Bundeswettbewerb Informatik 2023

Robert Vetter

November 2022

Inhaltsverzeichnis

Hagen Jacob, Magnus Kirbach,
Edgar König und Robert Vetter

1 Aufgabe 2

1.1 Lösungsidee

In der Aufgabe wurde gefordert, zufällige Kristallisationskeime auf eine graphische Oberfläche zu zeichnen, welche nach links, rechts, oben sowie unten wachsen. Hierbei bietet es sich an, zufällige Vierecke zu generieren, welche sich

1. bei Kollision mit den Rändern des Fensters aufhören zu wachsen,
2. bei Kollision mit anderen Kristallisationskeimen aufhören zu wachsen sowie
3. sich nicht überlappen und nicht ineinander erscheinen.

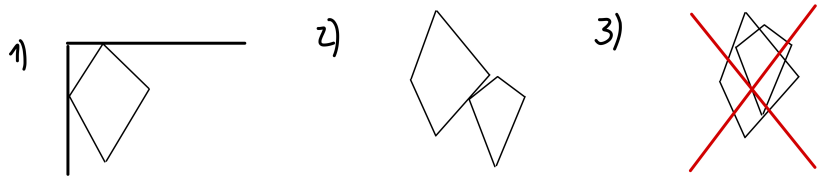

Abbildung 1: Veranschaulichung der Simulationsvorschriften

Zu jedem Zeitpunkt muss die genaue Position eines Vierecks abrufbar sein. Für die Bewegung ist es dabei notwendig, dass die x- und y-Koordinaten der Punkte ihre räumliche Lage als Wachstumsbewegung des sich in der Fläche wachsenden Vierecks verändern. Um die Überlappung mit anderen Kristallisationskeimen zu vermeiden, habe ich mich speziell mit der Berechnung von geometrischen Formen auseinandergesetzt. So war meine Idee, Vektoren zur genauen Positionsbestimmung zu verwenden, da diese sehr einfach implementiert werden können sowie speziell für graphische Probleme geeignet sind. Zudem lässt sich unter Anwendung einfacher Algorithmen schnell feststellen, ob sich ein Eckpunkt eines Vierecks innerhalb eines anderen Vierecks befindet und somit eine Kollision stattgefunden hat.

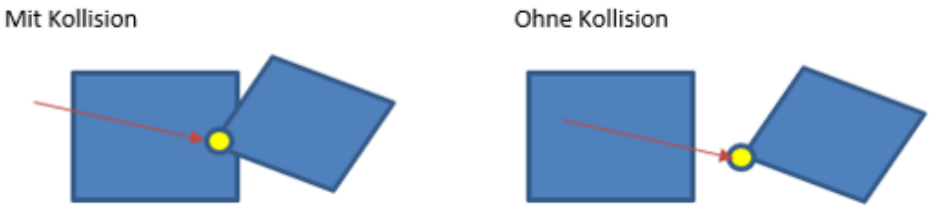

Abbildung 2: Lage Punkt mit Bezug auf Viereck

Weiterhin soll die Simulation in den folgenden unterschiedlichen Betrachtungsweisen variiert werden:

1. in den Wachstumsgeschwindigkeiten der Kristalle,
2. in der Anzahl der Keime,
3. in den Entstehungszeiten sowie
4. in den Erscheinungsorten.

Nach Prüfung weiterer Möglichkeiten entschied ich mich aufgrund des letzten Parameters zusätzlich einen manuellen Modus zu integrieren, in welchem man selbstständig Vierecke zeichnen kann.

1.2 Umsetzung

1.2.1 Grundlagen am Beispiel von Kreisen

Um diesen Ansatz umzusetzen, stellte sich mir zuerst die Frage nach einer geeigneten Programmiersprache. Da aus der Aufgabe deutlich wird, dass sehr viel graphisch gelöst werden muss, entschied ich mich für einen Ableger von Java namens Processing. Ich begann damit, erst einmal die Vierecke und deren Ausbreitung zu simulieren, bevor ich mich mit der Überlappung und sonstigen Charakteristiken beschäftigte. Dabei vereinfachte ich mir die Problemstellung und erstellte ein solches Programm für Kreise. Zuerst fügte ich eine zusätzliche Klasse "Circle"hinzu und übergab in den dazugehörigen Konstruktor zwei zufällige x- und y-Werte für die Position sowie einen Wert für den Radius. Weiterhin werden folgende Methoden benötigt:

1. "void() grow", welche den Kreis wachsen lässt
2. "boolean edges()", welche mithilfe von Fallunterscheidungen überprüft, ob ein Kreisrand über die Begrenzung hinausgeschritten ist als auch
3. "void show()", welche die Ellipse zeichnet.

Nachdem in der setup()-Methode alle statischen und formalen Werte festgelegt wurden, erstellte ich eine Array-Liste, die alle Kreise als Objekte beinhaltet und fügte dieser, immer wenn ein Kreis neu gezeichnet wurde, dieses Objekt hinzu. Das Ergebnis war folgendes:

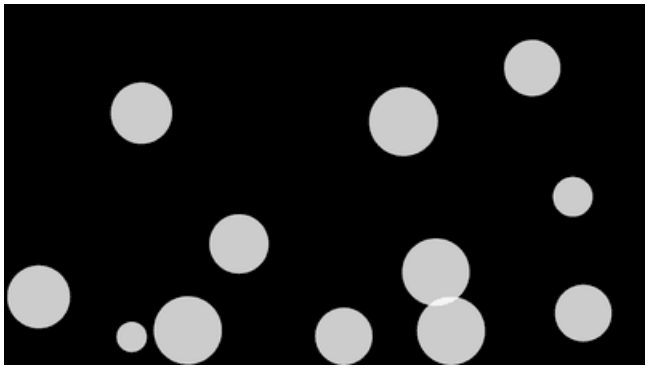

Abbildung 3: Kreise erste Ausgabe

Es entstanden zahlreiche Kreise, welche aufhörten zu wachsen, wenn sie die Ecken erreicht hatten. Dabei überlappten sie sich jedoch häufig und hatten oft auch nahezu denselben Ursprungspunkt. Um letzteres zu vermeiden, implementierte ich eine Methode "Circle newCircle()", welche die Array-Liste durchgeht. Sobald von einem neuen Kreis die Entfernung zwischen seinem Mittelpunkt und dem Mittelpunkt eines schon gezeichneten Kreises kleiner war, als der Radius des schon gezeichneten Kreises, sollte eine neue Position für den zu zeichnenden Kreis gesucht werden (Skizze 1). Um das Überlappen beim wachsen zu vermeiden, habe ich die Distanz zwischen zwei Ursprungspunkten berechnet. Sobald diese kleiner war, als der Radius des neuen Kreises und des schon bestehenden Kreises addiert, durfte der Kreis nicht weiter wachsen (Skizze 2):

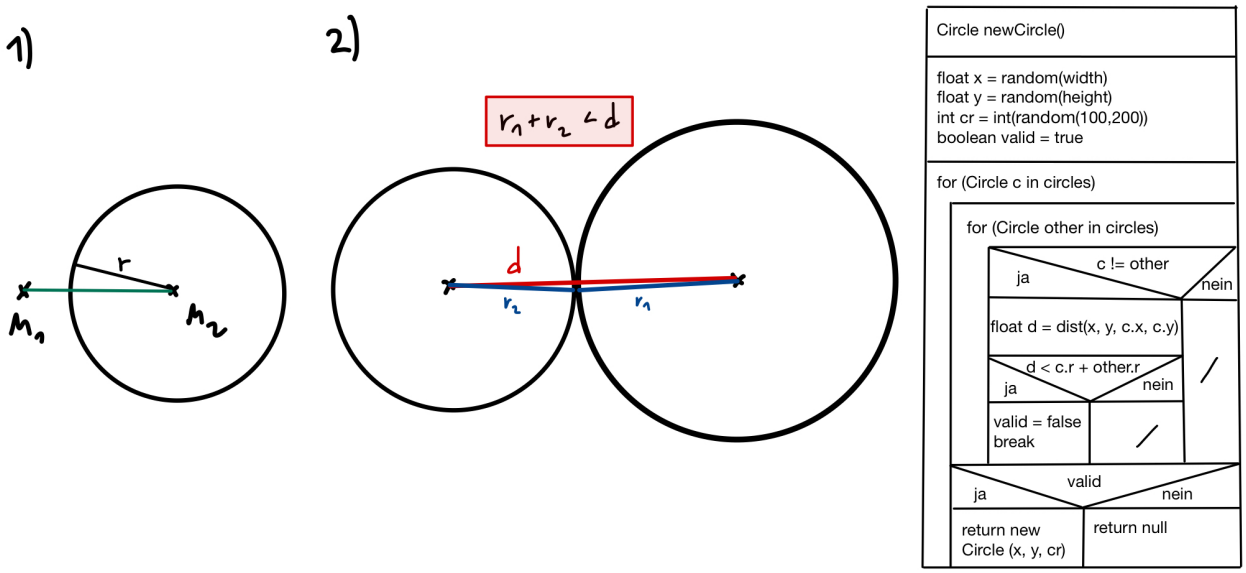

Abbildung 4: Veranschaulichung der Überlappung von Kreisen

Wenn man jetzt noch zufällige Grautöne über den Befehl "fill(r, g, b)"hinzufügt, erhält man folgendes als Ausgabe:

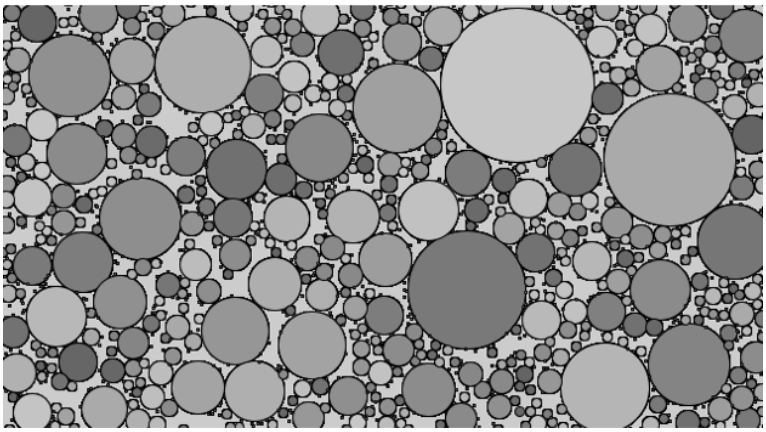

Abbildung 5: Zufälliges Generieren von grauen Kreisen

1.2.2 Erweiterung auf unregelmäßige Vierecke

Um das bestehende Programm nun auf unregelmäßige Vierecke zu erweitern, erhöhte ich die bisherigen zwei zufälligen x- und y-Startwerte auf insgesamt 4 Wertepaare an x- und y-Positionen. Dabei beschreibt ein Wertepaar immer eine Ecke. Ich verwendete diese Aufteilung der Koordinaten:

Hagen Jacob, Magnus Kirbach,
Edgar König und Robert Vetter

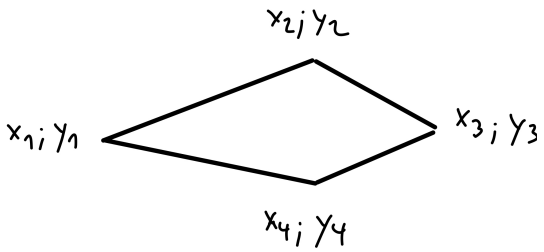

Abbildung 6: Aufteilung der Koordinaten an einem Viereck

Zusätzlich fügte ich vier zufällige Ausbreitungsgeschwindigkeiten nach links, oben, rechts und nach unten hinzu. Wenn eine Ecke des Vierecks eine Begrenzung des Fensters trifft, wird das Wachstum in diese Richtung gestoppt. Weiterhin musste ich mir etwas neues wegen der Überlappung beim Ausbreiten ausdenken. Die unregelmäßigen Vierecke haben im Gegensatz zu den Kreisen keinen klaren Mittelpunkt. Aus diesem Grund benutzte ich einen Algorithmus, der ausrechnet, ob sich ein Punkt auf einer Linie befindet oder nicht. Die Linie stellt in diesem Fall eine Viereckssseite dar und der Punkt einen Eckpunkt eines anderen Vierecks. Hier sollte zuerst die Länge der Linie (bzw. Viereckseite) berechnet werden. Dieses lässt sich über den Satz des Pythagoras machen oder einfacher über die schon in Processing eingebaute "dist(x1, y1, x2, y2)" Funktion. Zusätzlich wird auch der Abstand zwischen dem Punkt und den beiden Enden der Linie berechnet. Dieses macht man nach demselben Vorgehen. Wenn nun die Abstände zwischen dem Punkt und den beiden Enden gleich der Länge der Linie ist, liegt der Punkt auf der Linie. Zur besseren Veranschaulichung habe ich folgende Skizze erstellt:

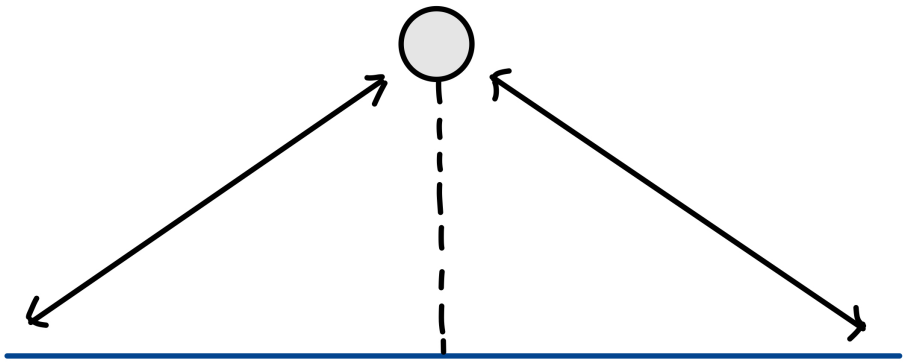

Abbildung 7: Veranschaulichung zur Punkt-Linien-Berechnung

Dieses habe ich für jede mögliche Überlappungsstelle gemacht. So kann bspw. der Punkt auf der linken Seite des Vierecks mit einer anderen Viereckssseite entweder

1. rechts oben oder
2. rechts unten kollidieren.

Wenn man dieses auf alle möglichen Kollisionen anwendet, erhält man 8 mögliche Fälle. Weiterhin erscheinen zurzeit die Vierecke auch noch in den anderen Vierecken. Um dieses zu vermeiden, erstellte ich eine Methode "SquarePointCollision()". Dabei wird zuerst ein Array mit 4 verschiedenen Vektoren kreiert, welche die Seiten des Vierecks darstellen. Anschließend werden die Vektoren einzeln abgerufen. Anfangs wird geprüft, ob der Punkt zwischen den beiden Scheitelpunkten der Vektoren in Y-Richtung liegt:

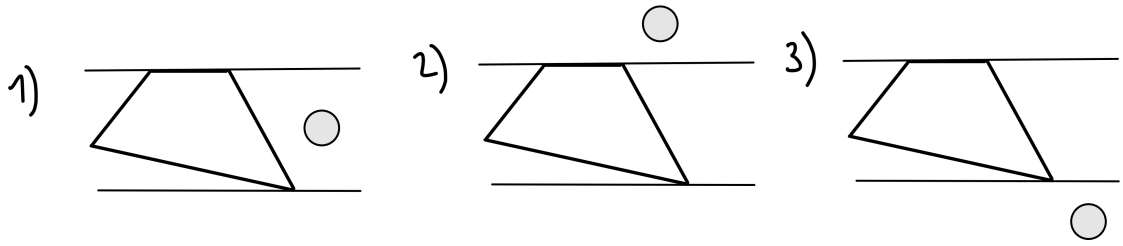

Abbildung 8: Lage des Punktes mit Bezug zu den Vektoren

Im ersten Fall liegt der Punkt zwischen beiden Vektoren, im Fall 2) über dem oberen und im Fall 3) unter dem unteren. Anschließend wird mithilfe des Jordan Curve Theorems die x-Position des Punktes überprüft. Wenn beide Überprüfungen wahr sind, wechselt die boolean-Variable "collision" zum entgegengesetzten Wert. Anfänglich war sie dabei "false".

1.2.3 Implementation der vorgegeben Parameter und zusätzlichen Features

Die Anzahl der gezeichneten Vierecke lässt sich mithilfe einer Fallunterscheidung bezüglich der Länge der Array-List anpassen. Da in der Aufgabenstellung deutlich wurde, dass auch der Ort der Keime angepasst werden soll, erstellte ich einen manuellen Modus, in welchem selbstständig Keime gezeichnet werden können. Dieses lässt sich über die schon von Processing vorgegebenen Funktionen "mousePressed" und "mouseClicked" verwirklichen. Zur Begrenzung der Anzahl implementierte ich wiederum eine Fallunterscheidung, welches die Länge der Array-List mit der angegebenen Anzahl n vergleicht. Bezüglich der Entstehungszeitpunkte der Kristalle überlegte ich mir folgendes: Die Methode draw() wird 60 Mal pro Sekunde aufgerufen. Ich zählte jeden Aufruf und sobald dieser modulo einer bestimmten Zahl 0 ergibt, soll ein weiterer Keim zur Array-Liste hinzugefügt werden. Ein Beispiel:
Möchte ich alle 0,5 Sekunden einen neuen Keim erscheinen lassen, dann sollte ich folgende Fallunterscheidung durchführen:

$$\text{if } (0 == frameCount \pmod{30})\,\{\dots\}$$

Um die Wachstumsgeschwindigkeiten der Kristalle zu variieren, müsste man für die bisher auf zufällig gesetzten Geschwindigkeiten entweder Werte einsetzen oder den Wertebereich nach Belieben verändern. Zusätzlich implementierte ich eine Funktion "keyPressed()", die bei Betätigung des Buchstabens 's' automatisch ein Bildschirmfoto der Simulation erstellt sowie beim Drücken des Buchstabens 'p' die Simulation unterbricht. Das Ziel sollte es sein, dass die in der Veranschaulichung entstandenen Bilder konvertiert zu Pixelbildern dem vorgegeben Bild ähneln. Hierzu schrieb ich ein kurzes Python-Skript, welches das angefertigte Bildschirmfoto importiert und verkleinert, sodass nur noch wenige Pixel abgebildet werden. In dem nächsten Gliederungspunkt werde ich auf von mir angefertigte Beispiele eingehen.

1.3 Beispiele

Um ein möglichst ähnliches Pixelbild zu generieren, versuchte ich, die Farbgebung der Vierecke bestmöglich an die vorgegebene Farbgebung anzugleichen. Hierzu implementierte ich einige Fallunterscheidungen bezüglich der Orte der Keime, welche imaginäre Bereiche des Ausgabefensters überdecken. Ich verwendete nicht den manuellen Modus, sondern den Simulationsmodus. In der nachfolgenden Abbildung sieht man das Ergebnis der Ausgabe sowie das dazugehörige Pixelbild:

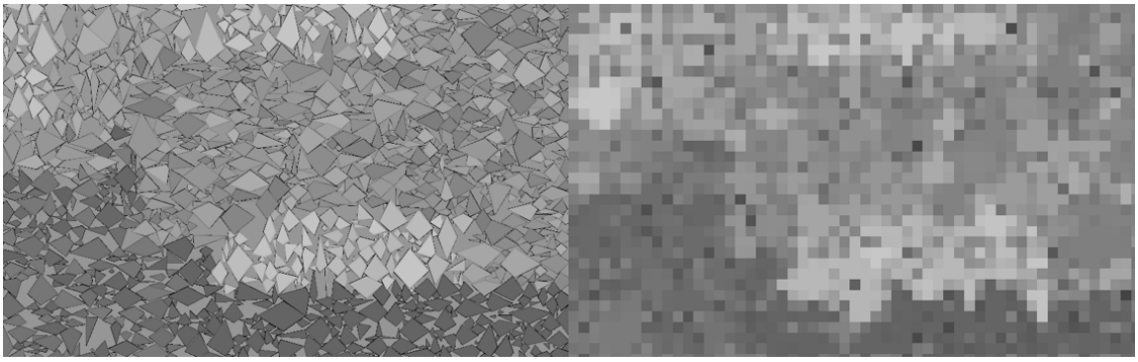

Abbildung 9: Ausgabe mit Ähnlichkeit zu in Aufgabe gegebenen Bild

Es weist gewisse Ähnlichkeiten zu dem vorgegebenen Bild in der Aufgabenstellung auf. Ich verwendete hierfür folgende Werte:

1. die Anzahl n an den Keimen setzte ich auf 1000,
2. ich ließ jeden 1/30-ten Teil einer Sekunde einen Keim generieren,
3. bei der Farbgebung entschied ich mich für eine entsprechende Aufteilung imaginärer Bereiche,
4. wobei ich die Geschwindigkeiten auf zufällig setzte.

Nun habe ich noch eine optische Vierteilung des Bildschirms durch die Kristalle erstellt:

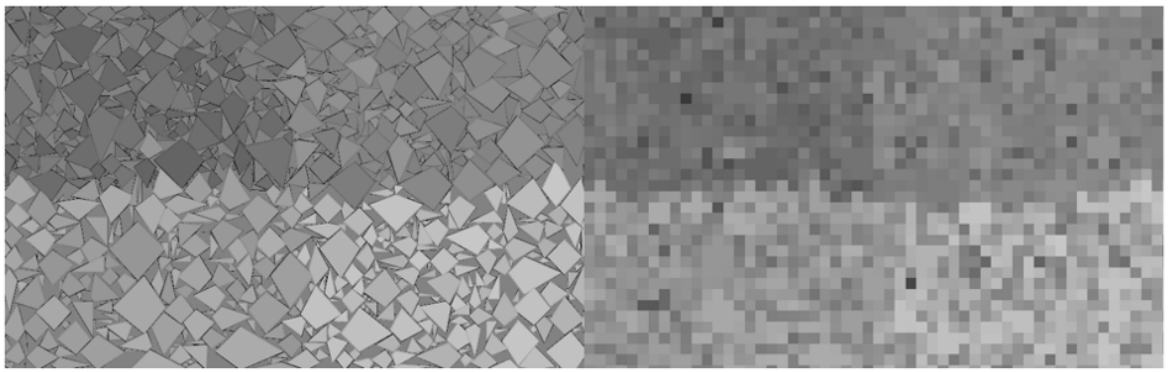

Abbildung 10: Muster mit farblicher Vierteilung

Die Anzahl n beträgt äquivalent zu dem Beispiel davor 1000, genauso verhält es sich mit der Erscheinungsszeit. Bei dem Muster entschied ich mich für eine Vierteilung und ich setzte alle Geschwindigkeiten auf einen Bereich zwischen 0.03 und 0.05. Zwei weitere generierte Beispiel finden sich im Anhang.

1.4 Anhang

1.4.1 Weitere Beispiele

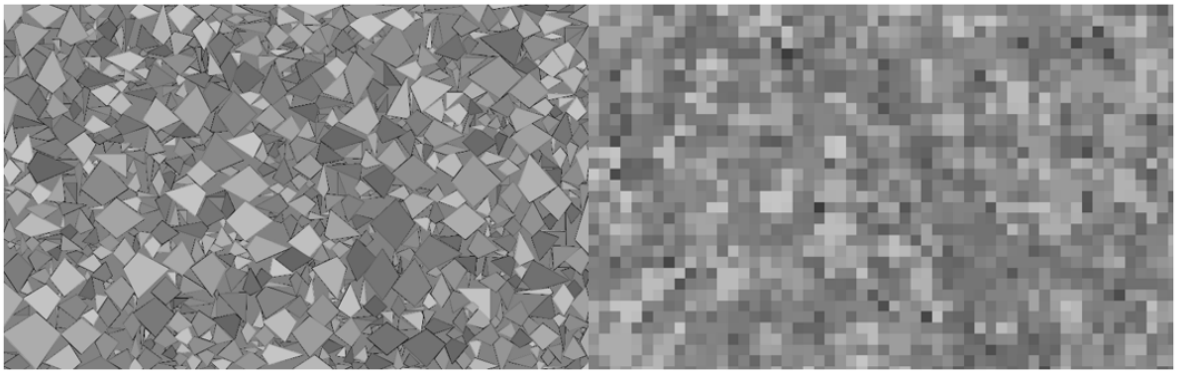

Abbildung 11: Muster mit zufälliger farblicher Verteilung

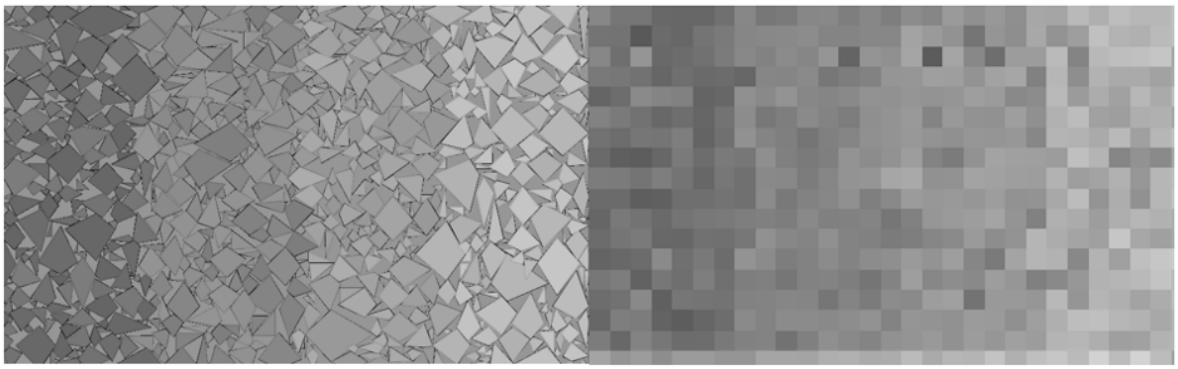

Abbildung 12: Muster mit einer Streifenverteilung der Grautöne

Abbildung 13: Selbsterstelltes Muster im manuellen Modus

Alle Abbildungen wurden dabei von mir selbst erstellt.

1.4.2 Quellcode

Aufgrund der Länge meines Quellcodes erachtete ich es nicht für sinnvoll, diesen an meine Dokumentation anzuhängen. Er ist in demselben Ordner wie meine Dokumentation zu finden.

```
//Zusaetzliche Informationen werden unten in der Konsole ausgegeben

//Parameter
boolean manualMode = false; //mithilfe des manuellen Modus koennen eigene
    Vierecke gezeichnet werden und so die genaue Position der Keime festgelegt
    werden

int n = 1000;//Anzahl an maximal erscheinenden Vierecken

//nur im Simulationsmodus verfuegbar
float time = 0.5; //Jede (time/60)s wird ein neuer Keim geschaffen

float vmax = 0.1; //Maximale Wachstumsgeschwindigkeit
float vmin = 0.05; //Minimale Wachstumsgeschwindigkeit

//nur im Simulationsmodus verfuegbar
int selection = 1; //Auswahl = 1: Muster wie in Aufgabe, Auswahl = 2:
    zufaelliges Muster, Auswahl = 3: senkrechte Vierteilung, Auswahl = 4:
    Aufteilung in Quadranten


//Beginn des eigentlichen Programms
//Liste von Objekten initialisieren(Viereck)
ArrayList<irrSquare> irrSquares;

//Farbe initialisieren
color cl;

//Geschwindigkeit
float vLeft;
float vUp;
float vRight;
float vDown;

//Vektoren zur UEberpruefung, ob Viereck in schon bestehendem Viereck erscheint
PVector[] vektoren = new PVector[4];

//Variable, die zur Klassifizierung der Screenshots verwendet wird
int num = 0;

//Variable, die Unterbrechung der Simulation angibt
boolean paused = false;

//Variable, die angibt, ob die Maus geklickt wurde
boolean varMouseClicked = false;

//Ortskoordinaten fuer die Vierecke
float x1;
float y1;
float x2;
float x3;
```

```
float x4;
float y2;
float y3;
float y4;

//jede timeX Sekunden wird ein neuer Keim generiert
float timeX = time/60;

void setup(){
  //grundlegende Eigenschaften festlegen
  size (800, 500);
  background(150, 150, 150);
  strokeWeight(0.05);
  frameRate(100);

  //Hinweise fuer den Benutzer ausgeben
  if(manualMode){
    println("Sie befinden sich im manuellen Modus und koennen nun selber unter
        Betaetigung der linken Maustaste Vierecke zeichnen");
  } else {
    println("Sie befinden sich im Simulationsmodus");
    println("Es werden insgesamt " + n + " Vierecke gezeichnet");
    println("Jede " + nf(timeX, 0, 4) + " Sekunden wird ein neuer Keim
        generiert");
    println("Sie haben das Muster " + selection + " gewaehlt");
  }
  println("Fuer einen Screenshot muessen Sie auf die Simulation klicken,
      anschliessend 's' druecken und zuletzt den entsprechenden Ordner
      auswaehlen, in welchem der Screenshot gespeichert werden soll (am besten
      derselbe Ordner wie diese Datei).");
  println("Fuer eine Unterbrechung bzw. Fortsetzung der Simulation sollten Sie
      wiederum auf die Simulation klicken und anschliessend 'p' druecken");

  //neue ArrayList erstellen
  irrSquares = new ArrayList<irrSquare>();

  //zu dieser ArrayList ein neues Viereck mit den passenden Parametern
      hinzufuegen
  irrSquares.add(new irrSquare(x1, y1, x2, y2, x3, y3, x4, y4, cl, vLeft, vUp,
      vRight, vDown));
}

//Hauptmethode, in welcher Polygone der Array-List hinzugefuegt werden und auf
    Kollisionen ueberprueft wird
void draw(){

  irrSquare newIS = newIrrSquare();
    if(!manualMode){
      //wenn nicht mehr als n Elemente enthalten sind, ein neues unregelmaessiges
          Viereck newIS gezeichnet wurde sowie Erscheinungszeit zutrifft, dann
          soll neues Viereck der ArrayList hinbzugefuegt werden
      //Simulationsmodus
      if (newIS!=null && irrSquares.size() < n+1 && round(frameCount % time) ==
          0){
         irrSquares.add(newIS);
```

```
    }
  } else {
   //wenn nicht mehr als n Elemente enthalten sind, ein neues unregelmaessiges
      Viereck newIS gezeichnet wurde sowie Maus gedrueckt wurde, dann soll
      neues Viereck der ArrayList hinbzugefuegt werden
   //manueller Modus
   if (newIS!=null && irrSquares.size() < n+1 && mousePressed){
      irrSquares.add(newIS);
    }
  }

  //for-Schleife, die fuer jedes Viereck ausgefuehrt wird
  for (irrSquare p : irrSquares){
   //solange sich mindestens eine Ecke des Viereck noch ausbreitet:
   if (p.growingLeft || p.growingRight || p.growingUp || p.growingDown){

     //wenn die linke Ecke eine Begrenzung trifft, stoppe das Wachstum nach
        links
     if (p.edges() == 0){
      p.growingLeft = false;
     }
     //wenn die obere Ecke eine Begrenzung trifft, stoppe das Wachstum nach oben
     if (p.edges() == 1){
      p.growingUp = false;
     }
     //wenn die rechte Ecke eine Begrenzung trifft, stoppe das Wachstum nach
        rechts
     if (p.edges() == 2){
      p.growingRight = false;
     }

     //wenn die untere Ecke eine Begrenzung trifft, stoppe das Wachstum nach
        unten
     if (p.edges() == 3){
      p.growingDown = false;
     }

     //fuer die Variable "other" in der ArrayList
     for (irrSquare other : irrSquares){
      //wenn "other" ein anderes Viereck als "p" beschreibt
      if (p != other){
         //Puffer implementieren, um Ausgabe visuell huebscher zu machen
         float buffer = 0.1;
         //UEberpruefung auf Kollision
         //Linke Ecke eines Vierecks beruehrt die rechte obere Seite eines
            anderen Vierecks
         float dL1 = dist(p.x1, p.y1, other.x3, other.y3);
         float dL2 = dist(p.x1, p.y1, other.x2, other.y2);
         float dLlineLen1 = dist(other.x2, other.y2, other.x3, other.y3);
         if (dL1 + dL2 >= dLlineLen1 - buffer && dL1 + dL2 <= dLlineLen1 +
            buffer){
            p.growingLeft = false;
            other.growingUp = false;
            other.growingRight = false;
         }
```

```
//Linke Ecke eines Vierecks beruehrt die rechte untere Seite eines
    anderen Vierecks
float dL3 = dist(p.x1, p.y1, other.x3, other.y3);
float dL4 = dist(p.x1, p.y1, other.x4, other.y4);
float dLlineLen2 = dist(other.x3, other.y3, other.x4, other.y4);
if (dL3 + dL4 >= dLlineLen2 - buffer && dL3 + dL4 <= dLlineLen2 +
    buffer){
   p.growingLeft = false;
   other.growingDown = false;
   other.growingRight = false;
  }
//Obere Ecke eines Vierecks beruehrt die rechte untere Seite eines
    anderen Vierecks
float dT1 = dist(p.x2, p.y2, other.x3, other.y3);
float dT2 = dist(p.x2, p.y2, other.x4, other.y4);
float dTlineLen1 = dist(other.x3, other.y3, other.x4, other.y4);
if (dT1 + dT2 >= dTlineLen1 - buffer && dT1 + dT2 <= dTlineLen1 +
    buffer){
   p.growingUp = false;
   other.growingDown = false;
   other.growingRight = false;
}
//Obere Ecke eines Vierecks beruehrt die linke untere Seite eines
    anderen Vierecks
float dT3 = dist(p.x2, p.y2, other.x1, other.y1);
float dT4 = dist(p.x2, p.y2, other.x4, other.y4);
float dTlineLen2 = dist(other.x1, other.y1, other.x4, other.y4);
if (dT3 + dT4 >= dTlineLen2 - buffer && dT3 + dT4 <= dTlineLen2 +
    buffer){
   p.growingUp = false;
   other.growingDown = false;
   other.growingLeft = false;
}
//Rechte Ecke eines Vierecks beruehrt die linke obere Seite eines
    anderen Vierecks
float dR1 = dist(p.x3, p.y3, other.x1, other.y1);
float dR2 = dist(p.x3, p.y3, other.x2, other.y2);
float dRlineLen1 = dist(other.x1, other.y1, other.x2, other.y2);
if (dR1 + dR2 >= dRlineLen1 - buffer && dR1 + dR2 <= dRlineLen1 +
    buffer){
   p.growingRight = false;
   other.growingUp = false;
   other.growingLeft = false;
}
//Rechte Ecke eines Vierecks beruehrt die linke untere Seite eines
    anderen Vierecks
float dR3 = dist(p.x3, p.y3, other.x1, other.y1);
float dR4 = dist(p.x3, p.y3, other.x4, other.y4);
float dRlineLen2 = dist(other.x1, other.y1, other.x4, other.y4);
if (dR3 +dR4 >= dRlineLen2 - buffer && dR3 + dR4 <= dRlineLen2 +
    buffer){
   p.growingRight = false;
   other.growingDown = false;
   other.growingLeft = false;
}
```

```
            //Untere Ecke eines Vierecks beruehrt die rechte obere Seite eines
               anderen Vierecks
            float dB1 = dist(p.x4, p.y4, other.x2, other.y2);
            float dB2 = dist(p.x4, p.y4, other.x3, other.y3);
            float dBlineLen1 = dist(other.x2, other.y2, other.x3, other.y3);
            if (dB1 +dB2 >= dBlineLen1 - buffer && dB1 + dB2 <= dBlineLen1 +
               buffer){
               p.growingDown = false;
               other.growingRight = false;
               other.growingUp = false;
            }
            //Untere Ecke eines Vierecks beruehrt die linke obere Seite eines
               anderen Vierecks
            float dB3 = dist(p.x4, p.y4, other.x1, other.y1);
            float dB4 = dist(p.x4, p.y4, other.x2, other.y2);
            float dBlineLen2 = dist(other.x1, other.y1, other.x2, other.y2);
            if (dB3 +dB4 >= dBlineLen2 - buffer && dB3 + dB4 <= dBlineLen2 +
               buffer){
               p.growingDown = false;
               other.growingLeft = false;
               other.growingUp = false;
            }

          }
        }
      }

      //fuehre durchgaengig die Methoden fuer das Zeichnen und fuer das Wachstum
         aus
      p.show();
      p.grow();

    }
  }

//wird aufgerufen, wenn Maustaste geklickt wird
void mouseClicked(){
  varMouseClicked = true;
}

//wird aufgerufen, wenn Maustaste gehalten wird
void mousePressed(){
 varMouseClicked = true;
}

irrSquare newIrrSquare(){

  //zufaellige Koordinaten des Ursprungs, wenn der Simulationsmodus aktiviert
     ist
  if(!manualMode){
    x1 = random(width);
    y1 = random(height);
    x2 = x1;
    x3 = x2;
    x4 = x3;
```

```
    y2 = y1;
    y3 = y2;
    y4 = y3;
}

//Zeichnen der Vierecke im manuellen Modus
if(varMouseClicked && mouseButton == LEFT){
  x1 = mouseX;
  y1 = mouseY;
  x2 = x1;
  x3 = x2;
  x4 = x3;
  y2 = y1;
  y3 = y2;
  y4 = y3;
}

//initialisieren und deklarieren der Farbe
int cl = 100;

vLeft = random(vmin, vmax);
vUp = random(vmin, vmax);
vRight = random(vmin, vmax);
vDown = random(vmin, vmax);


//nachfolgend werden 4 verschiedene Simulationsmuster beschrieben
//#1 Grautoene wie auf Muster in Aufgabe
if (selection == 1){
  if (x1 < 250 && y1 > 200 ){
       cl = int(random(110, 130));
   }
   if (x1 < 150 && (y1 > 100 && y1 < 250)){
      cl = int(random(125, 175));
   }
   if (x1 < 170 && y1 < 170 ){
       cl = int(random(180, 200));
   }
   if ((x1 < 300 && x1 > 75) && y1 < 175){
       cl = int(random(155, 175));
   }
   if ((x1 < 600 && x1 > 250) && y1 < 130){
       cl = int(random(180, 200));
   }
   if (x1 > 550 && y1 < 400){
      cl = int(random(125, 175));
   }
   if ((x1 < 650 && x1 > 200) && (y1 > 75 && y1 < 330)){
      cl = int(random(125, 175));
   }
   if (x1 > 300 && x1 < 650 && y1 > 300 && y1 < 400){
    cl = int(random(180, 200));
   }
   if (x1 > 500 && x1 < 650 && y1 > 230 && y1 < 300){
    cl = int(random(180, 200));
```

```
    }
    if (x1 > 650 && x1 < 700 && y1 > 400 && y1 < 450){
      cl = int(random(180, 200));
    }
    if (x1 > 210 && x1 < 260 && y1 > 240 && y1 < 290){
      cl = int(random(180, 200));
    }
    if (x1 > 370 && x1 < 600 && y1 > 370 && y1 < height){
      cl = int(random(130, 150));
    }
  }

  //#2 zufaellige Verteilung
  if (selection == 2){
      cl = int(random(100, 200));
  }

  //#3 senkrechte Vierteilung
  if(selection == 3){
    if(x1 < width/4){
      cl = int(random(100, 125));
    }
    else if(x1 < width/2 && x1 >= width/4){
      cl = int(random(125, 150));
    }
    else if(x1 < 3*width/4 && x1 >= width/2){
      cl = int(random(150, 175));
    } else {
      cl = int(random(175, 200));
    }
  }

  //#4 Aufteilung in Quadranten
  if(selection == 4){
    if(x1 < width/2 && y1 < height/2){
      cl = int(random(100, 125));
    }
    else if(x1 > width/2 && y1 < height/2){
       cl = int(random(125, 150));
    }
    else if(x1 < width/2 && y1 > height/2){
       cl = int(random(150, 175));
    } else {
       cl = int(random(175, 200));
    }
  }


  //boolean, die ausdrueckt, ob das Viereck gezeichnet werden darf oder nicht
  boolean valid = true;

  for (irrSquare p : irrSquares){
    //4 Vektoren werden erstellt
     vektoren[0] = new PVector(p.x1,p.y1);
     vektoren[1] = new PVector(p.x2,p.y2);
```

```
     vektoren[2] = new PVector(p.x3,p.y3);
     vektoren[3] = new PVector(p.x4,p.y4);

     boolean hit = SquarePointCollision(vektoren, x1,y1);
    //wenn das neue Viereck in einem schon bestehenden Viereck neu erscheinen
       will, wird dies nicht zugelassen
    if (hit){
      valid = false;
      break;
    }
  }

  //wenn das neue Viereck ausserhalb erscheint, wird das Zeichnen genehmigt,
     sonst nicht
  if (valid){
     return new irrSquare(x1, y1, x2, y2, x3, y3, x4, y4, cl, vLeft, vUp,
        vRight, vDown);
  } else {
    return null;
  }


}

//hier wird ueberprueft, ob ein Viereck in einem anderen erscheint
boolean SquarePointCollision(PVector[] vektoren, float x1, float y1) {
  int next = 0;

  boolean collision = false;

  //jeder Vektor wird geprueft
  for (int current=0; current<vektoren.length; current++) {

    next = current+1;
    if (next == vektoren.length) next = 0;

    PVector vc = vektoren[current]; // c steht fuer "current"
    PVector vn = vektoren[next]; // n steht fuer "next"

    //pruefe, wie Punkt zu Vektoren liegt
    if (((vc.y >= y1 && vn.y < y1) || (vc.y < y1 && vn.y >= y1)) &&
        (x1 < (vn.x-vc.x)*(y1-vc.y) / (vn.y-vc.y)+vc.x)) {
          collision = !collision;
    }
  }
  return collision;
}

//Bei Betaetigung des Buchstabens 's' wird Screenshot erstellt und bei
   Betaetigung des Buchstabens 'p' wird die Simulation unterbrochen bzw. weiter
   fortgefuehrt
void keyPressed(){
   if(key=='s'||key=='S'){
        selectFolder("Select a folder to process: ", "folderSelected");
   }
```

```
    if (key == 'p' || key=='P') {
      paused = !paused;
    if (paused) {
       noLoop();
    } else {
       loop();
    }
  }


}

//OEffnen des Ordners, in welchem der Screenshot abgelegt werden soll
void folderSelected(File selection){
  if (selection == null){
    return;
  } else {
    String dir2 = selection.getPath() + "\\";
    save(dir2 + "screenshot"+num+".jpg");
    num++;

  }

}


class irrSquare{

  //Vier Eckpunkte des Polygons
  float x1;
  float y1;
  float x2;
  float y2;
  float x3;
  float y3;
  float x4;
  float y4;

  //Geschwindigkeiten nach links (vLeft), nach oben (vUp), nach rechts (vRight)
     sowie nach unten (vDown)
  float vLeft;
  float vUp;
  float vRight;
  float vDown;

  //Farbe der Vierecke
  color cl;

  //Booleans als Indikatoren fuer das Wachstum in die vier Richtungen
  boolean growingLeft = true;
  boolean growingRight = true;
  boolean growingUp = true;
  boolean growingDown = true;
```

```
//Konstruktor um jeweiligen Objekte zu erstellen
irrSquare(float x1, float y1, float x2, float y2, float x3, float y3, float
    x4, float y4, color cl, float vx1, float vy2, float vx3, float vy4){
  this.x1 = x1;
  this.y1 = y1;
  this.x2 = x2;
  this.y2 = y2;
  this.x3 = x3;
  this.y3 = y3;
  this.x4 = x4;
  this.y4 = y4;
  this.vLeft = vx1;
  this.vUp = vy2;
  this.vRight = vx3;
  this.vDown = vy4;
  this.cl = cl;

}

//Methode, die die jeweiligen Punkte mit der entsprechenden Geschwindigkeit
    bewegt
void grow(){
  if (growingLeft){
       x1 -= vLeft;
  }
  if (growingUp){
       y2-=vUp;
  }
  if (growingRight){
       x3+=vRight;
  }
  if (growingDown){
       y4+=vDown;
  }
}

//Methode, die fuer das kollidieren mit den Fenster-Begrenzungen
    verantwortlich ist
int edges(){
  if (x1 < 0){
     return 0;
   } else if (y2 < 0){
     return 1;
   } else if (x3 > width){
     return 2;
   } else if (y4 > height){
     return 3;
   } else {
     return 4;
   }
}

//Methode, die die Vierecke auf das Canvas zeichnet
void show(){
  fill(cl);
```

```
    quad(x1, y1, x2, y2, x3, y3, x4, y4);
  }
}
```

1.5 Quellenverzeichnis

Literatur

[1] Processing, second edition: A Programming Handbook for Visual Designers and Artists, The MIT Press, 2014

[2] Learning Processing: A Beginner's Guide to Programming Images, Animation, and Interaction (The Morgan Kaufmann Series in Computer Graphics) 2. Auflage, Morgan Kaufmann, 2015

[3] Getting Started with Processing: A Hands-On Introduction to Making Interactive Graphics 2. Auflage, Make Community, LLC, 2015