arbeiten des Instituts für Leichte Flächentragwerke (IL) an der Universität Stuttgart, wo bereits vor 30 Jahren unter der Leitung von Frei Otto damit begonnen wurde, natürliche Prozesse und deren zugrunde liegende -oftmals mathematische- Strukturen zu erforschen und auf konstruktive und entwerferische Bereiche in der Architektur zu übertragen. Außerdem erlangen die Modelle und Theorien, die in den 60er Jahren unter dem Sammelbegriff Strukturalismus entstanden sind neues Interesse, da sie sich jetzt mit Hilfe des Computers als dynamische Prozesse darstellen und verstehen lassen.
Aus dieser Motivation entstand die Idee, eine Sammlung der verstreuten Programme anzulegen, zu untersuchen und deren Möglichkeiten aufzuzeigen und daraus im nächsten Schritt eigene arbeitsunterstützende, computergenerierte Entwurfsprogramme abzuleiten.

Ziel dieser Arbeit ist es, verschiedene, viel versprechende Ansätze die wir in Bereichen der Computergrafik und Programmierung finden zusammenzutragen und ihre Funktionsweise zu verstehen.
Im Anschluss daran versuchen wir über die Herstellung von strukturellen Analogien zu natürlichen oder künstlichen Gebilden und Strukturen - die einen abstrakten Bezug zu Themenbereichen der Architektur und des Städtebaus zulassen - mögliche Anwendungsgebiete abzustecken.
In einem weiteren Schritt sollen dann punktuelle Ziele für eine Umsetzung anhand computergenerierter Entwurfsmodelle formuliert werden um gezielt einzelne Anwendungen umzusetzen. Um zu verhindern bereits Bestehende Ansätze zu wiederholen wird diese Arbeitsphase von einer Recherche vorhandener Forschungsarbeiten begleitet werden.

Der erste Abschnitt beinhaltet die Sammlung vorgefundener Programme, die unserer Meinung nach das Potential für eine Anwendung auf architektonische oder städtebauliche Problemstellungen in sich tragen, oder zumindest eine Struktursimulation und Analogiebildung noch undefinierter Anwendungsbereiche zulassen.

Im zweiten Abschnitt finden sich erste Programme, die wir selbst erstellt haben um gezielte Studien durchzuführen und bereits in einer frühen Phase Möglichkeitsfelder auszuloten.

Reinhard König und Christian Bauriedel

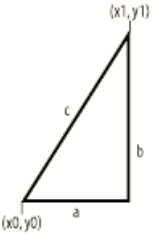
# Grundlagen

In diesem Abschnitt sind jene mathematischen Methoden zusammengefasst, die es erlauben einfache Operationen durchzuführen, die in den Programmbeispielen der folgenden Kapiteln immer wieder innerhalb komplexerer Code-Strukturen auftauchen werden.

## Entfernung zwischen zwei Punkten

Diese Berechnung wird sehr häufig benötigt und beruht auf dem Satz von Pythagoras:

$$a^2 + b^2 = c^2$$




Um nun die Länge der Strecke zwischen zwei beliebigen Punkten zu erhalten benötigt man die Quadratwurzel von ( $a^2 + b^2$).
In der ActionScript Syntax stellt sich das folgendermaßen dar:

```
var c = Math.sqrt(Math.pow(a, 2) + Math.pow(b, 2));
```

Für die Berechnung der beiden Seitenlängen a und b in einem Koordinatensystem verwenden wir die Differenz der x und y Koordinaten. Somit wird die Strecke b durch die Differenz der y Koordinaten definiert.
Hat man die Längen a und b lässt sich daraus mit der oben angegebenen Formel der Abstand der beiden Punkte berechnen.
Gestaltet man die Berechnung als eine wieder verwendbare Methode sieht der Code wie folgt aus:

```
Math.getDistance = function (x0, y0, x1, y1) {

  // Calculate the lengths of the legs of the right triangle.
  var dx = x1 - x0;
  var dy = y1 - y0;

  // Find the sum of the squares of the legs of the triangle.
  var sqr = Math.pow(dx, 2) + Math.pow(dy, 2);

  // Return the square root of the sqr value.
  return (Math.sqrt(sqr));
};
```

Die Methode kann dann beispielsweise so aufgerufen werden:

```
trace(Math.getDistance(300, 400, 0, 0));  // Displays: 500
```

## Schnittpunkt zweier Geraden

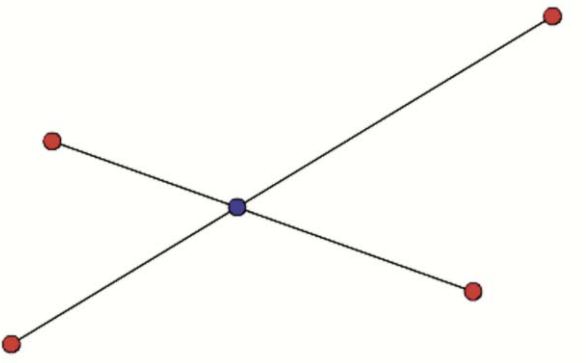
Für eine Gerade im x-y-Koordinatensystem gilt die Geradengleichung:

y=mx+b  (b= der Schnittpunkt mit der y-achse, m die Steigung)

Für die Schnittpunktberechnung stellt man zwei Geradengleichungen auf, die dann gleichgesetzt werden:

G1:
y=m1x+b1
G2:
y=m2x+b2

daraus kann man ableiten:
m1xs+b1=m2xs+b2
woraus sich im Koordinatensystem ergibt:
x=(m*a1x-a1y-b1x*m2+b1y)/(m-m2)



Übersetzt man diese Formel in ActionScript-Code ergibt sich:

```
m = (A2y-A1y)/(A2x-A1x); //steigung m=deltaY/deltaX
m2 = (B2y-B1y)/(B2x-B1x);
xstelle = (m*a1x-a1y-b1x*m2+b1y)/(m-m2);
ystelle = m*(xstelle-A1x)+A1y;
```

## Brownsche Bewegung

Mittels der Brownschen Bewegungsgleichung kann man eine zufällige Bewegung - beispielsweise die eines Punktes - erzeugen.

Hier der entsprechende Code in ActionScript:

```
damp = .95;
dot.onEnterFrame = brown;
//
function brown() {
      this.vx += Math.random()*0.6-0.3;
      this.vy += Math.random()*0.6-0.3;
      this.vx *= damp;
      this.vy *= damp;
      this._x += this.vx;
      this._y += this.vy;
}
```

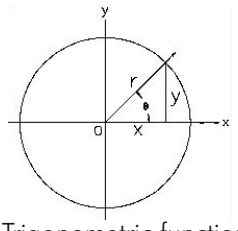
## Trigonometric Functions

This topic is a quick review for readers who need a reminder about this area of mathematics. If you''92re familiar with trigonometry, you can skip this topic. If you find this topic difficult to follow, you might consult a more basic reference on mathematics.

Trigonometric functions are principally used to model or describe:
· The relation between angles in a triangle (hence the name).
· Rotations about a circle, including locations given in polar coordinates.
· Cyclical or periodic values, such as sound waves.

The three basic trigonometric functions are derived from an angle rotating about a unit circle.

$$\sin\theta = \frac{y}{r} \quad \cos\theta = \frac{x}{r} \quad \tan\theta = \frac{y}{x}$$



Trigonometric functions based on the unit circle

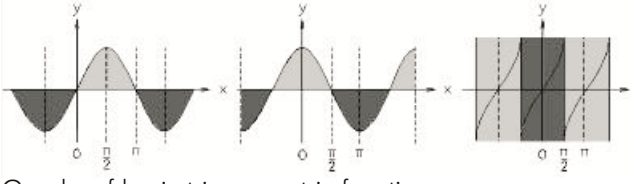
The tangent function is undefined for x = 0. Another way to define the target is:

$$\tan\theta = \frac{\sin\theta}{\cos\theta}$$

Because XYR defines a right-angled triangle, the relation between the sine and cosine is:

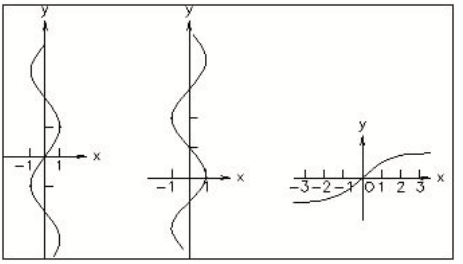
$$(\cos\theta)^2 + (\sin\theta)^2 = 1$$

The graphs of the basic trigonometric functions illustrate their cyclical nature.



Graphs of basic trigonometric functions

The sine and cosine functions yield the same values, but the phase differs along the X-axis by ?/2: in other words, 90 degrees.
The inverse functions for the trigonometric functions are the arc functions—'97the inverse only applies to values of x restricted by -?/2 = X = ?/2. The graphs for these functions appear like the basic trigonometric function graphs, but turned on their sides.



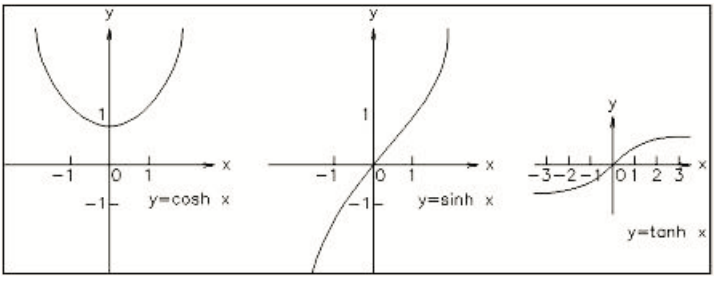
Graphs of basic arc functions

The hyperbolic functions are based on the exponential constant *e* instead of on circular measurement. However, they behave similarly to the trigonometric functions and are named for them. The basic hyperbolic functions are:

$$\sinh x = \frac{e^{x} + e^{-x}}{2}$$

$$\cosh x = \frac{e^{x} - e^{-x}}{2}$$

$$\tanh x = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} = \frac{\sinh x}{\cosh x}$$




Graphs of basic hyperbolic functions

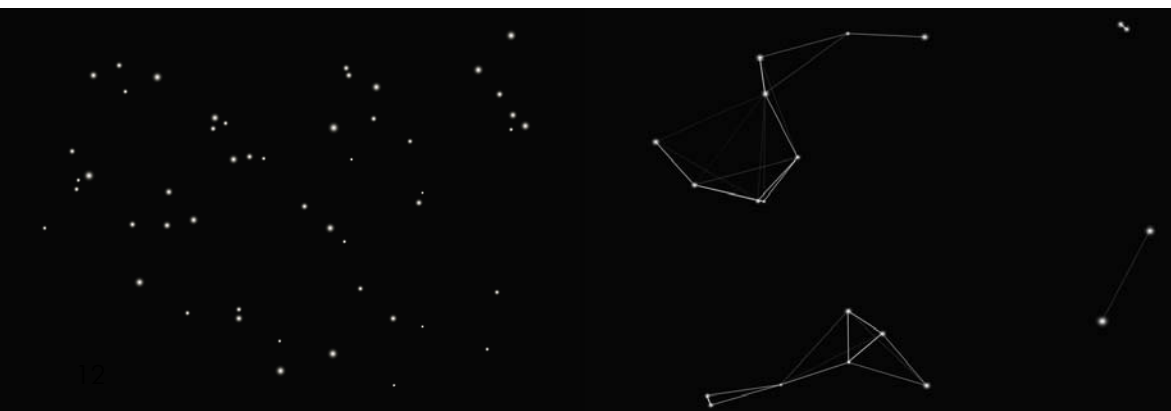
Der Abschnitt über Trigonometrische Funktionen ist der Hilfe von 3d Studio MAXscript 6.0 entnommen.

01

# node garden

code by bit-101 www.bit-101.com

```
damp = .999;
k = .001;
numNodes = 25;
nodez = [];
for(i=0;i<numNodes;i++){
    node = attachMovie("node", "n"+i, i);
    nodez[i] = node;
    node._x = Math.random()*550;
    node._y = Math.random()*400;
    node._xscale=node._yscale=Math.random()*100+50;
    node.vx = Math.random()*10-5;
    node.vy = Math.random()*10-5;
    node.onEnterFrame = nodeMove;
}
minDist = 120;
_root.onEnterFrame = function(){
    var i, j, nodeA, nodeB, dx, dy, dist, alpha, ax, ay;
    clear();
    for(i=0;i<nodez.length-1;i++){
        nodeA = nodez[i];
        for(j=i+1;j<nodez.length;j++){
            nodeB = nodez[j];
            dx = nodeB._x - nodeA._x;
            dy = nodeB._y - nodeA._y;
            dist = Math.sqrt(dx*dx+dy*dy);
            if(dist<minDist){
                alpha = 100-dist/minDist*100;
                lineStyle(1, 0xffffff, alpha);
                moveTo(nodeA._x, nodeA._y);
                lineTo(nodeB._x, nodeB._y);
                ax = dx*k*.5;
                ay = dy*k*.5;
                nodeA.vx += ax;
                nodeA.vy += ay;
                nodeB.vx -= ax;
                nodeB.vy -= ay;
            }
        }
    }
}
function nodeMove(){
    this.vx *= damp;
    this.vy *= damp;
    this._x += this.vx;
    this._y += this.vy;
    if(this._x>550){
        this._x = 0;
    } else if(this._x<0){
        this._x = 540;
    }
    if(this._y>400){
        this._y = 0;
    } else if(this._y<0){
        this._y = 400;
    }
}
```



Was ist ein node? Eine Einheit, eine Entität, die verschiedene Verhaltensweisen beinhalten kann. Nodes können sich verbinden, interagieren und miteinander kommunizieren. Ein node an sich hat keine besonders vielfältigen Möglichkeiten zu agieren, erst durch die Interaktion des gesamten Systems erzeugt interessante Strukturen.

Der erste Programmteil erzeugt eine Gruppe zufällig auf der Bühne verteilter Nodes:

```
numNodes = 50;
for(i=0;i<numNodes;i++){
    node = attachMovie("node", "n"+i, i);
    node._x = Math.random()*550;
    node._y = Math.random()*400;
    node._xscale=node._yscale=Math.random()*100+50;
}
```

Damit haben wir den ersten ‚node garden' erzeugt. Bis jetzt sieht dieser zwar lediglich wie ein Sternenfeld aus, da die nodes noch statisch verteilt sind. Aber jeder „Stern" kann mit Eigenschaften belegt und somit bewegt, skaliert und eingeblendet werden. Außerdem kann jeder node Funktionen aufnehmen, die tastatur- oder mausgesteuerte Animationen und vieles mehr erlauben.

## Interaktion

Im nächsten Schritt programmieren wir eine node Interaktion. Dafür wird eine Form der Kommunikation auf der untersten Ebene benötigt. Ein node kommuniziert mit einem Anderen - und jeder node kann mit jedem beliebigen node Informationen austauschen, muss dies aber nicht zwangsläufig.
Das Vorgehen gliedert sich folgendermassen:
1. Zwei nodes räumlich einander annähern.
2. Sie müssen sich entscheiden, ob sie miteinander kommunizieren wollen oder nicht.
3. Sie kommunizieren miteinander.
4. Zwei neue nodes werden betrachtet und mit Punkt 1. vortgefahren.

Die erste Herausforderung ist sicherzustellen, dass sich jeder node mit jedem anderen austauscht. Für eine einfache Handhabung platzieren wir die nodes in einem Array. Dadurch benötigen wir keine dynamische Namenszuweisung sondern könne einfach das Array mit einer Schleife durchlaufen. Die folgende Zuweisung in ein Array berücksichtigt bereits, dass kein node-Paar mehrfach miteinander verglichen wird:

```
for(i=0;i<nodez.length-1;i++){
    nodeA = nodez[i];
    for(j=i+1;j<nodez.length;j++){
        nodeB = nodez[j];
        // now match up nodeA and nodeB
        // and have them communicate
    }
}
```

Jetzt können wir zum zweiten Schritt übergehen:

2. Sie müssen sich entscheiden, ob sie miteinander kommunizieren wollen oder nicht.

Nun, was kann für einen node eine Entscheidung bedeuten? Jede Entscheidung beruht auf einer Bedingung. Wenn (irgendetwas) zutrifft, dann kommunizieren wir. Es bleibt freigestellt, was diese Bedingung ist. Die Tatsache, dass dies alles sein kann macht den Reiz eines node garden aus, da sich dadurch ein großer Raum an Möglichkeiten eröffnet:
- Wenn das eine node rot ist und das andere grün kommunizieren sie.
- Wenn beide eine bestimmte größe haben, kommunizieren sie.
- Wenn sich beide in dieselbe Richtung bewegen oder wenn sie beide den gleichen Wert besitzen (welcher durch vorhergehende Prozesse zugewiesen wurde) dann sollen sie kommunizieren.

Wir beginnen mit einem einfachen Beispiel. Wenn sie nah beieinander sind, sollen sie miteinander kommunizieren. Das ist eine der grundlegendsten Beziehungen. Wir messen den Abstand zwischen zwei nodes und wenn dieser geringer ist als ein bestimmter Wert, dann kommunizieren sie miteinander. Wir setzen dies folgendermaßen um:

```
minDist = 80;
for(i=0;i<nodez.length-1;i++){
    nodeA = nodez[i];
    for(j=i+1;j<nodez.length;j++){
        nodeB = nodez[j];
        dx = nodeB._x – nodeA._x;
        dy = nodeB._y – nodeA._y;
        dist = Math.sqrt(dx*dx+dy*dy);
        if(dist<minDist){
            // communicate
        }
    }
}
```

## Was ist Kommunikation?

Kann man in diesem Zusammenhang von Kommunikation sprechen? Generell versteht man darunter, dass eine beliebige Information von einer Person oder einem Terminal zu einem anderen gelangt. In unserem Fall können wir dies etwas präzisieren und sagen, dass es eine Reaktion zwischen den beiden oder bei beiden auslöst. Beispiele der Kommunikation zwischen nodes sind: die nodes ziehen sich gegenseitig an oder stoßen sich ab, sie können ihre Farbe oder Größe ändern. Sie können auch buchstäblich Informationen austauschen - jeder node besitzt beispielsweise einen Wert wie „Energie“, und wenn sie miteinander kommunizieren kann jener mit mehr Energie diese an den Anderen abgeben. Dieser Vorgang kann wahrscheinlich am besten durch eine Visualisierung repräsentiert werden, die den Energiezustand mittels einem Farbwert oder der Skalierung ausdrückt.

Wir greifen wieder ein sehr einfaches Thema für eine Kommunikation auf. Sie wird schlicht durch das zeichnen einer Linie zwischen den beiden nodes gezeigt. Zusätzlich verändern wir die Helligkeit der Linie in Abhängigkeit von der Entfernung. Der alpha-Wert wird zwischen 100 (für die Distanz 0) und 0 (wenn es die minDist erreicht). Der entsprechende Programmkode gestaltet sich wie folgt:

```
numNodes = 50;
nodez = [];
for(i=0;i<numNodes;i++){
    node = attachMovie(„node“, „n“+i, i);
    nodez[i] = node;
    node._x = Math.random()*550;
    node._y = Math.random()*400;
    node._xscale=node._yscale=Math.random()*100+50;
}
minDist = 80;
for(i=0;i<nodez.length-1;i++){
    nodeA = nodez[i];
    for(j=i+1;j<nodez.length;j++){
        nodeB = nodez[j];
        dx = nodeB._x – nodeA._x;
        dy = nodeB._y – nodeA._y;
        dist = Math.sqrt(dx*dx+dy*dy);
        if(dist<minDist){
            alpha = 100-dist/minDist*100;
            lineStyle(1, 0xffffff, alpha);
            moveTo(nodeA._x, nodeA._y);
            lineTo(nodeB._x, nodeB._y);
        }
    }
}
```

## Animierte Nodes

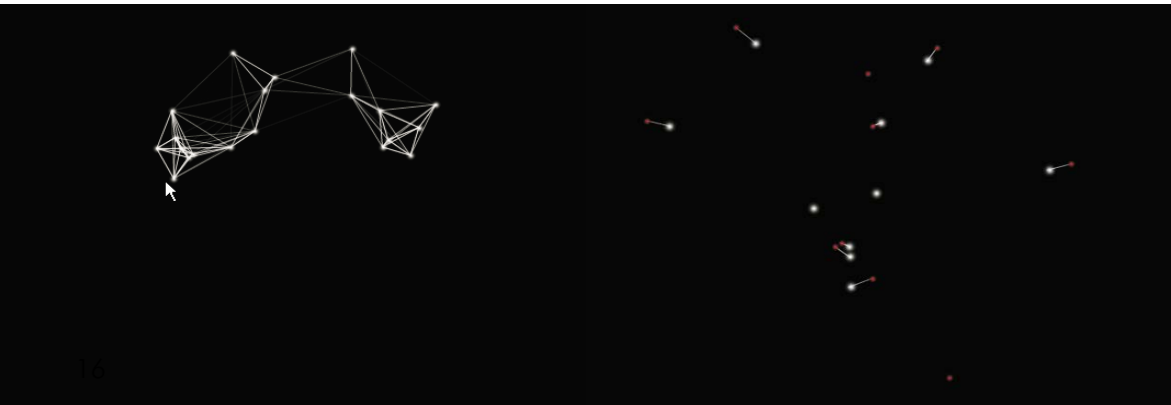
Soweit haben wir ein schönes, aber statisches Bild erzeugt. Dieses können wir jetzt animieren, indem wir jedem node eine Funktion zuweisen, die bei onEnterFrame ausgeführt wird. Diese Funktion nennen wir „nodeMover“. Sie wird dafür verantwortlich sein, die nodes über den Bildschirm zu bewegen. Zudem wird jedem node ein zufälliger vx und vy Wert zugewiesen, der ihm eine Geschwindigkeit auf jeder Achse zuweist. Zum Schluss verschachteln wir den Kommunikationsprogrammteil in einer Funktion die _root.onEnterFrame zugewiesen wird. Das sorgt dafür, dass die nodes ihre Beziehungen aktualisieren. Hier das gesamte Programm:

```
nodez = [];
for(i=0;i<numNodes;i++){
    node = attachMovie(“node”, “n”+i, i);
    nodez[i] = node;
    node._x = Math.random()*550;
    node._y = Math.random()*400;
    node._xscale=node._yscale=Math.random()*100+50;
    node.vx = Math.random()*10-5;
    node.vy = Math.random()*10-5;
    node.onEnterFrame = nodeMove;
}

minDist = 120;
_root.onEnterFrame = function(){
    var i, j, nodeA, nodeB, dx, dy, dist, alpha;
    clear();
    for(i=0;i<nodez.length-1;i++){
        nodeA = nodez[i];
        for(j=i+1;j<nodez.length;j++){
            nodeB = nodez[j];
            dx = nodeB._x – nodeA._x;
```

```
            dy = nodeB._y - nodeA._y;
            dist = Math.sqrt(dx*dx+dy*dy);
            if(dist<minDist){
                alpha = 100-dist/minDist*100;
                lineStyle(1, 0xffffff, alpha);
                moveTo(nodeA._x, nodeA._y);
                lineTo(nodeB._x, nodeB._y);
            }
        }
    }
}
function nodeMove(){
    this._x += this.vx;
    this._y += this.vy;
    if(this._x>550){
        this._x = 0;
    } else if(this._x<0){
        this._x = 540;
    }
    if(this._y>400){
        this._y = 0;
    } else if(this._y<0){
        this._y = 400;
    }
}
```

## Node Anziehung

Wie bereits erwähnt ist eine weitere Idee für eine node Kommunikation, dass sie sich gegenseitig anziehen oder abstoßen. Im folgenden Beispiel verwenden wir dafür Elastizität welche bereits im Grundlagenabschnitt vorgestellt wurde. Wenn die nodes nun kommunizieren werden sie nicht nur durch eine Linie verbunden sondern bewegen sich aufeinander zu. Um diese Bewegung zu definieren verwenden wir ein Federverhalten. Die eine Hälfte der Federspannung wird dabei auf den einen, die andere Hälfte auf den anderen node verteilt. Für die Definition einer Feder benötigen wir einen Faktor k für die Federkraft und einen Dämpfungswert (damp). Diese Werte führen wir als Variablen zu beginn des Programms ein.
Die Werte, die wir hierfür gewählt haben ergeben ein interessantes Ergebnis, da sich das System ausgewogen verhält. Spielen Sie mit den Werten herum und es wird sich zeigen, das es zum einen dazu kommen kann, das sich das System auflöst und zum anderen kann es kollabieren.



```
damp = .999;
k = .001;
numNodes = 25;
nodez = [];
for(i=0;i<numNodes;i++){
    node = attachMovie("node", "n"+i, i);
    nodez[i] = node;
    node._x = Math.random()*550;
    node._y = Math.random()*400;
    node._xscale=node._yscale=Math.random()*100+50;
    node.vx = Math.random()*10-5;
    node.vy = Math.random()*10-5;
    node.onEnterFrame = nodeMove;
}
minDist = 120;
_root.onEnterFrame = function(){
    var i, j, nodeA, nodeB, dx, dy, dist, alpha, ax, ay;
    clear();
    for(i=0;i<nodez.length-1;i++){
        nodeA = nodez[i];
        for(j=i+1;j<nodez.length;j++){
            nodeB = nodez[j];
            dx = nodeB._x – nodeA._x;
            dy = nodeB._y – nodeA._y;
            dist = Math.sqrt(dx*dx+dy*dy);
            if(dist<minDist){
                alpha = 100-dist/minDist*100;
                lineStyle(1, 0xffffff, alpha);
                moveTo(nodeA._x, nodeA._y);
                lineTo(nodeB._x, nodeB._y);
                ax = dx*k*.5;
                ay = dy*k*.5;
                nodeA.vx += ax;
                nodeA.vy += ay;
                nodeB.vx -= ax;
                nodeB.vy -= ay;
            }
        }
    }
}
function nodeMove(){
    this.vx *= damp;
    this.vy *= damp;
    this._x += this.vx;
    this._y += this.vy;
    if(this._x>550){
        this._x = 0;
    } else if(this._x<0){
        this._x = 540;
    }
    if(this._y>400){
        this._y = 0;
    } else if(this._y<0){
        this._y = 400;
    }
}
```

Die restlichen Veränderungen finden sich hauptsächlich im Kommunikationsteil. Hierfür möchten wir den Codeabschnitt genauer erläutern:

```
ax = dx*k*.5;
ay = dy*k*.5;
nodeA.vx += ax;
nodeA.vy += ay;
nodeB.vx -= ax;
nodeB.vy -= ay;
```

Zuerst berechnen wir den Beschleunigungsfaktor auf jeder Achse, ax und ay. Dazu nehmen wir eine Teil des Abstandes zur Zielposition, oder mul-

tiplizieren die Distanz um k. Anschließend multiplizieren wir den Wert mit 0.5, da jedem node nur die Hälfte der Beschleunigung zugewiesen wird. Die schwierigere Aufgabe besteht jetzt darin, die Kräfte richtig zu verteilen. Grundsätzlich möchten wir einem node eine Beschleunigung zuweisen und den anderen abbremsen, damit sie sich einander annähern. Welcher node die entsprechende Zuweisung benötigt hängt von der Subtraktionsreihenfolge bei der Abstandsberechnung ab. Vertauscht man die beide, dann kehrt sich der Effekt um und die nodes prallen voneinander ab:

```
dx = nodeB._x – nodeA._x;
dy = nodeB._y – nodeA._y;
```

In diesem Fall ziehen wir A von B ab. Somit müssen wir die resultierende Beschleunigung zu A hinzufügen und den entsprechenden node fortzubewegen. Ebenso muss er von node B abgezogen werden um diesen in die Richtung von A zu bewegen. Möchte man eine abstoßende Kraft erzeugen so sind A und B einfach zu vertauschen.

Des weiteren sollen hier einige Möglichkeiten knapp umrissen werden wie mit node-Eigenschaften noch umgegangen werden kann. Eine ist, zwei Typen von nodes zu erschaffen. Hierfür geben wir dem node movie clip zwei Frames und platzieren jeweils einen roten und einen blauen node darin. Während der node-Erstellung wird zufällig die Framenummer 1 oder 2 vergeben. Bei 1 erhält der node den Befehl `node.gotoAndStop(1)` und bekommt die Eigenschaft `node.color = „red“` zugewiesen. Im Auswertungsprozess können die Farben dann verglichen werden:

```
if(nodeA.color == nodeB.color){
```

Wenn sie die gleiche Farbe haben ziehen sie sich an, andernfalls stoßen sie sich ab - oder andersherum. Dieses Verhalten kann wiederum mit anderen, wie einer Abstandsmessung, kombiniert werden.

Eine andere Idee ist eine `„size“`-Eigenschaft für jeden node zu vergeben. Wir können hierfür einen Wert 50 annehmen. Dann kann bei jedem Mal wenn ein node kommuniziert dieser Wert um beispielsweise 5 erhöht werden. In der `nodeMove` Funktion wird zu diesem Zweck der `_xscale` und `_yscale` des nodes auf den `size`-Faktor gesetzt. Anschließend wird size wieder auf 50 gesetzt. Der Effekt der sich daraus ergibt ist, dass je mehr Verbindungen ein node hatte, dieser umso größer er wird. Nodes ohne Kommunikation werden kleiner. Ein entsprechender Effekt kann mit dem `_alpha`-Wert erreicht werden.

Zusätzlich können wir mit der node-Entstehung und Zerstörung experimentieren. Unser bisheriger Code kann dafür einfach verwendet werden, solange sich alle nodes in dem `nodez`-Array befinden, da diesem weitere nodes mit `nodez.push(newNode)` zugefügt werden können.
Um einen node wieder zu entfernen muss dieser zuerst aus dem Array gelöscht werden um dann den movie clip zu entfernen. Um ein Element aus einem Array zu löschen verwendet man gewöhnlich den `splice` Befehl. Dafür benötigt man allerdings den Index, wo das Element

gespeichert ist. Ist dieser bekannt:

```
deadNode = nodez[index];
nodez.splice(index, 1);
deadNode.removeMovieClip();
```

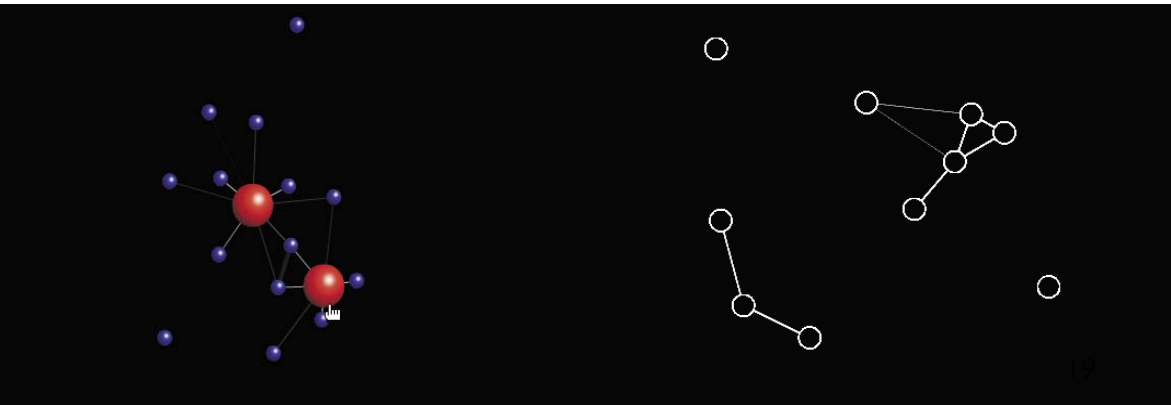
In der zweiten Zeile wird bei `index` begonnen und 1 Element gelöscht.

Ist dagegen der Index nicht bekannt kann der Referenzname des nodes in einer Schleife gesucht werden, die das Array durchläuft und die Namen vergleicht. Zum Beispiel kann dies in der nodeMove Funktion geschehen:

```
// this is inside the nodeMove function:
if(someCondition){
    for(i=0;i<nodez.length;i++){
        if(nodez[i] == this){
            nodez.splice(i, 1);
        }
    }
    this.removeMovieClip();
}
```

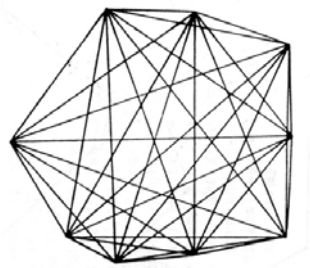
Andere Ideen wären, die nodes nicht nur gegenseitig sondern auch mit anderen Objekten auf der Bühne oder der Maus interagieren zu lassen. Unter den beigefügten Dateien findet sich eine Vielzahl an Beispielen.

Der verwendete Code sowie die Erläuterungen basieren auf einem Tutorial von Keith Peters welches auf Englisch unter www.bit-101.com zu finden ist.

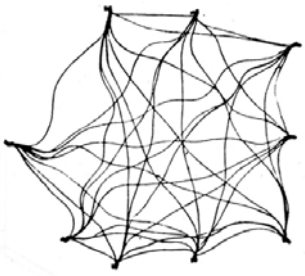


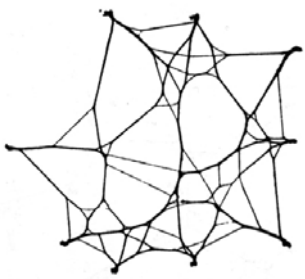
# Strukturanalogien zu ‚node garden'

Untersucht man die unterschiedlichen Wegesysteme, die in der natürlichen und der gebauten Umwelt vorkommen, so ergeben sich drei unterschiedliche Kategorien. Zum einen das Direktwegesystem, bei welchem jeder Knoten mit jedem anderen verbunden ist. Dies hat den Vorteil, dass der Weg zwischen zwei Punkten der kürzest mögliche ist und man keine Umwege zu gehen braucht. Die Gesamtwegelänge wird dagegen bei steigender Knotenzahl sehr hoch und damit steigt auch der Flächenverbrauch, was dieses System nicht sehr effizient macht.



Direktwegesystem

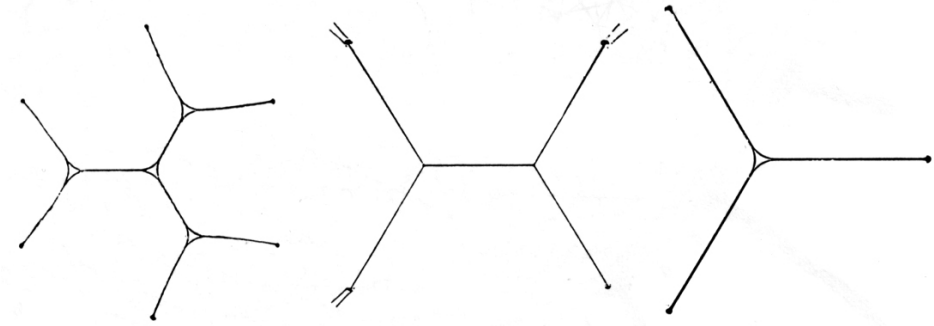




Systeme minimierter Umwege

node garden System

Des weiteren findet man Minimalwegesysteme, deren namensgebende Eigenschaft die minimale Gesamtsystemwegelänge ist. Der Nachteil bei dieser Wegeanordnung ist allerdings, dass die Strecke von einem Punkt zum anderen sehr groß ist, was für ein Wegenetz von Nachteil ist.

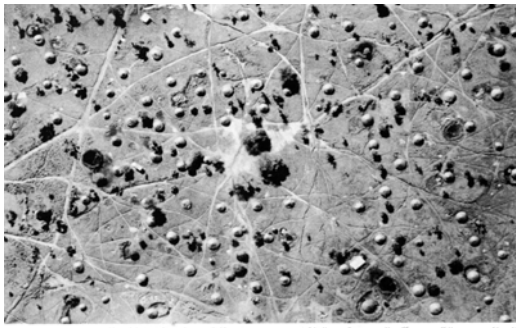


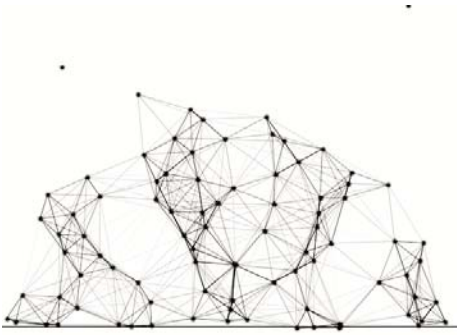
drei Beispiele für Minimalwegesysteme

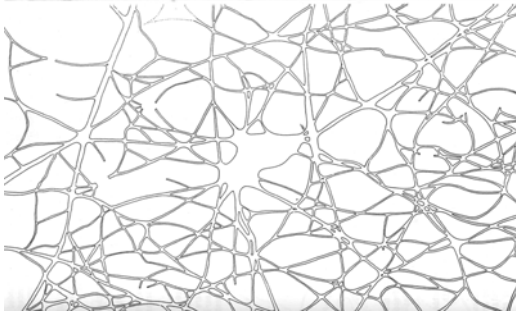
Das dritte System stellen die minimierten Direktwegesysteme dar, deren Prinzip auf dem Zusammenlegen einzelner Wege eines Direktwegesystems beruht. Ein solches Modell stellt die Synthese der günstigen Eigenschaften der beiden vorher genannten Wegenetze dar und verringert deren Nachteile.

Die Angaben beruhen auf der Publikation „Ungeplante Siedlungen, non-plannes Settlements" des Instituts für Leichte Flächentragwerke (IL 39) der Universität Stuttgart, Herausgeber ist Frei Otto. 1990 als Dissertation von Eda Schaur erschienen.

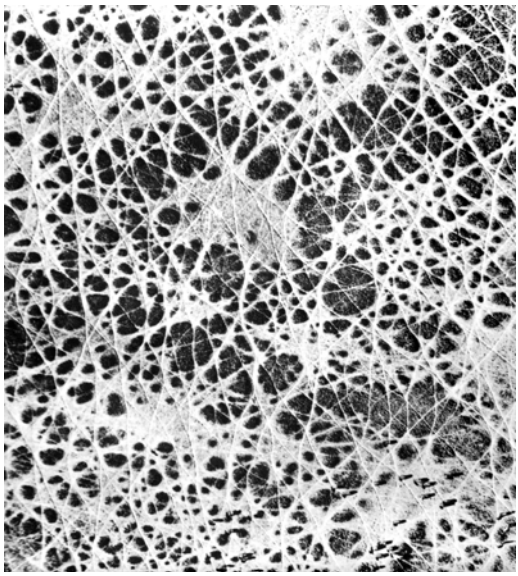
**rechts:** Siedlungsstruktur
**links:** Ergebnis einer node garden Simulation

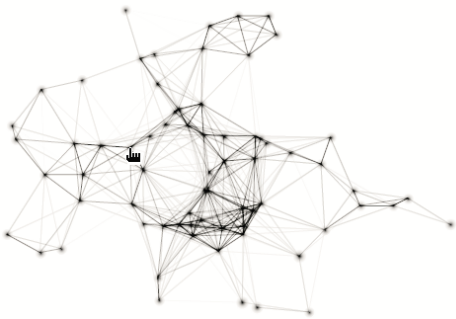


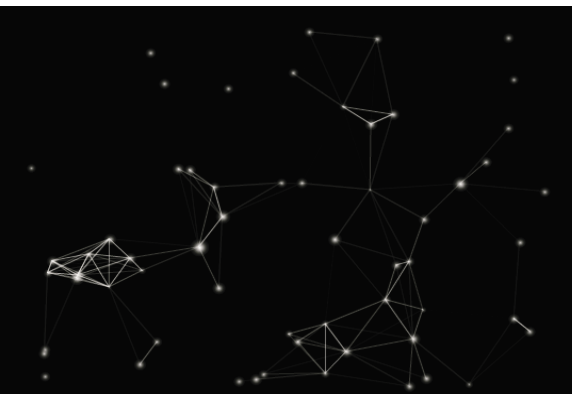




**rechts:** Tier-Trampelpfade als Luftbild und als Systemzeichnung (dickere Striche stehen für eine stärkere Nutzung)
**links:** Ergebnis einer node garden Simulation







# flächenbesetzung

## growing boxes

code by Jared Tarbell, 2002; from www.levitated.net

Another approach to filling a space with boxes uses exhaustive random sampling.

With a specially constructed Movieclip called the auto-expanding box, we can automatically fill a space up by randomly placing instances of the box onto the stage.

Click anywhere to create an auto-expanding box. Other boxes will be created automatically until the stage is near full.

The auto-expanding box increases it's size by precisely 2 pixels each frame. It also checks for collisions with any of the other already expanded boxes by sampling points along its perimeter. If a collision is detected, the box reduces it's size slightly (so as to no longer be in collision) and gives itself a small rotation (for a nice irregular quality).

The color of each box is selected according to its vertical position. There are 16 unique colors available and were taken from a palette photographed in Terlingua Ghost Town, Texas.

A known problem with this system exists in the way collisions are detected. Since only incremental points along the edges of the boxes are checked for collisions, it is sometimes possible for a box to expand beyond an obstacle. This is especially true for larger boxes. Also, two expanding boxes will not collide with each other because at that stage, they are not actually considered to be part of the 'collision construct'.

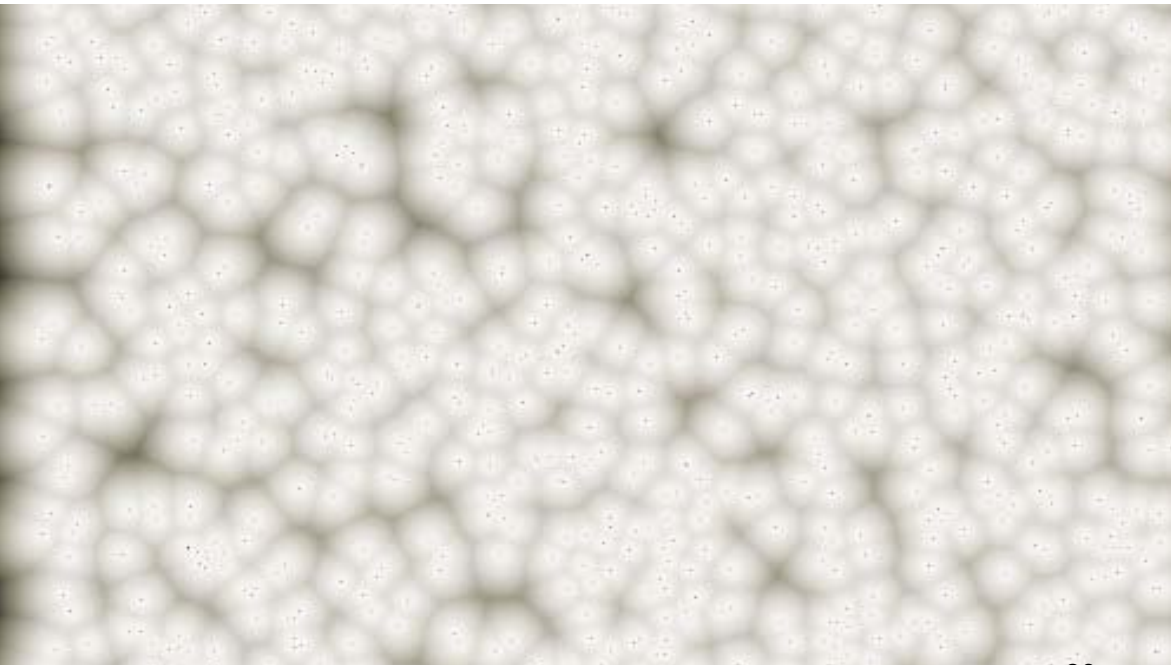
Obviously this is not a very efficient way to fill up a region with squares. It does however create an interesting generative pattern and is fun to watch. This recursive space filling algorithm is probably a better approach if speed and accuracy is important.

¡tarbell, april 2003

## sandpits

code by bit-101 www.bit-101.com

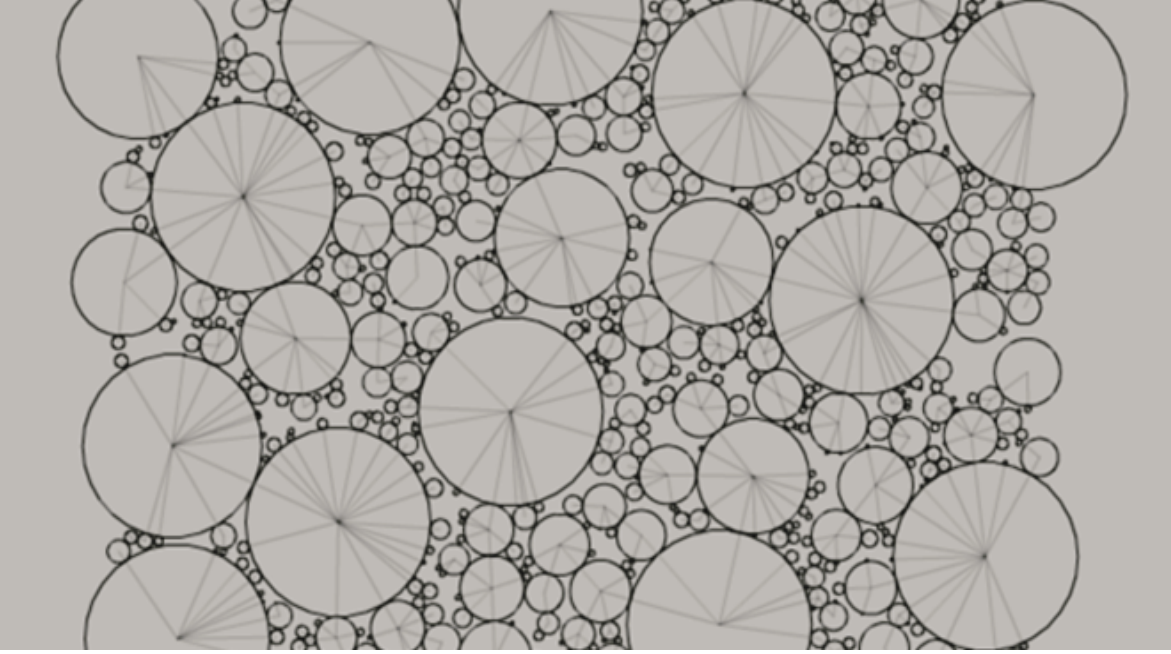
```
dots = [];
maxdep = 23;
clear();
function clear() {
dots[0] = {x:Math.random()*480+40, y:Math.random()*340+40};
    for (depth=0; depth<maxdep; depth++) {
        createEmptyMovieClip("p"+depth, 10000-depth);
    }
    i = 0;
}
onMouseDown = clear;
size = 1;
col = 0xffffff;
onEnterFrame = make;
function make() {
    for (dep=0; dep<maxdep; dep++) {
        plate = _root["p"+dep];
        dot = plate.attachMovie("dot", "d"+i, i);
        dot._x = dots[i].x;
        dot._y = dots[i].y;
        dot._width = dot._height=size;
        dotcol = new Color(dot);
        dotcol.setRGB(col);
        col -= 0x0a0a0b;
        size += 3;
    }
    i++;
    size = 1;
    col = xffffff;
    dots[i] = {x:Math.random()*480+40, y:Math.random()*340+40};
}
function makeFPO(boxobject) {
        // create a similar shape on hitTest object
    nomdos="fpobox"+String(construct.depth++);
    fpo = construct.attachMovie("growbox",nomdos,construct.depth)
    // position in same place
    fpo._x=boxobject._x;
    fpo._y=boxobject._y;
    // back up one size
    fpo._xscale=(boxobject._xscale-1.5)*.975;
    fpo._yscale=(boxobject._yscale-1.5)*.975;
    // rotate to same angle
    fpo._rotation=boxobject._rotation;
    // stop on same color
    fpo.square.gotoAndStop(boxobject.square._currentframe);
}
```



## growing circles

code by bit-101 www.bit-101.com

```
function makeBox(mx, my) {
    // place a square box
    var nombre=”growbox” + String(depth++);
    var neo =  this.attachMovie(“growbox”,nombre,depth);

    // size down to almost nothing
    neo._xscale=1;
    neo._yscale=1;
    // slight rotation (for irregular quality)
    neo._rotation=(random(101)-50)/60;
    // instruct new box to grow until an obstacle is hit
    neo.onEnterFrame = function() {
        this._xscale++;
        this._yscale++;
        if (this._parent.touchingConstruct(this)) {
            // obstacle touched
            // create identical graphic shape
            this._parent.makeFPO(this);
            // self-destruct
            removeMovieClip(this);
        }
    }
    // position until not touching anything
    if ((mx!=null) && (my!=null)) {
        neo._x=mx;
        neo._y=my;
        // set color according to vertical position
        neo.square.gotoAndStop(Math.ceil(neo._y*18/337));
        if (this.construct.hitTest(neo._x,neo._y,true)) {
            neo.removeMovieClip();
        }
    } else {
        // enforce a maximum try rule
        var limiter=0;
        do {
            // random screen position
            neo._x=random(100)+(depth%6)*100;
            neo._y=random(337);
            // set color according to vertical position
            neo.square.gotoAndStop(Math.ceil(neo._y*18/337));
            limiter++;
            if (limiter>200) {
                // give up
                neo.removeMovieClip();
                clearInterval(growInterval);
                break;
            }
        } while (this.construct.hitTest(neo._x,neo._y,true));
    }
}

function makeFPO(boxobject) {
        // create a similar shape on hitTest object
    nomdos=”fpobox”+String(construct.depth++);
    fpo = construct.attachMovie(“growbox”,nomdos,construct.depth)
    // position in same place
    fpo._x=boxobject._x;
    fpo._y=boxobject._y;
    // back up one size
    fpo._xscale=(boxobject._xscale-1.5)*.975;
    fpo._yscale=(boxobject._yscale-1.5)*.975;
    // rotate to same angle
    fpo._rotation=boxobject._rotation;
    // stop on same color
    fpo.square.gotoAndStop(boxobject.square._currentframe);
}
```



## additives Modell

Bei den drei ersten Beispiele dieses Abschnitts (growing boxes, sandpits und growing circles) funktioniert die Besetzung der Fläche auf ähnlicher Weise: Durch die Addition einzelner Elemente wird der freie Raum gefüllt, wodurch sich die jeweiligen charakteristischen Strukturen ergeben. Die sandpits Struktur berücksichtigt im Gegensatz zu den beiden anderen nicht die Positionen der bereits besetzten Elemente, sondern überlagert diese. Bei den growing boxes und circles hingegen können sich die neu gesetzten Elemente nur so weit ausbreiten, bis sie and die Grenze eines bereist Positionierten stoßen.
Allgemein können wir die Methode des Hinzufügens einzelner Elemente zur Flächenbesetzung als additiv bezeichnen. Wenn die zu verteilenden Elemente Präferenzen bezüglich ihrer Positionierung entwickeln können, hat das additive Verfahren den Vorteil, dass sich jedes Element den relativ zu den Anderen besten, freien Platz suchen kann. Dabei entsteht allerdings eine nach außen nicht abgeschlossene Form mit ungenutzten Zwischenräumen.

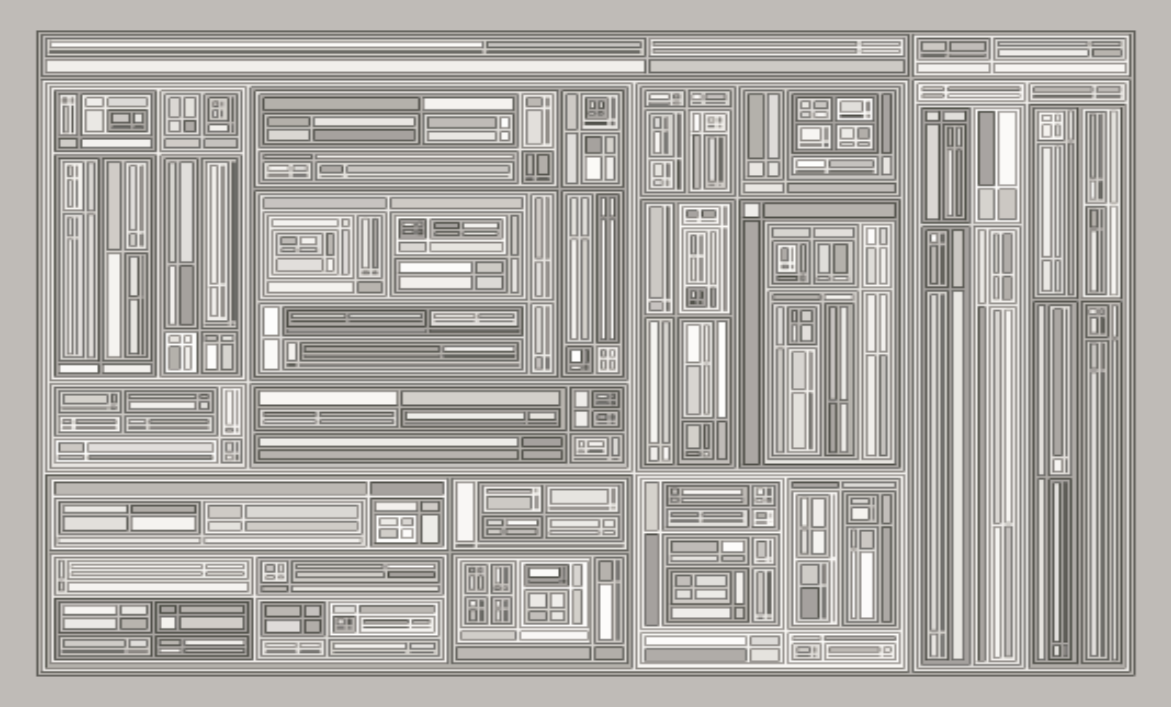
Einige weitere Anregungen zur Verwendung additiver Modelle für computergenereirte Entwurfsprozesse:

- Bei einem kumulativen Prozesses, der schrittweisen Einplanung einzelner Elemente in einen gesamten Entwurf, wird jedes neue Element vorerst auf eventuelle Verletzungen der bestehenden Restriktionen geprüft.
- Das kombinatorische Verfahren funktioniert in ähnlicher Weise. Hier wird eine Vielzahl von Lösungen durch konsequente Verfolgung der möglichen Kombinationen erzeugt. (Die ersten beiden Verfahren eignen sich besonders zur Kombination von Designelementen, bzw. zum Entwerfen mit Gebäude-Komponenten. Vgl. Matthias Castorphs Bauteilorientierte Entwurfsprozesse, Entwurfskonstruktionen)
- ein Ausgangspunkt wird definiert oder zufällig gesetzt und die Raumteile lagern sich entsprechend ihrer internen Abhängigkeiten an.
- ausgehend von einem definierten Beziehungsgefüge positionieren sich die Raumprogrammteile zueinander bis das Gefüge in ein stabiles Gleichgewicht gelangt. Vergleichbar ist dieses System mit dem einer Puppe, deren Glieder mit Gummibändern verbunden sind und somit die Position jedes einzelnen von der Positionierung der anderen und der Spannung der Gummibänder abhängt. (siehe Diplomarbeit Medienwerkstadt)
- ein nicht determiniertes System, indem keine Hierarchie der Raumbeziehungen nötig ist. Es würden alle Räume undifferenziert nach Größe und Funktion zueinander in Beziehung gesetzt. Vergleichbar mit pluripotenten Stammzellen, die sich zu jedem Raumtyp entwickeln können, wodurch sich die Relationsmuster anpassen müssen. Das System bewegt sich in einem Raumgitter, welches die Geschossigkeit und sinnvolle Raumtiefen vorgibt. Dabei kann bestimmt werden, dass bestimmte Bereiche vorrangig besiedelt werden, um z.B. eine Straßenfront zu bilden.

## recursive boxes

code by Jared Tarbell, www.levitated.net

```
onMouseDown = function () {
    boxes = [];
    createEmptyMovieClip("box", 0);
    box._x = 20+9;
    box._y = 20+9;
    box.x = 500;
    box.y = 360;
    box.depth = 0;
    count = 0;
    box._alpha = 10;
    box.onEnterFrame = divide;
};
function divide() {
    if (!this.oneTime) {
        this.lineStyle(1, 0, 50);
        shade=Math.random()*128+127;
        this.beginFill(shade<<16|shade<<8|shade, 100);
        this.lineTo(this.x, 0);
        this.lineTo(this.x, this.y);
        this.lineTo(0, this.y);
        this.lineTo(0, 0);
        this.endFill();
        this.oneTime = true;
    } else if (this._width>10 && this._height>10) {
        var x1 = Math.random()*(this._width-10)+4;
        var y1 = Math.random()*(this._height-10)+4;
        this.createEmptyMovieClip("m0", 0);
        this.m0._x = 2+9;
        this.m0._y = 2+9;
        this.m0.x = x1-2;
        this.m0.y = y1-2;
        this.m0._alpha = 0;
        this.createEmptyMovieClip("m1", 1);
        this.m1._x = x1+2+9;
        this.m1._y = 2+9;
        this.m1.x = this._width-x1-6;
        this.m1.y = y1-2;
        this.m1._alpha = 0;
        this.createEmptyMovieClip("m2", 2);
        this.m2._x = 2+9;
        this.m2._y = y1+2+9;
        this.m2.x = x1-2;
        this.m2.y = this._height-y1-6;
        this.m2._alpha = 0;
        this.createEmptyMovieClip("m3", 3);
        this.m3._x = x1+2+9;
        this.m3._y = y1+2+9;
        this.m3.x = this._width-x1-6;
        this.m3.y = this._height-y1-6;
        this.m3._alpha = 0;
        this.m0.onEnterFrame = divide;
        this.m1.onEnterFrame = divide;
        this.m2.onEnterFrame = divide;
        this.m3.onEnterFrame = divide;
        boxes.push(this.m0);
        boxes.push(this.m1);
        boxes.push(this.m2);
        boxes.push(this.m3);
        if (this == box) {
            this.onEnterFrame = fadein;
        } else {
            delete this.onEnterFrame;
        }
    }
}
function fadein() {
    if (this._alpha<100) {
        this._alpha += 34;
        this._x-=3;
        this._y-=3;
```



```
    } else {
        this._alpha = 100;
        delete this.onEnterFrame;
        boxes[count].onEnterFrame = fadein;
        count++;
    }
}
```

## dividierendes Modell

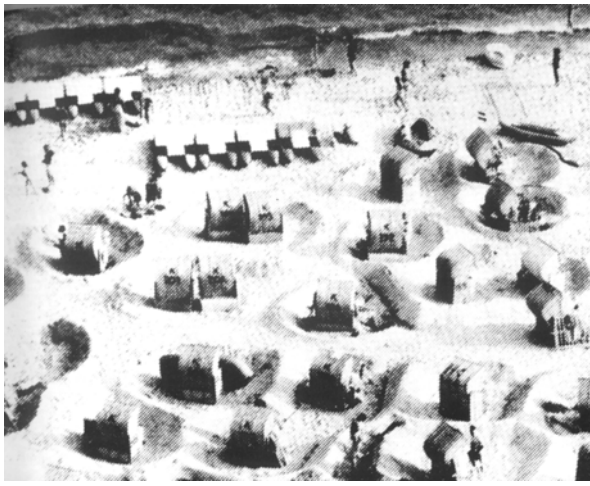
Zerlegung von Gebieten (dividierendes Modell). Ein gegebenes Volumen wird unterteilt, wobei die verschiedenen Räume gebildet werden. Eine anfängliche Hierarchie der Räume wird festgelegt die ausdrückt, welcher Raum das größte „Durchsetzungsvermögen" hat, um die ideale Position einzunehmen. Diese wiederum wird definiert durch die Gebäudeinternen Funktionsrelationen sowie den Gegebenheiten der Umwelt (z.B. Himmelsrichtung, Grundstück, umgebende Bebauung). Über die Ordnung der Raumhierarchie ist festgelegt, welcher Raum zuerst positioniert wird, was die Anordnung der darauf folgenden in Abhängigkeit stellt. Durch Ändern der Raumhierarchien oder der Relationsparameter kommt man zu unterschiedlichen Ergebnissen. Ein weiterer Faktor ist die Festlegung der Möglichkeiten der Raumkubatur. Ob sich ein Raum über mehrere Geschosse erstrecken kann oder auf eines festgelegt ist, ebenso die Definition der einzelnen Raumhöhen und deren Ordnung in den Geschossen (bzw. eine freie Anordnung in der Höhe ohne feststehende Geschossdefinitionen z.B. splitted level house). Diese Überlegungen klammern wir allerdings vorerst für eine primär zweidimensionale Betrachtung aus.

### subtrahierendes Modell

Aus einem vorgegebenen Volumen oder einer definierten Fläche werden Teile herausgeschnitten um Raumstrukturen und/oder entsprechende Freiräume zu definieren. Diese Methode erscheint besonders für die Generierung städtebaulicher Strukturen interessant zu sein. (vgl. Induction Cities von Watanabe).

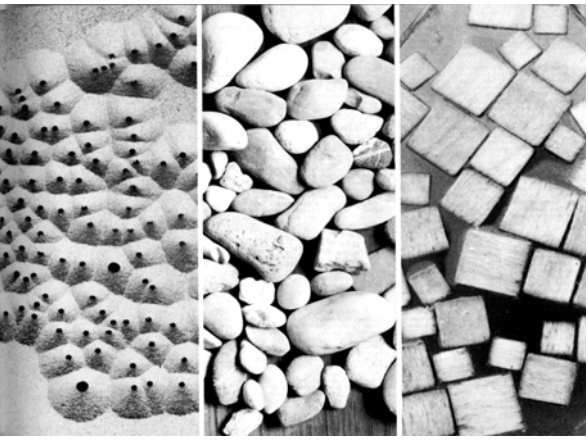
## Strukturanalogien zu 'Flächenbesetzung'

Im Gegensatz zu den Strukturbildenden Wegesystemen, die im Abschnitt ‚node garden' besprochen wurden, stehen die Prinzipien der „Besetzung" einer Fläche durch eine Vielzahl von Einheiten.



Die nebenstehende Abbildung zeigt eine solche Flächenaufteilung am Beispiel einer „Strandbesetzung". Hier berühren sich die einzelnen Einheiten - besetzte Bereiche (Strandburgen) in der Regel nicht und bilden somit eine lockere Packung. Eine dichte Packung entsteht, wenn die Einheiten aneinander grenzen, so dass praktisch keine Restflächen übrig bleiben.

Je nach Eigenschaften der Einheiten, die einer Packung bilden, können diese sich gegenseitig in ihren Formen, aber auch Größen beeinflussen oder, wenn sie nur begrenzt oder gar nicht anpassungsfähig sind, durch bestimmte Anordnungen nur eine Packung mit möglichst wenig Restfläche bilden.



**von inks nach rechts:** Sandschüttung, Kieselsteine, schwimmende Holzplättchen

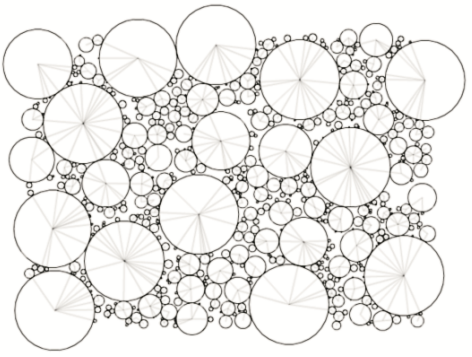
Betrachten wir zwei Modelle von Packungen, die von Einheiten mit unterschiedlich hohem Grad der Anpassungsfähigkeit gebildet sind. Beim ersten Modell wurde die Größe der einzelnen Einheiten vorgegeben, die sich dann in Selbstorganisation zu einer dichten Packung arrangierten. Beim zweiten Modell wurde von der Anordnung der Einheiten auf der Fläche ausgegangen. Die mehr oder weniger dichte Packung wurde durch die Ausdehnung einzelner Einheiten oder durch Hinzukommen neuer Einheiten erreicht.

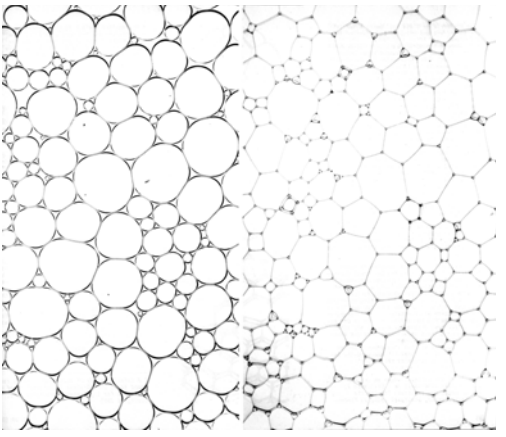
Die einzelnen die Packung bildenden Einheiten simulieren eine Besetzung eines Terrains, wie sie beim Siedlungsvorgang in der Anfangsphase stattfindet. Die Grenzlinien der Einheiten, beziehungsweise die verbleibende Restfläche, geben den Raum für das Wegesystem her. Im Gegensatz zu den Systemen der Verknüpfung (im Abschnitt ‚node garden') ist bei diesen Modellen für das Verkehrssystem kaum Spielraum vorhanden.

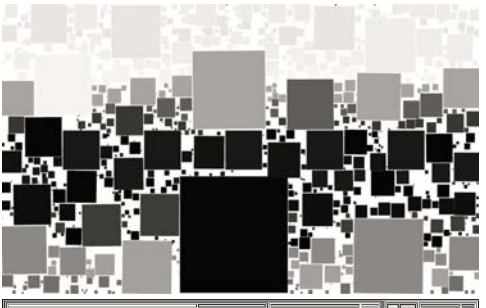
Die Angaben beruhen auf der Publikation „Ungeplante Siedlungen, non-plannes Settlements" des Instituts für Leichte Flächentragwerke (IL 39) der Universität Stuttgart, Herausgeber ist Frei Otto. 1990 als Dissertation von Eda Schaur erschienen.

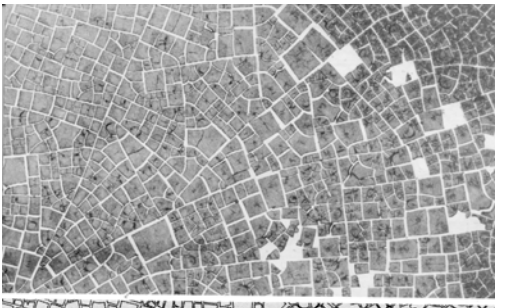
## Blasenflösse und Sandschüttungen - Modelle dichter Packung

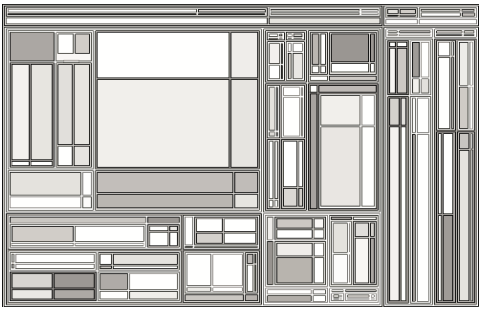
Auf einer Flüssigkeit schwimmende Blasen haben die Tendenz, sich zusammenzuschließen und dichte Packungen zu bilden. Das Bestreben der Flüssigkeit, eine minimale Oberfläche zu bilden (Gleichgewicht aller wirkende Kräfte), entsprechend der Eigenschaft bei den Minimalwegen, spielt auch bei der Formbildung eines Blasenfloßes eine entscheidende Rolle.

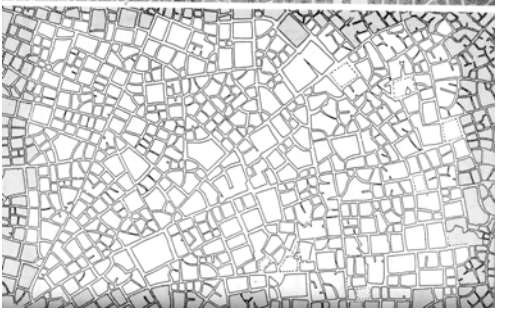




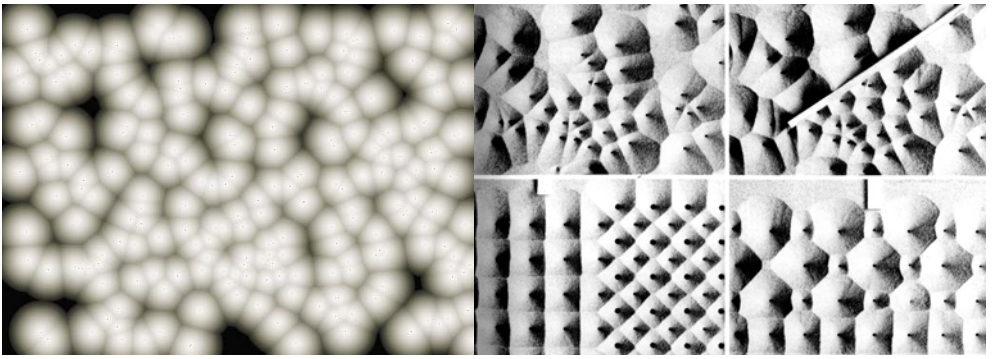








Will man das Problem der Flächenteilung durch Besetzung studieren, so kann man von der Anordnung einzelner „Punkte“ der Besetzung, vergleichbar der Strandbesetzung, ausgehen. Die Strukturbildung vollzieht sich durch die Ausdehnung dieser „Keimpunkte“ und durch Verdichtung infolge neu hinzukommender Besetzungen.



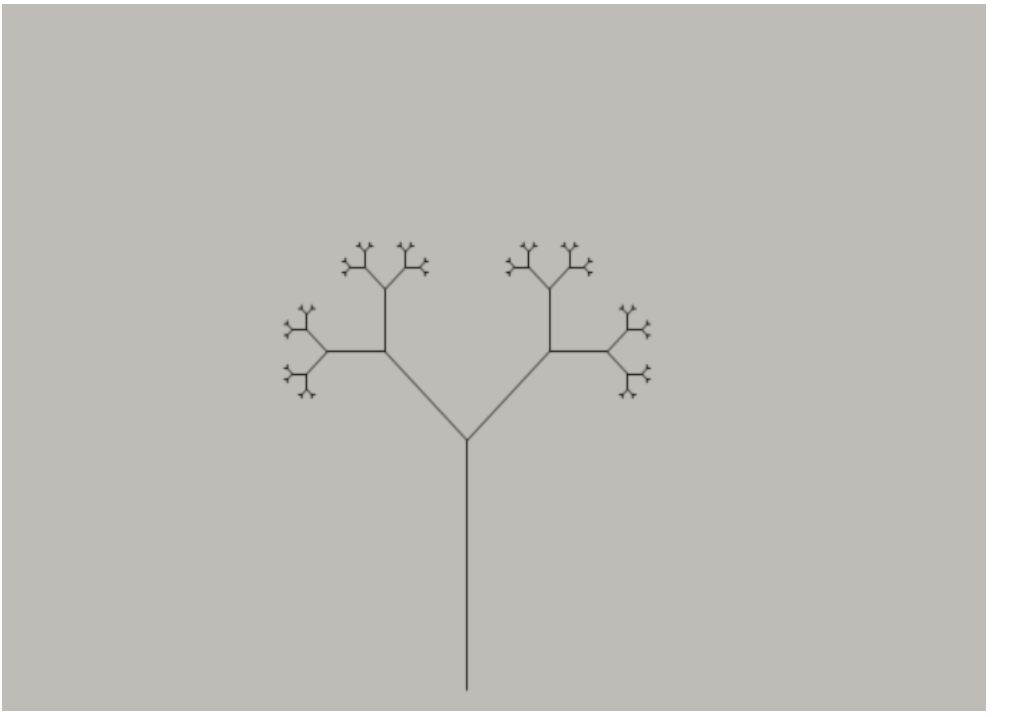
## fractal folia

code by keith peters, www.bit-101.com

Der Code bei diesem Beispiel liegt auf den ersten drei Schlüsselbildern des mocieClips “line”, indem sich die Ausgangslinie befindet.

Im ersten Schlüsselbild wird die Anzahl der fraktalen Schritte definiert:

```
limit = 6;
```

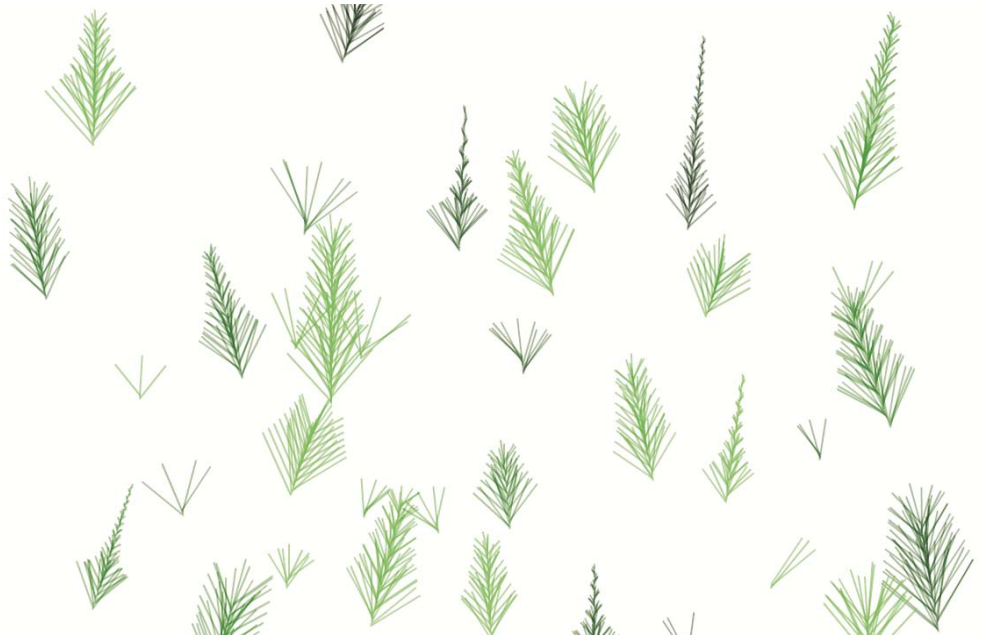


Die Berechnung und das Kopieren der Linien findet im zweiten Frame statt:

```
if(level<limit){
    t1.attachMovie(“line”, “line”+1, 1);
    t1.attachMovie(“line”, “line”+2, 2);
    t1.line1.level=level+1;
    t1.line1._xscale=t1.line1._yscale=50;
    t1.line1._rotation=-45;
    t1.line2.level=level+1;
    t1.line2._xscale=t1.line2._yscale=50;
    t1.line2._rotation=45;
}
```

Durchden Stopbefehl im letzten Frame wird der Prozess angehalten:

```
stop();
```



## tuft of grass

code by keith peters, www.bit-101.com

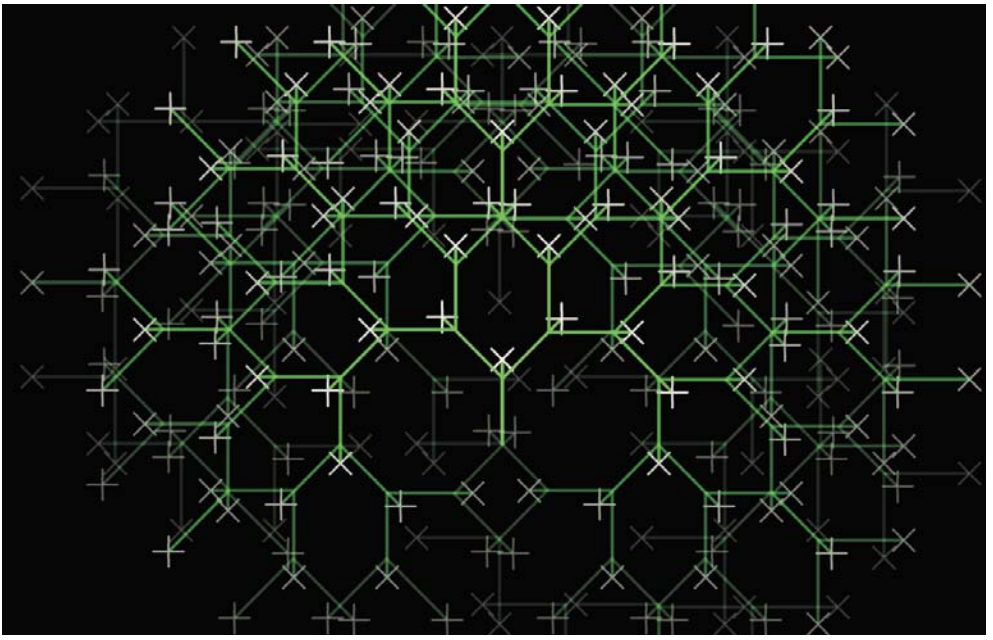
```
onMouseDown=init;
function init() {
    feather = createEmptyMovieClip(“f”+i, 10000+i++);
    feather.swapDepths(Math.random()*10000);
    feather._x = _xmouse;
    feather._y = _ymouse;
    feather._rotation = -90+Math.random()*40-20;
    col = Math.random()*255 << 8;
    radius = Math.random()*20+20;
    twist = Math.random()+.5;
    len = Math.random()*100+50;
    taper = Math.random()*.05+.95;
    x=0;
    onEnterFrame=grow;
}
function grow() {
    angle = Math.sin(fa += twist)*Math.PI/4;
    feather.moveTo(x, y);
    feather.lineStyle(1, col, 50);
    feather.lineTo(x+Math.cos(angle)*radius, y+Math.sin(angle
)*radius);
    radius *= taper;
    if (x++>len) {
        delete onEnterFrame;
    }
};
```

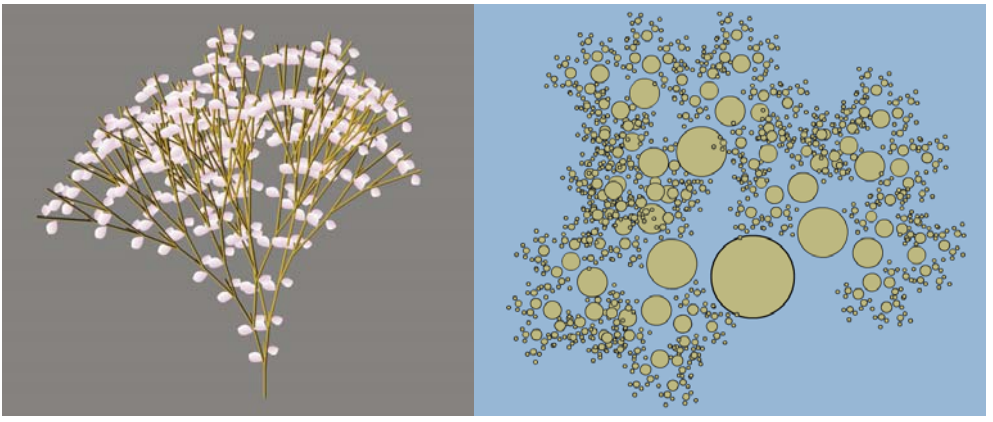
## iteration

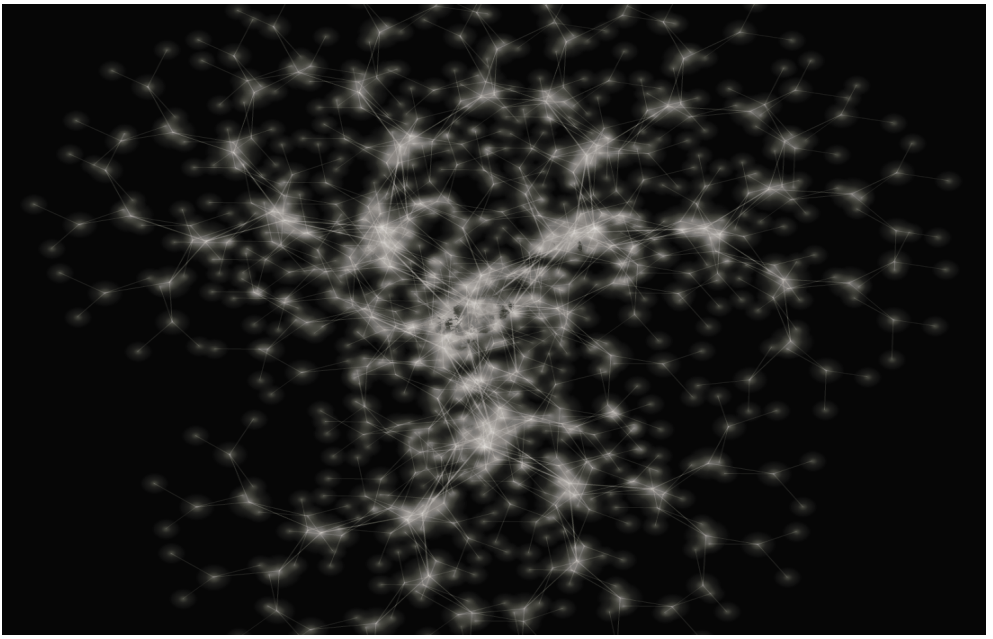
code by

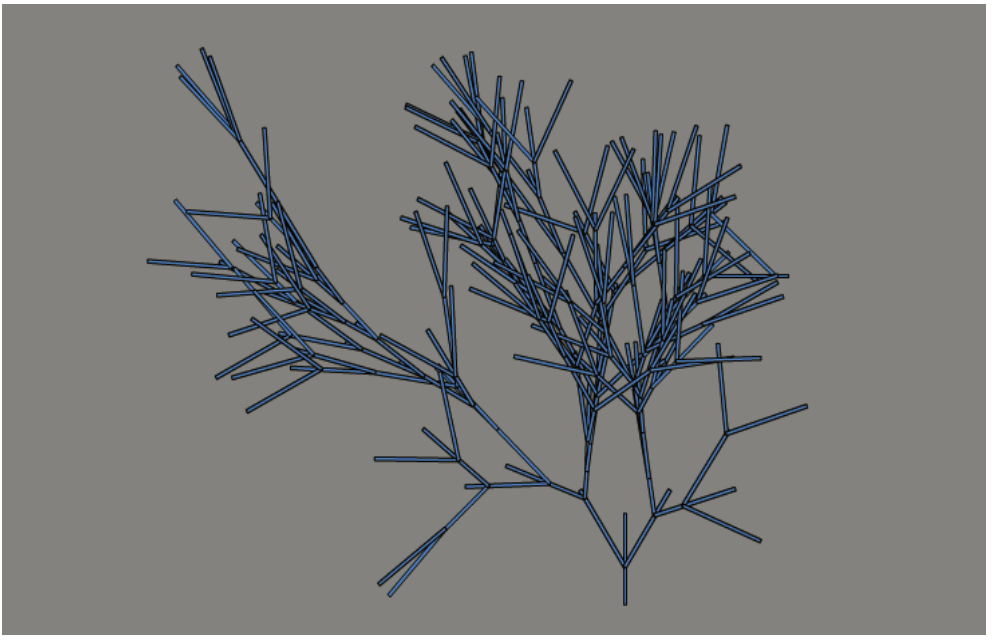
```
levels = 4;
a0 = -1;
a1 = 0
a2 = 1;
l0 = 60;
l1 = 60;
l2 =60;
function fract() {
    var i, dot;
    for (i=0; i<3; i++) {
        dot = this.attachMovie(“dot”, “d”+i, i);
        dot.num = i;
        dot._xscale = dot._yscale=50;
        dot.deep = this.deep+1;
        if (this.deep<levels-1) {
            dot.fract = fract;
            dot.fract();
        }
        dot.adjust = adjust;
        dot.onPress = doDrag;
        dot.onRelease = noDrag;
        dot.onReleaseOutside = noDrag;
    }
}
function adjust() {
    this.clear();
    this.lineStyle(1, 0, 100);
    if (this.deep>0) {
        this._x = Math.cos(_
root[“a”+this.num])*_root[“l”+this.num];
        this._y = Math.sin(_
root[“a”+this.num])*_root[“l”+this.num];
        this._rotation = _
root[“a”+this.num]*180/Math.PI;
        this._parent.moveTo(0, 0);
        this._parent.lineTo(this._x, this._y);
    }
    if (this.deep<levels) {
        for (var i = 0; i<3; i++) {
            this[“d”+i].adjust();
        }
    }
}
function doDrag() {
    this.startDrag();
    if (this.deep>0) {
        this.onMouseMove = findNewAngle;
    }
}
function noDrag() {
    stopDrag();
    delete this.onMouseMove;
}
function findNewAngle() {
    var angle = Math.atan2(this._y, this._x);
    var dist = Math.sqrt(this._x*this._x+this._
y*this._y);
    _root[“a”+this.num] = angle;
    _root[“l”+this.num] = dist;
    base.adjust();
}
attachMovie(“dot”, “base”, 0);
base._x = 270;
```

```
base._y = 200;
base.deep = 0;
base.fract = fract;
base.fract();
base.adjust = adjust;
base.adjust();
base._rotation=-90;
```









## treemaker

code by glen rhodes, www.glenrhodes.com

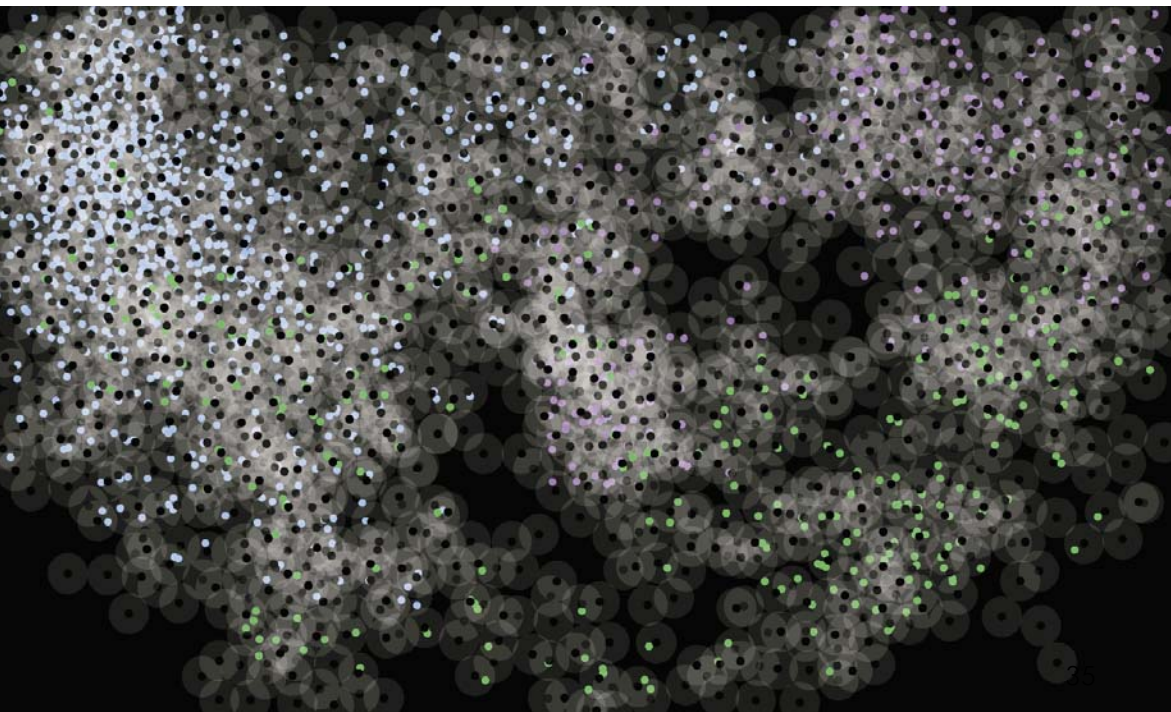
```
onClipEvent(enterFrame)
{
    if (destr < -180) destr += 360;
    if (destr > 180) destr -= 360;

    if (_rotation < -180) _rotation += 360;
    if (_rotation > 180) _rotation -= 360;

    if (_rotation > destr) _rotation--;
    if (_rotation < destr) _rotation++;

    _rotation = Math.floor(_rotation);

    if (_rotation == destr && _root.numsticks < _root .maxsticks
&& dead != true)
    {
        nm = “st” + _root.numsticks;
        duplicateMovieClip (_root.stick, nm, _root.numsticks++);
        ra = _rotation * (Math.PI / 180);

        len = (Math.random() * (_root.sticklen - 10)) + 10;

        dx = Math.cos(ra) * len;
        dy = Math.sin(ra) * len;
        _root[nm]._x = _x - dx;
        _root[nm]._y = _y - dy;
        _root[nm]._rotation = _rotation;

        randa = Math.floor(Math.random() * _root.da);
        _root[nm].destr = _rotation - randa;

        nm = “st” + _root.numsticks;
        duplicateMovieClip (_root.stick, nm, _root.numsticks++);
        _root[nm]._x = _x - dx;
        _root[nm]._y = _y - dy;
        _root[nm]._rotation = _rotation;
        _root[nm].destr = _rotation + randa;

        dead = true;
    }
```

## mycelium model



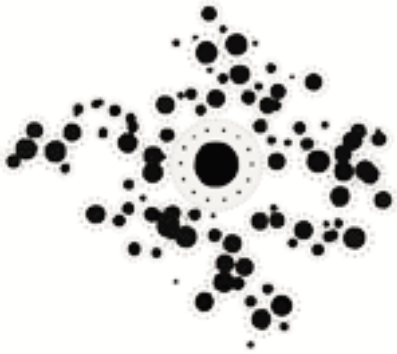
Mycelium is the vegetative phase of mushroom growth.
Among plants, mycelium has the unique property of having no leaves and no flowers, consisting, instead, of millions of self similar strands. This simple mycelium growth simulation uses a 'place and test' model for reproduction. Strand growth is represented here with circular forms.
CLICK inactive nodes to respawn dead areas.
Distinct colonies of fungi are initiated on a barren surface. Through recursive, teleologic reproduction, they expand across the surface, carefully placing each new node in it's own unique location. Colonies are assigned unique colors. Small dots of color appear where newly created nodes overlap previous instances of themselves.
It causes me anxiety to watch frames drop, so I have put a cap on the rate of growth. This cap imposes a maximum growth rate of 22 spores per second. Areas of fungi growth attempting to grow beyond this limit are truncated. For this reason, these fungi colonies often form asymmetrically.
Within the source, a framework exists for detecting interaction between colonies. For the purposes of this demonstration, it has been disabled because of it's expensive computation.

jtarbell
november 3, 2001



# diffusion limited aggregation





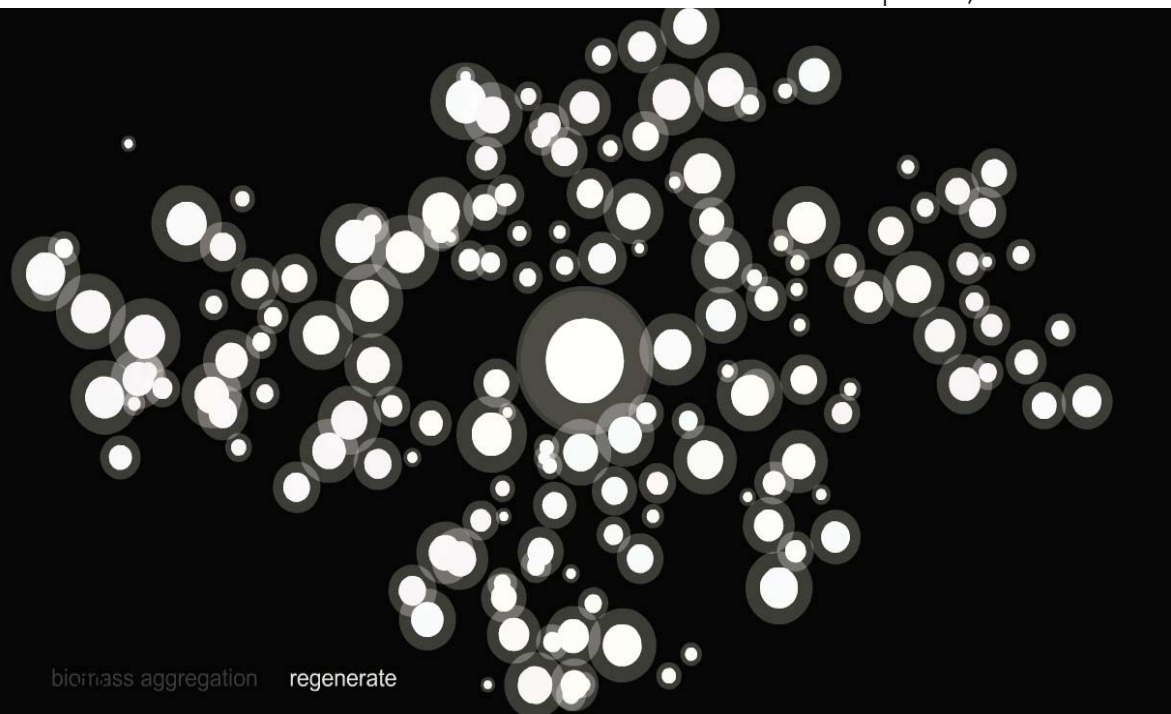
DLA is a simulated growth process.
A particle sticky seed exists in a two dimensional field. Wandering particles are generated along the edges of the field. Particles stick to the seed to form a cluster. Every time a particle bumps into the cluster, it sticks and stays put.

The resulting pattern is a typical fractal object, looking much like something in the deep sea. An interesting phenomena is the consistent creation of tremendous empty spaces, often in the form of long channels or 'fnords'.

Press the REGENERATE button to start with a fresh, new seed object.

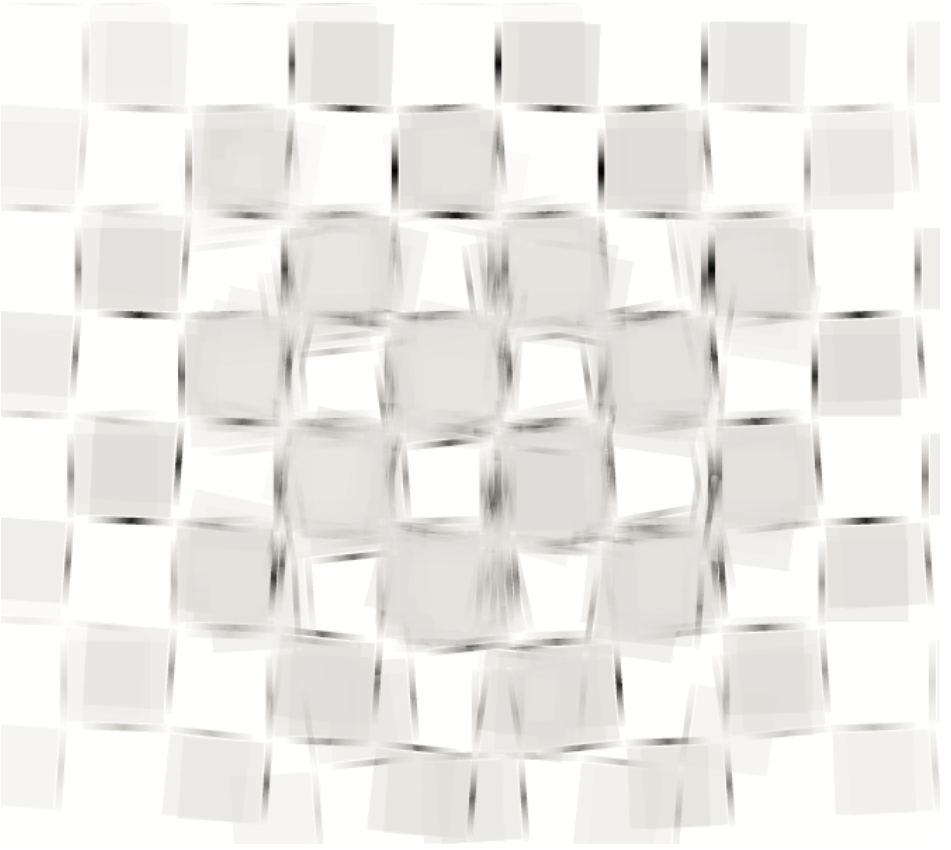
This simulation is called diffusion-limited aggregation. It is an effective, although slow, growth model for natural systems such as the deposition of metals during an electrochemical reaction.

jtarbell
juli 23, 2001



## Strukturanalogiene zu ‘wachstumsstrurukturen’

Schaltpläne
Luftbilder v. Städten
Nachtaufnahme der Erde mit Beleuchtung

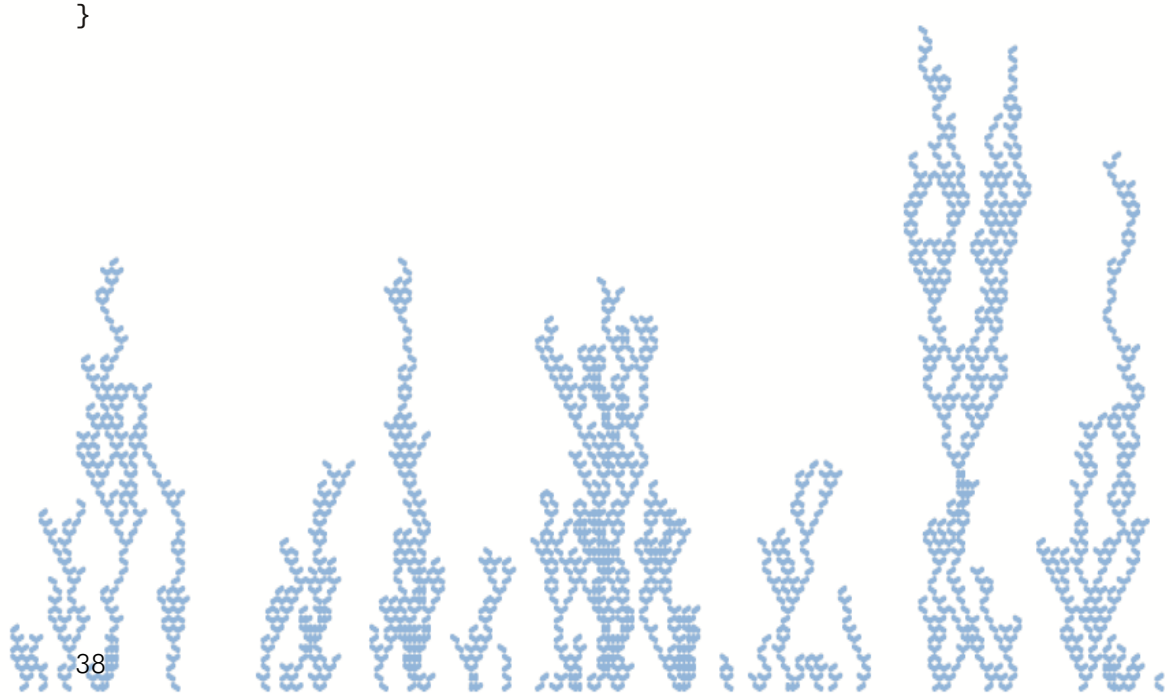


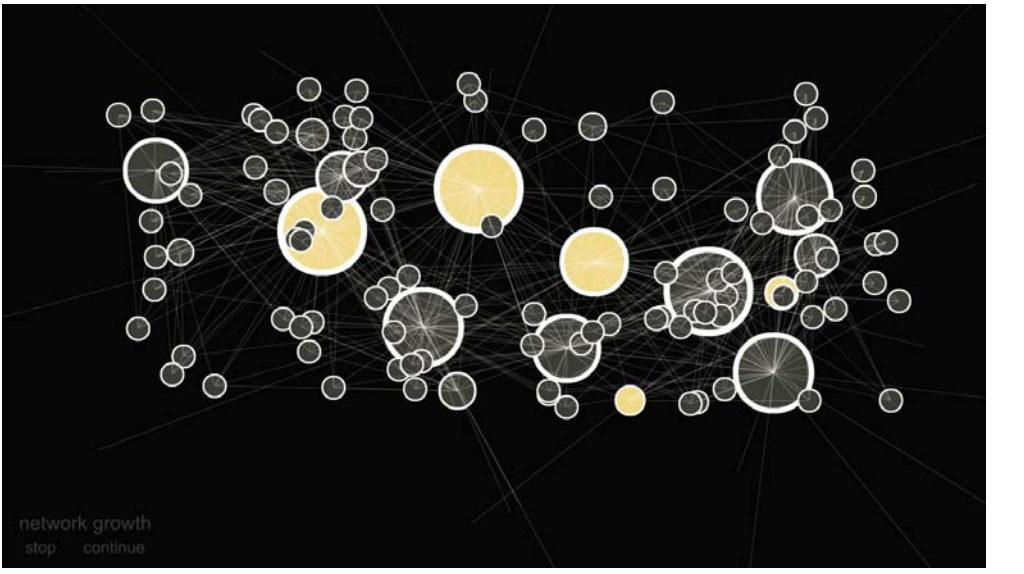
# zelluläre automaten

## automat 01

code by keith peters, www.bit-101.com

```
init();
function init() {
    panel = createEmptyMovieClip(“p”+count, count++);
    clear();
    j = 0;
    col = Math.random()*0xffffff;
    panel.lineStyle(2, col, 100);
    lines = [];
    lines[j] = [];
    for (i=0; i<250; i++) {
        lines[j][i] = Math.random()>.8;
    }
}
onEnterFrame = function () {
    j++;
    lines[j] = [];
    found = false;
    for (i=0; i<250; i++) {
        if (lines[j-1][i]) {
            if (lines[j][i-1]=Math.random()>.38) {
                panel.moveTo(20+i*2, 380-j*3);
                panel.lineTo(20+(i-1)*2, 379-j*3);
                found = true;
            }
            if (lines[j][i+1]=Math.random()>.38) {
                panel.moveTo(20+i*2, 380-j*3);
                panel.lineTo(20+(i+1)*2, 379-j*3);
                found = true;
            }
        }
    }
    if (!found) {
        panel.onEnterFrame = fade;
        init();
    }
};
function fade() {
    this._alpha-=2;
    if (this._alpha<1) {
        removeMovieClip(this);
    }
}
```






## binary network v.1

code by Jared Tarbell, 2002; from www.levitated.net

This randomly connected binary network is the first in a series of experiments with self-organization. At progressive intervals, binary nodes (switches which have only two states, *lit* or *unlit*) are created and connected to each other at random.

**figure a.** typical network of switching nodes.

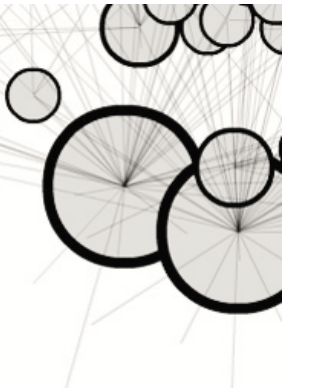


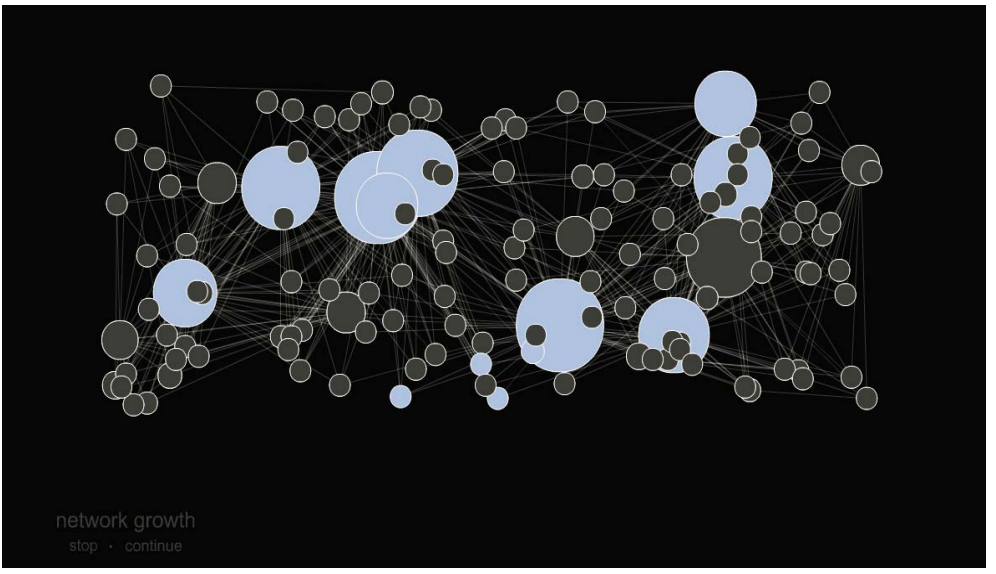
**figure b.** some connections appear to extend beyond the node. this is a result of lines scaling proportionally with their parental node

Nodes are coupled to each other in such a way that the activity of each node is governed by the prior activity of those nodes connected to it. The ‚switching rule' that determines the state of the node is simple: an odd number of adjacently lit nodes switches the node ‚on', while an even number switches the node ‚off'.

As the network slowly grows, arbitrary connection modulus begins to trigger lighting events. Some lighting events are instantaneous, ending only moments after beginning. Other lighting events lock into steady, sometimes alternating states. Most lighting events fall into waving patterns of random activity.

The network simulation can be started and stopped. In this version, interactivity with individual nodes is not possible.

jtarbell, August 5, 2001




## binary network v.2

code by Jared Tarbell, 2002; from www.levitated.net

The second in the series of randomly connected binary networks, this one introduces the ability to switch nodes on and off manually.

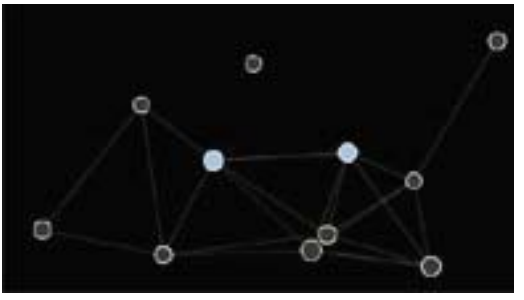
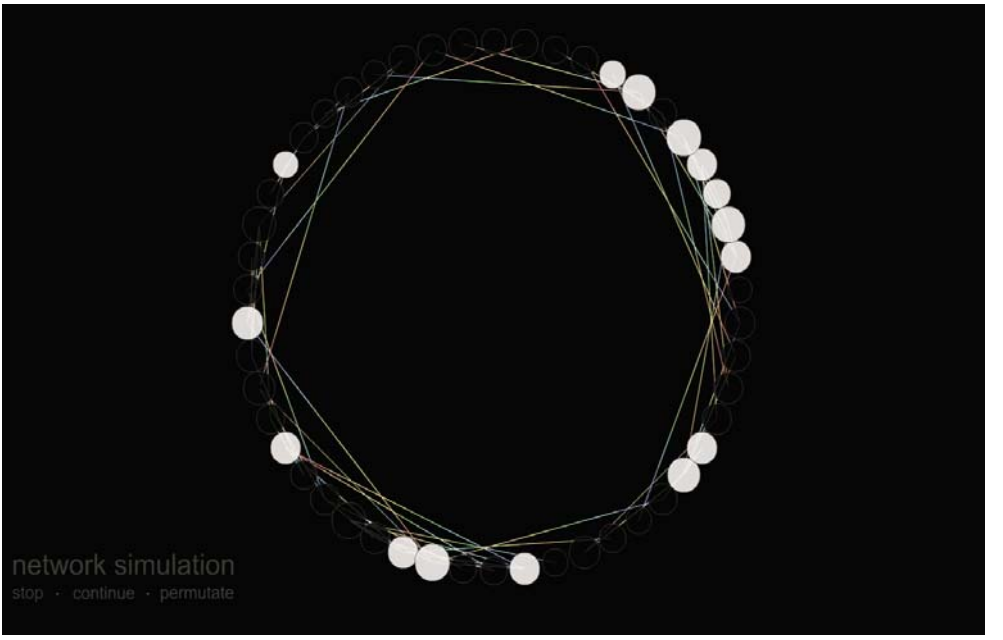


**figure a.** the two lit nodes in this figure have been manually activated by clicking

Manually lighting nodes through interaction shows the consistent nature caused by latency in the implementation. No longer are we dependent on the continual growth of the network to observe switching behavior.

A known deficiency of this particular version is the method in which the nodes are connected. Since nodes are only connected at their time of creation, couplings always have historic connection paths. That is, it is not possible for older nodes to be connected to newer nodes. This deficiency produces a kind of 'one way' connection scheme, inhibiting the chances of a closed loop. Thus, very little emergent behavior ever arises.

jtarbell, August 6, 2001




## binary network v.3

code by Jared Tarbell, 2002; from www.levitated.net

This version of the randomly connected binary network brings us closer to some good examples of emergent behavior and self-organization. The circular orientation of the binary nodes and the method in which the nodes are connected often creates closed loops. Closed loops are node connection paths that eventually loop back on themselves. This looping provides a source of feedback. Feedback is the key to all self-organizing systems.

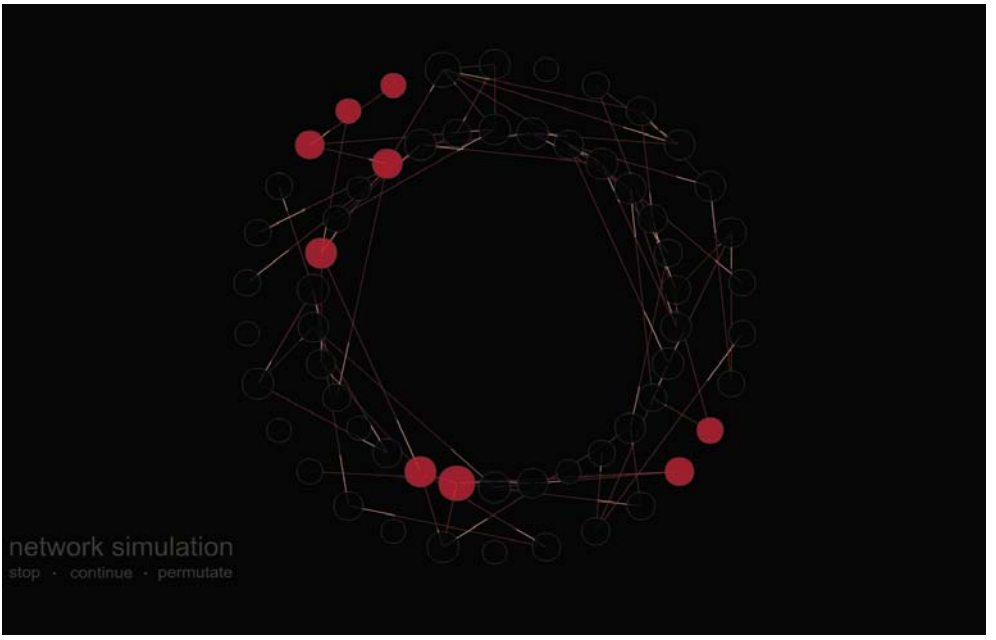
**figure a.** A close up of four 'unlit' binary nodes. The centers of the nodes have been exposed to show that connection polarity exists.

Connections have been colorized to show their one-way nature. In this particular instance, connection logic reads from red to blue. That is, a node will check all locally red connections to determine state. The blue connections of a node are indications of dependencies from other nodes.

Notice the effects certain nodes will have on the rest of the system. Some very peculiar arrangements can be discovered with enough experimentation. Certain arrangements will produce harmonic lighting patterns, complicated waves of activity, or induce external looping with seemingly random influences.

Sounds were assigned to individual nodes to help the user discern repeating patterns. However, the particular choice of sounds is probably not well suited for this task.

jtarbell, August 7, 2001




## binary network v.5

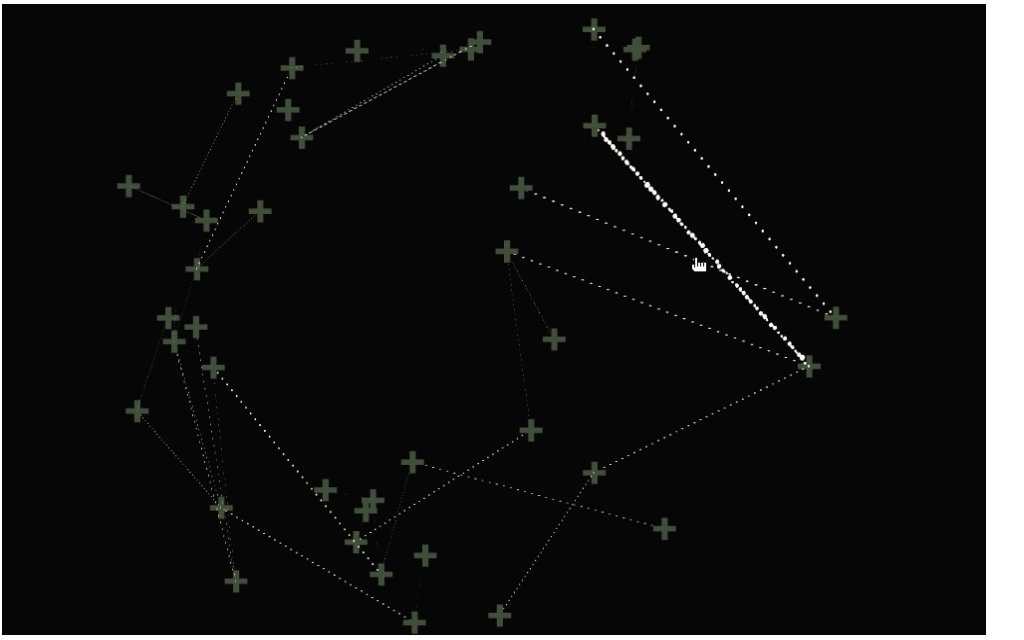
code by Jared Tarbell, 2002; from www.levitated.net

The concept of self-organization originated in the early years of cybernetics. In an attempt to understand the complexity of our brains, experiments by Warren McCulloch and Walter Pitts introduced idealized neurons represented by binary switching elements. Massive numbers of these elements were tied together in random fashion to produce highly complex networks. During simulation, to their great amazement, they discovered that after a short time of random flickering, some ordered patterns would emerge in most networks. This spontaneous emergence of order became known as 'self-organization'.

In some form or another the principles of self-organization can be used to describe almost every living system, from the living cell to the 3 pound fiber bundle we call our brain.

This humble Flash assembly of a mere 42 switching elements is an infinitesimal simulation of actual self-organizing systems, yet patterns and strange behavior can still be seen in it's execution.

jtarbell, August 9, 2001



## barslund repulsion network

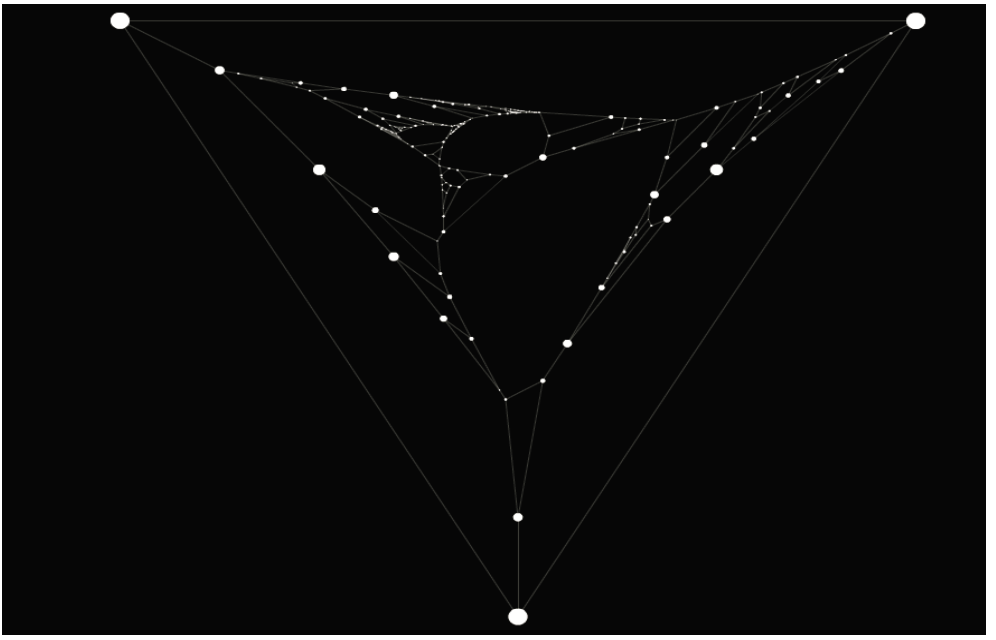
code by Jared Tarbell, 2002; from www.levitated.net

A tangentially biased distributed network of special repulsive nodes as defined by the equations of Morten Barslund.

Nodes exhibit Brownian motion when not being repelled. As a result, the network gradually decays from it's initial spherical form.

Connection lines between nodes activate when moused over. As a challenge, try to enter the center of the network without activating any of the connections. Good luck.

¡tarbell, September 6, 2001



## network evolution

code by Jared Tarbell, 2002; from www.levitated.net

Here we have evolved an intricately connected network of nodes using a single substitution rule. The network will gradually zoom in as it becomes more and more complex.

Use the mouse to explore. Click individual nodes to accelerate evolution.

Stephen Wolfram suggests many interesting methods of evolving a network using substitution systems in chapter 9 of his book *A New Kind of Science*. The simplest of these, a three node sub-network substitution (page 509), has been implemented in Flash MX.

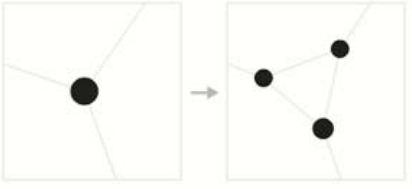


**figure a.** the node substitution rule

**figure b.** an evolved network using 210 substitutions

The network is evolved by replacing random nodes with a subnetwork of three nodes. One prerequisite of this substitution rule is that the node being replaced must have precisely three connections.

Starting with an initial network of precisely four nodes, each connected to all three others, repeated substitutions will create a network exhibiting intricate localized structures. The nodes are intentionally left still after substitutions to add a sense of legacy. The size of the nodes decrease proportional to the length of their connections.

The network will occasionally drift into vast areas of nothing. If this should happen, reload the page and the network will be reset.

jtarbell, march, 2003

## künstliche neuronale Netze

*„Maschinen, die aus ihren eigenen Erfahrungen lernen*
*Ein wesentlicher Bestandteil des in unserer Gegenwartskultur verankerten Begriffs der Maschine ist die Vorstellung, dass das Verhalten einer Maschine insofern trivial ist, als sie nur das ausführen kann, was ihr von ihrem menschlichen Erbauer vorher einprogrammiert wurde. Diese Vorstellung ist veraltet, weil es in den Forschungslabors, in der Industrie, und sogar schon in der Unterhaltungselektronik eine Reihe von Programmen gibt, die es Rechnern ermöglichen, aus ihrer eigenen Erfahrung zu lernen. Insbesondere können solche lernenden Maschinen eine durch ihre vorhergehenden Erfahrungen geformte Individualität hervorbringen, die selbst von ihrem menschlichen Erbauer nicht vollkommen vorhersehbar ist. Das einzige, was der menschliche Erbauer einer solchen Maschine kennt, ist deren Lernalgorithmus, also das von ihm einprogrammierte Verfahren, mit dem die Maschine neue Verhaltensmuster aufgrund von vorhergehenden Erfahrungen entwickeln kann. Eine Vielfalt solcher Lernalgorithmen ist inzwischen bekannt.* Viele der in den Entwicklungslabors bisher erfolgreichsten Lernalgorithmen sind den Lernmechanismen von lebendigen Organismen abgeschaut worden. Als Beispiele erwähne ich das *Reinforcement Lernen*, sowie das Lernen mittels *künstlicher Neuronaler Netzwerke*."

Aus: Rainer E. Burkhard, Wolfgang Maas, Peter Waibl (Hg); Zur Kunst des formalen Denkens; Wolfgang Maass, Das menschliche Gehirn - nur ein Rechner?, S. 217, Passagen Verlag, Wien 2000

Künstliche neuronale Netze folgen in ihrer Architektur eher den Prinzipien biologischer Systeme, können aber in jedem üblichen digitalen Rechner simuliert, oder direkt in neuartiger elektronischer Hardware implementiert werden. Eines der entscheidenden Eigenschaften eines solchen Netzwerks ist, dass es mittels einem Lernalgorithmus aus Erfahrung lernen kann.

Schematische Darstellung der Funktionsweise eines künstlichen neuronalen Netzwerks:

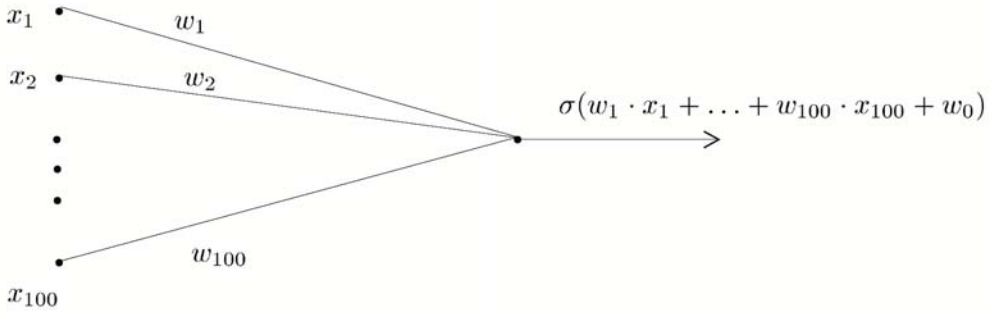



Abbildung 8.1. Schematische Arbeitsweise eines künstlichen Neurons. Der Input besteht in diesem Fall aus 100 Input Zahlen $x_1, \ldots, x_{100}$ (wobei die Zahl 100 willkürlich gewählt wurde; ein biologisches Neuron erhält bis zu 10 000 Input Zahlen). Der Output besteht aus einer einzigen Zahl zwischen 0 und 1, die in der Zeichnung mit $\sigma(w_1 \cdot x_1 + \ldots + w_{100} \cdot x_{100} + w_0)$ bezeichnet ist. Sie entsteht also dadurch, daß man zunächst die gewichtete Summe $w_1 \cdot x_1 + \ldots + w_{100} \cdot x_{100} + w_0$ berechnet, und auf diese Zahl dann die Quetschungsfunktion $\sigma$ anwendet.

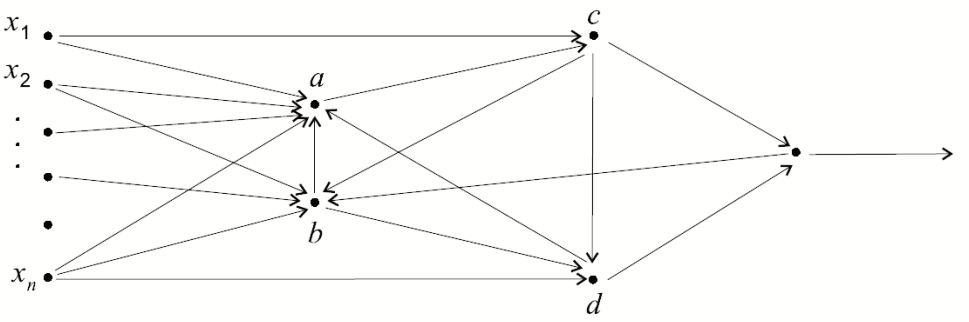



Abbildung 8.4. Beispiel für ein Neuronales Netz mit einer komplexeren Verbindungsstruktur. Der Netzwerk Input ist wieder links abgebildet, und der Netzwerk Output ist wieder rechts. Intern ist der Informationsfluß aber komplizierter. Das Neuron $a$ erhält als Inputs nicht nur Netzwerk Inputs, sondern auch die Outputs der Neuronen $b$ und $d$. Die Neuronen $b$ und $d$ bekommen als Input unter anderem den Output von Neuron $c$, das wiederum den Output von Neuron $a$ als Input bekommt. Wegen der vielfältigen "Rückkopplung" im Netz ist es nicht mehr so einfach zu beschreiben, wie die einzelnen Rechenschritte im Netz koordiniert werden. Komplizierte Netzwerk-Strukturen mit "Rückkopplungen" sind typisch für biologische Netzwerke von Neuronen, werden aber für eine große Anzahl von Anwendungen auch künstlich im Rechner simuliert.

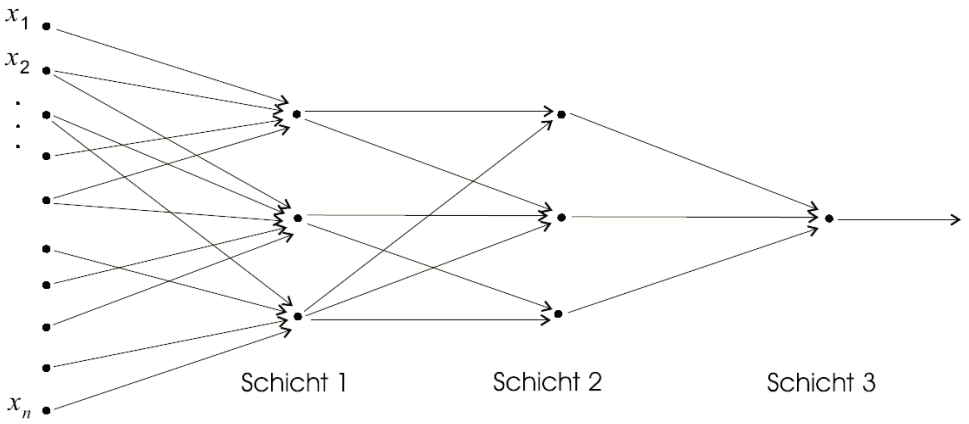



Abbildung 8.3. Verbindungsstruktur eines typischen in einer Richtung ausgerichteten ("feedforward") Neuronalen Netzes. Die Richtung der Berechnung in diesem Neuronalen Netz geht von links nach rechts. Jeder Punkt auf den Schichten 1,2,3, zusammen mit den in diesen Punkt gerichteten Pfeilen, symbolisiert ein künstliches Neuron wie in Abb. 8.1. Die Inputs der Neuronen auf Schicht 1 bestehen aus Netzwerk Inputs $x_1, \ldots, x_n$. Im 1. Zeitschritt berechnen alle Neuronen auf Schicht 1 gleichzeitig ("parallel") ihre jeweilige Output Zahl. Die drei von den Neuronen auf Schicht 1 berechneten Output Zahlen werden weiter verarbeitet von Neuronen auf Schicht 2, d.h. sie liefern die Inputs für Neuronen auf Schicht 2. Die Neuronen auf Schicht 2 errechnen ihre Output Zahlen während dem 2. Zeitschritt, und aus diesen berechnet dann das Neuron auf Schicht 3 seinen Output, der gleichzeitig den Output des gesamten Neuronen Netzes (bestehend aus 7 Neuronen) bildet.

Die Abbildungen stammen ebenfalls aus dem Buch: Rainer E. Burkhard, Wolfgang Maas, Peter Waibl (Hg); Zur Kunst des formalen Denkens; Wolfgang Maass, Das menschliche Gehirn - nur ein Rechner?, S. 217, Passagen Verlag, Wien 2000

# partikel, schwärme und agenten

Zu beginn wollen wir zu klären versuchen, worin der Unterschied zwischen Partikelsystemen, Schwärmen und Agentensystemen liegt.

Unter Partikelsystemen versteht man allgemein eine Menge autonomer Einheiten, die alle über gleiche Eigenschaften und somit über ein gemeinsames Verhalten verfügen. Die Einheiten des Systems können mit Objekten in ihrer Umgebung in Wechselwirkung treten. Man kann die Partikel mit Wassermolekülen vergleichen, die alle über die selben Eigenschaften verfügen und beispielsweise mit einem Stein in einem Wasserstrom insofern in Wechselwirkung treten, als sie von diesem abgelenkt werden oder der Stein vom Strom bewegt wird.

Ein Partikelsystem wird zum Schwarm, wenn die Einheiten des Systems miteinander Informationen austauschen können und sich so gegenseitig beeinflussen, um von außen betrachtet zu einer übergeordneten Einheit zu werden. Vergleiche finden sich in der Tierwelt etwa bei Insekten oder Fischen. Die individuellen Teile eines solchen Schwarmes können zwar unabhängig voneinander handeln, sich aber auch auf Grundlage einfacher, festgesetzter Regeln so verhalten, dass ein Schwarm entsteht, der als Ganzes ein anderes Verhalten aufweist, als eines seiner Elemente.

Das Agentensystem unterscheidet sich wiederum dadurch, dass die ihm zugehörigen Elemente mit ihrer Umwelt, ähnlich den Schwarmelementen, kommunizieren können. Durch den Informationsaustausch kann ein Agent allerdings sein Verhalten und seine Eigenschaften ändern, sowie eine solche Änderung in seiner Umwelt bewirken. So ist ein Agent in der Lage sein Wissen über seine Umgebung zu vergrößern, was als eine Form des Lernens bezeichnet werden kann. Aus diesem Grund wird einem solchen System eine gewisse Art von Intelligenz zugesprochen.

Allen Systemen gemeinsam ist die Fähigkeit zur Selbstorganisation, was sie für unsere Betrachtung interessant machen.

Im Ordner ‚06 partikel, schwärme und agenten' findet sich eine Datei namens ‚Particle.as' (code by keith peters, www.bit-101.com), welche eine Partikelklasse darstellt. Wird diese in Flash (ab v.7) eingelesen kann man auf deren Eigenschaften und Verhalten zugreifen.

## partikel

code by keith peters, www.bit-101.com

```
for(i=0; i<20; i++){
    dot = attachMovie("dot", "d" + i, i);
    dot._x = Math.random()*500+20;
    dot._y = Math.random()*50+200;
    dot.vx = Math.random()-.5;
    dot.vy = Math.random()-.5;
    dot.oldx = dot._x;
    dot.oldy = dot._y;
    dot.grav = Math.random()*.1+.3;
    dot.onEnterFrame = move;
}
createEmptyMovieClip("pad", 100);
pad.lineStyle(1,0,10);
onMouseDown = function(){
    pad.clear();
    pad.lineStyle(1,Math.random()*0x666666, 10);
}
function move(){
    var dx = _xmouse - this._x;
    var dy = _ymouse - this._y;
    var distSQ = dx*dx+dy*dy;
    var dist = Math.sqrt(distSQ);
    var force = 10000/distSQ;
    if(force>20) force = 20;
    var ax = force*dx/dist;
    var ay = force*dy/dist;
    this.vx += ax;
    this.vy += ay;
    this.vy+=this.grav;
    this.vx*=.98;
    this.vy*=.98;
    this._x += this.vx;
    this._y += this.vy;
    if(this._x >520){
        this._x = 520;
        this.vx*=-.9;
    }else if(this._x<20){
        this._x = 20;
        this.vx*=-.9;
    }
    if(this._y >380){
        this._y = 380;
        this.vy *=-.9;
    }else if(this._y<20){
        this._y =20;
        this.vy*=-.9;
    }
    pad.moveTo(this.oldx, this.oldy);
    pad.lineTo(this._x, this._y);
    this.oldx = this._x;
    this.oldy = this._y;
}
```

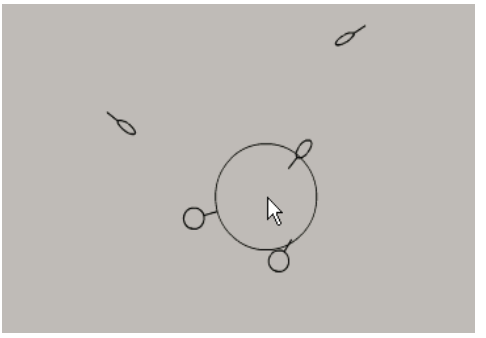
## schwärme

code by keith peters, www.bit-101.com

## agenten

code by keith peters, www.bit-101.com

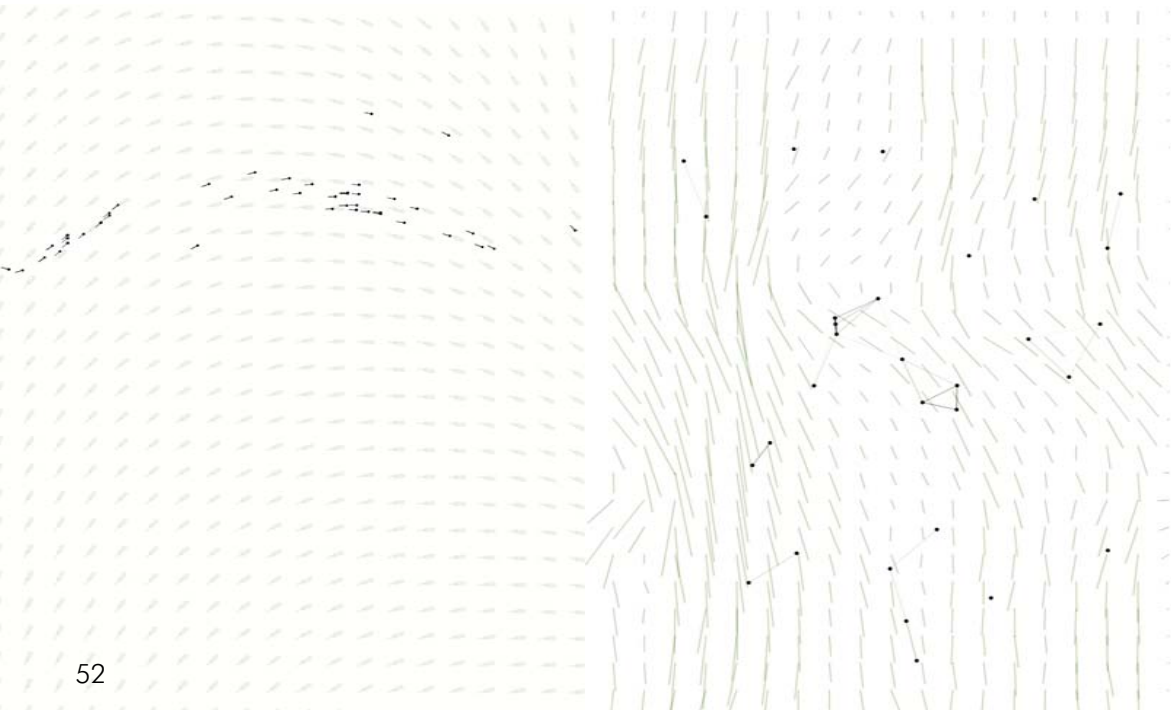
```
onClipEvent (load) {
    //position randomly
    _x = Math.random()*500+20;
    _y = Math.random()*360+20;
    //how much he can eat
    capacity = 10;
    //how fast he is (higher number = slower movement)
    speed = 10;
}
onClipEvent (enterFrame) {
    //if you are full
    if (full) {
        //head to a resting spot (determined below)
        xdist = xrest-_x;
        ydist = yrest-_y;
        //work off that food
        ate -= .2;
        //hungry yet???
        if (ate<1) {
            full = false;
        }
    } else {
        //if not full, head to food
        xdist = _root.food._x-_x;
        ydist = _root.food._y-_y;
    }
    //orient yourself right direction
    angle = Math.atan2(ydist, xdist);
    _rotation = angle*180/Math.PI+90;
    //if you are at the food, and not full, EAT!
    if (_root.food.hitTest(_x, _y, 1) && !full) {
        //food gets smaller
        _root.food._xscale -= .5;
        _root.food._yscale -= .5;
        //you get bigger
        ate++;
        //if you ate as much as you can...
        if (ate>capacity) {
            full = true;
            //choose a random resting spot
            xrest = Math.random()*200-100+_x;
            yrest = Math.random()*200-100+_y;
        }
    } else {
        //if you're not at the food OR you are full
        //just keep going wherever you are going.
        _x += xdist/10;
        _y += ydist/10;
    }
    //show just how full you are
    _xscale = 40+ate*10;
}
```



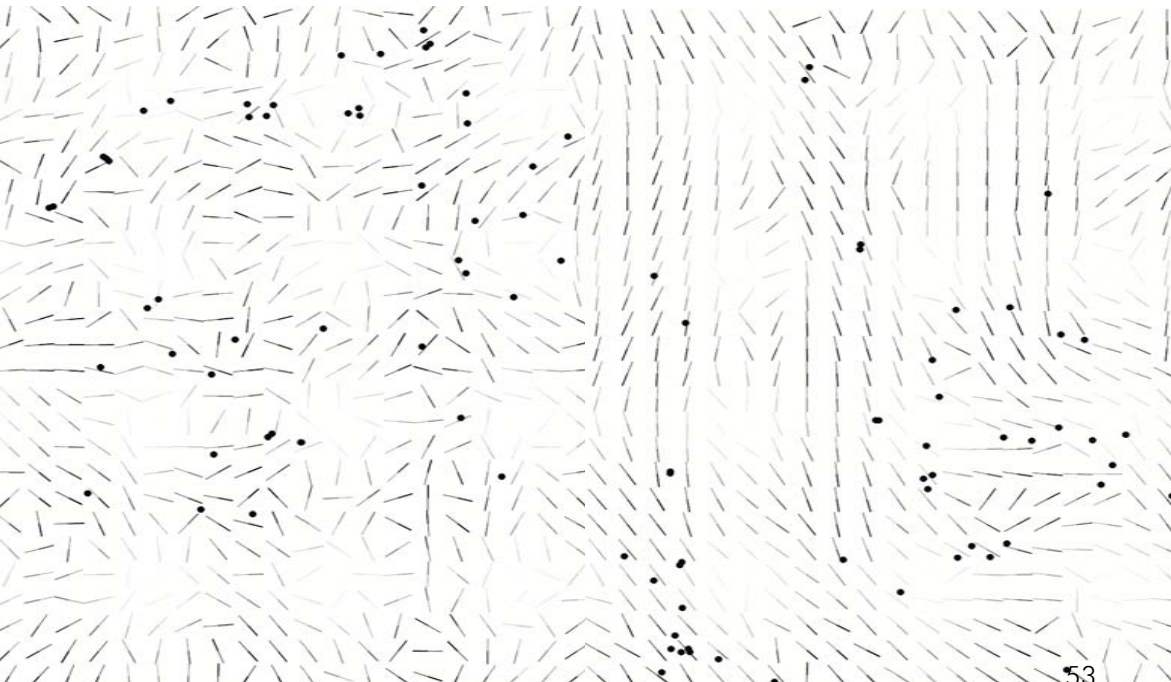
# vector field

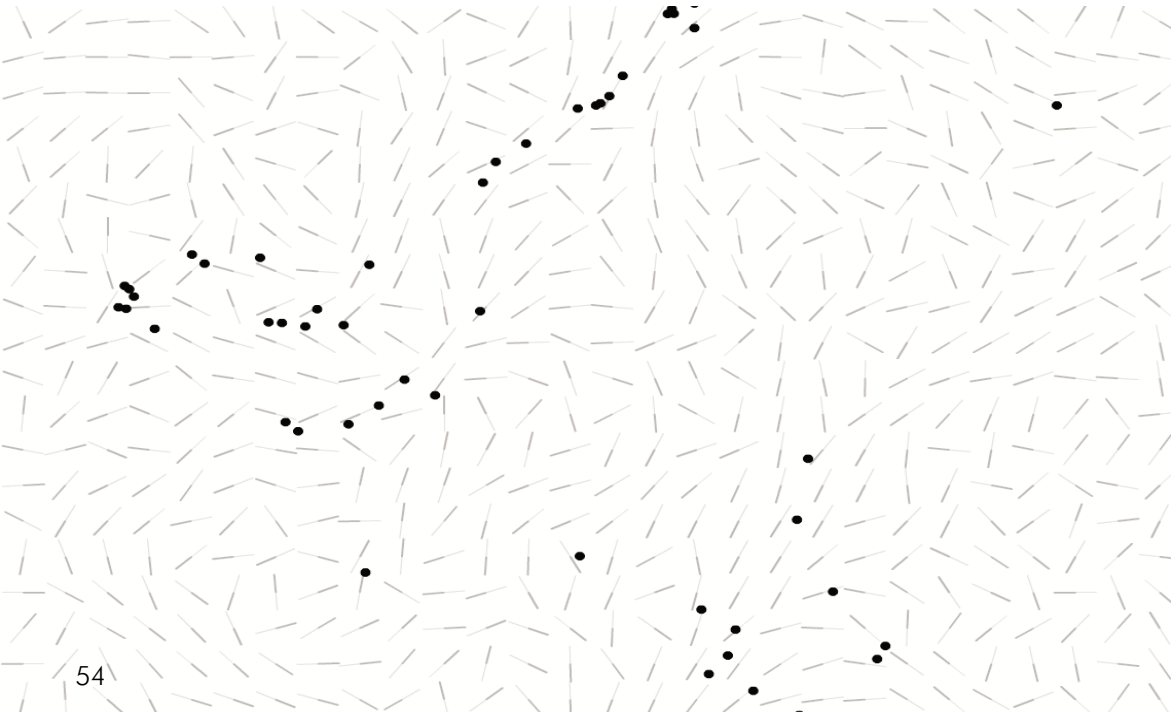
code by keith peters, www.bit-101.com

```
for (i = 0; i < 100; i++) {
    dot = attachMovie("dot", "dot" + i, i++);
    dot._x = Math.random() * 540;
    dot._y = Math.random() * 400;
    dot.vx = 0;
    dot.vy = 0;
    dot.onEnterFrame = move;
}
createEmptyMovieClip("field", 10);
field._visible = false;
Key.addListener(this);
onKeyDown = function () {
    field._visible = !field._visible;
};
init();
onMouseDown = init;
function init() {
    mult = Math.random() * .0001;
    xoffset = Math.random() * 540;
    yoffset = Math.random() * 400;
    field.clear();
    for (y = 0; y < 400; y += 20) {
        for (x = 0; x < 540; x += 20) {
            angle = ((x + xoffset) * (y + yoffset)) * mult;
            field.moveTo(x, y);
            field.lineStyle(1, 0, 10);
            field.lineTo(x + Math.cos(angle) * 5, y +
        Math.sin(angle) * 5);
            field.lineStyle(3, 0, 10);
            field.lineTo(x + Math.cos(angle) * 10, y +
    Math.sin(angle) * 10);
        }
    }
    for (i = 0; i < 100; i++) {
        _root["dot" + i].vx += Math.random() * 20 - 10;
        _root["dot" + i].vy += Math.random() * 20 - 10;
    }
}
```

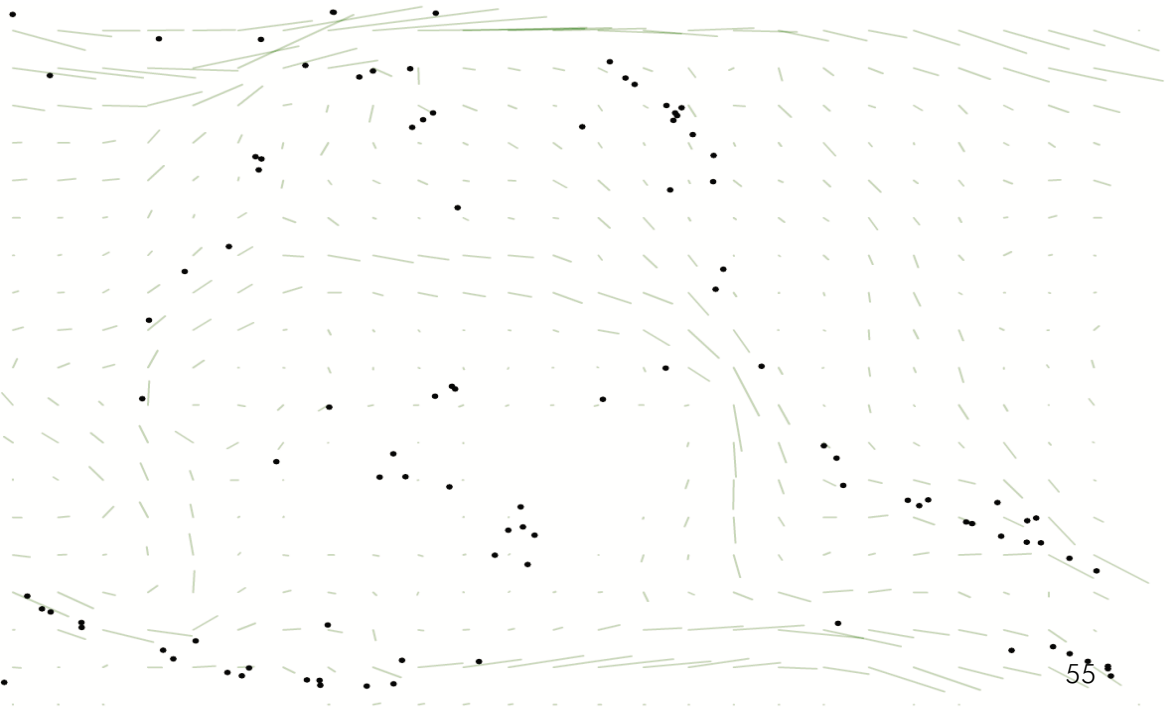


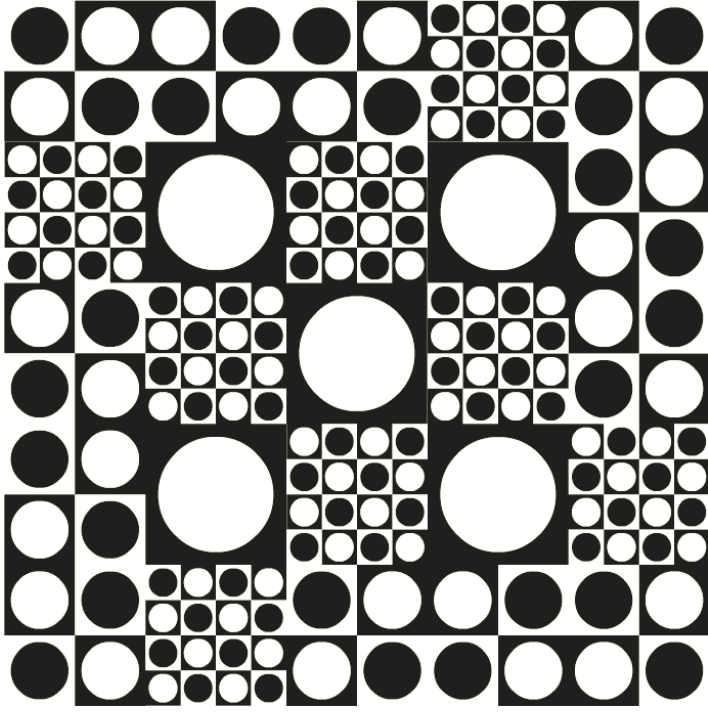
```
function move() {
    var dx = _xmouse - this._x;
    var dy = _ymouse - this._y;
    var dist = Math.sqrt(dx * dx + dy * dy);
    if (dist < 100) {
        var mouseAngle = Math.atan2(dy, dx);
        var tx = _xmouse - Math.cos(mouseAngle) * 100;
        var ty = _ymouse - Math.sin(mouseAngle) * 100;
        this.vx += (tx - this._x) * .2;
        this.vy += (ty - this._y) * .2;
    }
    var angle = ((this._x + xoffset) * (this._y + yoffset))
* mult;
    this.vx += Math.cos(angle) * 2;
    this.vy += Math.sin(angle) * 2;
    this.vx *= .8;
    this.vy *= .8;
    this._x += this.vx;
    this._y += this.vy;
    if (this._x > 540) {
        this._x = 0;
    }
    if (this._x < 0) {
        this._x = 540;
    }
    if (this._y > 400) {
        this._y = 0;
    }
    if (this._y < 0) {
        this._y = 400;
    }
    this._rotation = Math.atan2(this.vy, this.vx) * 180 /
Math.PI;
}
```





Strukturanalogien zu ‚vector field'





## panton geometri 1

code by Jared Tarbell, 2002; from www.levitated.net

An arbitrary construction of Verner Panton's original textile design, Geometri 1 using computational methods.

Click anywhere above to generate a reorganization the design.

The unusual nature of the pattern can first be described as being composed of only positive and negative squares. The squares also contain a circle within them. The charge of the square is determined by the color of the circle such that a white circle makes the square positive and a black circle makes the square negative.

Squares of both charges are arranged into larger square groups using an alternating grid system. The dimensions of each group are always a multiple of 2.

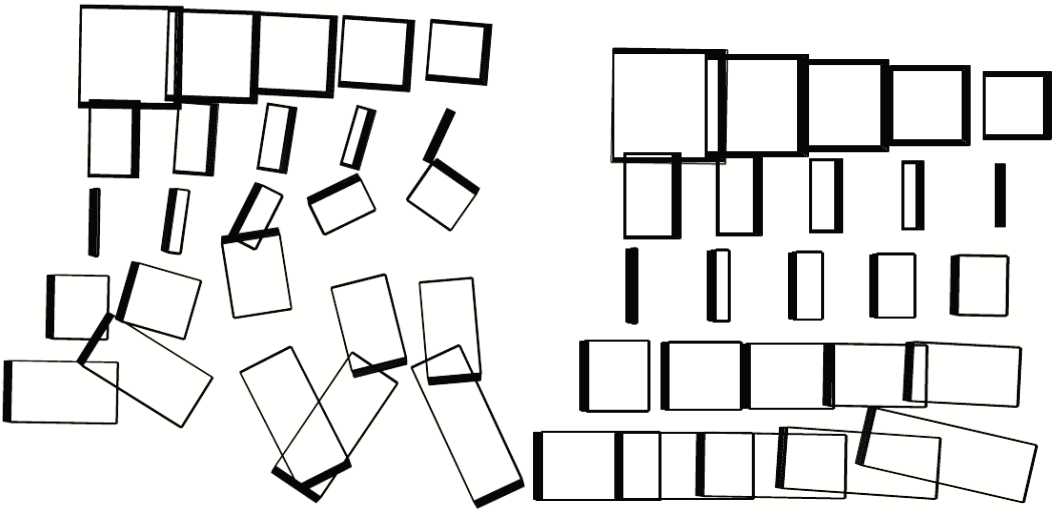
Geometri' I Verner Panton, 1960

The pattern as a whole maintains radial symmetry. This property makes it especially well suited for variation. Starting with a single center bit of positive or negative charge, we can complete the rest of the pattern by generating mirrored square groupings of random charge and depth complexity.

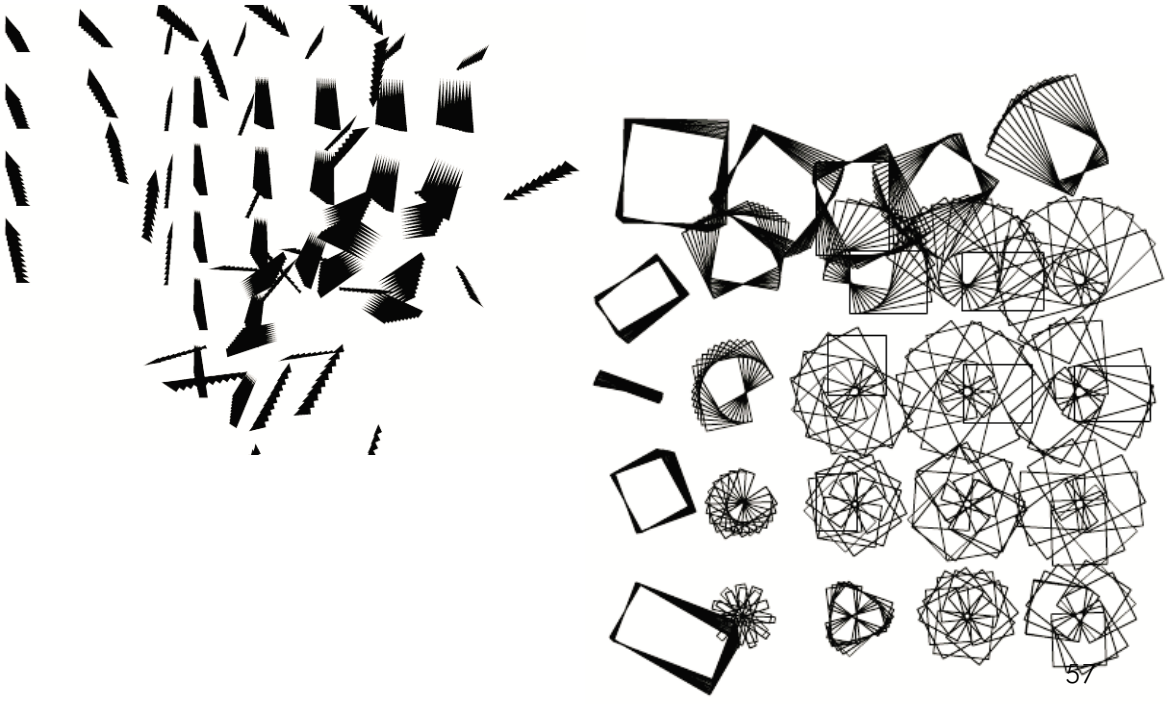
This project was inspired by the paradigm shifting design work of Verner Panton. His strong use of color and focus on geometric simplicity are qualities of a good computational substrate.

jtarbell, february 2003

## squares



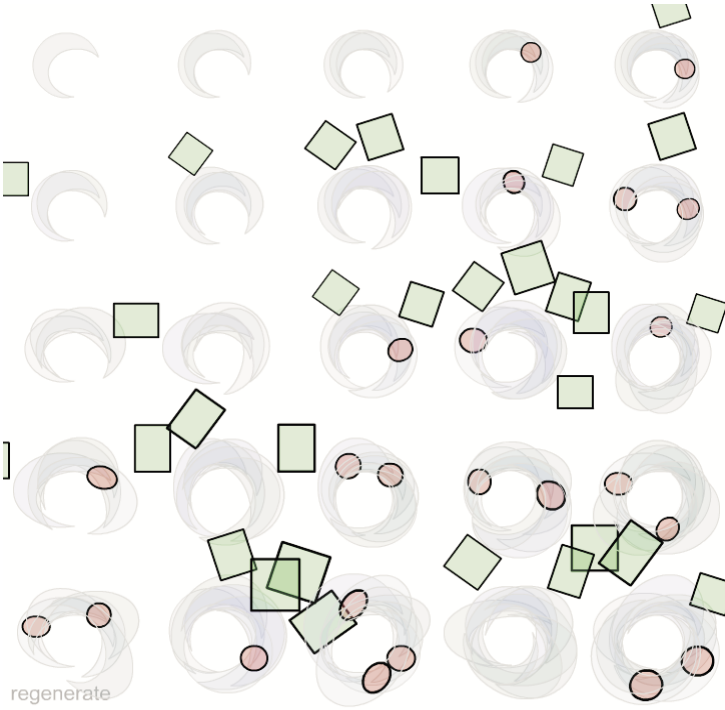
```
onClipEvent (load) {
    d = 0;
    for (b=1; b<6; b++) {
        for (a=1; a<6; a++) {
            for (i=0; i<10; i++) {
                d++;
                duplicateMovieClip (_root.square, “square”+d, d);
                _root[“square”+d]._x = 60*a;
                _root[“square”+d]._y = 60*b;
                _root[“square”+d]._rotation += Math.round(
Math.pow(a,b)*d);
                _root[“square”+d]._xscale -= (d/a);
                _root[“square”+d]._yscale -= (d/b);
            }
        }
    }
}
```

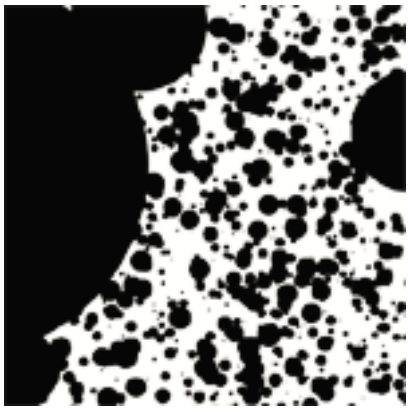



## hair generator

code by keith peters, www.bit-101.com

## iterative inspiration



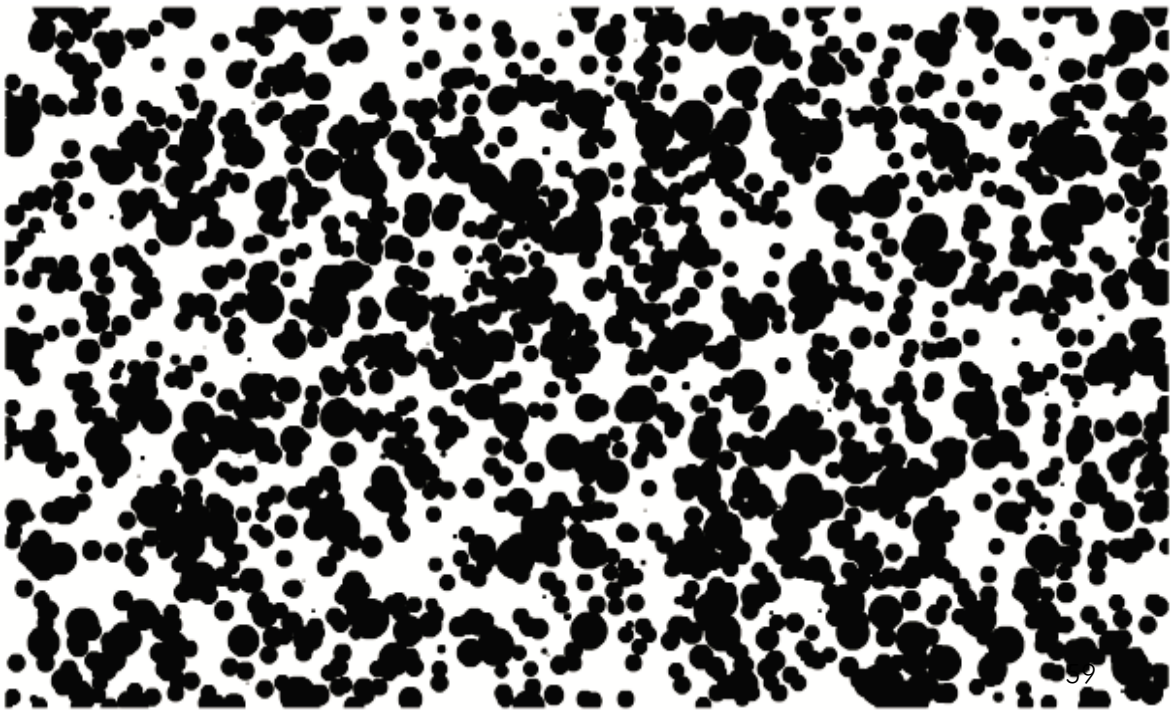


## trema generator

code by Jared Tarbell, 2002; from www.levitated.net

A 4000 element Mandlebrot texturing technique that sweeps away the extreme thirds of a finite space [0,1].
There is no user interaction.
To construct a Mandlebrot trema generator, imagine a space that is uniformly distributed over an initial domain [0,1]. Define this space as an object. Divide this object into three equal sections [0,1/3,2/3,1] and cut out the middle third. Represent the area cut as a sphere of blackness. Repeat the process for the remaining two thirds (one lower and one upper).

jtarbell, january 24, 2002



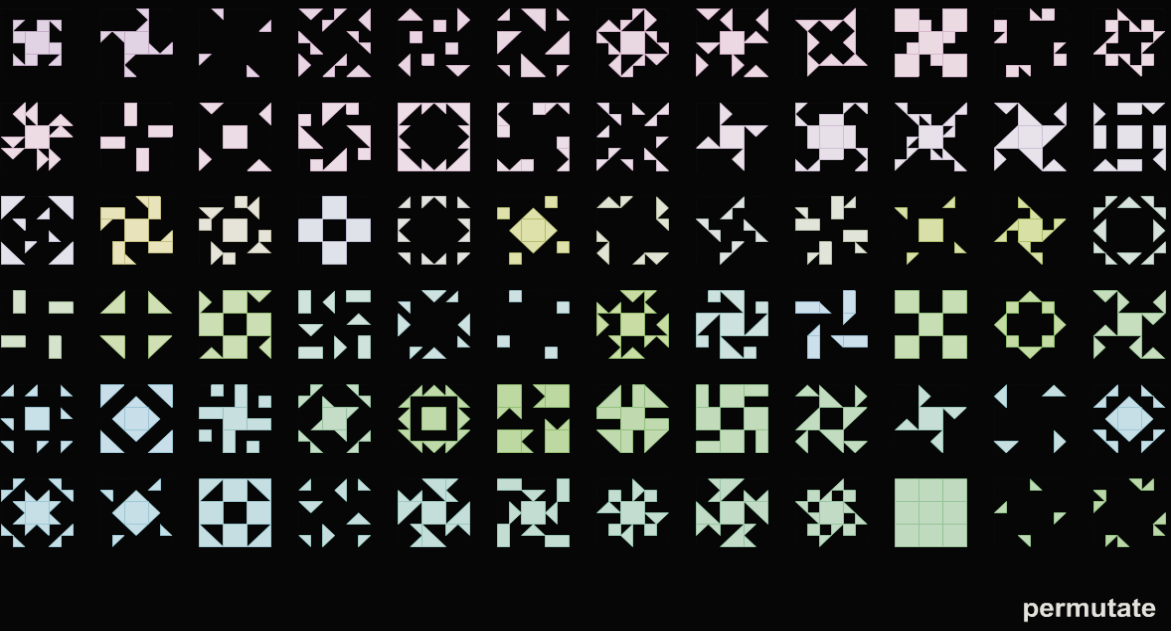
# genetische verfahren

Der Code bei diesen Beispielen verteilt sich komplex in den jeweiligen Dateien, dass wir bis auf weiteres auf die Darstellung und Erläuterung verzichten und nur die zugrunde liegende Ideen und die daraus erwachsenden Möglichkeiten zu Erläutern versuchen.

## flora



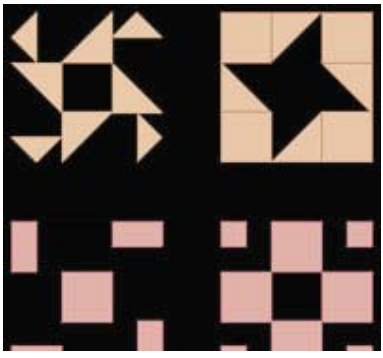
Aus verschiedenen Elementen wird durch die Kombination von entweder geziehlt oder zufällig ausgewählter Bestandteile wie Blütenblätter und deren Stengel eine sich ständig variierende Gesamtstruktur erzeugt.




## nine blocks



The nine block is a common design pattern among quilters. It's construction methods and primitive building shapes are simple, yet produce millions of interesting variations.



Each block is composed of 9 squares, arranged in a 3 x 3 grid. Each square is composed of one of 16 primitive shapes. Shapes are arranged such that the block is radially symmetric. Color is modified and assigned arbitrarily to each new block.

**figure A.** four 9 block patterns, arbitrarily assembled, show the grid composition of the block

The basic building blocks of the nine block are limited to 16 unique geometric shapes. Each shape is allowed to rotate in 90 degree increments. Only 4 shapes are allowed in the center position to maintain radial symmetry.

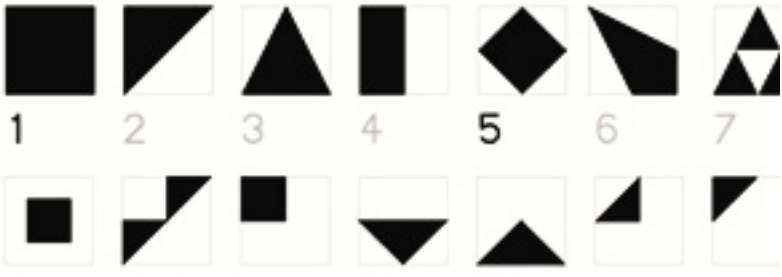



**figure B.** the 16 possible shapes allowed for each grid space. The 4 shapes allowed in the center have bold numbers.

This has got to be one of my favorite permutations simply because of the astonishing results for a very simple set of rules and primitives. If each generated nine block was a physical tile 1" in height and width, I'd have an overflowing cart of them at the bazaar, downtown, this weekend.

jtarbel| May 15, 2002

## legged creatures



A collection of small, one legged swimming automatons which combine on collision.

An organism moves about using the kicking of it's leg (or legs). Each kick produces an impulse force and a bit of faked angular momentum. Both the force and angular momentum translate into velocities which are gradually reduced with friction.

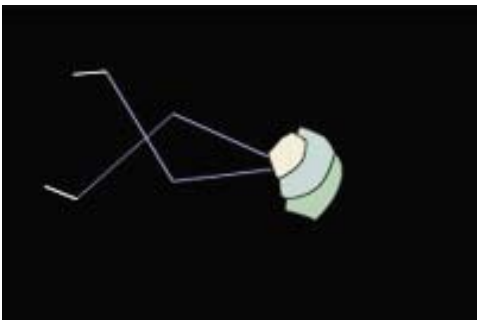


**figure a.** a well formed kicking machine

Upon collision, the larger organism will consume the smaller, aggregating it's leg structure. The most interesting result of this process is the formation of multi-legged organisms. Since the position of the aggregated legs depends upon the angle of collision, each multi-legged organism is unique. Some leg arrangements allow enhanced locomotion while others produce nonsensical movement.

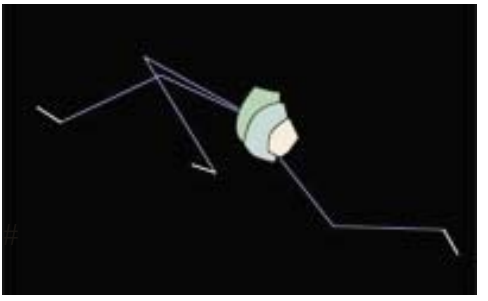


**figure b.** a slightly less practical configuration

CLICK ANYWHERE on the stage to create a new, one-legged swimming organism. There is a maximum of 11 concurrent organisms on stage. Organisms with more than 4 legs will eventually swim off the stage in search of nirvana (actually, they're ugly, so I remove them when no one is looking).

¡tarbel| September 27, 2001


## legged creatures



An environment for generating intelligent, six legged, walking insects of diverse appearance.
Press the GENERATE NEW WALKING INSECT button to create a new insect with random attributes.
Each insect is composed of 9 basic parts: a head, which decides where to go; a body, which computes how to get there; six legs, which move the body; and an ornamental tail, to get attention. The walking motion for each insect is randomly permutated across the foot shuffle order, leg length, and a fixed determination.
This project was inspired by the NINE BLOCK PATTERN GENERATOR of May 15.

¡tarbel| May 2001

## Strukturanalogien zu ‚genetische verfahren'

In der Robotik können wir Anregungen für die Bewältigung der komplexen Beziehungen in der Architektur finden:

*"Hier versucht man nicht mehr, ein immer komplizierter werdendes Modell der Wirklichkeit im Rechner nachzubauen und bei Entscheidungen zu befragen. Stattdessen besteht das Innenleben dieser neuen Wesen aus einem geschickt koordinierten Bündel von spezialisierten Reflexen zum Beispiel dem Reflexe ein unmittelbar vor ihm befindliches Hindernis zu umgehen, oder dem Reflex eine Ladestation aufzusuchen, sobald die Spannung am Akku des Roboters eine Schwelle unterschreitet, oder dem Reflex ein bestimmtes vom Benutzer vorgegebenes Ziel aufzusuchen. Das Prinzip von Brooks' Subsumptionsarchitektur besteht darin, daß zunächst jeder einzelne dieser Reflexe für sich möglichst direkt durch geeignete Verbindungen von den Sinnesorganen (d.h. von Lichtsensoren, sonaren Sensoren, Kameras oder Laser-Abstandsmessern) zu den Motoren des Roboters in möglichst einfacher und*
*stabiler Weise implementiert wird. Für den Fall, daß verschiedene dieser Reflexe zu gegensätzlichen Motor-Kommandos führen, werden solche Konflikte intern in einer Weise gelöst, die das langfristige Überleben des Roboters optimiert. Zum Beispiel: wenn die Akku-Spannung niedrig ist, aber ein Hindernis auf dem direkten Weg zur Ladestation auftaucht, dann ist es momentan wichtiger diesem Hindernis auszuweichen als zu versuchen, mit dem Kopf durch die Wand auf dem direktesten Weg zur Ladestation zu gelangen. Der geschilderte Neuansatz von Rodney Brooks ist aus der gegenwärtigen Robotik nicht mehr wegzudenken. In Schwierigkeiten gerät dieser Ansatz dort, wo die Vielfalt der zu kontrollierenden Reflexe oder die Kompliziertheit der zu bewältigenden Aufgaben es dem Ingenieur nicht mehr erlauben, die Regeln für ein geeignetes Zusammenspiel der Reflexe mittels seiner eigenen Intuition festzulegen. Ein menschlicher Betrachter dieser Problematik ist vielleicht erinnert an nicht ganz unähnliche Konflikte zwischen kurz- und langfristigen Zielen verschiedener Art, die sich teilweise unbewußt im Menschen abspielen."*

### genetischer algorithmus

*"Bei Käfern und anderen Lebewesen ist das Zusammenspiel der Reflexe im Laufe der Evolution über Millionen von Jahren hinweg soweit optimiert worden, daß sie damit überleben können. Diese Methode hat man auch in der gegenwärtigen Robotik erfolgreich eingesetzt: Indem man eine Evolution künstlich simuliert, also Teilstücke der Reflex-Koordination der überlebensfähigsten Exemplare der gegenwärtigen Generation von Robotern in verschiedenen Weisen rekombiniert (Kreuzung.) und lokal zufällig verändert (.Mutation.). Aus der so entstehenden neuen Generation von Robotern wählt man wiederum die überlebensfähigsten aus und*

*iteriert dann das geschilderte Verfahren. Schöne Beispiele von Anwendungen dieser sogenannten genetischen Algorithmen in der Robotik werden zum Beispiel in den Arbeiten der Robotik-Gruppe der EPFL Lausanne geschildert. Der wesentliche Nachteil dieser Technik besteht darin, daß sie in der Praxis erfordert, die individuell Überlebensfähigkeit von Hunderten und Tausenden verschiedener Roboter-Varianten zu evaluieren. Das nimmt sehr viel Zeit in Anspruch. Daher ist es sinnvoll, diese Evolutionstechnik zu ergänzen durch schnellere Lernmethoden, die schrittweise die Überlebensfähigkeit eines einzelnen Roboter-Individuums verbessern können. Solche Techniken werden im nächsten Unterabschnitt skizziert."*

Entwicklung durch Lernalgorithmen:

*„Ein wesentlicher Bestandteil des in unserer Gegenwartskultur verankerten Begriffs der Maschine ist die Vorstellung, daß das Verhalten einer Maschine insofern trivial ist, als sie nur das ausführen kann, was ihr von ihrem menschlichen Erbauer vorher einprogrammiert wurde. Diese Vorstellung ist veraltet, weil es in den Forschungslabors, in der Industrie, und sogar schon in der Unterhaltungselektronik eine Reihe von Programmen gibt, die es Rechnern ermöglichen, aus ihrer eigenen Erfahrung zu lernen. Insbesondere können solche lernenden Maschinen eine durch ihre vorhergehenden Erfahrungen geformte Individualität hervorbringen, die selbst von ihrem menschlichen Erbauer nicht vollkommen vorhersehbar ist. Das einzige, was der menschliche Erbauer einer solchen Maschine kennt, ist deren Lernalgorithmus, also das von ihm einprogrammierte Verfahren, mit dem die Maschine neue Verhaltensmuster aufgrund von vorhergehenden Erfahrungen entwickeln kann. Eine Vielfalt solcher Lernalgorithmen ist inzwischen bekannt."*

*siehe auch neuronale Netze im Abschnitt ‚netzwerke'*

Aus: Rainer E. Burkhard, Wolfgang Maas, Peter Waibl (Hg); Zur Kunst des formalen Denkens; Wolfgang Maass, Das menschliche Gehirn - nur ein Rechner?, Passagen Verlag, Wien 2000

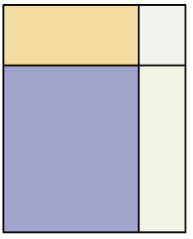
02

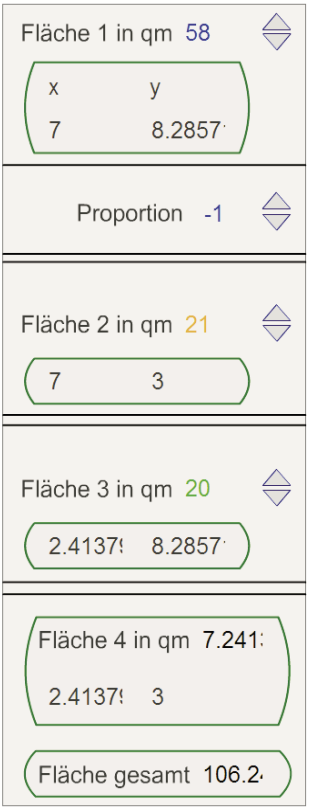
„Denken ist Simulieren“ - Oswald Wiener

# proportionsstudie

code by Reinhard König und Christian Bauriedel, www.entwurfsforschung.de

Bei dieser Studie wollten wir uns verdeutlichen, wie es möglich ist, Räume in vorgegebenen Hirarchien zu organisieren, damit bei einer Veränderung des Volumens, beziehungsweise der Abmessungen oder des Proportinsverhältnisses bei einem Raum, die rechteckige Umrisslinie erhalten bleibt und die anderen Räume sich entsprechend anpassen.






```
// initialisierung
startX = 150;
startY = 300;
// flächenZeichner
function zeichne (beginX,beginY, x, y, nr, farb) {
    _root.createEmptyMovieClip( “flaeche”, nr );
    with (_root.flaeche){
        beginFill (farb, 50);
        lineStyle (1, 0x000000, 100);
        moveTo (beginX, beginY);
        lineTo (beginX, beginY-y);
        lineTo (beginX+x, beginY-y);
        lineTo (beginX+x, beginY);
        lineTo (beginX, beginY);
        endFill();
    }
}
// proportionierung des rechtecks
function rechne () {
    eing1 = _root.qm;
    eing2 = _root.qmR2;
    eing3 = _root.qmR3;
    p = _root.prop*1;
    w = Math.sqrt(eing1);
    wR = Math.round(w);
    // raum1 blau
    rx1 = wR + p;
    ry1 = eing1/rx1;

    // raum2 orange
    if (rx1 < ry1) {
        rx2start = startX;
```

```
        ry2start = (startY-10*ry1)
        raum2x = rx1;
        raum2y = eing2/raum2x;
    } else {
        rx2start = startX+(10*rx1);
        ry2start = startY;
        raum2y = ry1;
        raum2x = eing2/raum2y;
    }
    // raum3 grün
    if (rx1 >= ry1) {
        rx3start = startX;
        ry3start = startY-10*ry1;
        raum3x = rx1;
        raum3y = eing3/raum3x;
    } else {
        rx3start = startX+(10*rx1);
        ry3start = startY;
        raum3y = ry1;
        raum3x = eing3/raum3y;
    }
    // raum4 grau
    rx4start = startX+(10*rx1);
    ry4start = startY-(10*ry1);

    if (rx1 >= ry1) {
        raum4x = raum2x;
        raum4y = raum3y;
    } else {
        raum4x = raum3x;
        raum4y = raum2y;
    }
    qmR4 = raum4x*raum4y;
    qmGes = eing1*1+eing2*1+eing3*1+qmR4*1
}
// schalter raum1
_root.propUp.onRelease = function() {
    _root.prop = _root.prop*1+1;
}
_root.propDown.onRelease = function() {
    _root.prop = _root.prop*1-1;
}
_root.eingUp.onRelease = function() {
    _root.qm = _root.qm*1+1;
}
_root.eingDown.onRelease = function() {
    _root.qm = _root.qm*1-1;
}
// raum 2
_root.eing2Up.onRelease = function() {
    _root.qmR2 = _root.qmR2*1+1;
}
_root.eing2Down.onRelease = function() {
    _root.qmR2 = _root.qmR2*1-1;
}
// raum 3
_root.eing3Up.onRelease = function() {
    _root.qmR3 = _root.qmR3*1+1;
}
_root.eing3Down.onRelease = function() {
    _root.qmR3 = _root.qmR3*1-1;
}
// ausführen
onEnterFrame = function () {
    rechne ();
    zeichne (startX, startY, 10*rx1, 10*ry1, 1, 0x0000FF);
    zeichne (rx2start, ry2start, 10*raum2x, 10*raum2y,
2,0xFF9900);
    zeichne (rx3start, ry3start, 10*raum3x, 10*raum3y, 3,
0xB4FBA4);
    zeichne (rx4start, ry4start, 10*raum4x, 10*raum4y, 4,
0xDBDBDB);
}
```

# fibonacci strukturen

code by Reinhard König und Christian Bauriedel, www.entwurfsforschung.de

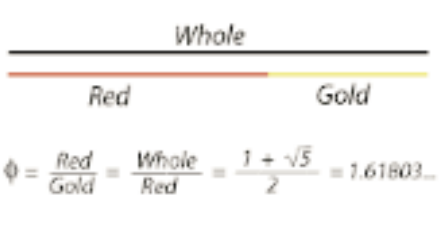
Viele Proportionen innerhalb natürlicher Strukturen lassen sich mit dem Goldenen Schnitt bzw. den Fibonacci Folgen beschreiben. Bei der bekanntesten Fibonaccifolge ergibt sich der jeweilige Wert aus der Summe der beiden vorherigen Zahlen:

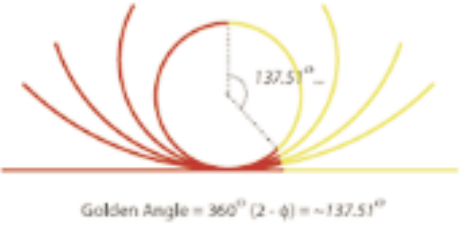
1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597,...´

Teilt man eine Zahl aus der Reihe durch die darauffolgende, so ergibt sich ein Näherungswert des Proportionsverhältnisses des Goldenen Schnitts. Um so größer die gewählten Zahlen sind, desto genauer ergibt sich der Wert.

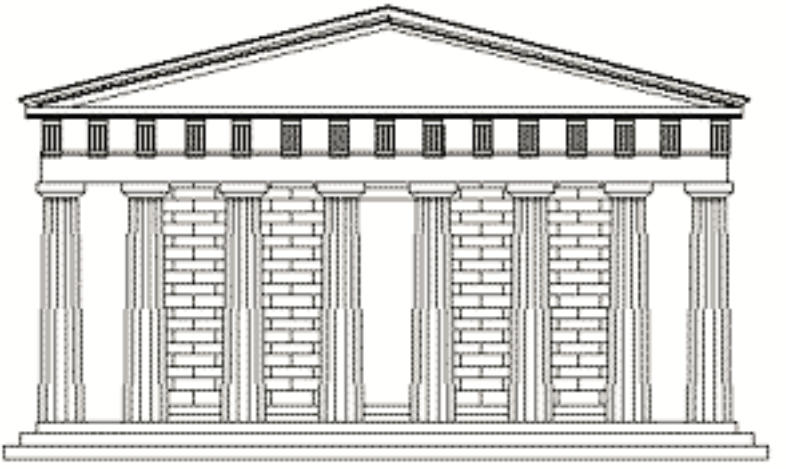
610 : 987 = 0,61803
entspricht dem Verhältnis 1 : 1,61803

Beispielsweise verwendete Iannis Xenakis die Fibonaccireihe zur Fassadengestaltung des im Büro von Le Corusier geplanten Kloster La Tourette bei den sogenannten 'Ondulatoires'.



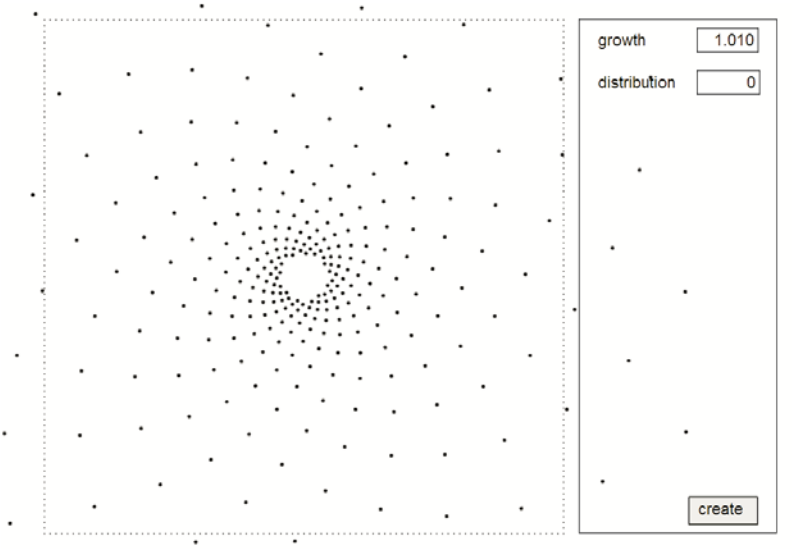




Der Goldene Schnitt diente bereits im Altertum der Gestaltung harmonischer Proportionen, hier am Beispiel des Parthenon.



weiterführende Links zur Zahlentheorie und anderem:
http://www.mscs.dal.ca/Fibonacci/
http://ccins.camosun.bc.ca/~jbritton/jbfunpatt.htm#TOPIC4

```
//start Variablen werden initialisiert
function init() {
    maxleaves = Number (_root.feldMax);
    //
    G = 1/1.618033989;
    GA = 360-360*G - Number (_root.feldDistribution);
    degenerate = Number (_root.feldDegenerate);
    //
    rad = Number (_root.feldRad);
    rgrowth = Number (_root.feldGrowth);
    growX = Number (_root.feldGrowX);
    growY = Number (_root.feldGrowY);
    xscl = Number (_root.feldXScale);
    yscl = Number (_root.feldYScale);
    //
    cur = maxleaves;
}
// Zeichenfunktion
function drawFib() {
    _root.container.createEmptyMovieClip("holder",1);
    this.onEnterFrame = function() {
        if (cur>0) {
           cur--;
           //
           GA *= degenerate;
           rot += GA;
           rot -= int(rot/360)*360;
           //
           rad *= rgrowth;
           x = Math.cos(rot*Math.PI/180)*rad;
           y = Math.sin(rot*Math.PI/180)*rad;
           //
           _root.container.holder.attachMovie("leaf","l"+cur, cur)
           mc = _root.container.holder["l"+cur];
           //
           xscl *= growX;
           yscl *= growY;
           mc._xscale = xscl;
           mc._yscale = yscl;
           mc._x = x;
           mc._y = y;
           mc._rotation = rot;
           //
        }
    };
}
// Start Schalter
buttCreate.onRelease = function() {
    init();
    drawFib();
};
// Schalter zum abbrechen und neubeginnen
buttClear.onRelease = function() {
    cur = 0;
    removeMovieClip (_root.container.holder);
}
```



`maxleaves` Anzahl der generierten Punkte

`G = 1/1.618033989` Teilungsverhältnis des goldenen Scnitts

`GA = 360-360*G - Number` Berechnung der Positionierung innerhalb des 360° Winkels des Kreises. Die variable `Number` sorgt für eine Veränderung der Sonnenblumensymmetrie wodurch z.B. ein strudelförmiges Gebilde entsteht.

Die folgenden Werte sind bei der zweitern Sruktur einstellbar (siehe Bild unten)

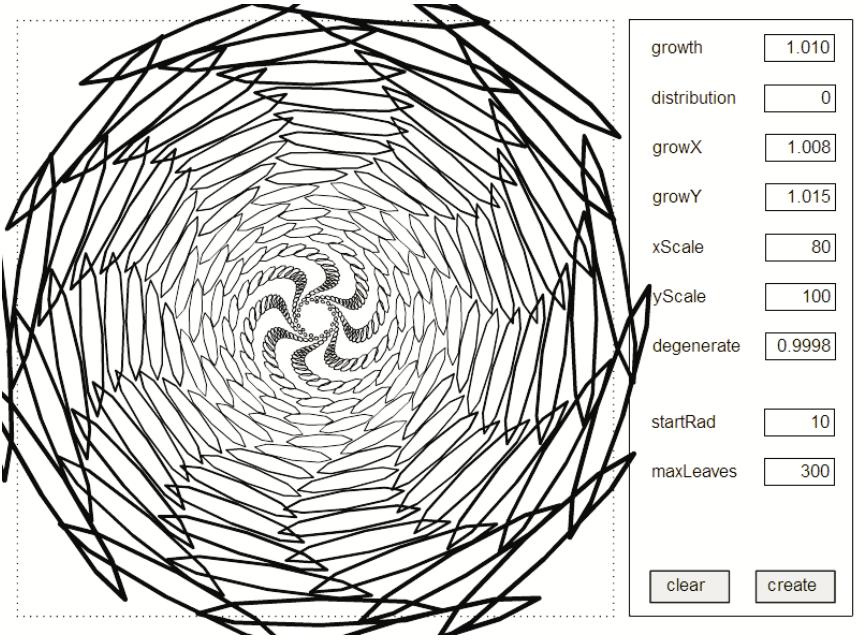
```
degenerate = Number (_root.feldDegenerate);
//
rad = Number (_root.feldRad);
rgrowth = Number (_root.feldGrowth);
growX = Number (_root.feldGrowX);
growY = Number (_root.feldGrowY);
xscl = Number (_root.feldXScale);
yscl = Number (_root.feldYScale);
```

`degenerate` Dieser Wert sorgt für eine zeitliche Veränderung des Proportionsverhältnisses des goldenen Schnitts. Bei jeder Iteration wird das Verhältnis etwas beeinflusst, bis sich in der Animation an einem bestimmten Zeitpunkt eine Art 'Verdrehung' erkennen lässt.

| | |
|---|---|
| `rad` stellt | den Anfangsradius ein. |
| `rgrowth` | setzt die Erweiterungsschritte des Radius fest. |
| `growX` | Vergrößerung des Elements im zeitlichen Verlauf |
| `growY` | Vergrößerung des Elements im zeitlichen Verlauf |
| `xscl` | Anfangsscalierung des verwendeten Elements in x-Richtung |
| `yscl` | Anfangsscalierung des verwendeten Elements in y-Richtung |

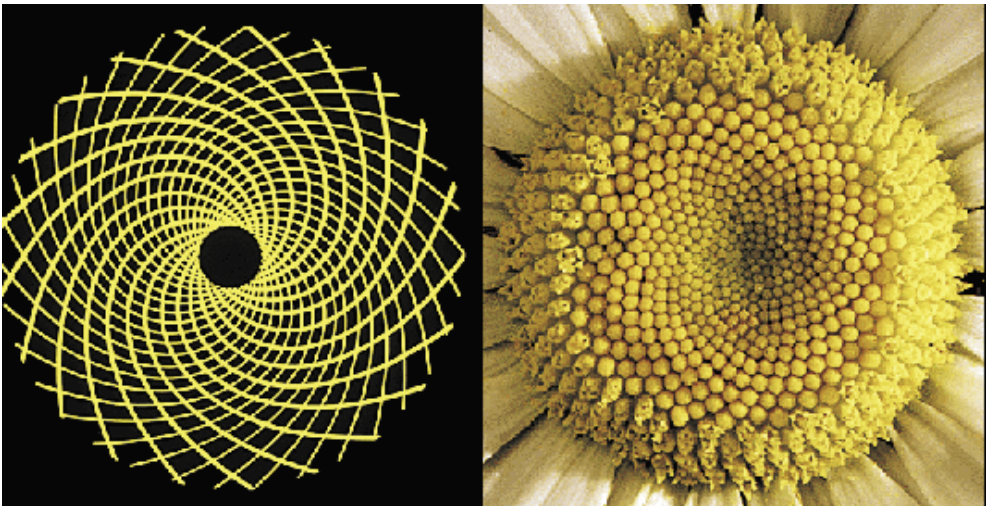
In diesen Beiden Zeilen werden die x- und die y-Position des zu setzenden Elements auf Grundlage der vorher definierten Variablen berechnet.

```
x = Math.cos(rot*Math.PI/180)*rad;
y = Math.sin(rot*Math.PI/180)*rad;
```



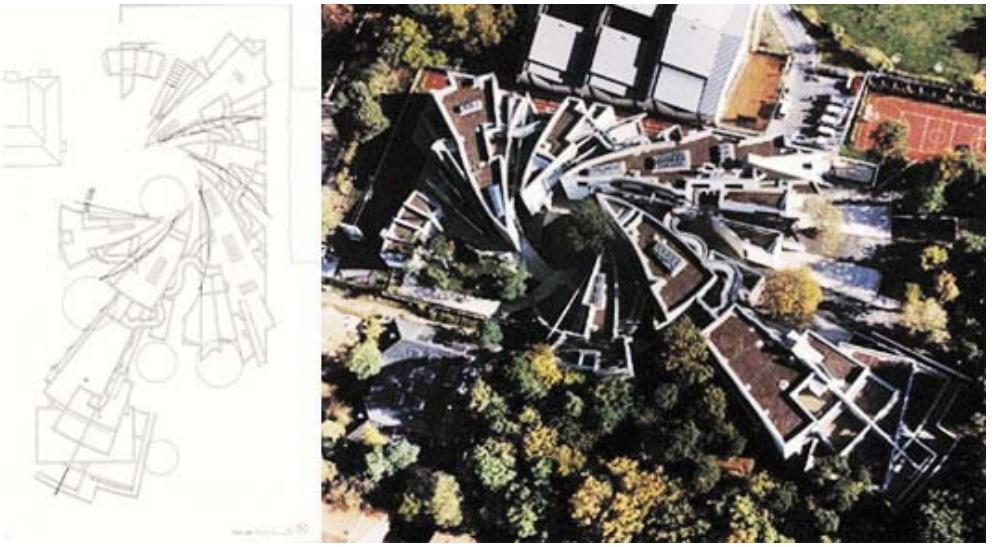
Bei Veränderung der Verteilung (distribution z.B. um 3 oder -4) verliert das System seine Sonnenblumensymmetrie und erzeugt eine Wirbelstruktur. Der Wert growth definiert die Zunahme des Abstandes nach jeder Iteration. Im zweiten Beispiel werden weitere Parameter eingeführt, um das System besser manipulieren zu können und Störungen einfließen zu lassen.

Mittels der Einführung weiterer Parameter kann die Veränderung nach jedem Iterationsschritt präziser gesteuert werden. Die Werte growX und Y verändern die Scalierung der "Blätter" nach jedem Schritt und können somit eine Verzerrung bewirken. x- und yScale dienen der Verzerrung am Beginn und über den Wert degenerate kann das Verhältnis des Goldenen Schnitts über die Zeit beeinfußt werden. Über startRad wird der Radius des inneren Kreises angegeben, maxLeaves definiert die Höchstzahl der "Blätter" bzw. die Anzahl der Iterationsschritte. Durch das experimentieren mit den Werten können die verschiedensten Strukturen erzeugt werden, welche meist an die Blüten verschiedener Pflanzen erinnern.



Ein Beispiel für die Verwendung solcher Strukturen in der Architektur stellt die Jewish Primary School Berlin (Heinz-Galinski-Schule Berlin) von Zvi Hecker dar.

http://www.zvihecker.com/frameset.htm und http://www.philiploskant.de/zvihecker

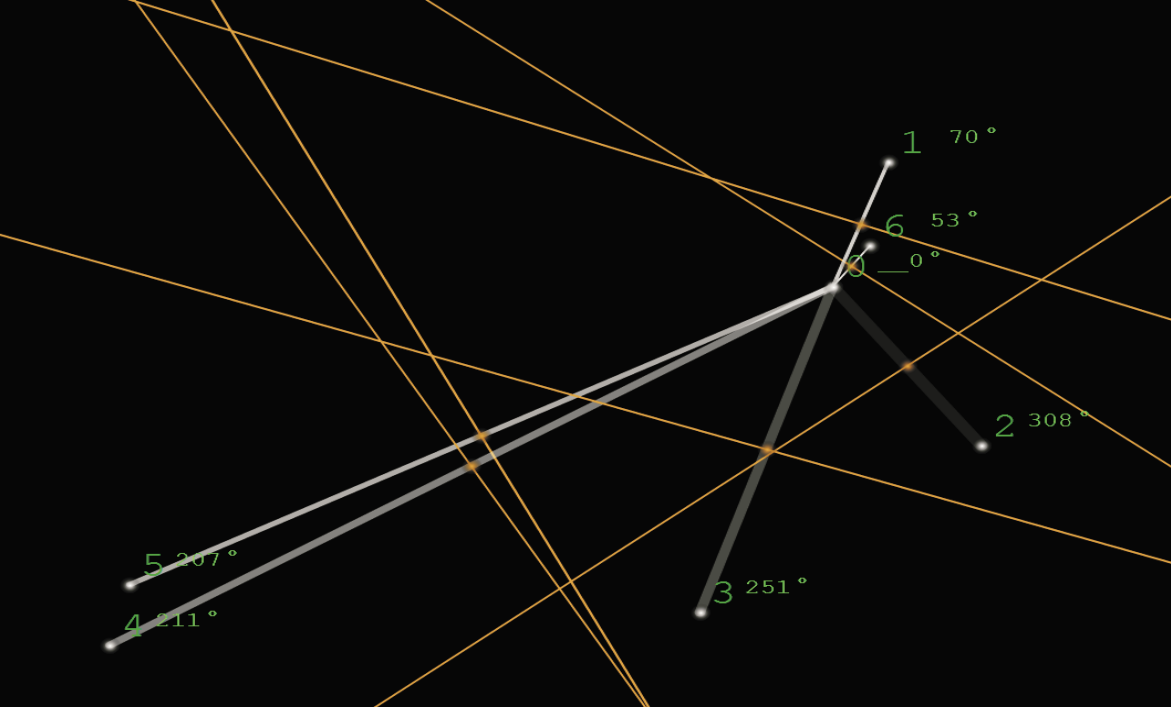


# voronoi diagramm

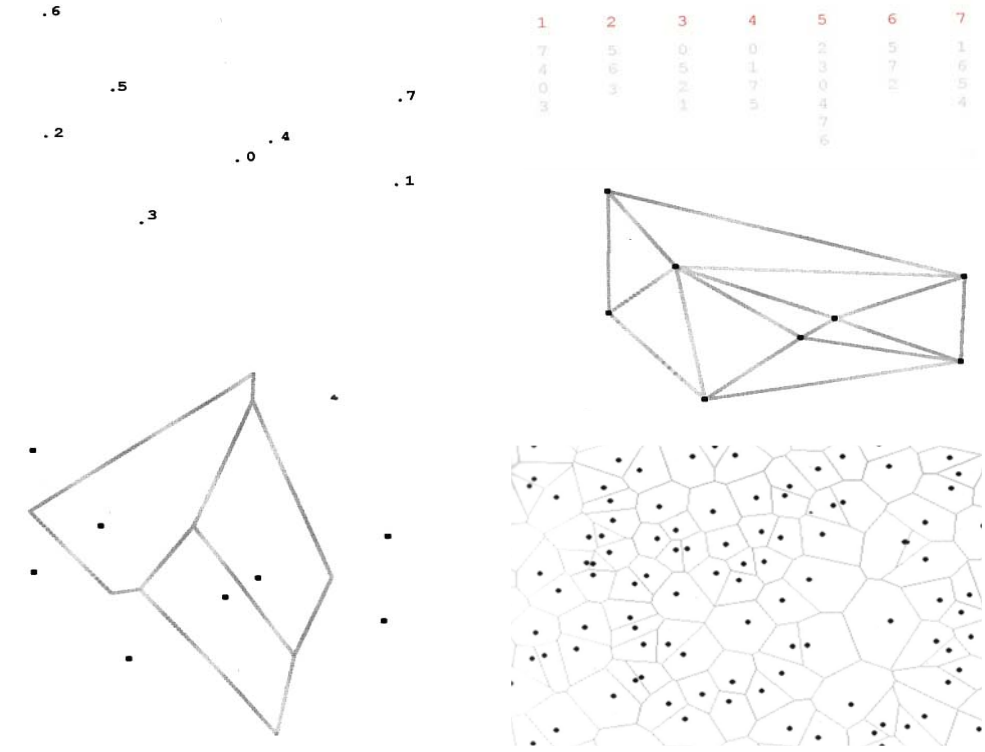
code by Reinhard König und Christian Bauriedel, www.entwurfsforschung.de

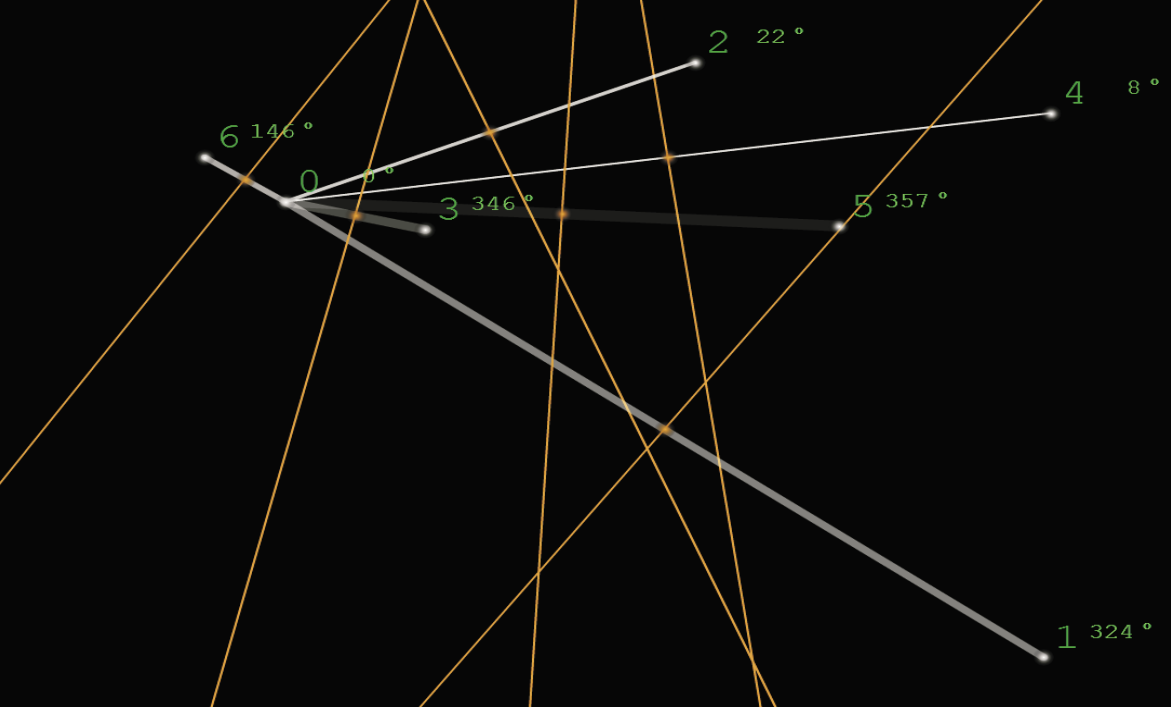
```
numNodes = 7;
nodez = [];
nodeWi = [];
nodeInd = [0, 1, 2, 3, 4, 5, 6];
midNode = [];
midNodeNx = [];
midNodeNy = [];
idxA=[];
sPkt = [];
q=0;
nachb = 0;
for(i=0;i<numNodes;i++){
    node = attachMovie(“node”, “n”+i, i);
    nodez[i] = node;
    node._x = Math.random()*550;
    node._y = Math.random()*400;
    node.nodeNr=i;
}
// Winkel berechnen
for(i=1;i<nodez.length;i++){
    nodeWink = nodez[i];
    riWiX = nodez[i].riWiX = nodeWink._x - nodez[0]._x;
    riWiY = nodez[i].riWiY = nodeWink._y - nodez[0]._y;
    delta = (riWiX) / (Math.sqrt((riWiX*riWiX)+(riWiY*riWiY))
);
    if (nodeWink._y > nodez[0]._y) {
        Winkel = 360-((Math.acos(delta))*180/Math.PI);
    } else  Winkel = (Math.acos(delta))*180/Math.PI;
    nodeWi[i] = Math.round(Winkel);
    nodeWink.nodeW = Math.round(Winkel)
}
// Reihenfolge der Nachbarn über die Winkel ermitteln
function sortZahlen(a, b){
    if(nodeWi[a] < nodeWi[b]) return -1;
    if(nodeWi[a] > nodeWi[b]) return 1;
    return 0;
}
// function für Schnittpunkt Berechnung
function schnittP (m, m2, a1x, a1y, b1x, b1y) {
    xStelle = (m*a1x-a1y-b1x*m2+b1y)/(m-m2);
    yStelle = m*(xstelle-A1x)+A1y;
}
// function distanzberechnung
function dist (p1x, p1y, p2x, p2y) {
    dx = p2x - p1x;
    dy = p2y - p1y;
    dd = Math.sqrt(dx*dx+dy*dy);
    return dd;
}
// Array über indexArray nodeWi sortieren
nodeInd.sort(sortZahlen);
nodez[0].nachbar = [];
for(var k=1; k<(2*nodeInd.length); k++){
    if (k>5) {
        idxA[k+1] = nodeInd[k-5];
        if (k>11) idxA[k+1] = nodeInd[k-11];
    } else idxA[k+1] = nodeInd[k+1];
}
for(var z=1; z<nodeInd.length; z++){
    wX = (nodez[0]._x + nodez[idxA[z+1]]._x)/2;
    wY = (nodez[0]._y + nodez[idxA[z+1]]._y)/2;
    midNodeNx[z] = wX;
    midNodeNy[z] = wY;
}

for(var i=1; i<nodeInd.length; i++){
    var idx = nodeInd[i];
    //  verbindungen zu den anderen Punkten gegen den
    uhrzeigersinn (strichstärke und transparenz)
    lineStyle(i, 0xffffff, 100-15*i);
    moveTo(nodez[0]._x, nodez[0]._y);
```

```
    lineTo(nodez[idx]._x, nodez[idx]._y);
    // Mittelpunkte
    Mit_x = (nodez[0]._x + nodez[idx]._x)/2;
    Mit_y = (nodez[0]._y + nodez[idx]._y)/2;
    Mnode = attachMovie(„nodeMid“, „m“+i, 100+i);
    midNode[i] = Mnode;
    midNode[i]._x = Mit_x;
    midNode[i]._y = Mit_y;
// Schnittpunkterrechnung
        m = nodez[idx].riWiX/-nodez[idx].riWiY;
        // Bühnenclipping
        schnittP (m, 0, midNode[i]._x, midNode[i]._y, 0, 0)
        midNode[i].achsP1x = xStelle;
        midNode[i].achsP1y = 0;
        if (midNode[i].achsP1x < 0) {
            midNode[i].rS2 = (-midNode[i]._x)/(-
nodez[idx].riWiY);
            midNode[i].achsP1y = midNode[i]._y +
        (midNode[i].rS2*nodez[idx].riWiX);
            midNode[i].achsP1x = 0;
        }
        if (midNode[i].achsP1x > 550) {
            midNode[i].rS3 =
            550 - Mnode._x)/(-nodez[idx].riWiY);
            midNode[i].achsP1y = midNode[i]._y +
        (midNode[i].rS3*nodez[idx].riWiX);
            midNode[i].achsP1x = 550;
        }
        schnittP (m, 0, midNode[i]._x, midNode[i]._y, 0,
    400)
        midNode[i].achsP3x = xStelle;
        midNode[i].achsP3y = 400;
        if (midNode[i].achsP3x < 0) {
            midNode[i].rS2 =
            (-midNode[i]._x)/(-nodez[idx].riWiY);
            midNode[i].achsP3y = midNode[i]._y +
        (midNode[i].rS2*nodez[idx].riWiX);
            midNode[i].achsP3x = 0;
        }
        if (midNode[i].achsP3x > 550) {
            midNode[i].rS3 =
            (550 - Mnode._x)/(-nodez[idx].riWiY);
            midNode[i].achsP3y = midNode[i]._y +
        (midNode[i].rS3*nodez[idx].riWiX);
            midNode[i].achsP3x = 550;
        }
        // orthogonalen Zeichnen
        lineStyle(1, 0xF58803, 60);
        moveTo(midNode[i].achsP1x, midNode[i].achsP1y);
        lineTo(midNode[i].achsP3x, midNode[i].achsP3y);
    //schnittpunkte mit den nachbarn
    for(var v=1; v<nodeInd.length; v++){
        m2 = nodez[idxA[v+1]].riWiX/-nodez[idxA[v+1]].riWiY;
        schnittP (m, m2, midNode[i]._x, midNode[i]._y,
    midNodeNx[v], midNodeNy[v])
        entf = dist (nodez[0]._x, nodez[0]._y, xStelle,
    yStelle);
        if ((entf != infinity) && (entf > 0)) {
        nachb = 1;
        for(var k=1; k<nodeInd.length; k++){
            midNode[i].entf = dist (xStelle, yStelle,
        nodez[idxA[1+k]]._x, nodez[idxA[1+k]]._y);
            if (entf > midNode[i].entf) nachb = 0;
        }
        if (nachb == 1) {
            nodez[0].nachbar[q]= nodez[idx];
            nodez[0].nachbar[q].X = xStelle;
            nodez[0].nachbar[q].Y = yStelle;
            //
            schnittNode = attachMovie(“nodeSchnitt”,
        “s”+w, 1000+w++);
            sPkt[w] = schnittNode
            sPkt[w]._x = nodez[0].nachbar[q].X;
            sPkt[w]._y = nodez[0].nachbar[q].Y;
            q++;    }    }     }     }
```




Die Struktur des Programms ist als diskretes System konzipiert, wobei jeder Punkt ein Objekt dieses Systems verkörpert und im Austausch mit seinen unmittelbaren Nachbarn steht. Geometrische Grundlage ist die Tatsache, dass für jeden Punkt ein Polygon definiert werden kann, das die Fläche beschreibt, die sich näher bei diesem Punkt befindet als bei allen anderen Punkten der Menge. Jede Polygonseite trennt den Punkt von einem ihm ‚am nächsten' liegenden. Die dabei entstehende Struktur ist als Voronoi-Diagramm bekannt. Eine Punktmenge kann so auf zwei verschiedene Weisen dargestellt werden, durch ihre Umgebungspolygone oder mit den Verbindungslinien zu ihren ‚nächsten Punkten', den Nachbarn. Im Programm wird jeder Punkt zum Kopf einer Liste, in der jeder andere Punkt als weiteres Listenelement gespeichert wird, der als Nachbar erkannt worden ist. Die Datenstruktur macht somit möglich, dass ein Punkt die Daten seiner Nachbarn abfragen kann.






Für jeden Punkt wird zunächst der nächstliegende andere Punkt gesucht. Die Mittelsenkrechte der Verbindungslinie bildet die erste Polygonseite. Von diesem Punkt ausgehend werden jetzt gegen den Uhrzeigersinn die anderen Punkte untersucht: auch hier wird die Mittelsenkrechte durch die Verbindungslinie erstellt und der Schnittpunkt dieser mit der Mittelsenkrechten des vorigen Punktes - als Potentieller Eckpunkt des Polygons - ermittelt. Liegt dieser Schnittpunkt bei keinem anderen Punkt der Punktmenge näher als beim Bezugspunkt und beim aktuell untersuchten Punkt, handelt es sich um einen Eckpunkt des Polygons, und der aktuell untersuchte Punkt wird als ‚Nachbar' in die Liste aufgenommen.
Die Seitenzahl des fertig erstellen Polygons entspricht der Anzahl der ‚nächsten' Nachbarn.

Die Angaben und Diagramme zur Erläuterung des Voronoi Diagramms sind einer Arbeit von Eva Friedrich an der Technischen Universität Kaiserslautern entnommen.

# anhang

Zu den Kapiteln ‚node garden' und ‚Flächenbesetzung':
Aus: „Ungeplante Siedlungen, non-plannes Settlements" des Instituts für Leichte Flächentragwerke (IL 39) der Universität Stuttgart, Herausgeber ist Frei Otto. 1990 als Dissertation von Eda Schaur erschienen. S.64

*Vergleichende Betrachtung der selbstbildenden Strukturen*
*Die unterschiedlichen Gesetzmäßigkeiten der Strukturbildung - kürzeste Verbindungen der Punktepaare, Minimierung der Gesamtsystemlänge, Bündelung der einzelnen Strecken, dichte Packung oder die maximal mögliche Ausdehnung einzelner Einheiten - führten zur Entstehung jeweils charakteristischer Strukturformen.*
*Einzelne Strukturen der Direktwege, Minimalwege, minimierten Umwege, Blasenflöße, oder die von Sandschüttungen gebildeten Packungen zeichneten sich durch die soeben besprochenen, jeweils charakteristischen Merkmale aus. Dies waren, wie wir gesehen haben, immer topologische Eigenschaften, aber auch einige metrische Merkmale der Strukturen. So sind Direktwegesysteme immer geschlossene Systeme mit vier- oder mehrarmigen Knoten, Minimalwege durchweg offene Systeme mit nur dreiarmigen Knoten, oder Blasenflöße immer geschossene Systeme mit dreiarmigen Knoten.*

*Innerhalb dieser Grenzen kann durch quantitative und qualitative Variationen der Randbedingungen eine kaum übersehbare Vielfalt individueller Strukturformen entstehen, die für die jeweilige Gruppe (Familie) der selbstbildenden Strukturen die jeweils typische „Formwelt" bilden.*

*Die betrachteten Systeme der Verknüpfung und die Systeme der Packung stellen zwei grundsätzlich unterschiedliche, bei gleichzeitigem Auftreten konkurrierende Prinzipen der Strukturbildung dar. Während bei den Verknüpfungssystemen, den Direktwegen, den Minimalwegen und den minimierten Umwegen die Struktur durch Anordnung linearer Elemente zwischen Punkten auf einer ansonsten freien Fläche entsteht, wird sie bei den Packungen von flächigen Elementen durch Besetzen der Fläche gebildet.*

*Bei der Entstehung der Siedlungsstrukturen sind beide Aspekte, Packung und Verknüpfung, von Bedeutung. Es ist kaum anzunehmen, dass ausschließlich einer der beiden, wie bei den selbstbildenden Strukturen, die Entstehung der Struktur bewirkt. Siedlungsstrukturen können also nicht die aus den Experimenten bekannte Klarheit der Formen aufweisen, was sowohl in der Menge und in der Art der sie bewirkenden Einflüsse, wie im Wegfallen der engen Zwänge physikalischer Gesetzmäßigkeiten begründet sein kann. Sie können aber durchaus Ähnlichkeiten zeigen, aus denen Schlüsse gezogen werden können.*

Diese Feststellungen sollen auch für unsere Arbeit im zweiten Teil gelten, wenn wir versuchen eigene Struktursimulationen zu programmieren.