Vorgelegt an der PHBern
Institut Sekundarstufe SI

Schwarmsimulation mit JGameGrid

Spezielle Arbeit zur Veranstaltung „Entwicklungsprojekte“
bei Frau Jarka Arnold

von
Martin Schellenberg
Archivstrasse 1
3005 Bern
Matrikelnummer: 92-210-798

Bern
26. Oktober 2011

mit LaTeX gesetzt

Inhaltsverzeichnis

1 Abstract **4**

2 Das Spielfeld **4**

3 Schwarmverhalten **7**
3.1 Separation 7
3.2 Alignment 8
3.3 Cohesion 8
3.4 Ausseneinflüsse 8

4 Aspekte der Umsetzung in Java **9**
4.1 Variablen zur Spielsteuerung 9
4.2 Verhaltenssteuerung der Klasse Bird 9
4.3 Verhaltenssteuerung der Klasse Raptor 10
4.4 Verhalten im Detail 10
4.4.1 Bewegungsgleichungen 10
4.4.2 Separation 11
4.4.3 Alignment 11
4.4.4 Cohesion 12
4.4.5 Aggression 12
4.4.6 Escape 12
4.4.7 getAcceleration 12
4.5 Programmiertechnische Unschönheiten 13

5 Beobachtungen - interessante Simulationen **13**
5.1 Simulationen ohne Raubvögel 14
5.1.1 Schwarm 1 14
5.1.2 Schwarm 2 14
5.1.3 Schwarm 3 14
5.1.4 Schwarm 4 14
5.1.5 Schwarm 5 15
5.1.6 Schwarm 6 15
5.2 Simulationen mit Raubvögeln 15
5.2.1 Die Jäger formieren sich 15

6 Listing 15

Literatur 32

1 Abstract

Schwarmverhalten ist ein in der Natur häufig auftretendes Phänomen: einzelne Individuen einer Population verhalten sich so, als wären sie Teil eines grösseren Ganzen. Für die Individuen selbst entstehen verschiedene Vorteile: bei der Nahrungssuche, bei der Flucht vor Fressfeinden durch kollektive Wachsamkeit, beim Energieverbrauch der Bewegung etc. In dieser Arbeit wird beschrieben, wie eine Schwarmsimulation mit Hilfe des von Aegidius Plüss erstellten Java-Package „JGameGrid“ (in [Plu01]) erstellt werden kann.

2 Das Spielfeld

Das Spielfeld der Breite b und der Höhe h besteht aus $b \cdot h$ Zellen, welche jeweils ein Individuum (abgeleitet aus der Klasse Actor) enthalten kann. Hier formt sich ein erstes prinzipielles Problem: Schwärme halten sich selten immer am gleich Ort auf (mit Ausnahme gewisser Fischschwärme, die sich vor Fressfeinden schützen); die Individuen würden mit hoher Wahrscheinlichkeit das Spielfeld sehr schnell verlassen. Natürlich könnte man eine avoidance-Strategie für die einzelnen Akteure einführen, welche sie vor einer Randberührung schützt. Nur wäre dies ein relativ unnatürliches setting, vergleichbar mit einem viel zu kleinen Aquarium. Idealerweise sollte das Spielfeld unbegrenzt, aber zugleich vom Benutzer überall einsehbar sein.
Dies lässt sich durch einen topologischen Trick erreichen:
Verlässt ein Akteur das Spielfeld gegen oben $(y < 0)$ so betritt er es wieder von unten: $y = y + h$. Für das Verlassen des Spielfeldes gegen unten, links, rechts gilt analog:

- links: if $x < 0$ then $x = x + b$
- rechts: if $x > b - 1$ then $x = x - b$
- oben: if $y < 0$ then $y = y + h$
- unten: if $x > h - 1$ then $y = y - h$

Topologisch wird der linke und der rechte Rand sowie der obere und der untere Rand

zusammengeklebt: man betrachte die Kleberelation ρ auf der Spielfläche X:

$$\begin{aligned} X &:= [0, h-1] \times [0, b-1] \\ \rho \subseteq X^2 : (x_1, x_2)\rho(x_3, x_4) &:\Longleftrightarrow (x_1, 0)\rho(x_1, h-1) \text{ oder } (0, x_2)\rho(b-1, x_2) \\ Y &:= X/\rho \end{aligned}$$

$f : X \to \partial Y$ ist ein Homöomorphismus: das (nicht diskretisierte) Spielfeld ist homöomorph zur Oberfläche des Torus Y (Abbildung 1). Für die Simulation ist vor allem wichtig, dass die Torus-Oberfläche eine orientierte und geschlossene Fläche ist.

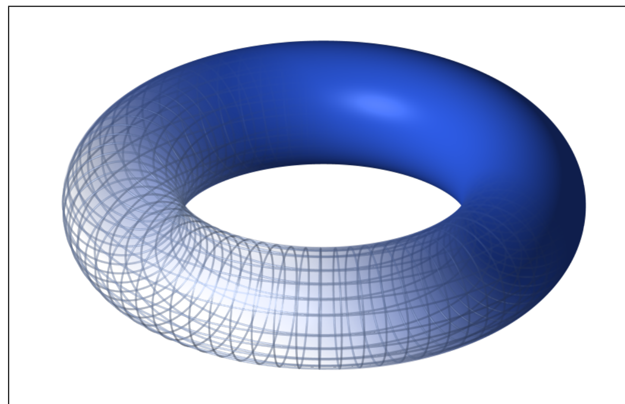

Abb. 1: *Torus*

Da für die Simulation die Entfernungen zwischen den Akteuren eine Rolle spielt, ergibt sich hier ein Problem: wie Abbildung 2 zeigt, liefert die direkte Verbindungslinie zwischen zwei Punkten $P(x_1, x_2)$ und $Q(x_3, x_4)$ nicht immer die kürzeste Entfernung! Soll sich ein Akteur auf einen anderen Akteur zubewegen, muss zuerst getestet werden, in welcher Richtung die kürzeste Entfernung liegt. Dafür müssen die Entfernungen von $P(x_1, x_2)$ zu folgenden Punkten bestimmt werden: $(x_3, x_4), (x_3 + b, x_4), (x_3 - b, x_4), (x_3, x_4 + h), (x_3, x_4 - h), (x_3 - b, x_4 - h), (x_3 - b, x_4 + h), (x_3 + b, x_4 + h), (x_3 + b, x_4 - h)$. Weiter kann man der Abbildung 2 entnehmen, dass die insgesamt 9 zu berechnenden Entfernungen 9 verschiedenen möglichen Richtungen von P nach Q entsprechen. Es ist klar, dass diese Berechnungen jeweils rechenintensiv sind, und daher nicht von allen Akteuren immer wieder von Neuem durchgeführt werden sollten.

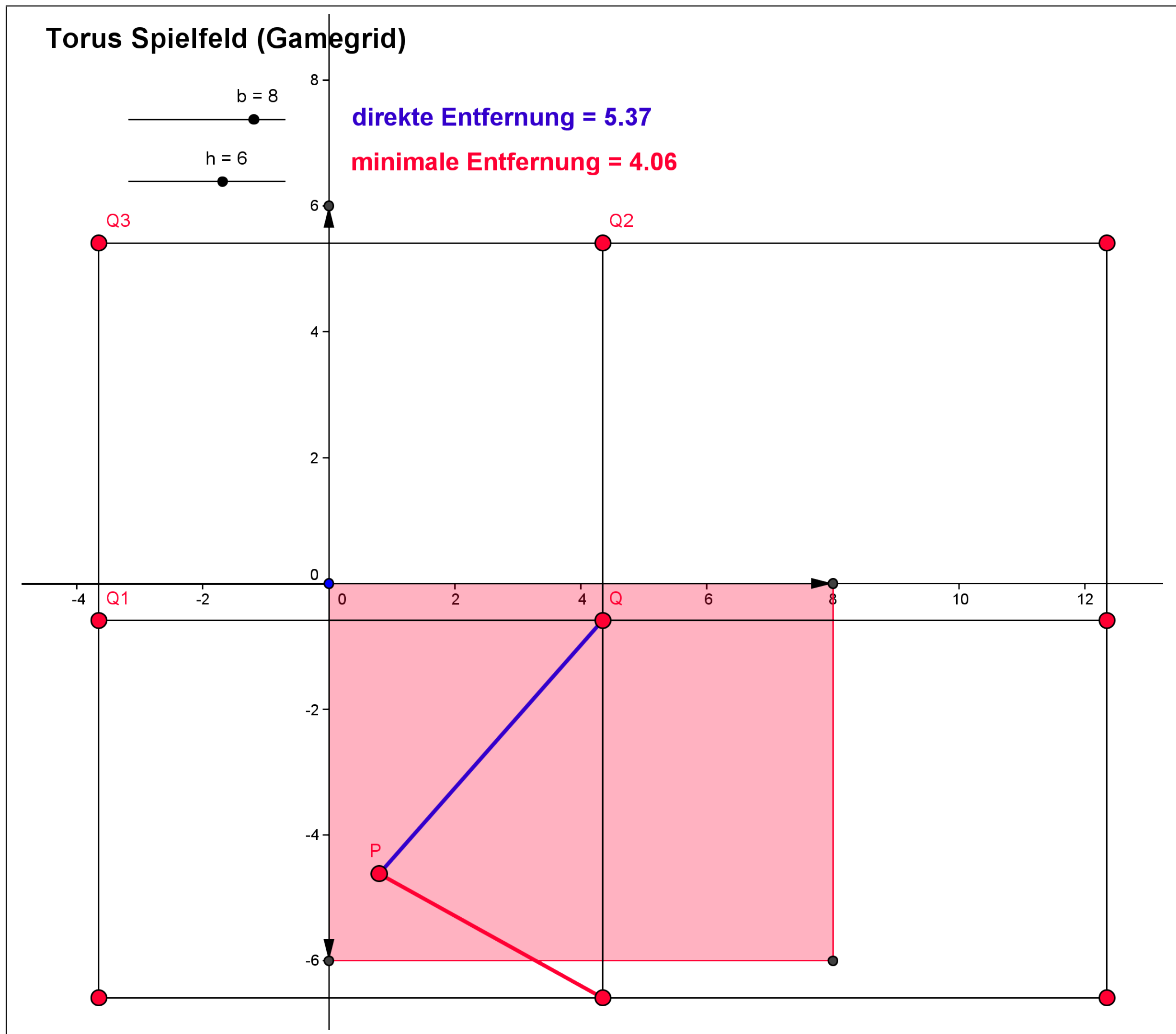

Abb. 2: *Entfernung zweier Punkte*

3 Schwarmverhalten

Der Informatiker und Animationsspezialist Craig W. Reynolds hat bereits 1986 Modelle entwickelt um Schwarmverhalten zu simulieren (siehe [Rey01]). Reynolds vertritt die Auffassung, dass Schwarmverhalten aus zum Teil widersprüchlichen Anforderungen entsteht: einerseits möchte der Akteur seinen Nachbarn nahe sein, andererseits möchte er natürlich Kollisionen vermeiden. Reynolds leitete aus diesen Anforderungen drei prinzipielle Regeln ab:

- Collision Avoidance (separation, Kollisionsvermeidung)
- Velocity Matching (alignment, Flugverhalten angleichen)
- Flock Centering (cohesion, Zusammenhalten)

Da diese Regeln sich zum Teil widersprechen, gibt es zwei Möglichkeiten dies algorithmisch umzusetzen: entweder man entwickelt eine Entscheidungsfunktion f(Parameter Akteur, Nachbarn) welche die für den momentanen Zeitpunkt für einen bestimmten Akteur wichtigste Regel auswählt, oder man arbeitet mit einer Gewichtungsfunktion g(Parameter Akteur, Nachbarn), welche aus allen drei Regeln einen gewichteten Mittelwert für die Beschleunigung errechnet. Reynolds hat mir der zweiten Methode bessere Erfahrungen gemacht. 1995 konkretisierte Reynolds die drei Regeln indem er jeweils ein konkretes Lenkungsverhalten (steering behavior) beschrieb. Er benannte sie auch um: Separation anstatt Collision Avoidance, Alignment anstatt Velocity Matching und Cohesion anstatt Flock centering.
Hier eine kurze Beschreibung der drei Lenkungsverhalten:

3.1 Separation

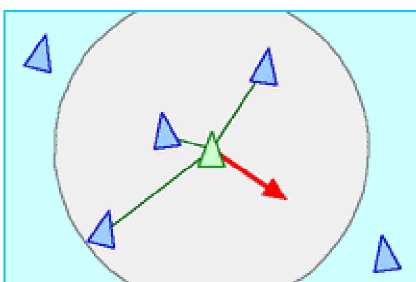

Abb. 3: *Separation*

Das separation-steering entsteht durch den Drang einen gewissen Abstand zu den Nachbarn zu halten, um mögliche Kollisionen zu vermeiden. Die Abbildung 3 zeigt, dass der Beschleunigungsvektor des Akteurs entgegengesetzt zum Vektor der Summe der Positionsvektoren der Nachbarn im Bezugssystem des Akteurs steht. Der Betrag des Beschleunigungsvektors wird durch die Gewichtungsfunktion g bestimmt.

3.2 Alignment

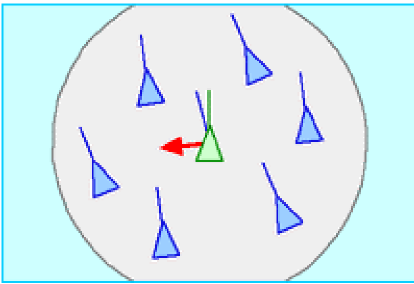

Abb. 4: *Alignment*

Das alignment-steering hat zum Ziel, die Geschwindigkeit und Richtung an die Bewegung der Nachbarn anzugleichen. Die Abbildung 4 zeigt, dass die Richtung des Beschleunigungsvektor des Akteurs aus der Differenz des Geschwindigkeitsvektors des Akteurs und des Mittelwert-Vektors der Geschwindigkeitsvektoren der Nachbarn gebildet wird. Der Betrag des Beschleunigungsvektors wird durch die Gewichtungsfunktion g festgelegt.

3.3 Cohesion

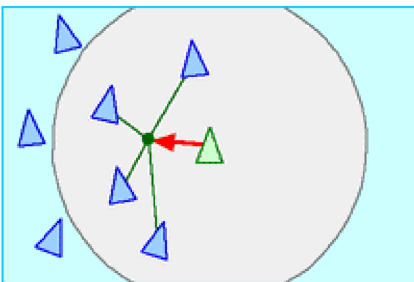

Abb. 5: *Cohesion*

Das cohesion-steering hat zum Ziel, den Schwarm zusammenzuhalten und lässt den Akteur möglichst nahe beim Nachbarn bleiben: der Akteur strebt zu seinen Nachbarn hin. Die Abbildung 5 zeigt, dass die Richtung des Beschleunigungsvektor des Akteurs in Richtung des gemeinsamen Schwerpunkts der Punktmassen der Nachbarn zeigt. Als Näherung können aber auch einfach die Positionsvektoren der Nachbarn gemittelt werden. Der Betrag des Beschleunigungsvektors wird auch hier durch die Gewichtungsfunktion g festgelegt.

3.4 Ausseneinflüsse

Bis anhin haben wir überhaupt keine Ausseneinflüsse: durch die Wahl der Torus-Oberfläche als Simulationsfläche gibt es keine Begrenzung! Es ist zu erwarten, dass die Akteure sich mit der Zeit zu einem stabilen Schwarm formen, welcher sich dann mit konstanter Geschwindigkeit in irgendeine Richtung bewegt, welche durch die Anfangspositionen und Anfangsgeschwindigkeiten der Akteure bestimmt ist. Dies hört sich nicht sehr spannend an! Als Störquelle könnte eine Klasse „Raubvogel“ dienen, deren Instanzen auf den nächsten Vogel zuflögen - die Vögel würden natürlich die Raubvögel meiden.

4 Aspekte der Umsetzung in Java

Die beiden Hauptklassen sind Bird (abgeleitet von Actor) und Raptor (abgeleitet von Actor). Die Instanzen von Bird und Raptor werden im Konstruktor der Klasse SimFlock ("Simulation of a flock of birds", abgeleitet von GameGrid") erzeugt. Beide Klassen funktionieren auf ganz ähnliche Weise wie die Klasse body aus newton.java. Der Hauptunterschied liegt darin, dass die Beträge der Geschwindigkeitsvektoren jeweils konstant sind: die Aktoren führen ausschliesslich Richtungsänderungen durch. Das Verhalten von Vogel/Raubvogel wird jeweils durch die Methode „GGVector setAcceleration()" bestimmt. Mit einer Vielzahl von Variablen, welche alle im Kopf der Klasse SimFlock stehen, lässt sich dieses Verhalten beeinflussen.

4.1 Variablen zur Spielsteuerung

- nh, nw : Höhe (Height) und Breite (Width) des Spielfeldes
- nb, nr : Anzahl Vögel (birds) und Anzahl Raubvögel (raptors)
  Bis zu etwa 300 birds bleibt die Animation auf meinem T400 (Dualcore 2.6 GHz) mit Windows 7 Professional flüssig.
- rseed: Eine Seed-Variable für den Zufallsgenerator, um die Animationen reproduzierbar zu machen.
- drawTrace: Die Flugbahn von den Aktoren zeichen: ein- oder ausschalten.
- timeFactor: Spiel beschleunigen: timeFactor ist das Δt in den Bewegungsgleichungen der Methode act().

4.2 Verhaltenssteuerung der Klasse Bird

- vbird : Betrag des Geschwindigkeitsvektors von Bird (zB. vbird = 1)
- rob : Radius of observation: Der Vogel erkennt Akteure nur, wenn sie nicht weiter als rob von ihm entfernt sind (zB. rob = 50)
- rcrit : Critical Radius: Nachbarvögel, welche näher als rcrit sind, sind dem Vogel zu nahe und er möchte ihnen ausweichen (zB. rcrit = 20)

- cohesionFactor : Mass für die Stärke des Wunsches nach Nähe zu anderen Vögeln innerhalb rob und ausserhalb rcrit (zB. cohesionFactor = 0.001)
- separationFactor : Mass für die Stärke des Wunsches nach grösserer Distanz zu anderen Vögeln innerhalb rcrit (zB. separationFactor = 0.05)
- alignmentFactor : Mass für die Stärke des Wunsches nach gleicher Flugrichtung im Bezug auf andere Vögel innerhalb rob (zB. alignmentFactor = 0.05)
- escapeFactor : Mass für die Stärke des Wunsches nach grösserer Distanz im Bezug auf andere Raubvögel innerhalb rob (zB. escapeFactor = 0.001). Befindet sich ein Raubvogel innerhalb rob, so möchte der Vogel ausschliesslich flüchten: die anderen Wünsche haben keine Bedeutung mehr.

4.3 Verhaltenssteuerung der Klasse Raptor

- vraptor : Betrag des Geschwindigkeitsvektors von Raptor (zB. vraptor = 1.1)
- ror : Radius of observation: Der Raubvogel erkennt Akteure nur, wenn sie nicht weiter als ror von ihm entfernt sind (zB. ror = rom = diagonale des Spielfeldes)
- aggressionFactor : Mass für die Stärke des Wunsches nach geringerer Distanz im Bezug auf den nächsten Vogel innerhalb ror (zB. aggressionFactor = 0.001). Der Raubvogel stürzt sich auf den nächsten Vogel innerhalb seiner Sichtweite ror.

4.4 Verhalten im Detail

4.4.1 Bewegungsgleichungen

Die Gleichungen wurden der Methode act() der Klasse body von Newton.java entnommen und angepasst.

$$
\begin{aligned}
\vec{a} &= \text{getAcceleration()} \\
\vec{v} &= \vec{v} + \vec{a} \cdot \Delta t \\
\vec{v} &= \frac{\vec{v}}{|\vec{v}|} \cdot \overrightarrow{v_{actor}} \quad \text{mit } \overrightarrow{v_{actor}} = \text{ konstant} \\
\vec{s} &= \vec{s} + \vec{v} \cdot \Delta t \\
\vec{s} &= \text{toTorusPosition}(\vec{s}) \\
\overrightarrow{location} &= \text{toLocation}(\vec{s})
\end{aligned}
$$

4.4.2 Separation

So wird die Beschleunigung $accSeparation$ durch Separation für den Vogel i berechnet:

$$
\begin{aligned}
\vec{r_j} &= \text{getPositionVecDiffOnTorusBird(Bird } j\text{) für } i \neq j \\
\vec{w} &= \sum_{|\vec{r_j}| \leq r_{crit}} r_j \\
\vec{w} &= \vec{w} \cdot (\text{Anzahl summierte } r_j)^{-1} \\
\overrightarrow{accSeparation} &= \vec{w} \cdot separationFactor
\end{aligned}
$$

4.4.3 Alignment

So wird die Beschleunigung $accAlignment$ durch Alignment für den Vogel i berechnet:

$$
\begin{aligned}
\vec{r_j} &= \text{getPositionVecDiffOnTorusBird(Bird } j\text{) für } i \neq j \\
\vec{v_j} &= \text{velocity(Bird } j\text{) für } i \neq j \\
\vec{w} &= \sum_{|\vec{r_j}| \leq r_{rob}} v_j \\
\vec{w} &= \vec{w} \cdot (\text{Anzahl summierte } v_j)^{-1} \\
\vec{w} &= \vec{v_i} - \vec{w} \\
\overrightarrow{accAlignment} &= -\vec{w} \cdot alignmentFactor
\end{aligned}
$$

4.4.4 Cohesion

So wird die Beschleunigung $accCohesion$ durch Cohesion für den Vogel i berechnet:

$$\begin{aligned}
\vec{r_j} &= \text{getPositionVecDiffOnTorusBird(Bird } j\text{) für } i \neq j \\
\vec{w} &= \sum_{r_{crit} < |\vec{r_j}| \leq r_{rob}} r_j \\
\vec{w} &= \vec{w} \cdot (\text{Anzahl summierte } r_j)^{-1} \\
\overrightarrow{accCohesion} &= -\vec{w} \cdot cohesionFactor
\end{aligned}$$

4.4.5 Aggression

Aggression ist ein Verhalten der Raubvögel (Klasse Raptor) und wird für den Raubvogel i so berechnet:

$$\begin{aligned}
\vec{r_j} &= \text{getPositionVecDiffOnTorusBird(Bird } j\text{)} \\
\vec{w} &= \min_{|\vec{r_j}| \leq r_{ror}} r_j \\
\overrightarrow{accAggression} &= -\vec{w} \cdot aggressionFactor
\end{aligned}$$

4.4.6 Escape

Escape ist ein Verhalten der Vögel (Klasse bird), die vor einem Raubvogel fliehen und wird für den Vogel i so berechnet:

$$\begin{aligned}
\vec{r_j} &= \text{getPositionVecDiffOnTorusBird(Raptor } j\text{)} \\
\vec{w} &= \sum_{|\vec{r_j}| \leq r_{rob}} r_j \\
\vec{w} &= \vec{w} \cdot (\text{Anzahl summierte } r_j)^{-1} \\
\overrightarrow{accEscape} &= \vec{w} \cdot escapeFactor
\end{aligned}$$

4.4.7 getAcceleration

Für die Raubvögel ist Berechnung der Beschleunigung $acceleration$ sehr einfach:

$$\overrightarrow{acceleration} = \overrightarrow{accAggression}$$

Für die Vögel macht man für die Berechnung der Beschleunigung $acceleration$ eine Fallunterscheidung:

$$\overrightarrow{acceleration} = \begin{cases} \overrightarrow{accEscape}, & \text{falls Raptor in Reichweite} \\ \overrightarrow{accCohesion} + \overrightarrow{accSeparation} + \overrightarrow{accAlignment}, & \text{sonst} \end{cases}$$

4.5 Programmiertechnische Unschönheiten

Die Klassen Raptor und Bird unterscheiden sich nur in wenigen, kleinen Details: z.B. die Methode „setAcceleration", welche das Verhalten von Vögeln, bzw. Raubvögeln bestimmt. Beim Versuch Raptor von Bird abzuleiten, scheiterte ich kläglich: das Problem ist der Aufruf des Konstruktors der Superklasse mit „super(true,"sprites/bird.gif")" im Konstruktor der Klassen. „super.super" existiert in JAVA aus guten Gründen nicht. Man müsste wohl zuerst eine Zwischenklasse „AllBirds" einführen.
In der Klasse Bird existieren die Methoden „getPositionVecDiffOnTorusBird(Bird bird)" und „getPositionVecDiffOnTorusRaptor(Raptor raptor)", welche den nächsten Vektor vom Aktor zu einem Nachbar-Vogel/Raubvogel zurückgeben. Diese Methoden sind identisch, ich sehe aber keine Möglichkeit, eine einzige Methode, welche als Argument die Superklasse Actor verwendet zu schreiben: in Actor wird die Position in einer Variablen vom Typ Location mit Integer-Werten gespeichert, während dem die Methoden mit GGVector und double arbeiten. Ein Typecast in jedem Zyklus würde zu einem Genauigkeitsverlust führen.
Solche Anpassungen überlasse ich gerne Personen, die sich mit der Java-Programmierung wirklich auskennen.

5 Beobachtungen - interessante Simulationen

Alle diese Simulationen lassen sich exakt replizieren - zumindest auf einer ähnlichen Hardware, welche mit identischer Implementierung vom Typ double funktioniert. Bedingung dafür ist aber, dass für die Zufallsvariable rand im Konstruktor der Klasse SimFlock der Seed gesetzt wird: „Random rand = new Random(rseed); „ und nicht „Random rand = new Random()".
Alle Simulationen basieren auf folgenden Variablen-Werten:

- Spiel: (nh = 600; nw = 1000; nb=100; nr=0; rseed = 123; timeFactor = 4)
- Bird: (vbird = 1; rob = 50; rcrit = 20)

- Bird: (cohesionFactor = 0.001; separationFactor = 0.05; alignmentFactor = 0.05)
- Bird: (escapeFactor = 0.001)
- Raptor: (vraptor = 1.1; ror = rom; aggressionFactor = 0.001)

Geänderte Variable werden jeweils aufgeführt.

5.1 Simulationen ohne Raubvögel

5.1.1 Schwarm 1

Dies ist auf Dauer ein Schwarm, welcher sich immer wieder in Teilschwärme auflöst. Die Aktoren sind sehr agil und wirken nervös. Änderungen an den Variablen-Werten im Bezug auf den Grundzustand:

- (keine Änderung)

5.1.2 Schwarm 2

Gewusel! Es kommt nur kurzfristig zur Schwarmbildung. Die Schwärme lösen sich sofort wieder auf. Änderungen an den Variablen-Werten im Bezug auf den Grundzustand:

- Bird: (alignmentFactor = 0)

5.1.3 Schwarm 3

Die Aktoren haben eine gemeinsame stabile Richtung, keine echte Schwarmbildung. Änderungen an den Variablen-Werten im Bezug auf den Grundzustand:

- Bird: (cohesionFactor = 0)

5.1.4 Schwarm 4

Die Aktoren sind auf kleinstem Raum zusammengedrängt haben eine gemeinsame stabile Richtung. Änderungen an den Variablen-Werten im Bezug auf den Grundzustand:

- Bird: (separationFactor = 0)

5.1.5 Schwarm 5

Dies ist auf Dauer ein kompakter aber wenig richtungsstabiler Schwarm mit grossen Abständen zwischen den Aktoren. Änderungen an den Variablen-Werten im Bezug auf den Grundzustand:

- Bird: (rob = 100; rcrit = 25)
- Bird: (separationFactor = 0.5; alignmentFactor = 0.3)

5.1.6 Schwarm 6

Dies ist auf Dauer ein kohärenter und relativ richtungsstabiler Schwarm mit grossen Abständen zwischen den Aktoren. Änderungen an den Variablen-Werten im Bezug auf den Grundzustand:

- Bird: (rob = 100; rcrit = 25)
- Bird: (cohesionFactor = 0.0001; separationFactor = 0.5; alignmentFactor = 0.3)

5.2 Simulationen mit Raubvögeln

5.2.1 Die Jäger formieren sich

Eine interessante Beobachtung: obwohl die Raubvögel untereinander unsichtbar sind, wählen sie auf Dauer denselben Vogel als Ziel! Sie haben dann alle die exakt gleiche Position und bewegen sich hinter dem Vogel auf einer Kreisbahn. Auch ein traversierender Schwarm stört ihre Bahn dann nicht mehr!
Änderungen an den Variablen-Werten im Bezug auf den Grundzustand:

- Spiel: (nr=10)

6 Listing

Die Datei „SimFlock.java“ kann auch von der Webseite heruntergeladen werden ([MarS02]).

```java
1  package simflock;
2
3  // Simflock.java by Martin Schellenberg (baal@gmx.net) 2011
4  // Based on the file newton.java from http://www.gamegrid.ch (Jarka Arnold)
5  // Simulation of the behavior of a flock of birds and raptors on the surface of a torus
6  // Based on the JGameGrid-Framework from Aegidius Plüss (http://www.aplu.ch)
7
8  import ch.aplu.jgamegrid.*;
9  import java.awt.Color;
10 import java.util.*;
11 import java.lang.Math;
12
13
14 public class SimFlock extends GameGrid
15 { private static final long serialVersionUID = -1113582265865921787L; // avoids warning in
       Eclipse: The serializable class...
16 // Change game parameters ////////////////////////////////////////////
17   protected static final int nh = 600;                 // Number of cells: Height of playground
18   protected static final int nw = 1000;                // Number of cells: Width of playground;
19   private static final int nb = 100;                   // Number of birds
20   private static final int nr = 0;                     // Number of raptors
21   protected static final double rom = Math.sqrt(nw*nw+nh*nh);   // Maximum Radius of observation
22   private static final int rseed=12355;                // Random Seed
23   protected static final  boolean drawTrace = false;   // Draw traces of birds/raptors
24   protected static final  double  timeFactor = 4;      // Descrete Timestep for calculation of
         acc/vel/pos
```

```java
25  // Change behavior of birds ////////////////////////////////////////
26  protected static final double vbird = 1;            // Magnitude of velocity: birds
27  protected static final double rob = 50;             // Radius of observation: birds
28  protected static final double rcrit = 20;           // Critical Radius: birds
29  protected static final double cohesionFactor = 0.001;
30  protected static final double separationFactor = 0.05;
31  protected static final double alignmentFactor = 0.05;
32  protected static final double escapeFactor = 0.001;
33  // Change behavior of raptors //////////////////////////////////////
34  protected static final double vraptor = 1.1;            // Magnitude of velocity: raptors
35  protected static final double ror = rom;              // Radius of observation: raptors
36  protected static final double aggressionFactor = 0.001;
37  ////////////////////////////////////////////////////////////////////
38  protected final Bird[] birds = new Bird[nb];
39  protected final Raptor[] raptors = new Raptor[nr];
40
41
42  public SimFlock()
43  {
44    super(nw,nh, 1, null, true);
45    setSimulationPeriod(50);
46    Random rand = new Random(rseed);    // use Random(rseed) for testing: change value of static
        final rseed
47
48    for (int i = 0; i < nb; i++)  // Generate birds
49    {
```

```
50        GGVector startVelocity = new GGVector(vbird,0);
51        startVelocity.rotate(rand.nextInt(360));
52        birds[i] = new Bird(startVelocity);
53        addActor(birds[i], new Location(rand.nextInt(nw), rand.nextInt(nh)));
54      }
55      for (int i = 0; i < nr; i++)  // Generate birds
56      {
57        GGVector startVelocity = new GGVector(vraptor,0);
58        startVelocity.rotate(rand.nextInt(360));
59        raptors[i] = new Raptor(startVelocity);
60        addActor(raptors[i], new Location(rand.nextInt(nw), rand.nextInt(nh)));
61      }
62      show();
63    }
64
65    public static void main(String[] args)
66    {
67      new SimFlock();
68    }
69  }
70
71  ////////////////////////////////////////CLASS Bird
        ///////////////////////////////////////////////////////////
72
73  class Bird extends Actor
74  {
```

```
75  private Location oldLocation = new Location(-1, -1);
76  private GGVector startVelocity;
77  public  Location location;
78  public  GGVector position;
79  public  GGVector velocity;
80  private GGVector acceleration;
81  private boolean  borderCrossed;
82
83  public Bird(GGVector startVelocity)
84  {
85    super(true,"sprites/bird.gif"); //NEU MarS
86        this.startVelocity = startVelocity;
87        this.velocity=startVelocity;
88  }
89
90  public void reset()
91  {
92        position = toPosition(getLocationStart());
93    setDirection(Math.toDegrees(startVelocity.getDirection()));
94    oldLocation.x = -1; oldLocation.y = -1;
95  }
96
97  private GGVector toPosition(Location location)
98  {
99    return new GGVector(location.x, location.y);
100 }
```

```
101
102  private Location toLocation(GGVector position)
103  {
104    return new Location((int)(position.x), (int)(position.y));
105  }
106
107  private GGVector toTorusPosition(GGVector position)
108  {
109       double x=position.x; double y=position.y;  borderCrossed=false;
110       if (x<0) {x = x + SimFlock.nw; borderCrossed=true;}
111       if (x>SimFlock.nw-1) {x = x - SimFlock.nw; borderCrossed=true;}
112       if (y<0) {y = y + SimFlock.nh; borderCrossed=true;}
113       if (y>SimFlock.nh-1) {y = y - SimFlock.nh; borderCrossed=true;}
114    return new GGVector(x,y);
115  }
116
117  public GGVector getPosition()
118  {
119    return position.clone();
120  }
121
122  public GGVector getVelocity()
123  {
124    return velocity.clone();
125  }
126
```

```
127
128 private GGVector getPositionVecDiffOnTorusRaptor(Raptor raptor)
129 {   GGVector delta = new GGVector();
130         double mag; double magmin = SimFlock.rom;
131     int imin = 0; int jmin = 0;
132
133         for (int i = -1; i < 2; i++)
134         {
135                 for (int j = -1; j < 2; j++)
136                 {
137                         delta.x = i*SimFlock.nw; delta.y =j*SimFlock.nh;
138                         mag = position.sub(raptor.getPosition().add(delta)).magnitude();
139                         if (mag<magmin) {magmin = mag; imin = i; jmin=j;}
140                 }
141         }
142
143         delta.x = imin*SimFlock.nw; delta.y =jmin*SimFlock.nh;
144         return position.sub(raptor.getPosition().add(delta));
145 }
146
147 private GGVector getPositionVecDiffOnTorusBird(Bird bird)
148 {   GGVector delta = new GGVector();
149         double mag; double magmin = SimFlock.rom;
150     int imin = 0; int jmin = 0;
151
152         for (int i = -1; i < 2; i++)
```

```
153             {
154                     for (int j = -1; j < 2; j++)
155                     {
156                             delta.x = i*SimFlock.nw; delta.y =j*SimFlock.nh;
157                             mag = position.sub(bird.getPosition().add(delta)).magnitude();
158                             if (mag<magmin) {magmin = mag; imin = i; jmin=j;}
159                     }
160             }
161
162             delta.x = imin*SimFlock.nw; delta.y =jmin*SimFlock.nh;
163             return position.sub(bird.getPosition().add(delta));
164     }
165
166     private GGVector setAccelerationToZero()   //Behavior of the bird: no acceleration
167     {
168             return new GGVector(0,0);
169     }
170
171
172     private GGVector setAcceleration()   //Behavior of the bird: acceleration
173     {
174             GGVector distVec = new GGVector();
175             GGVector cohDistSumVec = new GGVector(0,0);
176             GGVector sepDistSumVec = new GGVector(0,0);
177             GGVector escDistSumVec = new GGVector(0,0);
178             GGVector aligVelSumVec = new GGVector(0,0);
```

```
179         GGVector acceleration = new GGVector(0,0);
180         GGVector accCohesion = new GGVector(0,0);
181         GGVector accSeparation = new GGVector(0,0);
182         GGVector accAlignment = new GGVector(0,0);
183         GGVector accEscape = new GGVector(0,0);
184         int ccounter = 0; int scounter = 0; int acounter = 0; int rcounter = 0;
185         ArrayList<Actor> neighbourBirds = gameGrid.getActors(Bird.class);
186         neighbourBirds.remove(this);  // Remove self
187         ArrayList<Actor> neighbourRaptors = gameGrid.getActors(Raptor.class);
188
189         for (Actor neighbour : neighbourBirds)
190         {
191          Bird bird = (Bird)neighbour;
192          distVec = getPositionVecDiffOnTorusBird(bird);
193
194          if ((distVec.magnitude() <= SimFlock.rob) && (distVec.magnitude() > SimFlock.rcrit))
195             {cohDistSumVec.x+=distVec.x;cohDistSumVec.y+=distVec.y; ccounter++;}
196          if (distVec.magnitude() <= SimFlock.rcrit)
197             {sepDistSumVec.x+=distVec.x;sepDistSumVec.y+=distVec.y; scounter++;}
198          if (distVec.magnitude() <= SimFlock.rob)
199             {aligVelSumVec.x+=bird.velocity.x; aligVelSumVec.y+=bird.velocity.y; acounter++;}
200         }
201
202         for (Actor neighbour : neighbourRaptors)
203         {
204          Raptor raptor = (Raptor)neighbour;
```

```
205            distVec = getPositionVecDiffOnTorusRaptor(raptor);
206            if (distVec.magnitude() <= SimFlock.rob)
207               {escDistSumVec.x+=distVec.x;escDistSumVec.y+=distVec.y; rcounter++;}
208           }
209
210           if (ccounter != 0) {cohDistSumVec.mult(1/ccounter);}
211           if (rcounter != 0) {escDistSumVec.mult(1/rcounter);}
212           if (acounter != 0) {aligVelSumVec.mult(1/acounter);}
213           accCohesion = cohDistSumVec.mult(-SimFlock.cohesionFactor);
214           accSeparation = sepDistSumVec.mult(SimFlock.separationFactor);
215           accAlignment =  velocity.sub(aligVelSumVec).mult(-SimFlock.alignmentFactor);
216           accEscape = escDistSumVec.mult(SimFlock.escapeFactor);
217           acceleration.x = accCohesion.x + accSeparation.x + accAlignment.x;
218           acceleration.y = accCohesion.y + accSeparation.y + accAlignment.y;
219           if (rcounter>0) {acceleration.x = accEscape.x; acceleration.y = accEscape.y;}
220         return acceleration;
221   }
222
223   public void act()
224   {
225           acceleration = setAcceleration();
226           velocity = velocity.add(acceleration.mult(SimFlock.timeFactor));
227           velocity.normalize();
228           velocity = velocity.mult(SimFlock.vbird);
229           position = position.add(velocity.mult(SimFlock.timeFactor));
230           position = toTorusPosition(position);
```

```
231            location = toLocation(position);
232            if (borderCrossed) {oldLocation.x = -1; oldLocation.y = -1;}
233            setDirection(Math.toDegrees(velocity.getDirection()));
234            setLocation(location);
235
236            if (SimFlock.drawTrace)
237              {
238                    getBackground().setPaintColor(Color.blue);
239                if (oldLocation.x != -1)
240                  getBackground().drawLine(oldLocation.x, oldLocation.y, location.x, location.y);
241                  oldLocation.x = location.x; oldLocation.y = location.y;
242              }
243    }
244 }
245
246 // ////////////////////////////////////// CLASS RAPTOR
       //////////////////////////////////////////////////////////
247
248 class Raptor extends Actor
249 {
250   private Location oldLocation = new Location(-1, -1);
251   private GGVector startVelocity;
252   public  Location location;
253   public  GGVector position;
254   public  GGVector velocity;
255   private GGVector acceleration;
```

```
256   private boolean  borderCrossed;
257
258   public Raptor(GGVector startVelocity)
259   {
260     super(true,"sprites/raptor.gif");
261         this.startVelocity = startVelocity;
262         this.velocity=startVelocity;
263   }
264
265   public void reset()
266   {
267         position = toPosition(getLocationStart());
268     setDirection(Math.toDegrees(startVelocity.getDirection()));
269     oldLocation.x = -1; oldLocation.y = -1;
270   }
271
272   private GGVector toPosition(Location location)
273   {
274     return new GGVector(location.x, location.y);
275   }
276
277   private Location toLocation(GGVector position)
278   {
279     return new Location((int)(position.x), (int)(position.y));
280   }
281
```

```
282  private GGVector toTorusPosition(GGVector position)
283  {
284         double x=position.x; double y=position.y;  borderCrossed=false;
285         if (x<0) {x = x + SimFlock.nw; borderCrossed=true;}
286         if (x>SimFlock.nw-1) {x = x - SimFlock.nw; borderCrossed=true;}
287         if (y<0) {y = y + SimFlock.nh; borderCrossed=true;}
288         if (y>SimFlock.nh-1) {y = y - SimFlock.nh; borderCrossed=true;}
289    return new GGVector(x,y);
290  }
291
292  public GGVector getPosition()
293  {
294    return position.clone();
295  }
296
297  public GGVector getVelocity()
298  {
299    return velocity.clone();
300  }
301
302
303  private GGVector getPositionVecDiffOnTorus(Bird bird)
304  {    GGVector delta = new GGVector();
305          double mag; double magmin = SimFlock.rom;
306      int imin = 0; int jmin = 0;
307
```

```
308         for (int i = -1; i < 2; i++)
309         {
310                 for (int j = -1; j < 2; j++)
311                 {
312                         delta.x = i*SimFlock.nw; delta.y =j*SimFlock.nh;
313                         mag = position.sub(bird.getPosition().add(delta)).magnitude();
314                         if (mag<magmin) {magmin = mag; imin = i; jmin=j;}
315                 }
316         }
317
318         delta.x = imin*SimFlock.nw; delta.y =jmin*SimFlock.nh;
319         return position.sub(bird.getPosition().add(delta));
320 }
321
322 private GGVector setAccelerationToZero()   //Behavior of the RAPTOR: no acceleration
323 {
324         return new GGVector(0,0);
325 }
326
327
328 private GGVector setAcceleration()   //Behavior of the RAPTOR: acceleration
329 {
330         GGVector distVec = new GGVector();
331         GGVector nearestDistVecBird = new GGVector(0,0);
332         GGVector accAggression = new GGVector(0,0);
333         GGVector acceleration = new GGVector();
```

```
334             double mag; double minmag = SimFlock.rom;
335             ArrayList<Actor> neighbourBirds = gameGrid.getActors(Bird.class);
336 
337             for (Actor neighbour : neighbourBirds)
338             {
339              Bird bird = (Bird)neighbour;
340              distVec = getPositionVecDiffOnTorus(bird);
341              mag = distVec.magnitude();
342              if (mag <= SimFlock.ror)
343                 {if (mag<minmag) {minmag = mag; nearestDistVecBird = distVec.clone();}}
344             }
345             accAggression = nearestDistVecBird.mult(-SimFlock.aggressionFactor);
346             acceleration.x = accAggression.x;
347             acceleration.y = accAggression.y;
348           return acceleration;
349     }
350 
351     public void act()
352     {
353             acceleration = setAcceleration();
354             velocity = velocity.add(acceleration.mult(SimFlock.timeFactor));
355             velocity.normalize();
356             velocity = velocity.mult(SimFlock.vraptor);
357             position = position.add(velocity.mult(SimFlock.timeFactor));
358             position = toTorusPosition(position);
359             location = toLocation(position);
```

```
360             if (borderCrossed) {oldLocation.x = -1; oldLocation.y = -1;}
361             setDirection(Math.toDegrees(velocity.getDirection()));
362             setLocation(location);
363 
364             if (SimFlock.drawTrace)
365               {
366                     getBackground().setPaintColor(Color.blue);
367                 if (oldLocation.x != -1)
368                   getBackground().drawLine(oldLocation.x, oldLocation.y, location.x, location.y);
369                   oldLocation.x = location.x; oldLocation.y = location.y;
370               }
371     }
372 }
```

Abbildungsverzeichnis

1 Torus 5
2 Entfernung zweier Punkte 6
3 Separation 7
4 Alignment 8
5 Cohesion 8

Literatur

[Plu01] Aegidius Plüss, „JGameGrid“(Java-Framework)
http://www.aplu.ch/home/apluhomex.jsp?site=45
(Version von 26. Oktober 2011) Link

[Rey01] Craig W. Reynolds, „Boids“
http://www.red3d.com/cwr/boids/
(Version von 26. Oktober 2011) Link

[MarS01] Martin Schellenberg, „GG_Torus_v1.ggb“ (GeoGebra-Datei)
http://mathematik-is1.phbern.ch/_html/Informatik/EntwProjekt/EntwProjekt.html
(Version von 26. Oktober 2011) Link

[MarS02] Martin Schellenberg, „SimFlock.java“ (GeoGebra-Datei)
http://mathematik-is1.phbern.ch/_html/Informatik/EntwProjekt/EntwProjekt.html
(Version von 26. Oktober 2011) Link