Praktische Informatik I WS 2012/13

Übungsblatt 8

Musterlösung

Aufgabe 1 Alles super hier (20%)

Es gibt nun die zwei leeren Klassen `Obstacle` und `Item`. `Wall` und `Door` erben nun von `Obstacle`, `Key` und `Bomb` von `Item`:

```
9 public class Wall extends Obstacle

9 public class Door extends Obstacle

9 public class Key extends Item

9 public class Bomb extends Item
```

Die Klasse `Robot` testet nun nur noch auf Kollisionen mit Hindernissen, statt direkt auf Wände oder Türen. Diese Zeile ist wegen der Bearbeitung von Aufgabe 2 nicht mehr im Quelltext enthalten:

```
55 if (getOneIntersectingObject(Obstacle.class) != null) {
```

Das `Inventory` speichert nun eine Instanz von `Item`:

```
12     private Item contents;

25     public void store(Item object)

45     public Item get()
```

Die Klasse `Robot` hebt nun `Item`s auf (auch nicht mehr vorhanden):

```
152                 Item object = (Item) getOneIntersectingObject(Item.class);
```

Tests. Der Roboter bleibt immer noch vor Wänden und Türen stehen. Schlüssel und Bomben können weiterhin aufgehoben und abgelegt werden. Mit dem Schlüssel können immer noch Türen geöffnet und mit der Bombe Wände gesprengt werden.

Aufgabe 2 Massenkarambolage (30%)

Führt eine Superklasse für alle Klassen ein, die auf Kollisionen testen.

```
13 public class Collider extends Actor
14 {

162 }
```

1) *Eine Methode, die das Objekt einer bestimmten Klasse zurückliefert, mit dem gerade eine Kollision besteht, bzw.* `null`*, wenn keine Kollision besteht. Vorerst ist diese Methode einfach nur ein Synonym für* `getOneIntersectingObject`*.* (Zeile 33 ist wegen Aufgabe 3 nicht mehr vorhanden):

```
31      public Actor getCollidingObject(Class clazz)
32      {
33          return getOneIntersectingObject(clazz);
34      }
```

2) *Eine Komfortmethode, die zurückliefert,* ob *gerade eine Kollision mit einem Objekt einer bestimmten Klasse besteht. Sie soll sich auf die erste Methode abstützen.*

```
49      public boolean collidesWith(Class clazz)
50      {
51          return getCollidingObject(clazz) != null;
52      }
```

3) *Eine Methode, die zurückliefert, ob gerade eine neue Kollision mit einer Instanz einer bestimmten Klassen begonnen hat. Da diese Methode für verschiedene Klassen aufgerufen werden kann, muss sie darüber Buch führen, ob und wenn ja, mit welcher Instanz einer Klasse gerade eine Kollision besteht. So kann sie auch zurückliefern, wenn plötzlich mit einer anderen Instanz derselben Klasse eine Kollision begonnen hat. Auch diese Methode soll sich auf die erste Methode abstützen.*

Der aktuelle Kollisionszustand wird pro Klasse in einer `HashMap` gespeichert. Der Einfachheit halber wird auch der Wert `null` in der `HashMap` abgelegt, wenn es gerade keine Kollision gibt. Eine neue Kollision liegt immer vor, wenn wir uns gerade in einer Kollision befinden und das Objekt ein anderes ist als zuvor, wobei dies auch den Fall einschließt, dass wir zuvor keine Kollision hatten:

```
21      /** Merkt sich pro Actor-Klasse, mit welchem Objekt gerade eine Kollision besteht. */
22      private HashMap<Class, Actor> activeCollisions = new HashMap<Class, Actor>();

61      public boolean newCollisionWith(Class clazz)
62      {
63          Actor lastCollision = activeCollisions.get(clazz);
64          Actor currentCollision = getCollidingObject(clazz);
65          activeCollisions.put(clazz, currentCollision);
66          return currentCollision != null && currentCollision != lastCollision;
67      }
```

Baut alle Kollisionstests in eurem Code so um, dass sie nur noch diese drei Methoden benutzen.

Die Klassen `Robot`, `Exit`, `Smiley` und `Skull` erben nun von `Collider`. In diesen Klassen wurde, wenn vorhanden, auch das Attribut `intersecting` entfernt, weil es nicht mehr benötigt wird:

```
9  public class Robot extends Collider

9  public class Exit extends Collider

9  public class Smiley extends Collider

9  public class Skull extends Collider
```

In der Klasse `Robot` führen die Methoden `act`, `countCollisions`, `unlockDoor`, `blowUpObstacle` und `pickUpOrDropObject` Kollisionstests durch:

```
52              if (collidesWith(Obstacle.class)) {

89          if (newCollisionWith(Smiley.class)) {
90              score.setScore(score.getScore() + 10);
91          } else if (newCollisionWith(Skull.class)) {

117         if (collidesWith(Door.class) && !inventory.isEmpty()
118                 && inventory.get() instanceof Key) { // Übungsblatt 3 Bonusaufgabe
119             inventory.clear();
120             getWorld().removeObject(getCollidingObject(Door.class));
```

```
131         if (collidesWith(Wall.class) && !inventory.isEmpty() && inventory.get()
              instanceof Bomb) {
132             inventory.clear();
133             getWorld().removeObject(getCollidingObject(Wall.class));

150                 if (collidesWith(Item.class)) {
151                     inventory.store((Item) getCollidingObject(Item.class));
```

In den Klassen `Exit`, `Smiley` und `Skull` sind nur die Methoden `act` vom Umbau betroffen:

```
38          if (newCollisionWith(Robot.class)) {
39              Greenfoot.playSound("fanfare.wav");
40              switchWorld((Robot) getCollidingObject(Robot.class));

18          if (newCollisionWith(Robot.class)) {

18          if (newCollisionWith(Robot.class)) {
```

Tests. Grundsätzlich werden alle Kollisionen erkannt, die vor dem Umbau auch bereits erkannt wurden, d.h. der Roboter kann sich nicht durch Hindernisse bewegen, kann weiterhin Schlüssel und Bombe aufnehmen, Berührungen mit Smileys lassen den Spielstand um 10 Punkte wachsen und eine Berührung des Totenschädels beendet das Spiel. Ebenso spielen Smileys und Totenschädel weiterhin ihren Sound ab und der Ausgang schaltet den Level um. Zusätzlich kann man aber auch mehrere Smileys dicht nebeneinander platzieren und mit dem Roboter immer zwischen ihnen hin und herfahren. Obwohl der Roboter dabei immer mindestens einen der Smileys berührt, wächst der Punktestand dennoch bei jedem Wechsel von einem Smiley zum nächsten, d.h. es wird erkannt, dass das Objekt, mit dem der Roboter kollidiert, gewechselt hat. Platziert man Smiley und Totenschädel dicht beieinander, kann man auch testen, dass trotz einer andauernden Kollision mit einem Smiley sofort festgestellt wird, wenn der Totenschädel berührt wird.

Aufgabe 3 Richtig kollidieren (50%)

1) *Erweitert die erste Methode aus Aufgabe 2 so, dass sie die Liste aller Schnittobjekte durchläuft, d.h. die Rückgabe von* `getIntersectingObjects` *verarbeitet. Erst, wenn für eines der Objekte bestätigt wird, dass wirklich eine Kollision besteht, wird dieses als Ergebnis zurückgeliefert. Besteht zu keinem Objekt eine Kollision, ist die Rückgabe weiterhin* `null`*.*

```
31      public Actor getCollidingObject(Class clazz)
32      {
33          List<Actor> actors = getIntersectingObjects(clazz);

35          for (Actor actor : actors) {
36              if (confirmCollisionWith(actor)) {
37                  return actor;
38              }
39          }
40          return null;
41      }
```

2) *Um festzustellen, ob tatsächlich eine Kollision besteht, muss für jedes nicht-transparente Pixel des einen Objekts (z.B. jenem, das den Kollisionstest durchführt) überprüft werden, ob sich an derselben Stelle (in Weltkoordinaten) in dem anderen Objekt auch ein nicht-transparentes Pixel befindet. Sobald dies der Fall ist, gibt es tatsächlich eine Kollision. Um dies herauszubekommen, müssen die Pixelkoordinaten des Bildes des ersten Objekts in Weltkoordinaten umgerechnet werden und von dort weiter in Pixelkoordinaten des Bildes des zweiten Objekts.*

```
78      private boolean confirmCollisionWith(Actor other)
79      {

82          for (int y = 0; y < getImage().getHeight(); ++y) {
83              for (int x = 0; x < getImage().getWidth(); ++x) {
84                  if (getImage().getColorAt(x, y).getAlpha() > 0) {
85                      int[] inWorld = toWorld(this, new int[] {x, y});
86                      int[] inOther = toImage(other, inWorld);
87                      if (inOther[0] >= 0 && inOther[0] < other.getImage().getWidth() &&
88                          inOther[1] >= 0 && inOther[1] < other.getImage().getHeight() &&
89                          other.getImage().getColorAt(inOther[0], inOther[1]).getAlpha() >
                              0) {

93                              return true;

95                      }
96                  }
97              }
98          }
99          return false;
100     }
101
102     /**
103      * Transformation der Pixelkoordinaten eines Actors in Weltkoordinaten.
104      * @param actor Der Actor, zu dem das Pixel gehört
105      * @param inImage Die Koordinaten des Pixels innerhalb des Bildes des Actors.
106      * @return Die Koordinaten desselben Pixels innerhalb der Welt.
107      */
108     private static int[] toWorld(Actor actor, int[] inImage)
109     {
110         int dx = inImage[0] - actor.getImage().getWidth() / 2;
111         int dy = inImage[1] - actor.getImage().getHeight() / 2;
112         double theta = Math.toRadians(actor.getRotation());
113         return new int[] {
114             (int) (actor.getX() + dx * Math.cos(theta) - dy * Math.sin(theta)),
115             (int) (actor.getY() + dx * Math.sin(theta) + dy * Math.cos(theta))
116         };
117     }
118
119     /**
120      * Transformation von Weltkoordinaten in die Pixelkoordinaten eines Actors.
121      * @param actor Der Actor, in dessen Pixelkoordinaten transformiert wird.
122      * @param inWorld Die Weltkoordinaten des Pixels.
123      * @return Die Koordinaten desselben Pixels innerhalb des Bilds des Actors.
124      */
125     private static int[] toImage(Actor actor, int[] inWorld)
126     {
127         int dx = inWorld[0] - actor.getX();
128         int dy = inWorld[1] - actor.getY();
129         double theta = Math.toRadians(actor.getRotation());
130         return new int[] {
131             (int) (actor.getImage().getWidth() / 2 + dx * Math.cos(theta) + dy * Math.sin
                  (theta)),
132             (int) (actor.getImage().getHeight() / 2 - dx * Math.sin(theta) + dy * Math.
                  cos(theta))
133         };
134     }
```

3) *Da man bei den Transformationen viel falsch machen kann, baut ihr einen Testmodus ein. Ist dieser aktiv, werden überhaupt keine Kollisionen mehr gemeldet. Stattdessen werden im Bild des Actors, der auf Kollisionen testet, alle Pixel farblich markiert, die mit einem anderen Objekt kollidieren, z.B. in* `Color.RED`.

```
15      /** Ist der Testmodus aktiv? */
16      private static boolean testMode = false;
17
18      /** Bild bei der nächsten Kollision im Testmodus zurücksetzen? */
19      boolean resetImage;

34          resetImage = testMode;
```

```
80          resetImageInTestMode();

90                          if (testMode) {
91                              getImage().setColorAt(x, y, Color.RED);
92                          } else {
93                              return true;
94                          }

136     /**
137      * Aktivieren oder Deaktivieren des Testmodus'. Im Testmodus werden keine Kollisionen
138      * mehr gemeldet und alle Kollisionstests zeichnen Überlappungen mit anderen Objekten
139      * bei sich in rot ein.
140      * @param active Testmodus aktivieren?
141      */
142     public static void setTestMode(boolean active)
143     {
144         testMode = active;
145     }
146
147     /**
148      * Ersetzt das Bild dieses Objekts wieder durch das zuletzt geladene.
149      * Das funktioniert nur, wenn der Dateiname des Bildes keine zwei aufeinander
150      * folgenden Leerzeichen enthält.
151      */
152     private void resetImageInTestMode()
153     {
154         if (resetImage) {
155             String description = getImage().toString();
156             int endIndex = description.indexOf("  ");
157             String filename = description.substring(17, endIndex);
158             setImage(filename);
159             resetImage = false;
160         }
161     }
```

Tests. Der Roboter kann nun deutlich dichter an andere Objekte heranfahren, z.B. kann er die Ecke einer Wand mit seinen zwei Strahlern auch umschließen (siehe Abb. 1 links). Dies ist nicht immer offensichtlich, da er sich ja mit einer Schrittweite von fünf durch die Welt bewegt und deshalb manchmal immer noch in größeren Abständen von Objekten zum Stehen kommt. Mit aktiviertem Testmodus kann man sehen, dass die überlappenden Teile rot eingefärbt werden. Um diesen Test mit beliebigen Rotationen beider an einer Kollision beteiligten Objekten durchführen zu können, drehen sich die Totenschädel nun langsam, so dass man den sich drehenden Roboter mit einem sich drehenden Totenschädel überlappen lassen kann (siehe Abb. 1 rechts).

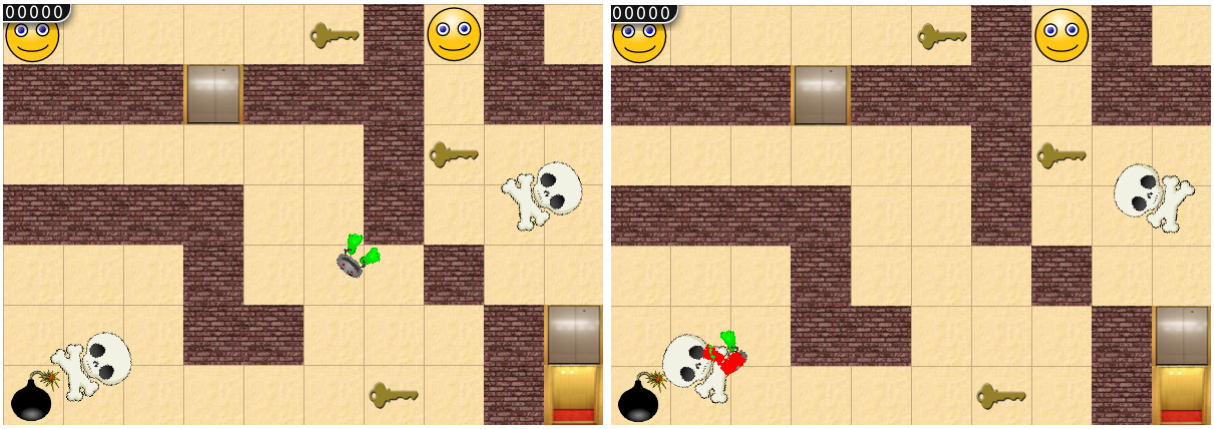

Abbildung 1: Kollision mit einer Wand bei deaktiviertem Testmodus und Überlappung mit einem Totenschädel (in rot) bei aktiviertem Testmodus