Zweidimensionale Spieleanwendung

Sven Eric Panitz

12. Juli 2022

Inhaltsverzeichnis

In diesem Kapitel wird eine sehr kleine Bibliothek zum Entwickeln von zweidimensionalen Spieleanwendungen entwickelt.

1 Punkte im 2-Dimensionalen Raum

Eine sehr einfache Klasse für Punkte im zweidimensionalen Raum macht den Anfang.

Vertex.java

```
class Vertex{
  double x;  double y;
  Vertex(double x,double y){this.x=x;this.y=y;}
  void add(Vertex that){x+=that.x;y+=that.y;}
  void moveTo(Vertex that){x=that.x;y=that.y;}
}
```

Es gibt zwei Koordinaten, die in einem Konstruktor übergeben werden. Diese Koordinaten können modifiziert werden: dadurch, dass sie auf neue Werte gesetzt werden oder dadurch, dass sie um Werte erhöht werden.

2 Spielobjekte

Mit der Klasse `Vertex` lassen sich Punkte im zweidimensionalen Raum beschreiben. Nun sollen Objekte definiert werden, die in diesem Raum verortet sind.

2.1 Allgemeine Schnittstelle

Hierfür wird zunächst eine geeignete Schnittstelle definiert.

2.1.1 Abstrakte Methoden

Jedes Spieleobjekt soll eine Position auf dem Spielfeld haben. Dabei ist zu beachten, dass das Koordinatensystem auf dem Bildschirm für einen Fensterinhalt in der Ecke links oben den Ursprung hat. Die x-Koordinate geht von dort navch rechts, die y-Koordinate geht in navch unten.

Desweiteren soll ein Objekt eine Bewegungsrichtung in beiden Koordinaten haben. Diese ist auch gebündelt in einem `Vertex`-Objekt.

Desweiteren soll ein Spielobjekt eine Weite und Höhe haben. Dieses entspricht bei Objekten, die selbst kein Rechteck sind, das kleinste das Objekt umschließende Rechteck.

Alle diese Eigenschaften sollen erfragt werden können.

GameObj.java

```
interface GameObj{
  Vertex pos();
  Vertex velocity();
  double width();
  double height();
```

Wir Beschränken uns darauf, die Spiele nur im Swing Gui-Framework zu spielen. Jedes Spieleobjekt soll somit eine Methode enthalten, in der definiert ist, wie es sich auf einem `Graphics` Objekt visualisiert.

GameObj.java

```
void paintTo(java.awt.Graphics g);
```

2.1.2 Standardmethoden

Eine ganze Reihe von Eigenschaften für Spieleobjekte lassen sich bereits als Standardmethoden umsetzen.

Das Objekt bewegt sich, um die gespeicherte Bewegungsgeschwindigkeit. Damit ist diese auf die Position aufzuaddieren.

GameObj.java

```
default void move(){pos().add(velocity());}
```

Desweiteren lassen sich Objekte in der Lage zueinander. Eine einfache methode kann testen, ob ein Objekt komplett überhalb einer Linie im Koordinatensystem liegt.

GameObj.java

```
default boolean isAbove(double y){return pos().y+height()<y;}
```

Damit lässt sich ausdrücken, ein das Objekt komplett oberhalb eines anderen Objekts liegt.

GameObj.java

```
default boolean isAbove(GameObj that){return isAbove(that.pos().y);}
```

Die Frage, ob ein Objekt komplett unterhalb eines anderen Objekts liegt, kann durch vertauschen der Objekte `this` und `that` mit der zuvor gelösten Frage, es ein Objekt Oberhalb liegt beantwortet werden.

GameObj.java

```
default boolean isUnderneath(GameObj that){return that.isAbove(this);}
```

Auf analoge Weise lässt sich prüfen ob zwei Objekte links oder rechts voneinander liegen. Hierzu ist lediglich die Weite durch die Höhe und die y-Koordinate durch die x-Koordinate zu ersetzen.

GameObj.java

```
default boolean isLeftOf(double x){return pos().x+width()<x;}
default boolean isLeftOf(GameObj that){return isLeftOf(that.pos().x);}
default boolean isRightOf(GameObj that){return that.isLeftOf(this);}
```

Ein wichtiges Konzept in Spielen ist die Frage, ob zwei Spielobjekte miteinander kollidieren. Mit den obigen Methoden ist diese Frage so gut wie beantwortet. Der trick ist, sich zu überlegen,

wannn sich zwei Objekte nicht berühren. Das ist der Fall, wenn sie über oder untereinander oder eben rechts oder links voneinander liegen.

Das ist zu negieren. Dann ist ausgedrückt, dass sie sich auf irgendeine Weise berühren.

GameObj.java

```
  default boolean touches(GameObj that){
    return
     ! (    isAbove(that)  || isUnderneath(that)
         || isLeftOf(that) || isRightOf(that)    );
  }
}
```

2.1.3 Bildobjekte

Am schnellsten kommt man zu einem bunten Spiel, wenn man als Spielobjekte Bilddateien nimmt.

Am einfachsten sind die in der Schnittstelle verlangten Methoden implementiert, wenn wir eine Datenklasse mit den entsprechenden Feldern definieren. Die vier Felder definieren Position, Geschwindigkeit und Größe. Zusätzlich soll ein Bildobjekt einen Dateinamen der Bilddatei haben und das fertige Bild für eine Darstellung auf einem `Graphics`-Objekt.

ImageObject.java

```
import java.awt.*;
import javax.swing.ImageIcon;
record ImageObject( Vertex pos, Vertex velocity
                  , double width, double height
                  , String fileName, Image image)
   implements GameObj{
```

Der kanonische Konstruktor verlangt einiges beim Aufruf ab. Man muss nicht nur die Bilddatei kennen, sondern schon das passende `Image`-Objekt und die korrekte Weiten- und Höhenangabe für das Bild. Die letzten drei lassen sich aber errechnen, wenn man das Bild aus einer Datei liest. Deshalb bekommt der kanonische Konstruktor zusätzlichen Code, in dem `width`, `height` und `image` nicht aus den Argumenten initialisiert werden, sondern diese Daten der Bilddatei entnommen werden.

Zum Laden der Bilddatei nehmen wir die Standardklasse `ImageIcon`, die uns die meiste Arbeit abnimmt und Größe, Weit eund Bildobjekt bereit stellt.

ImageObject.java

```
  public ImageObject{
    var iIcon
     = new ImageIcon(getClass().getClassLoader().getResource(fileName));
    width = iIcon.getIconWidth();
    height=iIcon.getIconHeight();
```

```
    image = iIcon.getImage();
  }
```

Da diese drei Felder nun bei der Initialisierung errechnet werden und nicht aus den Argumenten des Konstruktors übernommen werden, ist es hilfreich einen Konstruktor so zu überladen, dass er die Argumente nicht benötigt.

ImageObject.java

```
  public ImageObject(Vertex pos,Vertex velocity, String fileName){
    this(pos,velocity,0,0,fileName,null);
  }
```

Spielobjekte die direkt im Ursprung liegen und sich nicht bewegen können über einen noch einfacheren Konstruktor erzeugt werden. Da genügt die Angabe der Bilddatei. Dies ist meistens bei einem Hintergrundbild sinnvoll.

ImageObject.java

```
  public ImageObject(String fileName){
    this(new Vertex(0,0),new Vertex(0,0),fileName);
  }
```

Eine abstrakte methode verbleibt zu implementieren. Die Methode, die das Objekt auf dem `Graphics`-Objekt anzeigt. Hier ist direkt eine Methode zum Aufrufen bereit, die den gewünschten Effekt hat.

ImageObject.java

```
  public void paintTo(Graphics g){
    g.drawImage(image, (int)pos.x, (int)pos.y, null);
  }
}
```

2.1.4 Textobjekte

Ein weiteres Spielobjekt soll hauptsächlich dazu dienen Textinformationen auf dem Spiel zu platzieren.

Zu den durch die Schnittstelle festgelegten Feldern kommen noch Felder über Schriftart und Größe, sowie den zu platzierenden Text.

TextObject.java

```
import java.awt.*;
record TextObject( Vertex pos, Vertex velocity
                 , double width, double height
                 , int fontSize, String fontName, String text)
    implements GameObj{
```

Auch dieser kanonische Konstruktor wird in einem überladenen Konstruktor für die meisten Argumente mit Standardwerten belegt.

TextObject.java

```
  TextObject( Vertex pos, String text){
    this(pos,new Vertex(0,0),0,0,20,"Helvetica",text);
  }
```

Auf `Graphics`-Objekten können Texte gesetzt werden.

TextObject.java

```
  public void paintTo(Graphics g){
    g.setFont(new Font(fontName, Font.PLAIN, fontSize));
    g.drawString(text, (int)pos().x, (int)pos().y);
  }
}
```

3 Allgemeine Spielschnittstelle

In einem nächsten Schritt überlegen wir, wie man das Zusammenspiel der einzelnen Spielobjekte für ein Spiel ausdrücken kann. Auch das lässt sich gut allgemein über eine Schnittstelle ausdrücken.

Game.java

```
import java.util.List;
import java.awt.event.*;
import java.awt.*;

interface Game{
```

3.1 Abstrakte Methoden

Wir gehen von Spielfeldern mit einer festen Größe und Höhe aus. Diese müssen erfragt werden können.

Game.java

```
  int width();
  int height();
```

Desweiteren erwarten wir einen Spieler, der über die Tastatur steuerbar ist.

Game.java

```
GameObj player();
```

Alle anderen Spielfiguren sind in Listen zusaamengefasst. Es soll mehere Listen von Spielfiguren geben. So lassen sich unterschiedliche Figuren in unterschiedlichen Listen sammeln. Wir erwarten eine Liste von weiteren Listen in denen die Spielfiguren sind.

Game.java

```
List<List<? extends GameObj>> goss();
```

Dabei ist festgelegt, dass Spielfiguren aus späteren Listen, die aus früheren Listen übermalen. In der Liste `goss()` sollten also Hintergrundobjekte als erstes erscheinen.

Es ist sinnvoll für ein Spiel eine definierte Funktion zur Initialisierung bereitzustellen. Diese kann auch bei einem eventuellen `reset` verwendet werden.

Game.java

```
void init();
```

Die wahrscheinlich zentralste Methode soll Checks auf allen Spielobjekten vorsehen und so ins Spielgeschehen eingreifen.

Game.java

```
void doChecks();
```

Eine letzte abstrakte Methode, dass in einem konkreten Spiel implementiert ist, wie auf Tastatureingaben reagiert werden soll.

Game.java

```
void keyPressedReaction(KeyEvent keyEvent);
```

3.2 Standardmethoden

Auch in dieser Schnittstelle lassen sich bereits eine Reihe von Standardmethoden definieren. Wenn Spiel einen Tick weiter geht, solen sich alle Objekte einen Schritt weiterbewegen. Hierzu ist durch alle Listen zu iterieren und in jeder Liste jedes Objekt zu bewegen.

Ebenso soll auch die Spielfigut einen Bewegungsschritt machen.

Game.java

```
default void move(){
  for (var gos:goss()) gos.forEach(go -> go.move());
  player().move();
}
```

Auf ähnliche Weise ist dfür zu sorgen, dass alle am Spiel beteiligten Spielfiguren sich auf dem Spielfeld visuell darstellen. Es ist für alle Spielobjekte inklusive der Spielfigur die Methode `paintTo` aufzurufen.

Game.java

```
  default void paintTo(Graphics g){
    for (var gos:goss()) gos.forEach( go -> go.paintTo(g));
    player().paintTo(g);
  }
```

In einer letzten Standardmethode wird umgesetzt, wie das Spiel auf dem Bildschirm gestartet wird. Das Spiel wird hierzu neu initialisiert, ein Fensterrahmen erzeugt. In diesem wird das Spiel auf einer Spielfläche eingefügt. Die dazu verwendete Klasse wird im nachfolgenden Abschnitt enwickelt.

Game.java

```
  default void play(){
    init();
    var f = new javax.swing.JFrame();
    f.setDefaultCloseOperation(javax.swing.JFrame.EXIT_ON_CLOSE);
    f.add(new SwingScreen(this));
    f.pack();
    f.setVisible(true);
  }
}
```

3.3 Spiel in Swing spielen

Als nächstes soll unser Spiel visualisiert werden und mit einem Zeitgeber gestartet werden. Der Zeitgeber gibt Ticks ab. Zu jedem Tick werden die Checks des Spiel durchgeführt und der neue Spielzustand dann wieder auf dem Bildschirm angezeigt.

Wir schreiben eine Unterklasse von `JPanel`. Auf der können wir die Methode `paintComponent` überschreiben.

Die Klasse braucht zwei Dinge: die Spiellogik und den Zeitgebenr.

SwingScreen.java

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
public class SwingScreen extends JPanel{
  Game logic;
  Timer t;
```

Die Spiellogik wird im Konstruktor übergeben.

SwingScreen.java

```
public SwingScreen(Game gl) {
  this.logic = gl;
```

Dann wird der Zeitgeber erzeugt, der bei jedem Tick das Spiel einen Schritt weiter bewegt, dann die Checks macht und schließlich ein Neudarstellen des Spieles veranlasst. Anschließend wird der Zeitgeber gestartet.

SwingScreen.java

```
  t = new Timer(13, (ev)->{
      logic.move();
      logic.doChecks();
      repaint();
      getToolkit().sync();
      requestFocus();
    });
    t.start();
```

Für die Tastatursteuerung wird eine Tastaturereignisbehandlung zugefügt, die die Tastaturbehandlung des Spiels übernimmt.

SwingScreen.java

```
  addKeyListener(new KeyAdapter() {
      @Override public void keyPressed(KeyEvent e) {
        logic.keyPressedReaction(e);
      }
    });
  setFocusable(true);
  requestFocus();
  }
```

Die Graphikkomponente bekommt die für das Spiel erforderliche Größe.

SwingScreen.java

```
@Override public Dimension getPreferredSize() {
  return new Dimension((int)logic.width(),(int)logic.height());
}
```

Zum Zeichnen des Spielfeldes wird die entsprechende Methode des Spielobjekts verwendet.

SwingScreen.java

```
@Override protected void paintComponent(Graphics g) {
  super.paintComponent(g);
  logic.paintTo(g);
}
```

```
}
```

4 Kleines Beispiel-Spiel

Mit den abstrakt entworfenen Klassen soll ein erstes kleines Spiel umgesetzt werden.

Die einfachste Art, ein Spiel zu implmentieren, ist, eine Datenklasse mit den Feldern, die von der Schnittstelle `Game` vorgegeben sind, zu schreiben. Zusäztliche Felder beinhalten spezifische Information für das Spiel.

Das sind die einzelnen Listen mit den Spielobjekten. In einer sind Hintergrundobjekte, eine enthält Gegner der Spielfigur, eine soll ein paar athmosphärische Wolken enthalten und in einer letzten werden Textinformationen gesammelt.

SimpleGame.java

```
import java.util.List;
import java.util.ArrayList;
import java.awt.*;
import java.awt.event.*;
import static java.awt.event.KeyEvent.*;

record SimpleGame
  ( GameObj player, List<List<? extends GameObj>> goss
  , int width, int height, int[] schaden
  , List<GameObj> hintergrund, List<GameObj> gegner
  , List<GameObj> wolken, List<GameObj> texte)
    implements Game{
```

So praktisch es ist, auf diese Weise die wichtigsten Felder und Zugriffsmethoden zu definieren und damit schon einmal große Teile der Schnittstelle umzusetzen, so erwarten wir aber nicht, dass der kanonische Konstruktor vom Anwendungsprogrammierer aufgerufen wird, sondern der Konstruktor ohne Argumente, der schon einmal die wichtigsten Werte als Standardwerte erzeugt, verwendet werden kann.

SimpleGame.java

```
  SimpleGame(){
    this
     ( new ImageObject(new Vertex(200,200),new Vertex(1,1),"hexe.png")
     , new ArrayList<>(), 800, 600, new int[]{0}
     , new ArrayList<>(), new ArrayList<>()
     , new ArrayList<>(), new ArrayList<>());
  }
```

Zur Initialisierung werden alle Listen von Spielobjekten neu erzeugt:

SimpleGame.java

```
public void init(){
  goss().clear();
  goss().add(hintergrund());
  goss().add(gegner());
  goss().add(wolken());
  goss().add(texte());
  hintergrund().clear();
  gegner().clear();
  wolken().clear();
  texte().clear();
```

Für den Hintergrund setzen wir ein Bild einer Wiese in den Ursprung:

SimpleGame.java

```
  hintergrund().add(new ImageObject("wiese.jpg"));
```

Vier Bildern von Wolken wehen mit leicht unterschiedlichen Geschwindikeiten durch den oberen Bildausschnitt:

SimpleGame.java

```
  wolken().add(new ImageObject(
               new Vertex(800,10),new Vertex(-1,0),"wolke.png"));
  wolken().add(new ImageObject(
               new Vertex(880,90),new Vertex(-1.2,0),"wolke.png"));
  wolken().add(new ImageObject(
               new Vertex(1080,60),new Vertex(-1.1,0),"wolke.png"));
  wolken().add(new ImageObject(
               new Vertex(980,110),new Vertex(-0.9,0),"wolke.png"));
```

Und zwei Bienen sind die Gegener der Spielfigur.

SimpleGame.java

```
  gegner().add(new ImageObject(
               new Vertex(800,100),new Vertex(-1,0),"biene.png"));
  gegner().add(new ImageObject(
               new Vertex(800,300),new Vertex(-1.5,0),"biene.png"));

  texte().add(new TextObject(new Vertex(10,30)
                             ,"Bienenstiche: 0"));
}
```

Es folgen die Checks, die für jeden Spieltick gemacht werden.

SimpleGame.java

```
public void doChecks(){
```

Die Wolken werden, wenn sie aus dem linken Bildschirmrand geweht wurden, wieder ganz an den rechten Rand gesetzt.

SimpleGame.java

```
    for (var w:wolken()) if (w.isLeftOf(0)) {w.pos().x = width();}
```

Ebenso wird mit den zwei Gegnern, den Bienen verfahren.

SimpleGame.java

```
    for (var g:gegner()){
      if (g.isLeftOf(0))     {g.pos().x = width();}
```

Zusätzlich ist für jeden Gegner zu testen, ob er die Spielfigur berührt. Wenn das der Fall ist, setzen wir den Gegner hinter den rechten Bildschirmrand, erhöhren den Schaden und legen ein neues Textobjekt mit dem neuen Schaden an.

SimpleGame.java

```
      if (player.touches(g)) {
        g.pos().moveTo(new Vertex(width()+10,g.pos().y));
        schaden[0]++;
        texte().clear();
        texte().add(new TextObject(new Vertex(10,30)
                                   ,"Bienenstiche: "+schaden[0]));
      }
    }
    }
```

Es verbleibt die Tastatureingabe. Die Spielfigur wird über die Pfeiltasten gesteuert. Dabei verändert man die Geschwindigkeit in die entsprechende Richtung.

SimpleGame.java

```
  public void keyPressedReaction(KeyEvent keyEvent){
    switch (keyEvent.getKeyCode()){
      case VK_RIGHT -> player().velocity().add(new Vertex(1,0));
      case VK_LEFT  -> player().velocity().add(new Vertex(-1,0));
      case VK_DOWN  -> player().velocity().add(new Vertex(0,1));
      case VK_UP    -> player().velocity().add(new Vertex(0,-1));
    }
  }
```

Das Spiel kann entweder direkt in der JShell oder kompiliert über folgende Haiptmethode gestartet werden.

SimpleGame.java

```
  public static void main(String... args){
    new SimpleGame().play();
  }
}
```

Ein erstes kleines Spiel ist entstanden. Eine Spielfigur muss anderen ausweichen, der Schaden durch Kollisionen wird angezeigt. Das ganze Spiel ist in Abbildung 1 zu sehen.

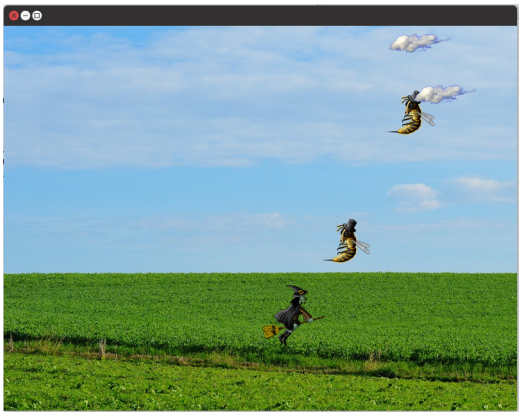

Abbildung 1: Ein erstes einfaches Spiel.

5 Simulationsspiel Pandemie

Schon mit diesem einfachen Klassen lassen sich ganz unterschiedliche Spielideen realisieren. Wir machen als zweites Beispiel ein kleines Spiel, das die Ausbreitung einer Infektionskrankheit simuliert.

5.1 Spielobjekt

Wir implementieren dieses Mal die Schnittstelle `GameObj` nicht über eine Datenklasse sondern einer Klasse mit modifizierbaren Feldern. Deshalb müssen wir die Informationen, die von der Schnittstelle verlangt werden, manuell umgesetzt werden. Um weitere unterschiedliche Implementierungen von `GameObj` zu unterstützen, die sich nicht einfach durch Datenklassen realisieren lassen, weil sie viel veränderliche Informationen tragen, geben wir eine abstrakte Klasse. Sie stellt Felder für Position, Geschwindigkeit und Weite wie Höhe mit den Selektionsmethoden bereits zur Verfügung.

AbstractGameObj.java

```
abstract class AbstractGameObj implements GameObj{
  protected Vertex pos;
  protected Vertex velocity;
  protected double width;
  protected double height;

  public Vertex pos(){return pos;}
  public Vertex velocity(){return velocity;}
  public double width(){return width;}
  public double height(){return height;}

  public AbstractGameObj(Vertex p, Vertex v, double w, double h){
    pos=p; velocity=v; width=w; height=h;
  }
}
```

Die Spielobjekte sind die Personen einer Population.

Person.java

```
import java.awt.*;
class Person extends AbstractGameObj{
```

Für diese interessiert in der Simulation der Infektionsstatus. Wir unterscheiden fünf Stück. Alle sind mit einer Farbe assoziiert. Hierzu eignet sich eine Aufzählungsklasse, die als innere Klasse der Klasse Person realisiert wird.

Person.java

```
  static enum Status {
    ungeimpft(Color.BLUE), geimpft(Color.GREEN), infiziert(Color.RED),
    genesen(Color.ORANGE), gestorben(Color.BLACK);

    Color c;
    Status(Color c){this.c=c;}
  }
```

Eine Person hat einen solchen Infektionsstatus.

Person.java

```
  public Status status = Status.ungeimpft;
```

Ein weiteres Feld soll bei infizierten Personen ein Zähler sein, der nach einer Infektion bei jedem Tick eins runter gezählt wird. Wenn dieser Zähler dann 0 erreicht hat, gilt eine Person als genesen.

Person.java

```
  public int restInfektionnsDauer;
```

Im Konstruktor wird eine zufällige Bewegung gesetzt und mit einer Wahrscheinlichkeit von 10% ist eine Person initial infiziert.

Person.java

```
Person(Vertex pos) {
  super(pos, new Vertex(Math.random()-0.7, Math.random()-0.7),10 ,10);
  if (Math.random()<0.03) infizieren();
}
```

Die Personen werden nur als kleine Quadrate in der Farbe, die den Infektionsstatus darstellt, gezeichnet.

Person.java

```
public void paintTo(Graphics g){
  g.setColor(status.c);
  g.fillRect((int)pos().x,(int)pos().y,(int)width(),(int)height());
}
```

Es folgen nun die Methoden, die sich mit dem Infektionsstatus beschäftigen.

Personen können geimpft werden. Mehrfachimpfungen und sogenannten Booster sind noch nicht vorgesehen.

Person.java

```
void impfen() {
  if (status==Status.ungeimpft) status=Status.geimpft;
  }
```

Personen können sich infizieren. In unserem Fall schützt die Impfung vollstaändig vor einer Infektion. Eine Infektion zieht einen Infektionsdauer nach sich, in dem die Person ansteckend für andere ist.

Person.java

```
void infizieren() {
  if (status==Status.ungeimpft) {
    status=Status.infiziert;
    restInfektionnsDauer=800;
  }
}
```

Personen können genesen, wobei beim Genesen eine Wahrscheilnichkeit von 2% besteht, dass die Person verstirbt.

Person.java

```
void genesen() {
  if (status==Status.infiziert) {
    if (Math.random()<0.02) {
```

```
        status=Status.gestorben;
        velocity = new Vertex(0, 0);
      }else status=Status.genesen;

      restInfektionnsDauer=0;
    }
  }
}
```

5.2 Simulation

Für das Simulationsspiel können wir zur Umsetzung wie bereits bei `SimpleGame` eine Datenklassen verwenden. Diese hat für die Spielobjekte eine Liste von Personen. Die Spielfigur ist ein Bildobjekt mit der Darstellung einer Spritze.

Corona.java

```
import java.util.List;
import java.util.ArrayList;
import java.awt.*;
import java.awt.event.*;
import static java.awt.event.KeyEvent.*;

record Corona( GameObj player, List<List<? extends GameObj>> goss
             , int width, int height, List<Person> personen)
    implements Game{
```

Die Felder des kanonischen Konstruktors werden in einem überladenen Konstruktor gesetzt.

Corona.java

```
  Corona(){
   this
    ( new ImageObject(new Vertex(200,200),new Vertex(0,0),"spritze.png")
    , new ArrayList<>(), 1200, 700, new ArrayList<>());
  }
```

Bei der Initialisierung werden 500 Personen auf zufälligen Positionen in der Simulation erzeugt.

Corona.java

```
  public void init() {
    goss().clear();
    personen().clear();
    goss.add(personen());
    for (int i = 0; i < 500; i++) {
      personen().add(new Person
        (new Vertex( Math.random()*(width()-20)
                   , Math.random()*(height()-20))));
    }
```

```
}
```

Die Spielfigur, die mit einer Spritze die Personen impfen kann, steuern wir wie bereits die Hexe im ersten Spiel.

Corona.java

```
public void keyPressedReaction(KeyEvent keyEvent){
  switch (keyEvent.getKeyCode()){
    case VK_RIGHT -> player().velocity().add(new Vertex(0.2,0));
    case VK_LEFT  -> player().velocity().add(new Vertex(-0.2,0));
    case VK_DOWN  -> player().velocity().add(new Vertex(0,0.2));
    case VK_UP    -> player().velocity().add(new Vertex(0,-0.2));
  }
}
```

Wir schauen in den Checks auf jede Person im Spiel:

Corona.java

```
public void doChecks() {
  for (var m1 : personen()) {
```

Zunächst wird verhindert, dass die Personen aus dem Spielfeld fliegen.

Corona.java

```
    if (m1.pos().x<0||m1.pos().x+m1.width()>width()) {
      m1.velocity().x *= -1;
    }
    if (m1.pos().y<0||m1.pos().y+m1.height()>height()) {
       m1.velocity().y *= -1;
    }
```

Personen die die Spielfigur berühren, werden geimpft:

Corona.java

```
    if (m1.touches(player())) m1.impfen();
```

Für infizierte Personen wird die Restinfektionsdauer verringert. hat diese 0 erreicht, dann genesen die Personen.

Corona.java

```
    if(m1.status==Person.Status.infiziert) {
      m1.restInfektionnsDauer--;
      if (m1.restInfektionnsDauer==0) m1.genesen();
    }
```

Und schließlich sollen alle Personen die sich begegnen, sprich berühren, gegenseitig anstecken können-

Corona.java

```
      for (var m2 : personen())
        if(m1.touches(m2) && m1.status==Person.Status.infiziert)
          m2.infizieren();
    }
  }
```

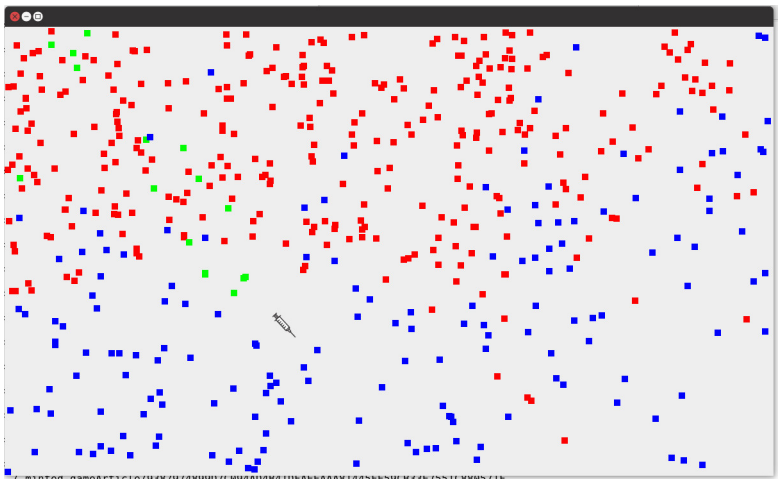

Abbildung 2: Eine kleine Simulation eines Infektionsgeschehens.

Jetzt können wir die Simulation einmal spielen. Ein Bildschirmfoto findet sich in Abbildung 2.

Interessant zu beobachten ist in dieser kleinen Simultation, wie schnell sich das Geschehen ändert, wenn man ein paar Parameter verändert.

Corona.java

```
  public static void main(String[] args) {
    new Corona().play();
  }
}
```