Grafische Animation in *Java*

Karl Stroetmann

23. Juni 2012

Die Modellierung komplexer Systeme spielt sowohl in der Wissenschaft als auch in der Praxis eine wichtige Rolle. Um die Ergebnisse einer solchen Modellierung zu veranschaulichen ist es hilfreich, ein Modell eines Systems grafisch zu animieren. Auch zum besseren Verständnis komplexer Algorithmen ist eine graphische Animierung nützlich.

Das vorliegende Skript zeigt, wie in *Java* Algorithmen grafisch animiert werden können. Dabei habe ich mich bewußt auf eine zweidimensionale graphische Darstellung beschränkt, denn eine Einführung in *Java3D* würde den Rahmen dieses Labors sprengen. Aufgrund des zur Verfügung stehenden Zeitrahmens ist eine vollständige Beschreibung und Bewertung der verschiedenen Konstrukte nicht möglich. Ziel ist es vielmehr, einen minimalen Satz an Klassen der *Java*-Bibliothek vorzustellen, die für eine elementare Animation von Algorithmen ausreichend sind. Die in diesem Skript besprochenen Beispiele finden Sie in dem folgenden Verzeichnis:

`http://wwwlehre.dhbw-stuttgart.de/~stroetma/FUEL/Java`

Die Programme wurden mit der Version *Java* 1.5 entwickelt und mit den Versionen 1.5 und 1.6 getestet.

1 Zeichnen einfacher geometrischer Formen

Wir zeigen zunächst, wie sich einfache geometrische Formen zeichnen lassen. Abbildung 1 zeigt die Implementierung der Klasse *DrawFrame*, die wir jetzt diskutieren.

1. Wir importieren die Pakete "`java.awt.*`", "`java.awt.geom.*`" und "`javax.swing.*`", in denen die zum Zeichnen benötigten Klassen *JFrame*, *JPanel*, etc. definiert werden.

2. Die Klasse *DrawFrame* erweitert die Klasse *JFrame*. Die Klasse *JFrame* stellt uns einen Rahmen zur Verfügung, auf dem wir die verschiedenen Komponenten eines GUI plazieren können.

3. In Zeile 11 setzen wir zunächst den Titel, der später vom Window-Manager im Rahmen des Fensters angezeigt wird.

4. In Zeile 12 legen wir Breite und Höhe des Fensters in Pixeln fest.

5. In Zeile 13 erzeugen wir ein Objekt der Klasse *DrawPanel*. Diese Klasse erweitert die Klasse *JPanel*. Objekte der Klasse *JPanel* bieten die Funktionalität einer *Leinwand*, auf der wir später malen können. Die Implementierung der Klasse *DrawPanel* wird in den Abbildungen 2 und 3 auf den Seiten 3 und 4 gezeigt.

6. Durch den Befehl "`add(panel)`" wird die Leinwand `panel` in den durch die Klasse *JFrame* gegebenen Rahmen eingespannt.

7. In Zeile 15 legen wir fest, dass das Programm beendet werden soll, wenn das Fenster, das den durch *JFrame* gegebenen Rahmen zeigt, geschlossen wird.

8. Bis jetzt wird das Fenster noch nicht am Bildschirm gezeigt. Der Aufruf "`setVisible(true)`" macht den Rahmen mit der eingebetteten Leinwand sichtbar.

```
1  import java.awt.*;
2  import java.awt.geom.*;
3  import javax.swing.*;
4
5  class DrawFrame extends JFrame
6  {
7     public static void main(String[] args) {
8        DrawFrame frame = new DrawFrame();
9     }
10    public DrawFrame() {
11       setTitle("DrawTest");
12       setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
13       DrawPanel panel = new DrawPanel();
14       add(panel);
15       setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
16       setVisible(true);
17    }
18    public static final int DEFAULT_WIDTH  = 400;
19    public static final int DEFAULT_HEIGHT = 400;
20 }
```

Abbildung 1: Zeichnen einfacher geometrischer Formen

Die Abbildungen 2 und 3 zeigen die Implementierung der Klasse *DrawPanel*, die die Klasse *JPanel* erweitert und folglich die Leinwand darstellt, auf der wir zeichnen.

1. Die Klasse *DrawPanel* enthält genau eine Methode, die Methode

   `public void paintComponent(Graphics g);`

   Jedesmal, wenn das die Leinwand enthaltende Fenster am Bildschirm abgebildet werden soll, wird diese Methode aufgerufen um den Inhalt des Fensters zu zeichnen. Wir dürfen die Methode *paintComponent*() später allerdings nicht selber aufrufen, der Aufruf dieser Methode erfolgt nur intern aus der GUI. Wenn wir das Fenster zeichnen wollen, dann rufen wir stattdessen die Methode *repaint*() auf. Dadurch wird der GUI signalisiert, dass diese bei der nächsten passenden Gelegenheit die Methode *paintComponent*() aufrufen soll. Der Hintergrund dieser Komplikation ist die Tatsache, dass die GUI und das restliche Programm in zwei unterschiedlichen Threads laufen. Wir gehen später noch näher auf diesen Punkt ein.

   Die Methode `paintComponent` erhält ein Objekt vom Typ *Graphics* als Argument. Dieses Objekt können Sie sich als einen Pinsel vorstellen, mit dem wir auf der Leinwand malen können.

2. Das Objekt "`g`", also unser Pinsel, hat in Wirklichkeit den Typ *Graphics2D*. Aus Gründen der Rückwärts-Kompatibilität ist der Typ von "`g`" in der Deklaration der Methode als `Graphics` angegeben. Um die volle Funktionalität unseres Pinsels nutzen zu können, casten wir in Zeile 25 das Objekt "`g`" auf den Typ *Graphics2D*.

3. In Zeile 26 setzen wir zunächst die Farbe des Hintergrunds. Allgemein erzeugt der Konstruktor

   `Color`(*red*, *green*, *blue*)

   eine Farbe mit den angegebenen Stärken der Farben Rot, Grün und Blau. Diese Stärken müssen in dem Intervall $[0, 1]$ liegen. Da wir alle Parameter auf 0.0 setzen, ist der Hintergrund schwarz.

```
21  class DrawPanel extends JPanel
22  {
23      public void paintComponent(Graphics g)
24      {
25          Graphics2D g2 = (Graphics2D) g;
26          setBackground(new Color(0.0F, 0.0F, 0.0F));
27          super.paintComponent(g);
28
29          // draw a rectangle
30          double upperLeftX = 100;
31          double upperLeftY = 100;
32          double width      = 200;
33          double height     = 150;
34          Rectangle2D rectangle =
35              new Rectangle2D.Double(upperLeftX, upperLeftY, width, height);
36          g2.setPaint(Color.BLUE);
37          g2.draw(rectangle);
38
39          // draw an ellipse enclosed in the rectangle
40          Ellipse2D ellipse = new Ellipse2D.Double();
41          ellipse.setFrame(rectangle);
42          g2.setPaint(Color.YELLOW);
43          g2.fill(ellipse);
44
45          // draw a diagonal line
46          Line2D line1 = new Line2D.Double(upperLeftX,
47                                           upperLeftY,
48                                           upperLeftX + width,
49                                           upperLeftY + height);
50          Line2D line2 = new Line2D.Double(upperLeftX,
51                                           upperLeftY + height,
52                                           upperLeftX + width,
53                                           upperLeftY);
54          g2.setPaint(Color.MAGENTA);
55          g2.draw(line1);
56          g2.setPaint(Color.BLACK);
57          g2.draw(line2);
```

Abbildung 2: Definition der Klasse *DrawPanel*, 1. Teil.

4. Nachdem wir den Hintergrund gesetzt haben, rufen wir in Zeile 27 die Methode *paintComponent()* für die Oberklasse *JPanel* auf, damit der Hintergrund gezeichnet wird. Diese Oberklasse wird durch das Schlüsselwort "`super`" angesprochen.

5. Als nächstes zeichnen wir verschiedene geometrische Figuren. In Zeile 34 erzeugen wir ein Rechteck. Dazu sind vier Parameter erforderlich:

   (a) Die X-Koordinate des linken oberen Eckpunktes.

   (b) Die Y-Koordinate des linken oberen Eckpunktes.

   Diese Koordinaten werden in Pixeln angegeben. Die linke obere Ecke der Leinwand hat die Koordinaten $(0, 0)$. Die X-Koordinaten nehmen nach rechts zu, die Y-Koordinaten nehmen nach unten hin zu.

```
58          double centerX = rectangle.getCenterX();
59          double centerY = rectangle.getCenterY();
60          double radius  = 0.5 * Math.sqrt(width * width + height * height);
61
62          Ellipse2D circle = new Ellipse2D.Double(centerX - radius,
63                                                  centerY - radius,
64                                                  2 * radius, 2 * radius);
65          g2.setPaint(Color.GREEN);
66          g2.setStroke(new BasicStroke(4.0F));
67          g2.draw(circle);
68
69          // draw another transparent circle
70          Ellipse2D sc = new Ellipse2D.Double(upperLeftY - 0.5 * radius,
71                                              upperLeftY - 0.5 * radius,
72                                              radius, radius);
73          g2.setPaint(new Color(0.0F, 0.0F, 1.0F, 0.6F));
74          g2.fill(sc);
75      }
76  }
```

Abbildung 3: Definition der Klasse *DrawPanel*, 2. Teil.

(c) Die Breite des Rechtecks.

(d) Die Höhe des Rechtecks.

6. Bevor wir das Rechteck zeichnen, müssen wir unseren Pinsel in den Farbtopf tauchen. Dies geschieht in Zeile 36, wo wir die Farbe Blau wählen. In *Java* sind die folgenden Farben vordefiniert:

   `BLACK`, `BLUE`, `CYAN`, `DARK_GRAY`, `GRAY`, `GREEN`, `LIGHT_GRAY`, `MAGENTA`, `ORANGE`, `PINK`, `RED`, `WHITE`, `YELLOW`.

7. Das Zeichnen geschieht nun mit der Methode `draw`() in Zeile 37.

8. Wir zeichnen nun eine Ellipse, die von dem eben gezeichneten Rechteck begrenzt wird. Die Ellipse erhält ihre Begrenzung durch den Aufruf von *setFrame*() in Zeile 41.

   Bei der Ellipse wollen wir nicht nur den Umriß zeichnen, sondern statt dessen die gesamte Fläche der Ellipse mit Farbe füllen. Daher benutzen wir in Zeile 43 statt der Methode *draw*() die Methode *fill()*.

9. Als nächstes zeichnen wir zwei Linien, und zwar die Diagonalen des Rechtecks. Der Konstruktor der Klasse *Line2D.Double* wird wie folgt aufgerufen:

   Line2D.Double(x_1, y_1, x_2, y_2).

   Hier ist (x_1, y_1) der Start-Punkt der Linie und (x_2, y_2) ist der Endpunkt.

10. Schließlich wollen wir noch einen Kreis zeichnen. Der Mittelpunkt dieses Kreises soll mit dem Mittelpunkt des Rechtecks übereinstimmen. Der Kreis soll gerade so groß sein, dass er die Eckpunkte des Rechtecks berührt. Wir berechnen in Zeile 60 den Radius des Kreises über den Satz des Pythagoras und können dann den Kreis mit dem Konstruktor

    `Ellipse2D.Double`(x, y, w, h)

    erzeugen. Die Parameter beschreiben das den Kreis umschließende Rechteck und haben die selbe Bedeutung wie die analogen Parameter des Konstruktors `Rectangle2D.Double`: x und

y spezifizieren die x- und y-Koordinaten des linken oberen Eckpunktes, w ist die Breite des Rechtecks und h gibt die Höhe des Rechtecks an.

Der Kreis, den wir zeichnen wollen, soll mit einer Strichdicke gezeichnet werden, die viermal so groß, wie normal ist. Wir setzen diese Strichdicke über den Aufruf

```
g2.setStroke(new BasicStroke(4.0F));
```

in Zeile 66.

11. Zum Schluß zeichnen wir einen transparenten Kreis um den linken oberen Eckpunkt des Rechtecks. Der Kreis selber wird in Zeile 70 definiert. Um diesen Kreis mit einer teilweise transparenten Farbe zeichnen zu können, benutzen wir den vierstelligen Konstruktor der Klasse `Color`, der die folgende Form hat:

    `Color`(`float` *red*, `float` *green*, `float` *blue*, `float` *alpha*).

    Die Werte *red*, *green* und *blue* legen dabei die Anteile der Grundfarben Rot, Grün und Blau an der Gesamtfarbe fest, während der Wert *alpha* den *Deckungsgrad* der Farbe bestimmt. Dieser Deckungsgrad liegt in dem Intervall [0.0, 1.0]. Dabei entsprich ein Wert von 1.0 einer Farbe, die komplett undurchsichtig ist, während 0.0 für eine Farbe steht, die Durchsichtig ist.

    Mathematisch gibt der Deckungsgrad die Wahrscheinlichkeit dafür an, dass ein Pixel des mit dieser Farbe zu zeichnenden Objektes ein Pixel eines unter diesem Objekt liegenden anderen Objektes überdeckt.

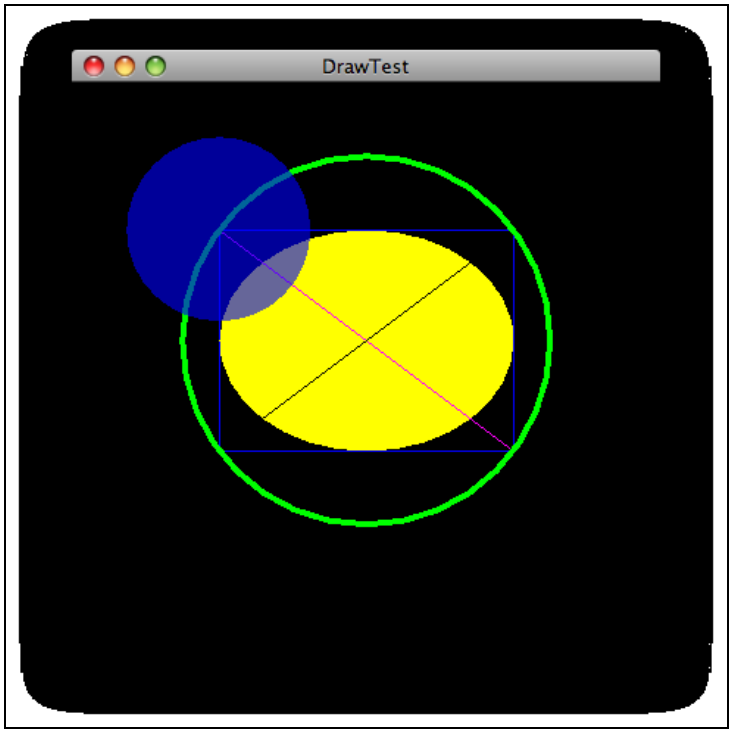

Abbildung 4: Ausgabe des in den Abbildung 1, 2 und 3 gezeigten Programms.

2 Zeichnen komplexerer Formen

Wir zeigen nun, wie sich komplexere Abbildungen in *Java* erstellen lassen. Abbildung 5 zeigt eine Grafik, die aus drei Elementen aufgebaut ist:

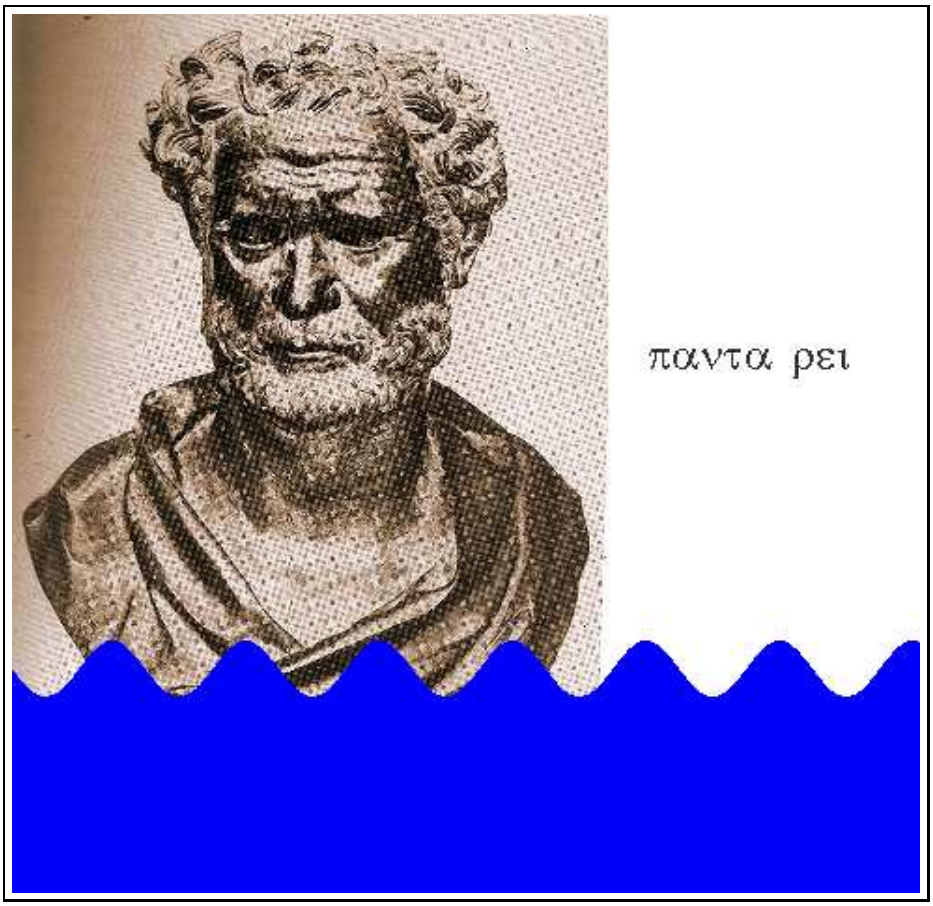

Abbildung 5: Eine komplexere Abbildung

1. einem Bild,
2. einem Text, der mit einem griechischen Zeichensatz gesetzt wurde und
3. einer wellenförmigen Figur in der unteren Hälfte des Bildes, die aus einer größeren Anzahl von Segmenten zusammen gesetzt worden ist.

Abbildung 6 auf Seite 7 zeigt zunächst die Import-Deklarationen und die Definition des auf der Klasse *JFrame* basierenden Rahmens. Bei der Definition der Klasse *DrawFrame2* gibt es gegenüber der Definition der Klasse *DrawFrame* aus Abbildung 1 auf Seite 2 nur einen Unterschied, der darin besteht, dass der Konstruktor der Klasse *DrawPanel* nun mit zwei Parametern aufgerufen wird, die die Breite und die Höhe des Rahmens angeben. Abbildung 7 auf Seite 8 zeigt die Implementierung der Klasse *DrawPanel*.

1. Zunächst zeichnen wir das Bild des griechischen Philosophen Heraklit. Das Bild liegt als `gif`-Datei vor. Mit dem Befehl in Zeile 16 kann das Bild aus dieser Datei eingelesen werden. Anschließend kann das Bild mit dem Befehl

   `drawImage`(*image*, x, y, `null`)

   gezeichnet werden. Hierbei geben die Parameter x und y die Koordinaten der linken oberen Ecke des Bildes an. Der vierte Parameter der Methode *drawImage*() gibt die Möglichkeit, einen sogenannten *ImageObserver* anzugeben. Hierbei handelt es sich um ein Objekt, dass regelmäßig über den Fortschritt beim Zeichnen des Bildes informiert wird. Einen solchen *ImageObserver* zu haben ist nur sinnvoll, wenn Bilder über ein Netz geladen werden und

```
1  import java.io.*;
2  import java.awt.*;
3  import java.awt.font.*;
4  import java.awt.geom.*;
5  import javax.swing.*;
6  import javax.imageio.*;
7
8  import static java.lang.Math.*;
9
10 class DrawFrame2 extends JFrame
11 {
12     public static void main(String[] args) {
13         DrawFrame2 frame = new DrawFrame2();
14     }
15
16     public DrawFrame2() {
17         setTitle("DrawTest");
18         setSize(WIDTH, HEIGHT);
19         DrawPanel panel = new DrawPanel(WIDTH, HEIGHT);
20         add(panel);
21         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
22         setVisible(true);
23     }
24     public static final int WIDTH  = 500;
25     public static final int HEIGHT = 500;
26 }
```

Abbildung 6: Zeichnen komplexerer geometrischer Formen

das Laden dadurch lange dauert. Wir benötigen keinen *ImageObserver* und geben daher als letzten Parameter `null` an.

Da beim Lesen der Bild-Datei ein *IOException* ausgelöst werden kann, wird der gesamte Code zum Lesen und Zeichnen des Bildes in einen `try-catch`-Block eingefaßt.

2. Als nächstes wollen wir den Text "*panta rei*" in griechischen Buchstaben setzen. Dazu öffnen wir in Zeile 22 die Datei "`greekfp.ttf`", die einen True-Type-Font für griechische Buchstaben enthält. Anschließend erzeugen wir in Zeile 23 mit der Methode *createFont*() den Font für griechische Buchstaben. In Zeile 24 skalieren wir den Font auf eine Größe von 24 *Punkten*. Ein *Punkt* ist dabei eine Längen-Einheit aus dem Druckgewerbe und entspricht der Länge von $\frac{1}{72}$ Inch. Ein Inch hat eine Länge von 2.54 Zentimetern, so dass ein Punkt eine Länge von $0.352\overline{7}$ Millimetern hat. Anschließend teilen wir in Zeile 25 unserem "Pinsel", also dem *Graphics2D*-Objekt `g2` mit, dass ab jetzt der eben erzeugte griechische Font zum Schreiben von Text benutzt werden soll. Das Schreiben des Textes geschieht mit der Methode

   `drawString`(msg, x, y).

   Diese Methode bewirkt, dass der Text *msg* so gesetzt werden soll, dass die linke obere Ecke eines Rechtecks, dass den Text umfaßt, die Koordinaten x und y hat.

   Da sowohl beim Lesen der Font-Datei als auch bei der Erzeugung des Fonts eine Ausnahme ausgelöst werden kann, fassen wir den Code in einem `try-catch`-Block ein.

3. Als letztes zeichnen wir den Fluss. Dazu erzeugen wir uns zunächst ein Objekt der Klasse *GeneralPath*. Für unsere Zwecke besteht ein *GeneralPath* aus einer geschlossenen Folge von

```
1  class DrawPanel extends JPanel
2  {
3      private int mWidth;
4      private int mHeight;
5
6      DrawPanel(int width, int height) {
7          mWidth  = width;
8          mHeight = height;
9      }
10     public void paintComponent(Graphics g)
11     {
12         Graphics2D g2 = (Graphics2D) g;
13         setBackground(new Color(1.0F, 1.0F, 1.0F));
14         super.paintComponent(g);
15         try {
16             Image image = ImageIO.read(new File("heraklit.gif"));
17             g2.drawImage(image, 0, 0,  null);
18         } catch (IOException e) {
19             e.printStackTrace();
20         }
21         try {
22             FileInputStream fis  = new FileInputStream("greekfp.ttf");
23             Font            font = Font.createFont(Font.TRUETYPE_FONT, fis);
24             font = font.deriveFont(24.0F);
25             g2.setFont(font);
26             g2.drawString("panta rei", 340, 190);
27         } catch (Exception e) {
28             e.printStackTrace();
29         }
30         GeneralPath path      = new GeneralPath();
31         float       startX    = 0.0F;
32         float       startY    = mHeight * 0.7F;
33         float       frequency = 7.0F;
34         float       amplitude = 15;
35         path.moveTo(startX, startY);
36         for (int i = 0; i <= mWidth; ++i) {
37             double y = amplitude * sin(2*PI*i*frequency/mWidth) + startY;
38             path.lineTo(i, (float)y);
39         }
40         path.lineTo(mWidth, mHeight);
41         path.lineTo(0, mHeight);
42         path.closePath();
43         g2.setPaint(Color.BLUE);
44         g2.fill(path);
45     }
46 }
```

Abbildung 7: Implementierung der Klasse *DrawPanel*

geraden Linien[1]. In Zeile 35 spezifizieren wir den Start-Punkt dieser Folge. Dieser Start-

[1]Im allgemeinen kann ein Objekt vom Typ *GeneralPath* auch noch quadratische und kubische-Bézier Kurven

Punkt ist auch gleichzeitig der *aktuelle Punkt* der Folge. In der anschließenden `for`-Schleife hängen wir mit der Methode

`lineTo`(x, y)

weitere Punkte an diese Linie an. Die Methode *lineTo*() erzeugt eine gerade Linie, die von dem *aktuellen Punkt* zu dem Punkt mit den Koordinaten (x, y) führt. Anschließend wird der aktuelle Punkt auf die Koordinaten (x, y) gesetzt.

In den Zeile 36 – 39 zeichnen wir eine Sinus-Kurve, die durch die Gleichung

$$y = a \cdot \sin(2 \cdot \pi \cdot f \cdot x)$$

beschrieben wird. Dabei gibt a die Amplitude und f die Frequenz der Kurve an. Um x in dem Fenster von 0 bis 1 laufen zu lassen, setzen wir für x den Wert $\frac{\texttt{i}}{\texttt{mWidth}}$ ein. In Zeile 40 verlängern wir die Kurve um eine gerade Linie, die zur unteren rechten Ecke des Bildschirms zeigt. In Zeile 41 fügen wir eine Linie an, die sich von der unteren rechten Ecke bis zur unteren linken Ecke des Bildschirms erstreckt. Wir schließen mit dem Aufruf der Methode

`closePath`()

die Figur: Die Methode *closePath*() fügt dem Pfad eine Linie hinzu, die von dem aktuellen Punkt zu dem Start-Punkt zurück geht. Der Start-Punkt ist der erste Punkt des Pfades, den wir in Zeile 35 mit der Methode *moveTo*() gesetzt haben. Anschließend füllen wir das von dem Pfad eingeschlossene Gebiet mit blauer Farbe.

3 Animierte Grafik

Das von dem Programm in Abbildung 7 gezeichnete Bild ist statisch und steht damit im Widerspruch zur Aussage der Philosophen Heraklit, der behauptet hat, dass alles fließt. Um diesen Widerspruch aufzulösen zeigen wir als nächstes, wie wir Grafiken animieren können. Abbildung 8 auf Seite 10 zeigt die Definition der Klasse *AnimatedFrame*. Hier ist der Code ab Zeile 20 neu. Zunächst laden wir in Zeile 21 eine animierte Gif-Datei, die ein zeitgenössisches Portrait des Philosophen zeigt, auf dem dieser in Bewegung ist. Hierzu benutzen wir die Klasse `ImageIcon`. Diese Klasse ist in der Lage, animierte Bilder zu verwalten. Um das Bild darzustellen, erzeugen wir in Zeile 27 ein Objekt der Klasse `JLabel`, deren Konstruktor mit dem Bild initialisiert wird.

Desweiteren animieren wir unsere Zeichnung des Flusses: Wir verändern dort in der Schleife in den Zeilen 32 - 41 immer wieder den Parameter `offset` des Objektes `panel`. Dieser Parameter bewirkt, dass die Grafik, die den Fluß repräsentiert, um `offset` nach rechts versetzt gezeichnet wird. Jedesmal, wenn wir den Parameter `offset` verändert haben, rufen wir anschließend die Methode `repaint`() auf. Dadurch veranlassen wir, dass die Grafik neu gezeichnet wird. Da das Zeichnen in einem anderen *Thread* statt findet, passiert das nicht sofort sondern erst dann, wenn die Kontrolle an den anderen *Thread* übergeben wird. Anschließend legen wir den Thread, der den Parameter geändert hat, durch den Aufruf der statischen Methode

`Thread.sleep`(d)

für d Millisekunden schlafen. Dadurch erhält der Thread, der für das Zeichnen zuständig ist, die Gelegenheit, die Grafik neu zu zeichnen. Außerdem können wir damit die Geschwindigkeit der Animation steuern.

Es ist hier wichtig zu wissen, welche Threads es gibt und welche Methoden in welchem Thread laufen. Jedes *Java*-Programm, das das Framework *Swing* benutzt, hat mindestens zwei Threads. Der erste Thread ist der eigentliche Programm-Thread, in dem die Methode *main*() aufgerufen wird. Der zweite Thread ist der GUI-Thread, der für die Darstellung der Fenster zuständig ist. Dieser Thread zeichnet die Fenster auf den Bildschirm. Dazu ruft dieser Thread die Methode *paintComponent*() auf. Der GUI-Thread hat allerdings eine geringere Priorität als der Programm-Thread. Er wird nur dann aktiviert, wenn der Programm-Thread gerade nichts zu tun hat. Wenn wir im Programm-Thread Parameter ändern und eine Grafik mit den geänderten Parametern neu

enthalten.

```
1  import java.io.*;
2  import java.awt.*;
3  import java.awt.geom.*;
4  import java.awt.image.*;
5  import javax.swing.*;
6  import javax.imageio.*;
7
8  import static java.lang.Math.*;
9
10 class AnimatedFrame extends JFrame
11 {
12     private ImageIcon image;
13
14     public static void main(String[] args) {
15         AnimatedFrame frame = new AnimatedFrame();
16     }
17     public AnimatedFrame() {
18         setTitle("panta rei");
19         setSize(WIDTH, HEIGHT);
20         try {
21             image = new Image("skeleton.gif");
22         } catch (Exception e) {
23             e.printStackTrace();
24         }
25         DrawPanel panel = new DrawPanel(WIDTH, HEIGHT);
26         add(panel);
27         JLabel heraklit = new JLabel(image);
28         panel.add(heraklit);
29         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
30         setVisible(true);
31         double x = 0.0;
32         while (true) {
33             panel.setOffset(x);
34             repaint();
35             try {
36                 Thread.sleep(DELAY);
37             } catch (InterruptedException e) {
38                 e.printStackTrace();
39             }
40             x += 0.01;
41         }
42     }
43     private static final int DELAY  =  20;
44     private static final int WIDTH  = 500;
45     private static final int HEIGHT = 500;
46 }
```

Abbildung 8: Zeichnen animierter Grafiken

zeichnen lassen wollen, so können wir mit dem Aufruf von *repaint*() der GUI signalisieren, dass die Grafik neu gezeichnet werden soll. Wir müssen der GUI aber auch eine Chance geben, ihre Arbeit zu erledigen. Dazu legen wir den Programm-Thread durch den Aufruf der statischen Methode

sleep() schlafen, so dass der GUI-Thread aktiviert werden kann.

```
1  class DrawPanel extends JPanel
2  {
3      private int       mWidth;
4      private int       mHeight;
5      private double    mOffSet;
6
7      public void setOffset(double offset) {
8          mOffSet = offset;
9      }
10
11     DrawPanel(int width, int height) {
12         mWidth       = width;
13         mHeight      = height;
14         mOffSet      = 0;
15     }
16
17     public void paintComponent(Graphics g)
18     {
19         Graphics2D g2 = (Graphics2D) g;
20         setBackground(new Color(1.0F, 1.0F, 1.0F));
21         super.paintComponent(g);
22
23         GeneralPath path      = new GeneralPath();
24         float       startX    = 0.0F;
25         float       startY    = mHeight * 0.7F;
26         float       frequency = 7.0F;
27         float       amplitude = 15;
28         path.moveTo(startX, startY);
29         for (int i = 0; i <= mWidth; ++i) {
30             double y = amplitude * sin(2*PI*(frequency*i/mWidth - mOffSet)) + startY;
31             path.lineTo(i, (float)y);
32         }
33         path.lineTo(mWidth, mHeight);
34         path.lineTo(0, mHeight);
35         path.closePath();
36         g2.setPaint(Color.BLUE);
37         g2.fill(path);
38     }
39 }
```

Abbildung 9: Implementierung der Klasse *DrawPanel*

Abbildung 9 auf Seite 11 zeigt die neue Implementierung der Klasse *DrawPanel*. Der wesentliche Unterschied gegenüber der alten Implementierung besteht darin, dass wir der Klasse den zusätzlichen Parameter `mOffSet` hinzugefügt haben. Dieser Parameter wird mit Hilfe der Methode *setOffset*() gesetzt. Wir zeichnen nun nicht mehr die Kurve

$$y = a \cdot \sin(2 \cdot \pi \cdot f \cdot x)$$

sondern statt dessen die um `mOffSet` nach rechts versetzte Kurve

$$y = a \cdot \sin(2 \cdot \pi \cdot f \cdot x - \texttt{mOffSet}).$$

Dadurch, dass wir den Parameter `mOffSet` alle 20 Millisekunden um 0.01 inkrementieren und

jedesmal das (leicht veränderte) Bild neu zeichnen, entsteht beim Betrachter der Eindruck eines bewegten Bildes.

Ansonsten hat sich gegenüber der alten Implementierung nichts wesentliches verändert: Um das Programm abzukürzen, verzichten wir allerdings darauf, den String “panta rei” auf den Bildschirm zu schreiben.

4 Eingabe von Daten

Wenn Algorithmen animiert werden sollen, dann ist es in der Regel erforderlich, Eingaben vom Benutzer an die Animation weiter zu reichen. Wir zeigen dies an einem Beispiel. Das Beispiel zeigt eine einfache Animation des mittelalterlichen Weltbilds, in der die Sonne um die Erde kreist. Dabei kann die Verzögerung (und damit die Geschwindigkeit des Kreisens) vom Benutzer verändert werden. Abbildung 10 auf Seite 12 zeigt die Grafik.

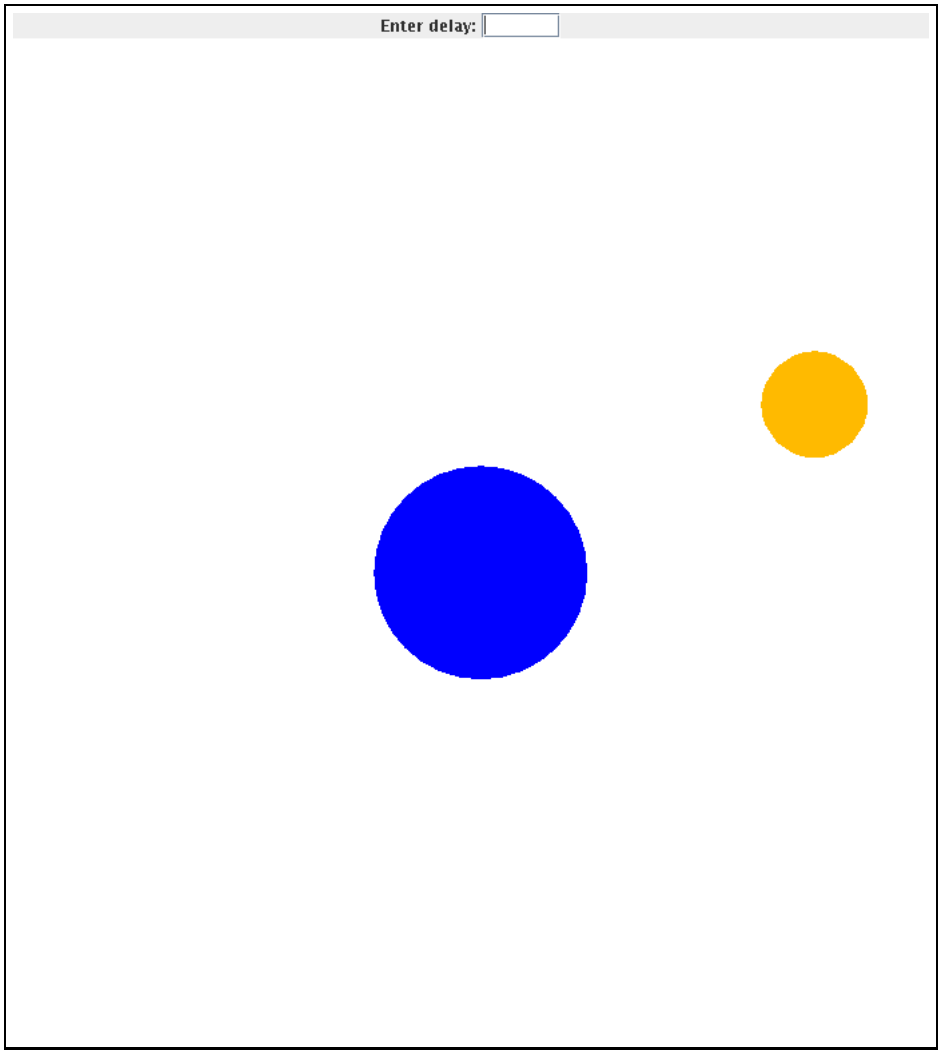

Abbildung 10: Vereinfachte Darstellung des mittelalterlichen Weltbildes

Um dieses Beispiel implementieren zu können brauchen wir eine Hilfsklasse `Wrap<T>`, die als Behälter für ein Objekt eines generischen Typs `T` fungiert. Abbildung 11 auf Seite 13 zeigt die Implementierung. Diese Klasse kann dazu benutzt werden, die *Call-by-Value*-Semantik der *Java*-Methoden-Aufrufe in eine *Call-by-Reference*-Semantik umzuwandeln. Wir zeigen später wie das

funktioniert.

```
1  class Wrap<T>
2  {
3      T mValue;
4
5      public void setValue(T value) {
6          mValue = value;
7      }
8      public T getValue() {
9          return mValue;
10     }
11 }
```

Abbildung 11: Die generische Klasse *Wrap*

Abbildung 12 auf Seite 14 zeigt die Implementierung der Klasse *PlanetFrame*. Diese Klasse beschreibt einen Rahmen, der aus zwei Teilen besteht. Dabei enthält der obere Teil ein Eingabe-Feld, in dem der Benutzer eine Zahl eingeben kann und der untere Teil enthält die Fläche, auf der später eine Animation zu sehen ist. Es geht darum, dass der Benutzer die Verzögerung der Animation interaktiv steuern kann. Das Eingabe-Feld wird in Zeile 6 als Objekt der Klasse *JTextField* deklariert, die Verzögerung wird in Zeile 7 als Objekt der Klasse `Wrap<Integer>` deklariert und wird in Zeile 16 zunächst auf den Wert 30 gesetzt. Vor das Eingabe-Feld haben wir einen Label gesetzt. Dieser wird als Objekt der Klasse *JLabel* in Zeile 17 definiert. Das Eingabe-Feld erzeugen wir in Zeile 18 als `JTextField(5)`. Dabei gibt die Zahl 5 die Breite des Eingabe-Feldes an: In unserem Fall ist das Feld so breit, dass 5 Buchstaben hineinpassen. Der Aufruf der Methode *setMaximumSize*() in Zeile 19 stellt sicher, dass das Eingabe-Feld auch nicht vergrößert werden kann. Zusätzlich müssen wir für das Eingabe-Feld einen sogenannten *ActionListener* definieren. Bei *ActionListener* handelt es sich um ein Interface, das die Existenz der Methode

`actionPerformed`(`ActionEvent` e)

vorschreibt. Nach jeder Eingabe in dem Eingabe-Feld ruft der Window-Manager diese Methode auf. Die Klasse *MyListener*, die in Abbildung 13 auf Seite 15 gezeigt wird, implementiert dieses Interface. Wir gehen später noch genauer auf die Implementierung dieser Klasse ein. In Zeile 20 erzeugen wir ein Objekt der Klasse `MyListener` und definieren dieses Objekt in Zeile 21 als das zu dem Eingabe-Feld gehörende *ActionListener*-Objekt.

Anschließend erzeugen wir in Zeile 22 ein Panel mit dem Namen `firstPanel`, in das wir das Label und das Eingabe-Feld nebeneinander setzen können. In Zeile 23 – 26 fügen wir das Label und das Eingabe-Feld in dieses Panel ein. Mit den Befehlen in Zeile 23 und Zeile 26 setzen wir vor das Label und hinter das Eingabe-Feld jeweils horizontalen Abstand. Dadurch werden Label und Eingabe-Feld zentriert gesetzt.

Als nächstes erzeugen wir in Zeile 27 das Panel `planetPanel`, das die eigentliche Animation enthält. In Zeile 28 erzeugen wir ein weiteres Panel mit dem Namen `topPanel`, in das wir die beiden Panels `firstPanel` und `planetPanel` einbinden. Damit wir die Position bestimmen können, an die `firstPanel` und `planetPanel` gesetzt werden, definieren wir in Zeile 29 den *Layout-Manager* des Panels als *BorderLayout*. Damit können wir dann in Zeile 30 angeben, dass `firstPanel` an den oberen "nördlichen" Rand von `topPanel` gesetzt wird und in Zeile 31 das `planetPanel` in das Zentrum von `topPanel` gesetzt wird. Als letztes wird das `topPanel` in Zeile 32 in den Rahmen eingebunden.

Der restliche Code des Konstruktors beschäftigt sich mit der Animation. Die Klasse *PlanetFrame* hat die Methode *setAngle*, mit der der Winkel verändert werden kann, unter dem die in der Grafik gezeigte Sonne am Himmel steht.

Als nächstes betrachten wir die Klasse *MyListener*, die wir innerhalb der Klasse *PlanetFrame*

```
class PlanetFrame extends JFrame
{
    private static final int WIDTH  = 700;
    private static final int HEIGHT = 800;

    private JTextField    mDelayField;
    private Wrap<Integer> mDelay;

    public static void main(String[] args) {
        PlanetFrame frame = new PlanetFrame();
    }
    public PlanetFrame() {
        setTitle("Planet Animation");
        setSize(WIDTH, HEIGHT);
        mDelay = new Wrap<Integer>();
        mDelay.setValue(30);
        JLabel labelDelay = new JLabel("Enter delay: ");
        mDelayField = new JTextField(5);
        mDelayField.setMaximumSize(mDelayField.getPreferredSize());
        MyListener listener = new MyListener(mDelayField, mDelay);
        mDelayField.addActionListener(listener);
        Box firstPanel = Box.createHorizontalBox();
        firstPanel.add(Box.createHorizontalGlue());
        firstPanel.add(labelDelay);
        firstPanel.add(mDelayField);
        firstPanel.add(Box.createHorizontalGlue());
        PlanetPanel planetPanel = new PlanetPanel(WIDTH, HEIGHT);
        JPanel topPanel = new JPanel();
        topPanel.setLayout(new BorderLayout());
        topPanel.add(firstPanel,  BorderLayout.NORTH);
        topPanel.add(planetPanel, BorderLayout.CENTER);
        add(topPanel);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setVisible(true);

        double angle = 0.0;
        while (true) {
            planetPanel.setAngle(angle);
            repaint();
            try {
                Thread.sleep(mDelay.getValue());
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            angle += 0.02;
        }
    }
}
```

Abbildung 12: Implementierung der Klasse *PlanetFrame*

```
49  class MyListener implements ActionListener
50  {
51      private JTextField    mTextField;
52      private Wrap<Integer> mWrapper;
53
54      MyListener(JTextField textField, Wrap<Integer> wrapper) {
55          mTextField = textField;
56          mWrapper   = wrapper;
57      }
58      public void actionPerformed(ActionEvent e) {
59          Integer value = Integer.parseInt(mTextField.getText().trim());
60          mWrapper.setValue(value);
61          mTextField.setText(null);
62      }
63  }
```

Abbildung 13: Die Implementierung eines *ActionListener*s

definiert haben. Diese Klasse hat zwei Member-Variablen:

1. `mTextField` ist das Eingabe-Feld, auf das der *ActionListener* hören soll.

2. `mWrapper` ist ein Objekt vom Typ `Wrap<Integer>` und liefert eine Referenz, über die ein mit der Klasse *Wrap* eingepackter Wert verändert werden kann.

In dem Konstruktor werden die beiden Member-Variablen gesetzt. In der Methode *actionPerformed*, die jedesmal aufgerufen wird, wenn der Wert in dem Eingabe-Feld geändert worden ist, passiert folgendes:

1. Zunächst lesen wir in Zeile 59 die vom Benutzer eingegebene Zahl. Wir erlauben dem Benutzer, dass er vor und hinter der Zahl noch Leerzeichen einfügt, die von der Methode *trim*() entfernt werden.

2. Anschließend setzen wir die in dem Wrap-Objekt abgespeicherte Zahl auf den vom Benutzer eingegebenen Wert. An dieser Stelle wird die Bedeutung der Klasse `Wrap` klar: Wenn wir in Zeile 20 das Objekt `listener` erzeugen, dann wollen wir erreichen, dass dieses Objekt in der Lage ist, die Variable `mDelay` zu verändern. In *Java* werden aber nur Werte übergeben. Hätten wir dem Konstruktor *MyListener* einfach ein Objekt vom Type *Integer* mitgegeben, so würde die Variable *mDelay* von späteren Änderungen dieses Objektes nichts merken. Daher haben wir *mDelay* als `Wrap<Integer>` definiert, denn in einem Objekt vom Typ `Wrap<Integer>` können wir den dort gespeicherten Wert mit Hilfe der Methode *setValue*() ändern.

3. Um dem Benutzer zu signalisieren, dass seine Eingabe verarbeitet worden ist, löschen wir in Zeile 61 den eingegebenen Text.

Die Implementierung der Klasse *PlanetPanel* wird in Abbildung 14 gezeigt. In den Zeilen 21 und 22 berechnen wir die Koordinaten des Mittelpunkts des Panels. An dieser Stelle wird in den Zeilen 24 – 28 die Erde als Kreis mit einem Radius von 80 Pixeln gezeichnet. Der Mittelpunkt der Sonne hat ausgehend vom Mittelpunkt der Erde die Koordinaten

$$\langle r \cdot \cos(\varphi), r \cdot \sin(\varphi) \rangle.$$

Der Winkel φ wird im Programm als `mAngle` bezeichnet, der Radius r entspricht der Variable `radiusOrbit`. In den Zeilen 40 – 42 wird die Farbe der Sonne abhängig von dem Winkel φ unter

```
1  class PlanetPanel extends JPanel
2  {
3      private int    mWidth;
4      private int    mHeight;
5      private double mAngle;
6
7      public void setAngle(double angle) {
8          mAngle = angle;
9      }
10     PlanetPanel(int width, int height) {
11         mWidth  = width;
12         mHeight = height;
13         mAngle  = 0;
14     }
15     public void paintComponent(Graphics g)
16     {
17         Graphics2D g2 = (Graphics2D) g;
18         setBackground(new Color(1.0F, 1.0F, 1.0F));
19         super.paintComponent(g);
20
21         int centerEarthX = mWidth  / 2;
22         int centerEarthY = mHeight / 2;
23         int radiusEarth  = 80;
24         Ellipse2D.Double earth =
25             new Ellipse2D.Double(centerEarthX - radiusEarth,
26                                  centerEarthY - radiusEarth,
27                                  2 * radiusEarth,
28                                  2 * radiusEarth);
29         double radiusOrbit = 0.40 * min(mWidth, mHeight);
30         double centerSunX  = centerEarthX + radiusOrbit * cos(mAngle);
31         double centerSunY  = centerEarthY + radiusOrbit * sin(mAngle);
32         double radiusSun   = 40;
33         Ellipse2D.Double sun =
34             new Ellipse2D.Double(centerSunX - radiusSun,
35                                  centerSunY - radiusSun,
36                                  2 * radiusSun,
37                                  2 * radiusSun);
38         g2.setPaint(Color.BLUE);
39         g2.fill(earth);
40         double red   = 1.0;
41         double green = 0.5 * (1.0 - sin(mAngle));
42         double blue  = 0;
43         g2.setPaint(new Color((float) red, (float) green, (float) blue));
44         g2.fill(sun);
45     }
46 }
```

Abbildung 14: Die Implementierung der Klasse *PlanetPanel*

dem die Erde am Horizont zu sehen ist, verändert. Die Rot-Wert der Farbe ist konstant bei 1.0, der Grün-Wert g variiert zwischen 0.0 und 1.0 gemäß der Formel

$$g(\varphi) = \tfrac{1}{2} \cdot \big(1 - \sin(\varphi)\big).$$

Dadurch erscheint die Sonne gelb, wenn der Winkel φ den Wert $\frac{\pi}{2}$ annimmt, denn $g\big(-\frac{\pi}{2}\big) = 1$. Für $\varphi = \frac{\pi}{2}$ gilt hingegen $g\big(\frac{\pi}{2}\big) = 0$ und daher wird die Sonne für diesen Wert rot dargestellt.

5 Anhalten der Animation

Gelegentlich möchte man dem Benutzer die Möglichkeit geben eine Animation zu unterbrechen und wieder zu starten[2]. Die Abbildungen 15 und 16 auf den Seiten 18 und 19 zeigen die Implementierung einer Animation, die sich mit einer Taste anhalten und wieder starten läßt. Gegenüber der in Abbildung 14 gezeigten Implementierungen ist folgendes hinzu gekommen.

1. In Zeile 8 definieren wir eine zusätzliche Member-Variable `mPause`. Die Variable ist ein mit *Wrap* verpackter Boolescher Wert. Die Animation läuft genau dann, wenn `mPause` den Wert "`false`" beinhaltet, sonst pausiert die Animation.

2. In Zeile 19 initialisieren wir `mPause` mit dem Wert "`false`", so dass die Animation sofort startet.

3. In Zeile 26 erzeugen wir ein Objekt vpm Typ *JButton*. Dabei handelt es sich um eine Schaltfläche, die mit dem Text "Pause" beschriftet ist.

4. In Zeile 27 erzeugen wir ein Objekt vom Typ *PauseListener*. Diese Klasse wird weiter unten definiert. Dieses Objekt registrieren wir dann in Zeile 28 als den zu der Schaltfläche "Pause" gehörenden *ActionListener*.

5. In Zeile 35 wird die Schaltfläche "Pause" auf das Panel `firstPanel` gesetzt.

6. Der Code zur Animation befindet sich in den Zeilen 48 – 60 der Abbildung 19. Neu hinzu gekommen ist die `if`-Abfrage in Zeile 57, wo wir den Wert von `mPause` testen. Nur wenn `mPause` den Wert `false` hat, wird der Winkel inkrementiert.

7. In den Zeilen 69 bis 87 zeigen wir die Implementierung der Klasse *PauseListener*, die dafür zuständig ist, den Wert von `mPause` bei jeder Betätigung der Schaltfläche "Pause" zu invertieren.

   (a) Die Klasse enthält zwei Member-Variablen: Den zu invertierenden Wert `mFlag` sowie die Schaltfläche *mButton*.

   (b) Die beiden Member-Variablen werden im Konstruktor initialisiert.

   (c) Die Methode *actionPerformed*() invertiert den Wert, der in dem Wrapper `mFlag` abgespeichert ist. Außerdem wird mit Hilfe der Methode *setText*() die Beschriftung der Schaltfläche von "Pause" zu "Run" und von "Run" wieder zu "Pause" geändert.

[2] Dies ist bei dem letzten Beispiel dann wichtig, wenn biblische Ereignisse (z.B. Josua 10:12) nachgespielt werden sollen.

```
1  class PlanetFrame2 extends JFrame
2  {
3      private static final int WIDTH  = 700;
4      private static final int HEIGHT = 800;
5
6      private JTextField    mDelayField;
7      private Wrap<Integer> mDelay;
8      private Wrap<Boolean> mPause;
9
10     public static void main(String[] args) {
11         PlanetFrame2 frame = new PlanetFrame2();
12     }
13     public PlanetFrame2() {
14         setTitle("Planet Animation);
15         setSize(WIDTH, HEIGHT);
16         mDelay = new Wrap<Integer>();
17         mDelay.setValue(30);
18         mPause = new Wrap<Boolean>();
19         mPause.setValue(false);
20
21         JLabel labelDelay = new JLabel("Enter delay: ");
22         mDelayField = new JTextField(5);
23         mDelayField.setMaximumSize(mDelayField.getPreferredSize());
24         MyListener myListener = new MyListener(mDelayField, mDelay);
25         mDelayField.addActionListener(myListener);
26         JButton       pauseButton   = new JButton("Pause");
27         PauseListener pauseListener = new PauseListener(mPause, pauseButton);
28         pauseButton.addActionListener(pauseListener);
29
30         Box firstPanel = Box.createHorizontalBox();
31         firstPanel.add(Box.createHorizontalGlue());
32         firstPanel.add(labelDelay);
33         firstPanel.add(mDelayField);
34         firstPanel.add(Box.createHorizontalGlue());
35         firstPanel.add(pauseButton);
36         firstPanel.add(Box.createHorizontalGlue());
37
38         PlanetPanel planetPanel = new PlanetPanel(WIDTH, HEIGHT);
39
40         JPanel topPanel = new JPanel();
41         topPanel.setLayout(new BorderLayout());
42         topPanel.add(firstPanel,  BorderLayout.NORTH);
43         topPanel.add(planetPanel, BorderLayout.CENTER);
44
45         add(topPanel);
46         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
47         setVisible(true);
```

Abbildung 15: Implementierung der Klasse *PlanetFrame2*, Teil I

```
48          double angle = 0.0;
49          while (true) {
50              planetPanel.setAngle(angle);
51              repaint();
52              try {
53                  Thread.sleep(mDelay.getValue());
54              } catch (InterruptedException e) {
55                  e.printStackTrace();
56              }
57              if (!mPause.getValue().booleanValue()) {
58                  angle += 0.02;
59              }
60          }
61      }
62  }
63
64  class MyListener implements ActionListener
65  {
66      // Code wir vorher
67  }
68
69  class PauseListener implements ActionListener
70  {
71      private Wrap<Boolean> mFlag;
72      private JButton       mButton;  // the associated button
73
74      PauseListener(Wrap<Boolean> flag, JButton button) {
75          mFlag   = flag;
76          mButton = button;
77      }
78      public void actionPerformed(ActionEvent e) {
79          Boolean flag = mFlag.getValue();
80          mFlag.setValue(!flag.booleanValue());
81          if (flag) {
82              mButton.setText("Pause");
83          } else {
84              mButton.setText("Run");
85          }
86      }
87  }
```

Abbildung 16: Implementierung der Klasse *PlanetFrame* Teil II

5.1 Probleme beim Multi-Threading

Wie in Abschnitt 3 bereits angesprochen, laufen bei jedem *Swing*-Programm zwei Threads:

1. Der eigentliche Programm-*Thread* wird von der Methode *main*() gestartet.
2. Zusätzlich gib es den GUI-Thread, in dem sowohl die Methode *paintComponent*() als auch die Methode *actionPerformed*() ausgeführt wird.

Der GUI-Thread ist also nicht nur für das Zeichnen der grafischen Oberfläche zuständig, sondern auch für die Abarbeitung von GUI-Ereignissen. Das heißt insbesondere, dass die Methode *actionPerformed*() von dem GUI-Thread ausgeführt wird. Deswegen sollte in einer solchen Methode nur Parameter gesetzt werden, die den Ablauf der Animation steuern; die eigentliche Animation darf keinesfalls in einer solchen Methode implementiert werden. Das liegt daran, dass es keinen Sinn macht, in einem *ActionListener* die Methode *repaint*() aufzurufen, denn der Aufruf von *repaint*() bewirkt ja lediglich, dass an den GUI-Thread eine Anforderung geschickt wird, den Bildschirm bei der nächsten Gelegenheit neu zu zeichnen. Da dies aber der selbe Thread ist, der auch die Methode *actionPerformed*() ausführt, kann der entsprechende Aufruf von *paintComponent*() erst dann durchgeführt werden, wenn die Methode *actionPerformed*() beendet wird. Hier kann dann auch ein Aufruf von *sleep*() nicht helfen, weil dieser Aufruf genau den Thread schlafen legen würde, der die Methode *paintComponent*() ausführen soll. Daher ist darauf zu achten, dass die Animation in dem *Programm-Thread* durchgeführt wird und nicht in dem GUI-Thread.

Falls Sie versuchen, die Animation doch innerhalb von Methoden zu implementieren, die im GUI-Threads laufen, so werden Sie statt der Animation immer nur das letzte Bild der Animation sehen.