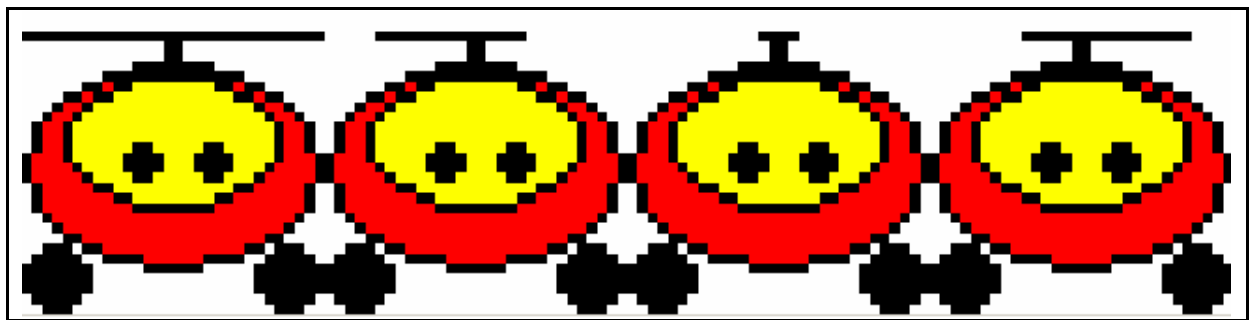


Spiele-Programmierung in Java

von Ralf Bauer (aka Quaxli)



Zum Inhalt:

Vorwort	4
Teil1: Los geht's	5
Das Spielfeld.....	5
Initialisierungen	8
Die Spielschleife.....	9
Zeitmessung	10
Weitere Methoden:	13
Die Klasse Sprite	14
Interface Movable	14
Interface Drawable	14
Die Methode drawObject(Graphics g)	16
Die Methode doLogic(long delta)	16
Die Methode move(long delta).....	17
Laden der Bilddaten	19
Das erste bewegte Objekt	21
doLogic():	22
moveObjects():.....	22
paintComponent(Graphics g):.....	22
Und so sieht's aus:	23
Die Steuerung.....	24
Und so sieht's aus:	28
Erster Feinschliff	29
Jetzt wird's abstrakt.....	29
Die neue Klasse Heli	31
Änderung des Programm-Starts.....	33
Weitere Objekte	38
Und so sieht's aus:	41
Weitere Objekte	42
Die Klasse Rocket	44
Die Logic-Methode:	45
Modifikationen der Klasse GamePanel:.....	47
Das Erzeugen der Raketen:.....	50
Und so sieht's aus:	52
Feinschliff 2. Teil	53
Objekte wieder löschen.....	53
Hintergrundbild	58
Und so sieht's aus:	61
Kollisionen.....	62
Abstrakte Methoden	62
So sieht's aus:	67
Spiel-Ende signalisieren	67
Explosionen	70
Und so sieht's aus:	74
Pixelgenaue Kollisionsermittlung.....	75
Und so sieht's aus:	77
Sound.....	78
Die Klasse SoundLib	79
Verwenden der Klasse SoundLib.....	80
Teil2 – tile-basierte Karten	83

GamePanel wird abstract	84
Die neue Hauptklasse	86
Ein erster Test	88
Und so sieht's aus:	89
Wiederverwendung bereits erstellter Klassen:	90
Eine SpriteLib	90
Laden eines einzelnen Images	91
Laden von Animationen	93
Das Spieler-Objekt	96
Und so sieht's aus:	101
Die Karte	102
Die Tiles	102
Die Datendatei	102
Verwaltung der Bilddaten	105
Das eigentliche Tile	108
Zusammenführung der einzelnen Komponenten zur Karte	109
Zwischentest	113
So sieht's aus:	115
Das Zeichnen der ganzen Karte	116
Und so sieht's aus:	117
Bewegung	118
So sieht's aus:	122
Steuerung	122
Anzeigeoptimierung	126
So sieht's aus:	134
Kollisionen	135
Änderung der Steuerung	135
Die „Schatten-Karte“	138
Abfrage der Farbinformationen	143
Verarbeitung der Informationen	145
Das war's	154
Nachträgliche Änderung	155
Fertig	156

Vorwort

Sinn und Zweck dieses Tutorials ist es, eine Möglichkeit aufzuzeigen, wie man Spiele bzw. Animationen in Java programmiert. Es handelt sich dabei aber nur um (m)eine Herangehensweise – andere Vorgehensweisen sind jederzeit denkbar und je nach Vorhaben, besser geeignet.

Das Tutorial ist für Einsteiger gedacht, daher wird auf einige Feinheiten vorerst verzichtet. Grundkenntnisse der Java-Programmierung werden aber vorausgesetzt. Darunter fällt vor allem, dass bekannt ist, wie man die API liest!!! ☺

Auf das Verwenden von Packages habe ich im ersten Beispiel bewusst verzichtet.

Das Tutorial wurde auf Basis Java 1.5 erstellt. Unbekannte oder nicht komplett dargestellte Methoden bitte ich in der entsprechenden API nachzulesen.

Die verwendete IDE ist Eclipse.

Teil1: Los geht's

Das Spielfeld

Das Spielfeld ist die Anzeigekomponente, hier wird „die ganze Action ablaufen“. Gleichzeitig ist es unsere Hauptklasse, welche die main-Methode und wichtige Methoden zur Initialisierung enthält.

Das Spielfeld erbt von JPanel, um eine vorhandene doppelt gepufferte Komponente zu verwenden und unnötigen Programmieraufwand zu sparen. Für komplexere Spiele wäre es denkbar, ein Canvas zu verwenden, um durch aktives Rendering die Animationen flüssiger ablaufen zu lassen, aber für kleinere Anwendungen ist ein JPanel ausreichend.

Im Konstruktor sollen lediglich Breite und Höhe des Spielfeldes übergeben werden.

```
1 import javax.swing.JPanel;
2
3 public class GamePanel extends JPanel{
4
5     private static final long serialVersionUID = 1L;
6
7
8     public GamePanel(int w, int h){
9
10    }
11
12
13 }
14
```

Als nächstes fügen wir eine main-Methode ein und hinterlegen im Konstruktor den Code um ein Fenster zu erzeugen, in das wir unser GamePanel reinpacken. Auch das Fenster ist aus dem Swing-Paket. Hier bitte keinen Frame verwenden, sondern einen JFrame, um das unnötige Mischen von leicht- und schwergewichtigen Komponenten zu vermeiden.

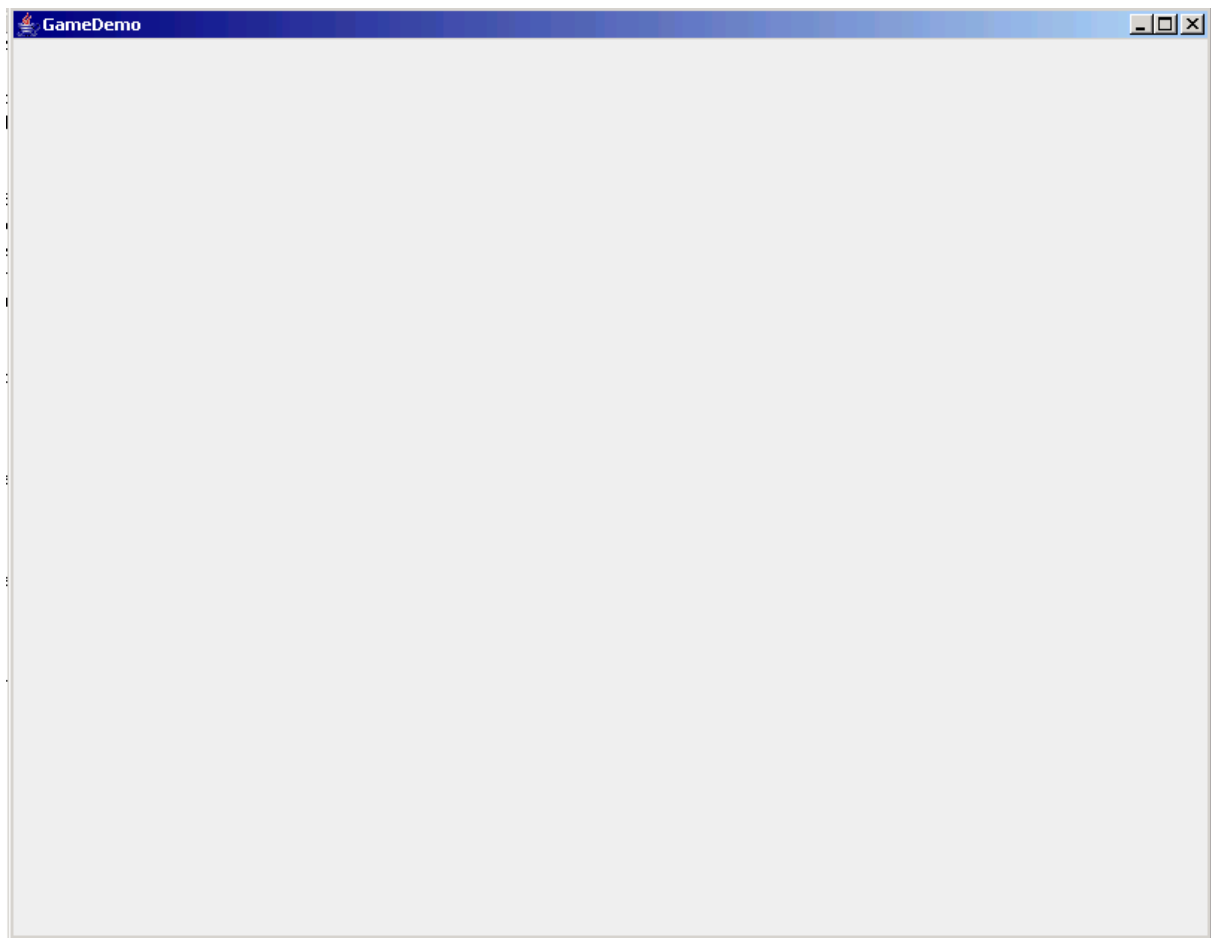
```
15 public class GamePanel extends JPanel implements Runnable, KeyListener{
16
17     private static final long serialVersionUID = 1L;
18
19     public static void main(String[] args){
20         new GamePanel(800,600);
21     }
22
23     public GamePanel(int w, int h){
24         this.setPreferredSize(new Dimension(w,h));
25         JFrame frame = new JFrame("GameDemo");
26         frame.setLocation(100,100);
27         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
28         frame.add(this);
29         frame.pack();
30         frame.setVisible(true);
31     }
32 }
```

Die markierten Zeilen enthalten den neu eingefügten Code.

Zeile 19 – 21: die main-Methode als Einsprungpunkt und Instanziierung unseres GamePanels. Hierbei wird für das GamePanel eine Breite von 800 Pixeln und eine Höhe von 600 Pixeln übergeben.

Zeile 24 – 30: Mit der von JPanel geerbten Methode setPreferredSize(...) übergeben wir die gewünschte Größe an unser GamePanel. Anschließend wird ein Fenster erzeugt und das GamePanel eingebunden. Über die pack()-Methode wird das Fenster an die gewünschte Größe des GamePanels angepasst.

Das Ergebnis unserer Mühen sieht bis jetzt so aus:



Noch nicht wirklich toll, aber das ändert sich in Kürze. 😊

Initialisierungen

Als nächstes basteln wir uns eine Methode für „einmalige Angelegenheiten“, wie das Laden von Images, etc.. . Diese nennen wir entsprechend `doInitializations()`. Im weiteren Verlauf werden wir diese noch weiter ausbauen:

```
1 import java.awt.Dimension;
2 import javax.swing.JFrame;
3 import javax.swing.JPanel;
4
5 public class GamePanel extends JPanel{
6
7     private static final long serialVersionUID = 1L;
8
9     public static void main(String[] args){
10         new GamePanel(800,600);
11     }
12
13     public GamePanel(int w, int h){
14         this.setPreferredSize(new Dimension(800,600));
15         JFrame frame = new JFrame("GameDemo");
16         frame.setLocation(100,100);
17         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
18         frame.add(this);
19         frame.pack();
20         frame.setVisible(true);
21         doInitializations();
22     }
23
24     private void doInitializations(){
25
26     }
27
28 }
29
```

`doInitializations()` werden wir sofort aufrufen, wenn das Fenster erzeugt ist. Da diese Methode später auch das Laden von Grafikdateien enthalten soll, haben wir dann zum Start des Programms z. B. alle Bilder geladen und müssen nicht mit Performance-Einbußen rechnen, wenn wir das während des Spiels machen wollten. Außerdem werden wir `doInitializations()` auch nach einem Neustart eines Spiels durch Tastatureingabe aufrufen – dazu später mehr.

Die Spielschleife

Kommen wir zur sog. Spielschleife oder neudeutsch: GameLoop. Innerhalb dieser Schleife werden wiederholt alle notwendigen Prüfungen vorgenommen und das Spiel neu gezeichnet oder vielmehr das Neuzeichnen angestoßen.

Um zu gewährleisten, dass unser Spiel „rund“ läuft, packen wir diesen Teil in einen eigenen Thread – mit Hilfe des Interface Runnable.

```
1 import java.awt.Dimension;
2
3
4
5 public class GamePanel extends JPanel implements Runnable{
6
7     private static final long serialVersionUID = 1L;
8     boolean game_running = true;
9
10    public static void main(String[] args){
11        new GamePanel(800,600);
12    }
13
14    public GamePanel(int w, int h){
15        this.setPreferredSize(new Dimension(800,600));
16        JFrame frame = new JFrame("GameDemo");
17        frame.setLocation(100,100);
18        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
19        frame.add(this);
20        frame.pack();
21        frame.setVisible(true);
22        doInitializations();
23    }
24
25    private void doInitializations(){
26
27        Thread t = new Thread(this);
28        t.start();
29    }
30
31    public void run() {
32
33        while(game_running){
34
35
36            try {
37                Thread.sleep(10);
38            } catch (InterruptedException e) {}
39
40
41        }
42
43    }
44
45
46
47 }
```

Zeile 5: Hier implementieren wir das Interface Runnable in unser GamePanel, damit wir unsere Spielschleife in einem eigenen Thread laufen lassen können.

Zeile 8: Hier definieren wir einen boolean, mit dessen Hilfe wir unsere Spielschleife später elegant beenden können, indem wir diesen auf false setzen und die Methode fertig laufen lassen.

Zeile 31 – 43: Dies ist die Ausprägung der Methode run() aus dem Interface Runnable. Sie enthält unsere Spielschleife. Innerhalb der dargestellten while-Schleife werden wir periodisch die notwendigen Methoden aufrufen, die unser Spiel am Leben erhalten. Damit auch andere Prozesse nicht zu kurz kommen, wird der Thread jeweils für 10 Millisekunden schlafen gelegt.

Zeile 27 – 28: Hier wird unsere Spielschleife in Ihren eigenen Thread gepackt.

Zeitmessung

Als nächstes werden wir eine Zeitmessung implementieren und ausgeben. Die Durchlaufzeit für die einzelnen Schleifendurchläufe werden wir später noch verwenden, z. B. um unsere Sprites in Abhängigkeit des letzten Schleifendurchlaufs zu bewegen und damit eine gleichmäßige Bewegung zu gewährleisten.

Aus Gründen der Übersichtlichkeit werde ich jetzt nur noch Code-Ausschnitte einfügen.

```
8 public class GamePanel extends JPanel implements Runnable{
9
10     private static final long serialVersionUID = 1L;
11     boolean game_running = true;
12
13     long delta = 0;
14     long last = 0;
15     long fps = 0;
16
17     public static void main(String[] args){
```

Folgende Klassenvariablen werden implementiert:

delta: zur Errechnung der Zeit, die für den letzten Durchlauf benötigt wurde

last: Speicherung der letzten Systemzeit

fps: Für die Errechnung der Bildrate (frames per second)

Die Variable last setzen wir doInitializations() auf Null - unserer Methode für alle notwendigen Voreinstellungen.

```

32 private void doInitializations() {
33
34     last = System.nanoTime();
35
36     Thread t = new Thread(this);
37     t.start();
38 }
39

```

Danach erweitern wir die Spielschleife um 2 Methodenaufrufe:

```

40 public void run() {
41
42     while (game_running) {
43
44         computeDelta();
45
46         repaint();
47
48         try {
49             Thread.sleep(10);
50         } catch (InterruptedException e) {}
51
52     }
53 }
54
55 }
56

```

Zeile 44: `computeDelta()`. Hier werden wir die Zeit für den jeweils vorhergehenden Schleifendurchlauf errechnen.

Zeile 46: `repaint()`. Von `Component` geerbte Methode, die ein Neuzeichnen anstößt – dieses wird aber nicht notwendigerweise sofort ausgeführt (vgl. API)

Die Methode `computeDelta()` erhält folgenden Code:

```

68 private void computeDelta() {
69
70     delta = System.nanoTime() - last;
71     last = System.nanoTime();
72
73     fps = ((long) 1e9) / delta;
74 }

```

Zeile 70: Errechnen der Zeit für den Schleifendurchlauf in Nanosekunden

Zeile 71: Speicherung der aktuellen Systemzeit

Zeile 73: Errechnung der Framerate

Weiterhin überschreiben wir jetzt noch die `paintComponent`-Methode unseres Panels um individuelle Zeichenoperationen vornehmen zu können. Vorerst begnügen wir uns aber damit die Framerate an den oberen Rand zu malen.

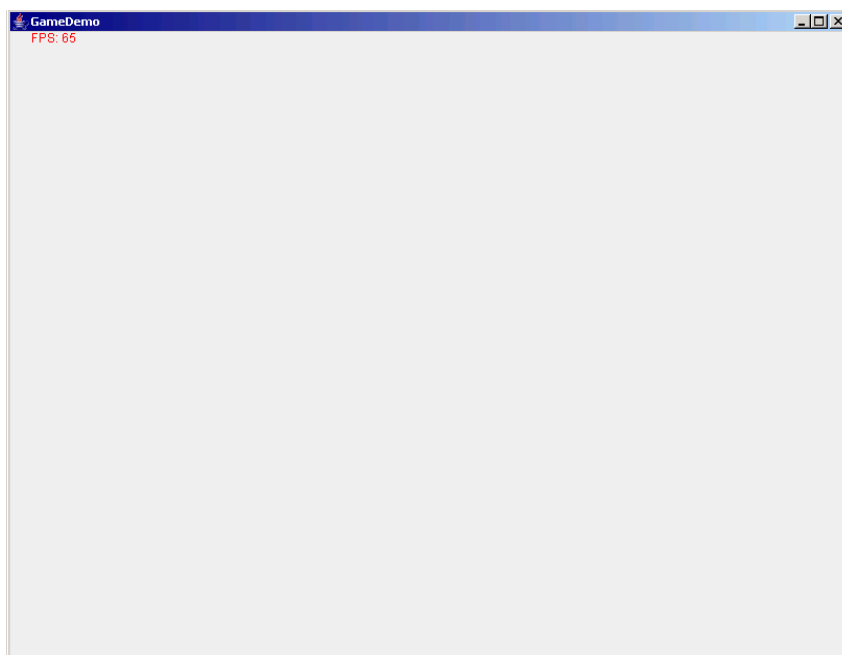
```
124 @Override
125 public void paintComponent(Graphics g) {
126     super.paintComponent(g);
127
128     g.setColor(Color.red);
129     g.drawString("FPS: " + Long.toString(fps), 20, 10);
130
131 }
```

Man kann darüber streiten, ob es notwendig ist, die Framerate anzuzeigen. Beim fertigen Spiel wird man sie sicherlich nicht verwenden. Im Verlauf der Spielentwicklung kann sie bei komplexen Berechnungen aber durchaus notwendig sein, um zu prüfen, ob die Animation weiterhin einigermaßen flüssig ausgeführt wird.

Wichtig:

1. Es ist wichtig, dass die `paintComponent`-Methode der Superklasse am Anfang aufgerufen wird. So vermeidet man, dass zusätzliche Komponenten, wie z. B. Menüs u.U. nicht gezeichnet werden.
2. Das `Graphics`-Objekt, welches hier vom System übergeben wird, wird später per Methodenaufruf an alle Objekte weitergegeben. Es ist sozusagen die Referenz auf die Zeichenfläche unseres `JPanel`s
3. Das `Graphics`-Objekt sollte immer mit übergeben werden und nie, nie, nie mit `getGraphics()` abholt werden (Ausnahmen bestätigen diese Regel 😊). Dies hilft unnötige Fehler zu vermeiden. Genauer bitte in diversen Foren nachlesen. (Empfehlung: www.java-forum.org 😊)

So sieht das Ganze bis jetzt aus:



Weitere Methoden:

Als nächstes sollen die für den Spielablauf notwendigen Methoden angelegt werden.

```
40 public void run() {  
41  
42     while(game_running) {  
43  
44         computeDelta();  
45         checkKeys();  
46         doLogic();  
47         moveObjects();  
48  
49  
50         repaint();  
51  
52         try {  
53             Thread.sleep(10);  
54         } catch (InterruptedException e) {}  
55  
56  
57     }  
58  
59 }  
60
```

Dazu wurden die folgenden Methoden angelegt:

Zeile 45: checkKeys(): Wird später zur Abfrage von Tastatureingaben verwendet

Zeile 46: doLogic(): Wird später zur Ausführung von Logik-Operation herangezogen

Zeile 47: moveObjects(): Verwenden wir später um unsere Objekte zu bewegen

Im Verlauf des Tutorials wird auf diese Methoden näher eingegangen, wir werden diese dann nach und nach mit Leben füllen.

Die Klasse Sprite

Kommen wir nun zu den bewegten Objekten. Hier wird zunächst einiges an Vorarbeit notwendig sein, bis wir unser erstes Objekt im Fenster anzeigen können – dafür haben wir es später etwas leichter.

Um für die Klasse Sprite einheitliche Methoden zur Verfügung zu stellen, definieren wir zunächst 2 Interfaces. Auch dies ist zunächst einmal zusätzlicher Aufwand (zugegebenermaßen relativ gering) wird uns aber später das Leben erleichtern.

Interface Movable

```
2 public interface Movable {  
3  
4     public void doLogic(long delta);  
5  
6     public void move(long delta);  
7  
8 }
```

Das Interface enthält 2 Methoden zum Bewegen unserer Objekte: Die Methode `move(long delta)` für die eigentliche Bewegung und die Methode `doLogic(long delta)` für Logikoperationen, wie z. B. Kollisionserkennung, etc.

Die Namensgebung wurde so gewählt, dass eindeutig ist, aus welcher Methode unserer Spielschleife diese Methoden aufgerufen werden, wenn wir sie im Sprite implementiert haben.

Interface Drawable

```
1 import java.awt.Graphics;  
2  
3 public interface Drawable {  
4  
5     public void drawObjects(Graphics g);  
6  
7 }
```

Das Interface Drawable stellt analog zum Interface Movable eine einheitliche Methode zum Zeichnen unseres Objektes zur Verfügung.

Jetzt endlich erstellen wir die Klasse Sprite. Sie soll die beiden Interfaces implementieren und uns darüber hinaus die Möglichkeit zur Verfügung stellen, Animation aus einem Image heraus zu erzeugen.

```
1 import java.awt.Graphics;
4
5 public class Sprite extends Rectangle2D.Double implements Drawable, Movable{
6
7     long delay;
8
9     GamePanel parent;
10    BufferedImage[] pics;
11    int currentpic = 0;
12    public Sprite(BufferedImage[] i, double x, double y, long delay, GamePanel p ){
13        pics = i;
14        this.x = x;
15        this.y = y;
16        this.delay = delay;
17        this.width = pics[0].getWidth();
18        this.height = pics[0].getHeight();
19        parent = p;
20    }
21
22
23    public void drawObjects(Graphics g) {
24    }
25
26    public void doLogic(long delta) {
27    }
28
29
30    public void move(long delta) {
31    }
32 }
```

Die Klasse Sprite erbt von Rectangle2D.Double, dadurch haben wir die Möglichkeit die aktuellen x- und y-Parameter genau zu errechnen und haben auch gleich einige grundlegenden Parameter und Methoden zur Verfügung. Unter anderem auch um später Kollisionen ermitteln zu können. (intersects(..)).

Zeile 7: Instanzvariable delay um das umschalten zwischen den einzelnen Bildern unseres Image-Arrays zu steuern. Die Variable soll einen Wert im Millisekundenbereich erhalten!

Zeile 9: Referenz auf unser GamePanel

Zeile 10: BufferedImage-Array zum Speichern der Animation

Zeile 11: Zähler für das aktuell anzuzeigende Bild

Zeile 17-24: Der Konstruktor mit der Übergabe des ImageArrays für die Animation, den Positionswerten, der Verzögerung für die Animation und der Referenz auf unser GamePanel. Außerdem holen wir uns aus dem ersten Bild unseres Array die Höhe und die Breite des Bildes (in der Annahme, dass diese durchgehend gleich bleiben).

Die Methode drawObject(Graphics g)

Die Methode drawObject(Graphics g) bekommt später das Graphics-Objekt unseres GamePanel übergeben um darauf zeichnen zu können. Innerhalb der Methode wird jetzt das aktuelle Bild gezeichnet.

```
21 public void drawObjects(Graphics g) {  
22     g.drawImage(pics[currentpic], (int) x, (int) y, null);  
23 }
```

Der x- und y-Parameter wird hier auf ganze Zahlen herunter gebrochen bzw. nach int gecastet. Das ist notwendig, weil Graphics ganze Zahlen verlangt (es gibt ja auch keine halben Pixel). Um die Positionen unsere Objekte genau berechnen zu können macht es aber Sinn, diese als Double-Werte vorzuhalten (daher u.a. Rectangle2D.Double).

Wichtig ist an dieser Stelle, sich das Graphics-Object nicht mit getGraphics() zu holen, da dies – kurz gesagt – unnötige Probleme verursachen kann (nicht muß). Mit der Übergabe des Graphics-Objektes ist man auf jeden Fall auf der sicheren Seite.

Die Methode doLogic(long delta)

Innerhalb der Methode doLogic hinterlegen wir vorerst nur den Code für die Animation des Objektes. Später werden wir hier zusätzlichen Code einbauen. Da wir hier die Zeit kumulieren müssen, legen wir uns eine weitere Instanzvariable namens animation an.

```
6 public class Sprite extends Rectangle2D.Double implements Drawable, Movable{  
7  
8     long delay;  
9     long animation = 0;  
10    GamePanel parent;  
11    BufferedImage[] pics;  
12    int currentpic = 0;  
..
```

Die Methode doLogic(long delta) prägen wir wie folgt aus:

```
29 public void doLogic(long delta) {  
30  
31     animation += (delta/1000000);  
32     if (animation > delay) {  
33         animation = 0;  
34         computeAnimation();  
35     }  
36  
37 }
```

Die Methode bekommt die Dauer des letzten Schleifen-Durchlaufs übergeben (vgl. Klasse GamePanel). Diesen kumulieren wir in der Variable animation. Delta wird hier durch 1000000 dividiert, weil wir hier einen Wert übergeben bekommen, der in Nanosekunden berechnet wurde. Für die bequemere Einstellung

der Animationsgeschwindigkeit sollen aber Millisekunden verwendet werden – daher die Umrechnung.

Ist der Wert der Variable animation größer als der voreingestellte Animationswert, setzen wir animation wieder auf Null und rufen die Methode computeAnimation() über die das nächste Bild ermittelt wird.

Die Methode doAnimation() ist momentan sehr sparsam ausgelegt, kann aber bei der Entwicklung komplexerer Spiele viel umfangreicher werden, beispielsweise könnte man entsprechenden Code hinterlegen, um die Animation rückwärts laufen zu lassen oder ähnliches. Auch wir werden diese Methode später noch etwas modifizieren.

```
41 private void computeAnimation() {  
42  
43     currentpic++;  
44  
45     if(currentpic>=pics.length) {  
46         currentpic = 0;  
47     }  
48  
49 }
```

Momentan wird unsere Variable currentpic bei jedem Aufruf um eins erhöht. Wenn Sie die Anzahl der vorhandenen Bilder überschreitet (Arrays beginnen mit dem Zähler 0!), wird sie auf Null gesetzt, d. h. die Animation beginnt von vorne.

Die Methode move(long delta)

Um unser Objekt bewegen zu können, müssen wir ihm noch mitteilen, wie schnell diese Veränderung stattfinden soll. Daher werden zwei weitere Instanzvariablen angelegt:

```
14 protected double dx;  
15 protected double dy;
```

- dx wird den Wert für die horizontale Veränderung enthalten (x-Wert)
- dy wird den Wert für die vertikale Veränderung enthalten (y-Wert)

Für diese Bewegungs-Variablen legen wir auch gleich noch die entsprechenden get- und set-Methoden an (geht in Eclipse z. B. relative automatisiert):

```
63 public void setVerticalSpeed(double d) {  
64     dy = d;  
65 }  
66  
67 public void setHorizontalSpeed(double d) {  
68     dx = d;  
69 }  
70  
71 public double getVerticalSpeed() {  
72     return dy;  
73 }  
74  
75 public double getHorizontalSpeed() {  
76     return dx;  
77 }
```

Jetzt aber die eigentliche Methode move(long delta):

```
39 public void move(long delta) {  
40  
41     if(dx!=0) {  
42         x += dx*(delta/1e9);  
43     }  
44  
45     if(dy!=0) {  
46         y += dy*(delta/1e9);  
47     }  
48  
49 }
```

Wenn unsere Delta-Werte nicht Null sind, verändern wir die Position des Objekts in die entsprechende Richtung – abhängig von der Zeit, die der letzte Durchlauf unseres GameLoop benötigt hat. Durch die Berücksichtigung der Durchlaufzeit wird eine gleichförmige Bewegung des Objektes gewährleistet, selbst wenn einmal umfangreichere Berechnungen während eines Schleifendurchlaufs anfallen sollten.

Jetzt sind wir fast fertig, das einzige was noch fehlt, damit wir endlich ein Sprite anzeigen können, sind die Bilddaten, die wir laden müssen.

Laden der Bilddaten

Für die Bestückung des Image-Arrays wollen wir folgende kleine Bildsequenz verwenden. Es handelt sich um ein GIF, dass die Vorderansicht eines „Hubschraubers“ enthält. Jedes Einzelbild ist 30 Pixel breit und hoch.



Das GIF speichern wir in einem eigenen Ordner „pics“ im gleichen Verzeichnis, in dem auch die class-Dateien erzeugt werden:

Dateiname	Größe	Typ	Geändert ▲
pics		File Folder	22.10.2007 11:03
GamePanel.class	3 KB	CLASS-Datei	18.10.2007 15:46
Drawable.class	1 KB	CLASS-Datei	18.10.2007 15:50
Movable.class	1 KB	CLASS-Datei	18.10.2007 15:50
Sprite.class	2 KB	CLASS-Datei	22.10.2007 10:54

Nun gilt es, dieses GIF zu laden. Dies erledigen wir in unserer Hauptklasse GamePanel. Hierzu erstellen wir eine Methode `loadPics(String path, int pics)`, die uns eine `BufferedImage`-Array zurück liefert und den Speicherort und die Anzahl der Einzelbilder übergeben bekommt:

```
95 private BufferedImage[] loadPics(String path, int pics){
96
97     BufferedImage[] anim = new BufferedImage[pics];
98     BufferedImage source = null;
99
100    URL pic_url = getClass().getClassLoader().getResource(path);
101
102    try {
103        source = ImageIO.read(pic_url);
104    } catch (IOException e) {}
105
106    for(int x=0;x<pics;x++){
107        anim[x] = source.getSubimage(x*source.getWidth()/pics, 0,
108            source.getWidth()/pics, source.getHeight());
109    }
110
111    return anim;
112 }
```

Zeile 95: Die Methode bekommt Speicherort und Anzahl der Einzelbilder übergeben

Zeile 97: Wir erzeugen ein BufferedImage-Array in der Größe der Einzelbilder

Zeile 98: BufferedImage zum Laden des ganzen Bildes

Zeile 100: Ermitteln der URL mit dem Speicherort

Zeile 103: Laden des Quellbildes

Zeile 106-109: Mit Hilfe der Methode getSubimage(...) aus BufferedImage, wird das Quellbild in die Anzahl der angegebenen Einzelbilder zerlegt.

Diese Methode kann nur Bildsequenzen laden, die hintereinander gezeichnet sind! Außerdem muß für jedes Objekt eine eigene Datei geladen werden.

Das eigentliche Laden führen wir vor dem Beginn des Spieles durch, damit uns die dafür benötigte Zeit nicht den Spielfluß stört. Hierfür haben wir schon früher die Klasse Methode doInitializations() angelegt.

```
36 private void doInitializations() {  
37  
38     last = System.nanoTime();  
39  
40     BufferedImage[] heli = this.loadPics("pics/heli.gif", 4);  
41  
42  
43     Thread t = new Thread(this);  
44     t.start();  
45 }
```

Wichtig dabei ist, dass die Pfadangabe wie oben dargestellt übergeben wird!

Das erste bewegte Objekt

Jetzt soll es aber endlich losgehen, damit wir das erste bewegte Objekt anzeigen können. Zunächst benötigen wir zwei neue Instanzvariablen in unserem GamePanel: Eine Collection (ich bevorzuge Vektoren (rein subjektiv)) und eine Klasse Sprite:

```
13 public class GamePanel extends JPanel implements Runnable{
14
15     private static final long serialVersionUID = 1L;
16     boolean game_running = true;
17
18     long delta = 0;
19     long last = 0;
20     long fps = 0;
21
22     Sprite copter;
23     Vector<Sprite> actors;
24
25     public static void main(String[] args){
26         new GamePanel(800,600);
27     }
28 }
```

Diese beiden Instanzvariablen initialisieren wir ebenfalls in unserer Methode doInitialization():

```
40 private void doInitializations(){
41
42     last = System.nanoTime();
43
44     actors = new Vector<Sprite>();
45     BufferedImage[] heli = this.loadPics("pics/heli.gif", 4);
46     copter = new Sprite(heli,400,300,100,this);
47     actors.add(copter);
48
49     Thread t = new Thread(this);
50     t.start();
51 }
```

Zeile 44: Instanzieren des Vectors

Zeile 46: Instanzieren des Sprites an der Position 400/300 mit einer Bildwechselrate von 100 ms.

Zeile 47: Sprite in den Vector packen

Jetzt müssen wir noch die GameLoop-Methoden mit Leben füllen und dann kann es los gehen.

Da wir zur Definition der wichtigsten Methoden Interfaces herangezogen haben, können wir diese jetzt wie Objekte verwenden und die entsprechende Methode aufrufen. Was jetzt schon vorteilhaft ist, wird später noch bequemer, wenn wir weitere Objekte einführen. ☺

doLogic():

```
78 private void doLogic() {
79     for (Movable mov:actors) {
80         mov.doLogic(delta);
81     }
82 }
```

moveObjects():

```
84 private void moveObjects() {
85     for (Movable mov:actors) {
86         mov.move(delta);
87     }
88 }
```

paintComponent(Graphics g):

```
124 @Override
125 public void paintComponent(Graphics g) {
126     super.paintComponent(g);
127
128     g.setColor(Color.red);
129     g.drawString("FPS: " + Long.toString(fps), 20, 10);
130
131     if(actors!=null) {
132         for (Drawable draw:actors) {
133             draw.drawObjects(g);
134         }
135     }
136
137 }
```

Da wir noch keinen Startbildschirm etc. haben, wurde hier noch zusätzlich eine If-Bedingung eingebaut, so dass ein nicht initialisierter Vector keine NullPointerException auslöst.

Und so sieht's aus:



Die Steuerung

Den Hubschrauber nur anzuzeigen ist aber nur der halbe Spaß. Jetzt wollen wir diesen auch noch über die Cursor-Tasten steuern können. Dafür benötigen wir 5 weitere Instanzvariablen:

```
27 | boolean up    = false;
28 | boolean down  = false;
29 | boolean left  = false;
30 | boolean right = false;
31 | int speed = 50;
```

4 Boolean für die 4 Richtungen und eine Variable für die Geschwindigkeit der Bewegung. Für komplexere Spiele mag dies nicht die richtige Vorgehensweise sein, für unser kleines Spiel hier ist das aber ausreichend.

Um die o. g. Steuervariablen beeinflussen zu können, implementieren wir das `KeyListener`-Interface in unser `GamePanel`:

```
15 | public class GamePanel extends JPanel implements Runnable, KeyListener{
16 |
17 |     private static final long serialVersionUID = 1L;
18 |     boolean game_running = true;
19 |
20 |     long delta = 0;
```

Außerdem müssen die entsprechenden Methoden angelegt werden, je nach IDE geschieht dies automatisch:

```
140 | public void keyPressed(KeyEvent e) {
141 |
142 | }
143 |
144 | public void keyReleased(KeyEvent e) {
145 |
146 | }
147 |
148 | public void keyTyped(KeyEvent e) {
149 |
150 | }
```


Jetzt noch im Konstruktor unseres GamePanels den KeyListener zu unserem Fenster hinzufügen und dann sind wir schon fast fertig:

```
37 public GamePanel(int w, int h) {
38     this.setPreferredSize(new Dimension(800, 600));
39     JFrame frame = new JFrame("GameDemo");
40     frame.setLocation(100, 100);
41     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
42     frame.addKeyListener(this);
43     frame.add(this);
44     frame.pack();
45     frame.setVisible(true);
46     doInitializations();
47 }
```

In der Methode keyPressed(KeyEvent e) setzen wir beim entsprechenden Tastendruck die jeweilige Steuervariable auf true.

```
public void keyPressed(KeyEvent e) {

    if (e.getKeyCode() == KeyEvent.VK_UP) {
        up = true;
    }

    if (e.getKeyCode() == KeyEvent.VK_DOWN) {
        down = true;
    }

    if (e.getKeyCode() == KeyEvent.VK_LEFT) {
        left = true;
    }

    if (e.getKeyCode() == KeyEvent.VK_RIGHT) {
        right = true;
    }

}
```

Und machen dies bei `keyReleased(KeyEvent e)` ggf. rückgängig:

```
public void keyReleased(KeyEvent e) {  
  
    if (e.getKeyCode() == KeyEvent.VK_UP) {  
        up = false;  
    }  
  
    if (e.getKeyCode() == KeyEvent.VK_DOWN) {  
        down = false;  
    }  
  
    if (e.getKeyCode() == KeyEvent.VK_LEFT) {  
        left = false;  
    }  
  
    if (e.getKeyCode() == KeyEvent.VK_RIGHT) {  
        right = false;  
    }  
  
}
```

Diese Vorgehensweise stellt sicher, dass unser Spiel auch richtig reagiert, wenn mehr als eine Taste gedrückt wird. Würden wir den Steuercode direkt in den `KeyListener`-Methoden hinterlegen, würde mit Sicherheit nicht jeder Tastendruck erkannt werden.

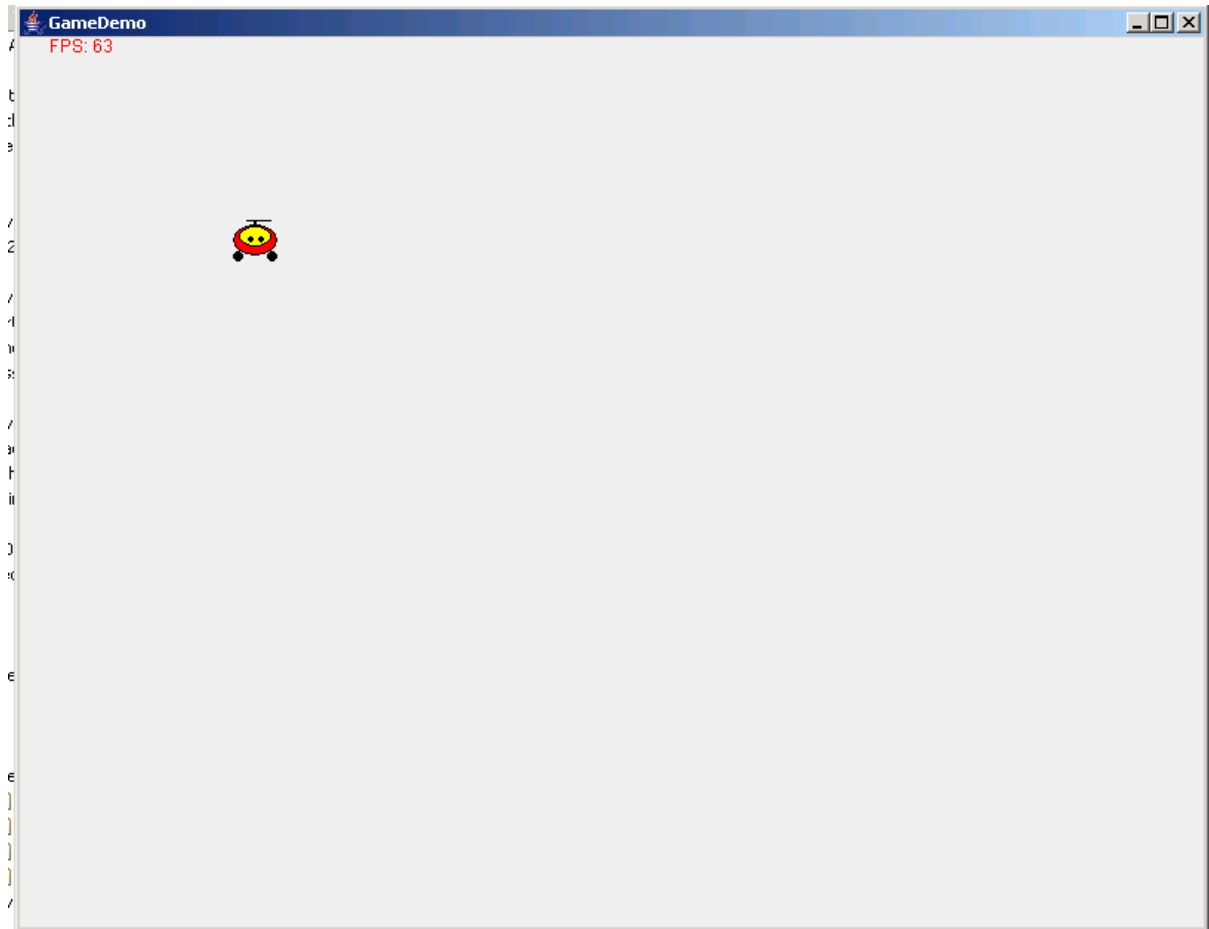
Innerhalb des GameLoop prüfen wir jetzt in der Methode checkKeys() die boolean-Werte ab und verändern die Geschwindigkeitsparameter unseres Sprites:

```
private void checkKeys() {  
  
    if(up) {  
        copter.setVerticalSpeed(-speed);  
    }  
  
    if(down) {  
        copter.setVerticalSpeed(speed);  
    }  
  
    if(right) {  
        copter.setHorizontalSpeed(speed);  
    }  
  
    if(left) {  
        copter.setHorizontalSpeed(-speed);  
    }  
  
    if(!up&&!down) {  
        copter.setVerticalSpeed(0);  
    }  
  
    if(!left&&!right) {  
        copter.setHorizontalSpeed(0);  
    }  
  
}
```

Wichtig ist, auch zu prüfen ob die beiden gegenüberliegenden Werte beide false sind, um unser Objekt ggf. wieder anzuhalten. Es wäre hier auch denkbar in unserer Helikopter-Klasse komplexere Methoden bereit zu stellen, um z. B. den Hubschrauber nach und nach zu beschleunigen. Für unser Anfänger-Spiel soll die normale Geschwindigkeitsänderung aber erst einmal ausreichen.

Und so sieht's aus:

Jetzt fliegt unser Heli nicht nur, wir können ihn auch beliebig auf dem GamePanel herum bewegen 😊



Erster Feinschliff

Jetzt wird's abstrakt

Momentan können wir unseren Hubschrauber problemlos aus dem Bild heraus fliegen. Dies soll sich nun ändern. Da diese Logik aber ausschließlich für den Hubschrauber gelten soll und nicht für spätere Objekte, wollen wir die Klasse Sprite ab jetzt nur noch als Basis-Klasse verwenden, die erweitert, aber nicht mehr instanziiert wird.

Dies erreichen wir ganz einfach, indem wir die Klasse Sprite abstract setzen.

```
1 import java.awt.Graphics;
2
3
4
5
6 public abstract class Sprite extends Rectangle2D.Double implements Drawable, Movable{
7
8     long delay;
9     long animation = 0;
10    GamePanel parent;
11    BufferedImage[] pics;
12    int currentpic = 0;
13
14    protected double dx;
15    protected double dy;
16
17    public Sprite(BufferedImage[] i, double x, double y, long delay, GamePanel p ){
18        pics = i;
```

Je nach verwendeter IDE, erhalten wir sofort oder spätestens beim Kompilieren unserer Klassen eine Fehlermeldung, weil für diese Klasse jetzt kein Objekt mehr erzeugt werden darf. Daher legen wir uns eine Klasse Heli an, die alle Eigenschaften von Sprite erbt!

```
1 import java.awt.image.BufferedImage;
2
3
4 public class Heli extends Sprite {
5
6     public Heli(BufferedImage[] i, double x, double y, long delay, GamePanel p) {
7         super(i, x, y, delay, p);
8     }
9
10
11 }
12
```

Wichtig ist dabei, den Konstruktor der Elternklasse aufzurufen (Zeile 7), damit alle Werte schön gesetzt werden.

Zusätzlich müssen wir natürlich noch die Klasse GamePanel anpassen und alle bisher verwendeten Objekte vom Typ Sprite in Objekte vom Typ Heli umbenennen:

```
14 import java.awt.Color;
15 public class GamePanel extends JPanel implements Runnable, KeyListener{
16
17     private static final long serialVersionUID = 1L;
18     boolean game_running = true;
19
20     long delta = 0;
21     long last = 0;
22     long fps = 0;
23
24     Heli copter;
25     Vector<Sprite> actors;
26
27     boolean up = false;
28     boolean down = false;
29     boolean left = false;
30     boolean right = false;
31     int speed = 50;
32
33     public static void main(String[] args){
34
35     public GamePanel(int w, int h){
36
37     private void doInitializations(){
38
39         last = System.nanoTime();
40
41         actors = new Vector<Sprite>();
42         BufferedImage[] heli = this.loadPics("pics/heli.gif", 4);
43         copter = new Heli(heli, 400, 300, 100, this);
44         actors.add(copter);
45
46         Thread t = new Thread(this);
47         t.start();
48     }
49 }
```

Das war's. Jetzt ist es problemlos möglich, individuelle Logik für unseren Hubschrauber zu hinterlegen!

Die neue Klasse Heli

An dieser neuen Klasse Heli zeigt sich nun, dass es durch den Aufwand, den wir bis jetzt mit der Klasse Sprite getrieben haben, relativ einfach ist, neue Objekte in unser Spiel einzufügen.

```
1 import java.awt.image.BufferedImage;
2
3
4 public class Heli extends Sprite {
5
6     public Heli(BufferedImage[] i, double x, double y, long delay, GamePanel p) {
7         super(i, x, y, delay, p);
8     }
9
10    @Override
11    public void doLogic(long delta) {
12        super.doLogic(delta);
13
14        if (getX() < 0) {
15            setHorizontalSpeed(0);
16            setX(0);
17        }
18
19        if (getX() + getWidth() > parent.getWidth()) {
20            setX(parent.getWidth() - getWidth());
21            setHorizontalSpeed(0);
22        }
23
24        if (getY() < 0) {
25            setY(0);
26            setVerticalSpeed(0);
27        }
28
29        if (getY() + getHeight() > parent.getHeight()) {
30            setY(parent.getHeight() - getHeight());
31            setVerticalSpeed(0);
32        }
33    }
34
35
36 }
```

Ab Zeile 11 überschreiben wir die Methode doLogic(long delta) der Vaterklasse um individuellen Code einzufügen.

Wichtig ist der Aufruf der Methode der Vaterklasse in Zeile 12, da sonst unsere Animation nicht mehr ausgeführt werden würde. (Tipp: Einfach mal kurz auskommentieren und den Effekt beobachten).

Danach überprüfen wir die Ränder unseres GamePanels:

Linker Rand:

Mit getX() (Zeile 14), welches wir übrigens von Rectangle2D geerbt haben, überprüfen wir die aktuelle x-Position. Ist diese kleiner als Null, setzen wir die horizontale Geschwindigkeit auf Null (Zeile 15) und verändern die x-Position unseres Hubschraubers auf Null (Zeile 16). Da es leider keine Methode setX(double i) gibt, die wir erben könnten, erstellen wir uns diese kurz selbst – allerdings in der Klasse Sprite, damit sie von allen Objekten geerbt wird:

```
79 public void setX(double i){  
80     x = i;  
81 }  
82  
83 public void setY(double i){  
84     y = i;  
85 }
```

Obige Methoden werden in die Klasse Sprite eingefügt, damit sie für alle Kind-Objekte verfügbar sind.

Rechter Rand:

Wenn die Summe aus x-Position und Breite unseres Objektes größer ist als die Breite unseres GamePanels (Zeile 19), setzen wir unser x so, dass der rechte Rand unseres Hubschraubers mit dem rechten Rand des GamePanels abschließt (Zeile 20) und setzen die Geschwindigkeit ebenfalls auf Null (Zeile 21).

Auch hier können wir jetzt Vorteile aus der geeigneten Wahl der Vaterklassen ziehen, die uns z. B. die Methode getWidth() schon zur Verfügung stellen, ohne dass wir diese selbst programmieren müssten.

Da wir hier beim Prüfen der Ränder auf die Dimensionen des GamePanels zugreifen ist es uns möglich, unserem Spiel jede denkbare Größe zu geben ohne zusätzliche Parameter zu ändern. Auch bei veränderter Größe wurde der o. a. Code ohne Probleme funktionieren.

Oberer und unterer Rand:

Analoge Vorgehensweise zu linkem und rechten Rand. ☺

Änderung des Programm-Starts

Sehr unschön ist momentan noch, dass unser Spiel sofort losläuft, wenn wir das Programm starten. Wir wollen das Ganze nun so ändern, dass das Spiel mit der Enter-Taste gestartet wird und mit der Escape-Taste abgebrochen bzw. komplett geändert werden kann.

Hierfür richten wir uns eine weitere Instanzvariable vom Typ boolean ein, die je uns den Zustand des Spiels beschreibt.

```
15 public class GamePanel extends JPanel implements Runnable, KeyListener{
16
17     private static final long serialVersionUID = 1L;
18     boolean game_running = true;
19     boolean started = false;
```

Für komplexere Spiele (mit Intro, Ladebildschirm, etc.) wird man hier eine andere Lösung bevorzugen (verschiedene Werte eines Integer, etc.), für unser kleines Tutorial ist ein einzelner Boolean aber ausreichend.

Für diese Variable definieren wir uns auch noch die get- und set-Methoden in unserem GamePanel:

```
169 public boolean isStarted() {
170     return started;
171 }
172
173 public void setStarted(boolean started) {
174     this.started = started;
175 }
176
```

Außerdem müssen wir den bisherigen Code noch etwas modifizieren:

```
63 public void run() {  
64     while(game_running) {  
65         computeDelta();  
66         if(isStarted()){  
67             checkKeys();  
68             doLogic();  
69             moveObjects();  
70         }  
71         repaint();  
72         try {  
73             Thread.sleep(10);  
74         } catch (InterruptedException e) {}  
75     }  
76 }  
77  
78  
79  
80  
81  
82  
83  
84  
85 }
```

Die Methoden

- checkKeys()
- doLogic() und
- moveObjects()

packen wir in eine If-Bedingung, die unsere neue Variable via get-Methode abfragt.

Das computeDelta() lassen wir außen vor. Dies ist aber eher ein subjektive Vorliebe, ähnlich wie die Anzeige der fps, die durch diesen Methodenaufruf ermöglicht wird. 😊 Wichtig ist, dass die 3 oben genannten Methoden erst abgearbeitet werden, wenn unser Spiel gestartet ist, andernfalls wäre es möglich, dass Objekte schon vor dem Spielstart beeinflusst werden.

Auch in die überschriebene paintComponent-Methode fügen wir noch eine Bedingung ein:

```
128 @Override
129 public void paintComponent(Graphics g) {
130     super.paintComponent(g);
131
132     g.setColor(Color.red);
133     g.drawString("FPS: " + Long.toString(fps), 20, 10);
134
135     if(!isStarted()){
136         return;
137     }
138
139     if(actors!=null){
140         for(Drawable draw:actors){
141             draw.drawObjects(g);
142         }
143     }
144 }
145 }
```

Alles nach Zeile 135 wird erst gezeichnet, wenn das Spiel gestartet ist. Jetzt müssen wir nur noch dafür sorgen, dass die Spielstatus-Variable auch auf Tastendruck verändert werden kann. Dazu müssen wir in unserem ActionListener noch Code einfügen – es empfiehlt sich die Methode keyReleased(..) da diese eindeutig ist.

```
228
229
230     if(e.getKeyCode()==KeyEvent.VK_ENTER){
231         if(!isStarted()){
232             doInitializations();
233             setStarted(true);
234         }
235
236         if(e.getKeyCode()==KeyEvent.VK_ESCAPE){
237             if(isStarted()){
238                 setStarted(false);
239             }else{
240                 setStarted(false);
241                 System.exit(0);
242             }
243         }
244     }
```

In Zeile 229 wird der Code für das Drücken (bzw. Richtigerweise: das Loslassen) der Enter-Taste hinterlegt. Wenn unser Spiel noch nicht gestartet ist, wird unsere Initialisierungs-Methode aufgerufen und der boolean-Wert auf true gesetzt.

Das Aufrufen der Initialisierungsmethode ist wichtig, damit bei einem Neustart, alles auf seinen Ausgangswert gesetzt wird. Damit wir mit dem wiederholten Aufruf der Initialisierungsmethode keine Probleme bekommen, wird gleich noch eine kleine Änderung nötig sein.

In Zeile 236 wird der Code für die Escape-Taste hinterlegt. Läuft das Spiel, wird es gestoppt. Ist das Spiel schon gestoppt wird unser Programm beendet.

Jetzt noch die erwähnte Änderung in der Methode doInitializations() und gut ist's (für den Augenblick):

```
15 public class GamePanel extends JPanel implements Runnable, KeyListener{
16
17     private static final long serialVersionUID = 1L;
18     boolean game_running = true;
19     boolean started = false;
20     boolean once = false;
21
22     long delta = 0;
23     long last = 0;
24     long fps = 0;
25
26     Heli copter;
27     Vector<Sprite> actors;
28
29     boolean up = false;
30     boolean down = false;
31     boolean left = false;
32     boolean right = false;
33     int speed = 50;
34
35+ public static void main(String[] args) {
36
37
38
39+ public GamePanel(int w, int h) {
40
41
42
43
44
45
46
47
48
49
50
51- private void doInitializations() {
52
53     BufferedImage[] heli = this.loadPics("pics/heli.gif", 4);
54
55     last = System.nanoTime();
56
57     actors = new Vector<Sprite>();
58     copter = new Heli(heli, 400, 300, 100, this);
59     actors.add(copter);
60
61     if(!once) {
62         once = true;
63         Thread t = new Thread(this);
64         t.start();
65     }
66 }
```

In Zeile 20 definieren wir einen weiteren boolean, der ab Zeile 61 verhindert, dass bei jedem Neustart ein neuer Thread gestartet wird.

Mit dieser Konstellation vernachlässigen wir, dass bei jedem Aufruf von doInitializations() unsere GIF-Dateien unnötigerweise neu geladen werden.

Dies lassen wir aus folgenden Gründen hier unberücksichtigt:

1. Aufgrund des geringen Umfangs der Datei, wird es nicht sonderlich ins Gewicht fallen und
2. Wird man bei komplexeren/fortgeschritteneren Spielen die Grafiken sowieso anders verwalten (i.d.R. mit einer eigenen Klasse, die Grafiken lädt und speichert)

Und noch eine letzte Änderung für den Feinschliff:

```
39 public GamePanel(int w, int h) {  
40     this.setPreferredSize(new Dimension(800, 600));  
41     this.setBackground(Color.cyan);  
42     JFrame frame = new JFrame("GameDemo");  
43     frame.setLocation(100, 100);  
44     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
45     frame.addKeyListener(this);  
46     frame.add(this);  
47     frame.pack();  
48     frame.setVisible(true);  
49     doInitializations();  
50 }
```

Schließlich fliegen wir am Himmel. ☺
(Color.blue geht auch, war mir aber zu dunkel☺)

Weitere Objekte

Nun wollen wir das Ganze mal etwas mit Leben füllen. Wir wollen ein paar Wolken einfügen, die über den Bildschirm wandern. Wenn Sie auf der einen Seite verschwunden sind, sollen Sie zur anderen Seite wieder rein kommen. Dazu erstellen wir eine neue Klasse, die von Sprite erbt. Innerhalb dieser Klasse überschreiben wir lediglich die Methode doLogic() um die Logik für das Verlassen des Bildschirms zu hinterlegen.

```
1 import java.awt.image.BufferedImage;
2
3
4 public class Cloud extends Sprite{
5
6     public Cloud(BufferedImage[] i, double x, double y, long delay, GamePanel p) {
7         super(i, x, y, delay, p);
8     }
9
10
11     @Override
12     public void doLogic(long delta) {
13         super.doLogic(delta);
14     }
15
16 }
17
18
19
20 }
```

Zu dieser Kind-Klasse von Sprite fügen wir jetzt noch folgenden Code hinzu:

- etwas Logik im Konstruktor zur Bestimmung der Richtung in die sich die Wolke bewegt (das überlassen wir dem Zufall)
- die Logik in doLogic(long delta) wo wir prüfen, ob sich die Wolke komplett außerhalb des sichtbaren Bereichs befindet. Ist das der Fall, versetzen wir sie auf die andere Seite – auch außerhalb des sichtbaren Bereichs.

```

1 import java.awt.image.BufferedImage;
2
3
4 public class Cloud extends Sprite{
5
6     final int SPEED = 20;
7
8     public Cloud(BufferedImage[] i, double x, double y, long delay, GamePanel p) {
9         super(i, x, y, delay, p);
10
11         if((int)(Math.random()*2)>1){
12             setHorizontalSpeed(-SPEED);
13         }else{
14             setHorizontalSpeed(SPEED);
15         }
16     }
17
18
19     @Override
20     public void doLogic(long delta) {
21         super.doLogic(delta);
22
23         if(getHorizontalSpeed()>0 && getX()>parent.getWidth()){
24             setX(-getWidth());
25         }
26
27         if(getHorizontalSpeed()<0 && (getX()+getWidth())<0){
28             setX(parent.getWidth()+getWidth());
29         }
30     }
31 }
32
33 }

```

Folgende Änderungen wurden eingefügt:

Zeile 6: Eine Konstante SPEED, welche die Geschwindigkeit der Wolke enthält. In diesem Fall für alle gleich.

Zeile 11 – 15: Hier bestimmen wir per Zufall die Bewegungsrichtung der Wolke

Zeile 23 – 27: Hier prüfen wir, ob die Wolke außerhalb des sichtbaren Bereichs ist und bewegen sie dann auf die andere Seite.

Jetzt fehlen noch 2 Dinge:

1. die Wolke
2. ein bisschen Code, mit dem wir die Wolken-Objekte in unseren Vektor im GamePanel werfen

Das wär's. 😊

Hier ist die Wolke:



Jetzt noch der restliche Code in unserem GamePanel:

```
52 private void doInitializations() {
53
54     BufferedImage[] heli = this.loadPics("pics/heli.gif", 4);
55
56     last = System.nanoTime();
57
58     actors = new Vector<Sprite>();
59     copter = new Heli(heli, 400, 300, 100, this);
60     actors.add(copter);
61
62     createClouds();
63
64     if(!once) {
65         once = true;
66         Thread t = new Thread(this);
67         t.start();
68     }
69 }
70
71 private void createClouds() {
72
73     BufferedImage[] ci = this.loadPics("pics/cloud.gif", 1);
74
75     for(int y=10; y<getHeight(); y+=50) {
76         int x = (int) (Math.random() * getWidth());
77         Cloud cloud = new Cloud(ci, x, y, 1000, this);
78         actors.add(cloud);
79     }
80
81
82 }
```

In der Methode `doInitializations()` fügen wir einen weiteren Methodenaufruf ein: `createClouds()`. Hier werden wir die Wolken für das Spiel erzeugen. Nach dem erweitern von `doInitializations()` müssen wir noch die Methode erstellen.

Zeile 73: Hier wird das GIF geladen – genau so, wie wir es für den Hubschrauber realisiert haben

Zeile 75: Beginnend bei 10 setzen wir in 50er-Schritten Wolken über die gesamte Höhe des Spielfeldes.

Zeile 76: die x-Position lassen wir über die Zufallsfunktion erzeugen.

Zeile 77: Hier wird unser Cloud-Objekt instanziiert – analog zu unserem Hubschrauber.

Zeile 78: Dann das Objekt in die Collection packen – fertig.

Alles andere (Logik-Aufrufe, Bewegungen, Zeichnen) haben wir schon lange in unserem GameLoop implementiert. An dieser Stelle sparen wir uns durch den Aufwand mit den Interfaces in der Klasse Sprite einiges an Arbeit, da nach einem Programmstart unsere Wolken jetzt problemlos mit verarbeitet werden.

Und so sieht's aus:



Es ist an dieser Stelle beabsichtigt, dass der Hubschrauber hinter den Wolken verschwindet. Wem dies nicht gefällt, der muß sicherstellen, dass alle Objekte vor dem Hubschrauber in die Collection eingefügt werden, damit der Hubschrauber als letztes gezeichnet wird!

Weitere Objekte

So langsam wird es Zeit, das Ganze etwas interessanter zu machen. Daher wollen wir nun ein paar Feinde in unser Spiel einbinden. Im Prinzip funktioniert das nicht anders, als wir es bisher mit allen Objekten gemacht haben:

1. Objekt von Sprite erben lassen
2. zusätzliche Logik einbauen
3. Instanzieren
4. In die Collection packen, die unsere Spielobjekte enthält

Das nächste Objekt soll nun eine Rakete sein, die unseren Hubschrauber zerstören könnte. Die entsprechend Grafik für die Animation sieht so aus:



Dabei fällt auf, dass die Grafikdatei 2 Animationsreihen enthält, um nicht für jede Bewegungsrichtung ein GIF ablegen zu müssen (man denke an den Aufwand für komplexe/flüssigere Animationen und detailliertere Richtungsänderungen).

4 Bilder für die Bewegung nach links und 4 Bilder für die Bewegung nach rechts.

Dafür wird es jetzt aber notwendig, die Animationsroutine zu ändern, so dass wir unseren Objekten mitteilen können, nur über einen bestimmten Bereich des Bildarrays zu „loopen“.

Da diese Funktion für alle Objekte sinnvoll sein kann, realisieren wir sie in der Klasse Sprite, so dass sie für alle Objekte zur Verfügung steht.

```
6 public abstract class Sprite extends Rectangle2D.Double implements Drawable, Movable{
7
8     long delay;
9     long animation = 0;
10    GamePanel parent;
11    BufferedImage[] pics;
12    int currentpic = 0;
13
14    protected double dx;
15    protected double dy;
16
17    int loop_from;
18    int loop_to;
19
20    public Sprite(BufferedImage[] i, double x, double y, long delay, GamePanel p ){
21        pics = i;
22        this.x = x;
23        this.y = y;
24        this.delay = delay;
25        this.width = pics[0].getWidth();
26        this.height = pics[0].getHeight();
27        parent = p;
28        loop_from = 0;
29        loop_to = pics.length - 1;
30    }
```

Zeile 17 + 18: Definition der Variablen, die Beginn und Ende des zu animierenden Intervalls angeben.

Zeile 28 + 29: Im Konstruktor werden diese Variablen defaultmäßig belegt. Da ein Array mit 0 beginnt, erhält `loop_from` den Wert Null. Entsprechend erhält bei dieser Vorgehensweise die Variable `loop_to` einen Wert, der der Länge des Arrays -1 entspricht (weil wir ja bei Null zu zählen beginnen und nicht bei 1).

Nun müssen wir noch die Animations-Routine anpassen und schon können wir aus einer Datei unterschiedliche Animationssequenzen verwenden.

```
46 public void move(long delta) {
47
48     if(dx!=0){
49         x += dx*(delta/1e9);
50     }
51
52     if(dy!=0){
53         y += dy*(delta/1e9);
54     }
55
56 }
57
58 private void computeAnimation(){
59
60     currentpic++;
61
62     if(currentpic>loop_to){
63         currentpic = loop_from;
64     }
65
66 }
67
68 public void setLoop(int from, int to){
69     loop_from = from;
70     loop_to   = to;
71     currentpic = from;
72 }
73
74 public void setVerticalSpeed(double d) {
75     dv = d;
```

Zeile 62-64: Hier haben wir jetzt die Bedingung so angepasst, dass `loop_from` und `loop_to` berücksichtigt werden.

Zeile 68-72: Zusätzlich wurde eine Methode implementiert, mit der das Animations-Intervall verändert werden kann. Damit es keine logischen „Kollisionen“ gibt, wird hier jedes Mal die Variable `currentpic` auf den Beginn des Intervalls gesetzt.

Die Klasse Rocket

Die Rakete soll folgende Eigenschaften haben:

- Sie soll zufällig von links oder rechts kommen
- Abhängig von der Startposition soll sie sinken oder steigen
- Wenn Sie den Hubschrauber erfasst hat, soll sie diesen „verfolgen“

Auch diese Klasse erbt von Sprite, somit müssen wir hauptsächlich wieder die doLogic-Methode überschreiben um das Verhalten der Rakete zu beeinflussen.

Zusätzlich möchte ich hier die Bewegung des Objektes erst nach dessen Erzeugung festlegen – einfach um mal eine unterschiedliche Vorgehensweise im Vergleich zur Klasse Cloud zu zeigen. Daher werden wir noch die Methoden zur Festlegung der Geschwindigkeit überschreiben, damit dort dann entschieden werden kann, welche Bilder des Arrays zu verwenden sind.

Der Rumpf der neuen Klasse sieht dann so aus:

```
4 public class Rocket extends Sprite{
5
6 public Rocket(BufferedImage[] i, double x, double y, long delay, GamePanel p) {
7     super(i, x, y, delay, p);
8
9 }
10
11 @Override
12 public void doLogic(long delta) {
13     super.doLogic(delta);
14 }
15
16
17 public void setHorizontalSpeed(double d) {
18     super.setHorizontalSpeed(d);
19 }
20
21
22
23
24 }
```

Die Rakete werden wir jeweils außerhalb des Bildschirms starten. Je nachdem setzen wir die vertikale Geschwindigkeit auf 70 oder -70 um den Sink- oder Steigflug zu simulieren. Dies werden wir schon im Konstruktor erledigen.

Zusätzlich benötigen wir für die Verfolgung des Helikopters zwei Klassenvariablen:

```
7 public class Rocket extends Sprite(  
8  
9     Rectangle2D.Double target;  
10    boolean locked = false;  
11  
12    public Rocket(BufferedImage[] i, double x, double y, long delay, GamePanel p) {  
13        super(i, x, y, delay, p);  
14  
15        if (getY() < parent.getHeight() / 2) {  
16            setVerticalSpeed(70);  
17        } else {  
18            setVerticalSpeed(-70);  
19        }  
20    }  
21
```

Zeile 9: Klassenvariable für die Zielverfolgung, dazu später mehr

Zeile 10: Klassenvariable, ob die Rakete ein Ziel erfasst hat, auch dazu später mehr

Zeile 15 – 19: Entscheiden, ob sich die Rakete nach oben oder unten bewegen soll.

Ich habe hier einmal die Geschwindigkeit bewusst nicht als Variable hinterlegt, so dass jeder der hier noch ein bisschen Tuning vornehmen möchte, einmal die Unterschiede erkennt (eine Variable ändern vs. alle Zahlen ändern).

Die Logic-Methode:

```
22 @Override  
23 public void doLogic(long delta) {  
24     super.doLogic(delta);  
25  
26     if (getHorizontalSpeed() > 0) {  
27         target = new Rectangle2D.Double(getX() + getWidth(), getY(),  
28             parent.getWidth() - getX(), getHeight());  
29     } else {  
30         target = new Rectangle2D.Double(0, getY(), getX(), getHeight());  
31     }  
32  
33     if (!locked && parent.copter.intersects(target)) {  
34         setVerticalSpeed(0);  
35         locked = true;  
36     }  
37  
38     if (locked) {  
39         if (getY() < parent.copter.getY()) {  
40             setVerticalSpeed(40);  
41         }  
42         if (getY() > parent.copter.getY() + parent.copter.getHeight()) {  
43             setVerticalSpeed(-40);  
44         }  
45     }  
46  
47 }
```

Zeile 26 - 31: Abhängig von der Bewegungsrichtung erzeugen wir hier ein Rechteck, dass den Raum von der Vorderseite der Rakete bis zum Bildschirm-Rand beinhaltet.

Zeile 33 – 36: Wenn die Rakete noch kein Ziel erfasst hat, sich unser Helikopter aber im gerade berechneten Rechteck befindet, beenden wir die vertikale Bewegung und setzen das Flag „locked“. Hier zeigt sich jetzt wieder der Vorteil aus der Verwendung der Klasse Rectangle2D als Basisklasse. Für diese einfache „Kollisionserkennung“ müssen wir keinen weiteren Code implementieren.

Zeile 38 – 45: Wenn die Rakete ein Ziel „erfasst“ hat, passt sie Ihre Höhe an den Hubschrauber an. Die vertikale Veränderung ist hier geringer gewählt, damit unser Hubschrauber noch eine Chance hat. (Sollte dieser Code von Rüstungskonzernen verwendet werden, sind hier Anpassungen nötig ☺).

Zu guter Letzt müssen wir noch die Methode setHorizontalSpeed(double d) überschreiben, damit das richtige Bildintervall angezeigt wird.

```
56 public void setHorizontalSpeed(double d) {  
57     super.setHorizontalSpeed(d);  
58  
59     if(getHorizontalSpeed()>0){  
60         setLoop(4, 7);  
61     }else{  
62         setLoop(0,3);  
63     }  
64 }
```

Zeile 59 – 63: Abhängig von der Bewegungsrichtung, wird ein anderes Bildintervall gesetzt.

Für den Test des „Zielrechtecks“ könnte man noch die paint-Methode überschreiben und diese anzeigen lassen. Dies ist dann sinnvoll, wenn der Code nicht wie gewünscht funktioniert.

```
49 @Override
50 public void drawObjects(Graphics g) {
51     super.drawObjects(g);
52     g.setColor(Color.red);
53     g.drawRect((int)target.x, (int)target.y, (int)target.width, (int)target.height);
54 }
```

Das Ergebnis sähe dann so aus:



Dies aber nur zum Testen.

Modifikationen der Klasse GamePanel:

Über einen Timer soll alle 3 Sekunden eine Rakete erzeugt werden. Dazu verwenden wir den Timer aus dem Paket javax.swing. Es muß darauf geachtet werden, dass das richtige Package importiert wird, da es in der API mehrere Klassen namens Timer gibt.

Da der Timer einen `ActionEvent` erzeugt, müssen wir noch das entsprechende Interface importieren. Zudem werden der Timer und ein Array vom Typ `BufferedImage` als Klassenvariable definiert. Dies ist notwendig damit der Timer aus

anderen Methoden heraus gestoppt werden kann und damit wir die Bilddaten für die Rakete zentral ablegen können.

Würden wir eine eigene Klasse zur Verwaltung der Bilddaten verwenden könnten wir uns dies sparen. Da wir aber bei jeder Erzeugung eines neuen Rocket-Objekts auf die Bilddaten zugreifen müssen, würde es uns einiges an Performance kosten, wenn wir diese jedes Mal laden müssten.

```
10 public class GamePanel extends JPanel implements Runnable, KeyListener, ActionListener{
11
12     private static final long serialVersionUID = 1L;
13     boolean game_running = true;
14     boolean started = false;
15     boolean once = false;
16
17     long delta = 0;
18     long last = 0;
19     long fps = 0;
20
21     Heli copter;
22     Vector<Sprite> actors;
23
24     boolean up = false;
25     boolean down = false;
26     boolean left = false;
27     boolean right = false;
28     int speed = 50;
29
30     Timer timer;
31     BufferedImage[] rocket;
32
33     public static void main(String[] args){
34         new GamePanel(800,600);
35     }
```

Zeile 10: Implementierung des ActionListeners

Zeile 30: Timer-Klasse als Klassen-Variable

Zeile 31: BufferedImage-Array als Klassenvariable

Hier noch die „nackte“ Methode für den ActionListener

```
268 public void actionPerformed(ActionEvent e) {
269
270
271 }
```


Modifikation in doInitializations():

```
50 private void doInitializations() {  
51  
52     BufferedImage[] heli = loadPics("pics/heli.gif", 4);  
53     rocket = loadPics("pics/rocket.gif", 8);  
54  
55     last = System.nanoTime();  
56  
57     actors = new Vector<Sprite>();  
58     copter = new Heli(heli, 400, 300, 100, this);  
59     actors.add(copter);  
60  
61     createClouds();  
62  
63     timer = new Timer(3000, this);  
64     timer.start();  
65  
66     if(!once) {  
67         once = true;  
68         Thread t = new Thread(this);  
69         t.start();  
70     }  
71 }
```

In der Methode doInitializations füllen wir die zusätzlichen Objekte mit Leben:

Zeile 53: Hier laden wir die Bilddaten, mit unserer selbst erstellten Methode. Dies wird dann allerdings bei jedem Spielstart durchgeführt und damit spätestens beim 2. Mal unnötigerweise. An dieser Stelle schmerzt das allerdings nicht so sehr, so dass wir dies vernachlässigen. Wie bereits öfter bemerkt sollte man bei größeren Spielen eine eigene „Sprite-Lib“ programmieren um das unnötige Laden von Bilddaten zu vermeiden.

Zeile 63 – 64: Hier wird das Timer-Objekt erzeugt und gestartet. Intervall: 3 Sekunden.

Das Erzeugen der Raketen:

Wir benötigen noch eine Methode, die wir zum Erzeugen der Raketen aufrufen:

```
208 private void createRocket() {
209
210     int x = 0;
211     int y = (int) (Math.random() * getHeight());
212     int hori = (int) (Math.random() * 2);
213
214     if (hori == 0) {
215         x = -30;
216     } else {
217         x = getWidth() + 30;
218     }
219
220
221     Rocket rock = new Rocket(rocket, x, y, 100, this);
222     if (x < 0) {
223         rock.setHorizontalSpeed(100);
224     } else {
225         rock.setHorizontalSpeed(-100);
226     }
227     actors.add(rock);
228
229 }
230
```

Diese Methode erzeugt uns Raketen an unterschiedlichen Orten:

Zeile 210: Definition der Variable für die x-Position

Zeile 211: Zufällig Ermittlung der y-Position

Zeile 212: Zufallszahl (0 oder 1) um zu entscheiden, ob die Rakete von Links oder Rechts einfliegen soll.

Zeile 214 – 218: Je nachdem, von welcher Seite die Rakete einfliegt, soll sie deutlich außerhalb des Bildschirms starten.

Zeile 221: Erzeugen eines Rocket-Objekts

Ziel 222 – 226: hier setzen wir die Geschwindigkeit

Zeile 227: Die Rakete noch in unseren Objekt-Pool packen.

Dies ist hier mehr oder weniger als Demonstration gedacht, wie zusätzlicher Code die Übersichtlichkeit vermindert. In diesem Fall würde es viel mehr Sinn machen, den Großteil des in der Methode implementierten Codes in den Konstruktor der Rakete zu packen!

2 letzte Modifikationen

In der Methode `keyRelease(KeyEvent e)` müssen wir noch dafür sorgen, dass der Timer beim Abbruch des Spiels gestoppt wird:

```
277     if (e.getKeyCode() == KeyEvent.VK_ESCAPE) {
278         if (isStarted()) {
279             setStarted(false);
280             timer.stop();
281         } else {
282             setStarted(false);
283             System.exit(0);
284         }
285     }
```

Zu guter Letzt müssen wir noch dafür sorgen, dass aus der `ActionPerformed`-Methode heraus ein Raketen-Objekt erzeugt wird:

```
359     public void actionPerformed(ActionEvent e) {
360
361         if (isStarted() && e.getSource().equals(timer)) {
362             createRocket();
363         }
364
365     }
```

Dabei ist es wichtig, abzufragen ob das Spiel gestartet ist, sonst würde unser Timer, der beim ersten Aufruf von `doInitializations` gestartet wurde, fleißig Objekte erzeugen, obwohl wir das Spiel noch gar nicht per Enter-Taste gestartet haben.

Und so sieht's aus:



Feinschliff 2. Teil

Hier wollen wir jetzt noch 2 Punkte abhaken:

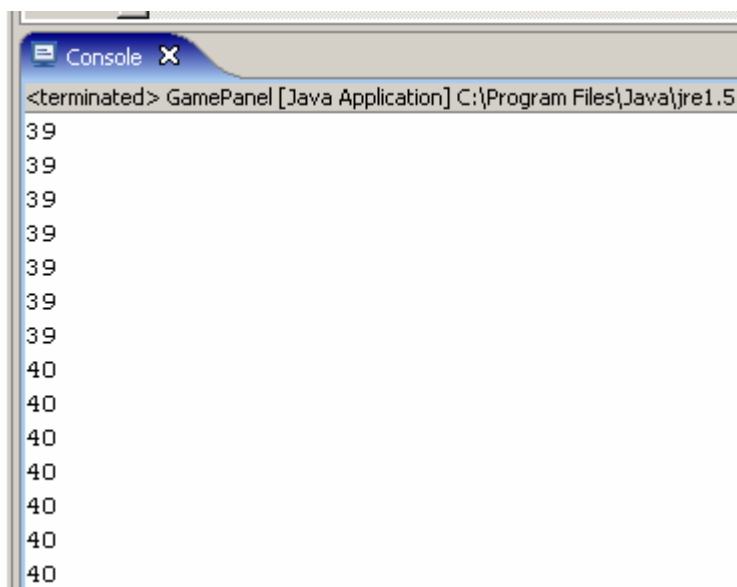
1. Es gibt im Spiel noch eine potentielle Fehlerquelle, die bisher noch nicht auffällt.
2. Außerdem wäre ein schöneres Hintergrundbild wünschenswert.

Objekte wieder löschen

Die potentielle Fehlerquelle lässt sich sehr rasch verdeutlichen. Dazu fügen wir in der Methode doLogic() in unserem GamePanel vorübergehend eine Zeile Code ein:

```
138 private void doLogic() {  
139     for (Movable mov:actors) {  
140         mov.doLogic(delta);  
141     }  
142     System.out.println(actors.size());  
143 }
```

In Zeil 142 geben wir die aktuelle Größe unseres Vectors in die Konsole aus. Wenn wir unser Programm dann eine Weile laufen lassen, sieht das Ergebnis ungefähr so aus:



Ursache ist, dass wir zwar fleißig Objekte in unserem Vector ablegen, diese aber niemals wieder entfernen. Hiermit steigt der Speicherbedarf des Spiels und die Anzahl der abzuarbeitenden Objekte kontinuierlich an.

Um dies zu beheben wollen wir ab sofort Objekte, die nicht mehr benötigt werden, wieder entfernen. Ob ein Objekt noch benötigt wird, entscheiden wir in der jeweiligen

Logik-Methode. Hierzu definieren wir uns ein entsprechendes Flag. Da wir dies für die meisten Objekte benötigen, packen wir es als Klassenvariable in unsere Klasse Sprite.

```
6 public abstract class Sprite extends Rectangle2D.1
7
8     long delay;
9     long animation = 0;
10    GamePanel parent;
11    BufferedImage[] pics;
12    int currentpic = 0;
13
14    protected double dx;
15    protected double dy;
16
17    int loop_from;
18    int loop_to;
19
20    boolean remove = false;
21
22    public Sprite(BufferedImage[] i, double x, doub
```

Anschließend hinterlegen wir in der Logik-Methode der Klasse Rocket zusätzlichen Code, der unseren Boolean auf true setzt, wenn die Rakete nicht mehr sichtbar ist.

```
20 @Override
21 public void doLogic(long delta) {
22     super.doLogic(delta);
23
24     if(getHorizontalSpeed()>0){
25         target = new Rectangle2D.Double(getX()+getWidth(),getY(),
26             parent.getWidth()-getX(),getHeight());
27     }else{
28         target = new Rectangle2D.Double(0,getY(),getX(),getHeight());
29     }
30
31     if(!locked&&parent.copter.intersects(target)){
32         setVerticalSpeed(0);
33         locked = true;
34     }
35
36     if(locked){
37         if(getY()<parent.copter.getY()){
38             setVerticalSpeed(40);
39         }
40         if(getY()>parent.copter.getY()+parent.copter.getHeight()){
41             setVerticalSpeed(-40);
42         }
43     }
44
45     if(getHorizontalSpeed()>0 && getX()>parent.getWidth()){
46         remove = true;
47     }
48
49     if(getHorizontalSpeed()<0 && getX()+getWidth()<0){
50         remove = true;
51     }
52
53 }
```

Zeile 45 – 47: Bewegt sich die Rakete nach rechts und ist der x-Wert größer als die Breite des Spielfeldes (d. h. Rakete ist vollständig außerhalb des sichtbaren Bereichs), setzen wir unser Flag auf true

Zeile 49 – 51: Andere Richtung, gleiche Bedingung. 😊

Jetzt benötigen wir in unserem GamePanel nur noch etwas Code, der uns die Objekte „aufräumt“. Die hinterlegen wir in doLogic() (wo sonst? 😊).

```

163 private void doLogic() {
164
165     Vector<Sprite> trash = new Vector<Sprite>();
166
167     for(Movable mov:actors){
168         mov.doLogic(delta);
169         Sprite check = (Sprite)mov;
170         if(check.remove){
171             trash.add(check);
172         }
173     }
174
175     if(trash.size()>0){
176         for(Sprite s: trash){
177             actors.remove(s);
178         }
179     }
180
181
182 }

```

Zeile 165: Wir definieren einen zusätzlichen (Müll-)Vector, in dem wir die aufzuräumenden Objekte zunächst sammeln. Dies ist notwendig, damit es keine Überschneidung gibt, dass ein Objekt u. U. noch geändert werden soll, dass wir löschen wollen oder bereits gelöscht haben.

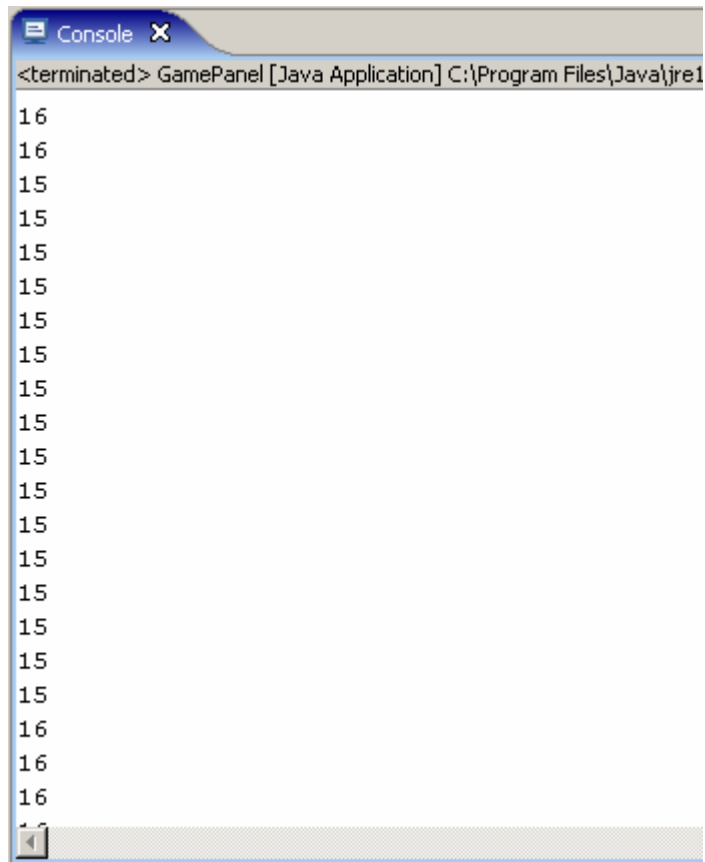
Zeile 169: Das aktuelle Objekt casten wir in ein Sprite

Zeile 170 – 172: Ist das remove-Flag gesetzt, packen wir das Sprite in den „Müll-Vektor“.

Zeile 175 – 179: Enthält der „Müll-Vector“ Objekte, löschen wir diese aus dem „Haupt-Vector“ actors. Anschließend machen wir den „Müll-Vector“ wieder platt.

Für die Überprüfung der Funktionsweise, lassen wir uns die Anzahl der gespeicherten Objekte in die Konsole ausgeben.

Ergebnis:

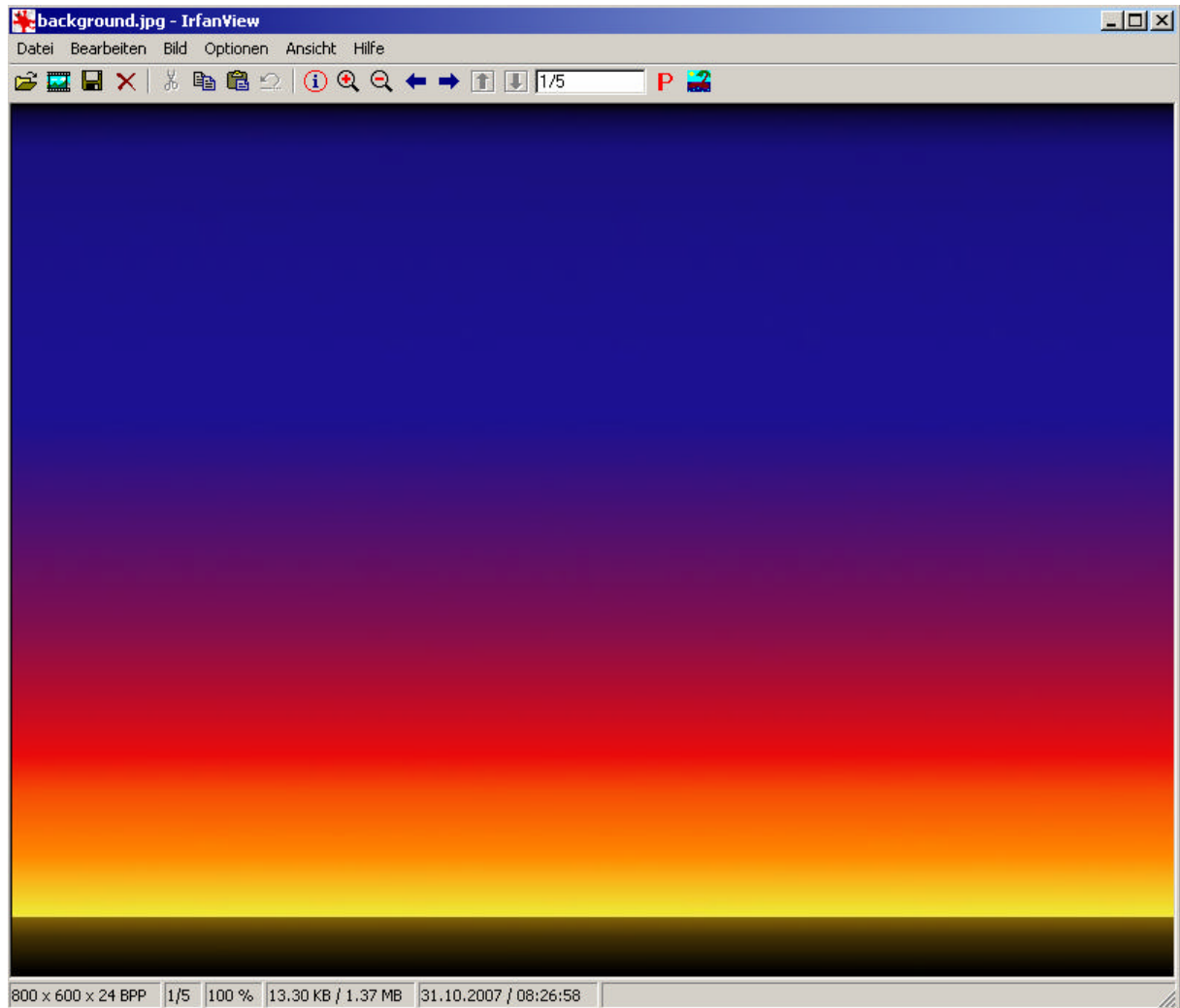


```
Console X
<terminated> GamePanel [Java Application] C:\Program Files\Java\jre1
16
16
15
15
15
15
15
15
15
15
15
15
15
15
15
15
15
15
15
15
16
16
16
16
```

Aktuell übersteigt die Anzahl der gespeicherten Objekte nicht die 16 (1 Hubschrauber, 12 Wolken und bis zu 3 Raketen). Der Code funktioniert also wie gewünscht. 😊

Hintergrundbild

Nun wollen wir noch ein schöneres Hintergrundbild einbauen. Mit einem beliebigen Grafik-Programm erstellen wir ein Hintergrundbild (hier ein Farbverlauf aus GIMP) und speichern das Bild in dem Ordner in dem alle Grafik-Dateien abgelegt werden.



Da das Hintergrundbild ständig verfügbar sein muß, definieren wir uns ein Objekt als Klassenvariable und laden/füllen dieses jeweils in der Methode `doInitializations()` mit Leben.

```
10 public class GamePanel extends JPanel implements Runnable, KeyListener, ActionListener {
11
12     private static final long serialVersionUID = 1L;
13     boolean game_running = true;
14     boolean started = false;
15     boolean once = false;
16
17     long delta = 0;
18     long last = 0;
19     long fps = 0;
20
21     Heli copter;
22     Vector<Sprite> actors;
23
24     boolean up = false;
25     boolean down = false;
26     boolean left = false;
27     boolean right = false;
28     int speed = 50;
29
30     Timer timer;
31     BufferedImage[] rocket;
32     BufferedImage background;
33
34     public static void main(String[] args) {
```

Zeile 32: `BufferedImage` zum Speichern des Hintergrundbildes

Ein Besonderheit gilt es in `doInitializations()` zu beachten:

```
51     private void doInitializations() {
52
53         BufferedImage[] heli = loadPics("pics/heli.gif", 4);
54         rocket = loadPics("pics/rocket.gif", 8);
55         background = loadPics("pics/background.jpg", 1)[0];
56
57         last = System.nanoTime();
58
59         actors = new Vector<Sprite>();
60         copter = new Heli(heli, 400, 300, 100, this);
61         actors.add(copter);
62
63         createClouds();
64
65         timer = new Timer(3000, this);
66         timer.start();
67
68         if(!once) {
69             once = true;
70             Thread t = new Thread(this);
71             t.start();
72         }
73     }
```

Da unsere Methode `loadPics(..)` ein Array zurück liefert, wir aber nur ein Bild wollen, müssen wir aus dem zurück gegebenen Objekt das erste (und einzige) Bild abgreifen. Dies geschieht in Zeile 55 durch die Null in eckigen Klammern, die auf das erste Objekt des Arrays verweist.

Jetzt müssen wir das Bild nur noch in unsere überschrieben `paintComponent(..)`-Methode packen:

```
166 @Override
167 public void paintComponent(Graphics g) {
168     super.paintComponent(g);
169
170     g.drawImage(background, 0, 0, this);
171
172     g.setColor(Color.red);
173     g.drawString("FPS: " + Long.toString(fps), 20, 10);
174
175     if(!isStarted()){
176         return;
177     }
178
179     if(actors!=null){
180         for(Drawable draw:actors){
181             draw.drawObjects(g);
182         }
183     }
184
185 }
```

Wichtig ist, dass das Bild als erstes nach dem `super`-Aufruf gezeichnet wird, damit es ganz hinten liegt. Alle folgenden Objekte werden dann darüber gezeichnet.

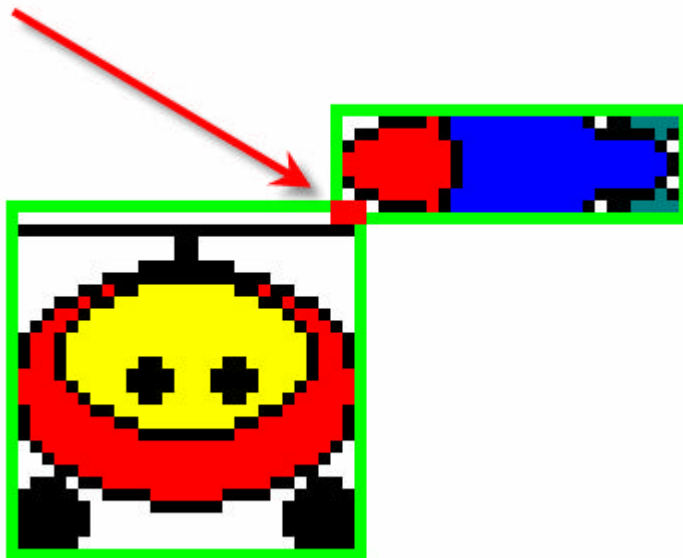
Und so sieht's aus:



Kollisionen

Wenn wir das Spiel bis jetzt testen, dann bleibt das Aufeinandertreffen von Hubschrauber und Rakete folgenlos. Das wollen wir jetzt abstellen.

Wir wollen uns im ersten Schritt zunächst auf eine sehr einfache Kollision beschränken, in dem wir das Überschneiden der Rectangle-Objekte prüfen. Dies wird allerdings eine ungenaue Kollisions-Ermittlung, weil wir unsere Objekte nun mal nicht bis in die letzte Ecke gezeichnet haben. Dies wird schnell deutlich, wenn wir uns die Ränder der Bilder einmal einzeichnen:



Auch o. a. Überschneidung wird bei uns zunächst als Kollision gelten! Wir werden dies aber ändern, wenn unsere erste „Primitiv-Kollisionsprüfung“ funktioniert.

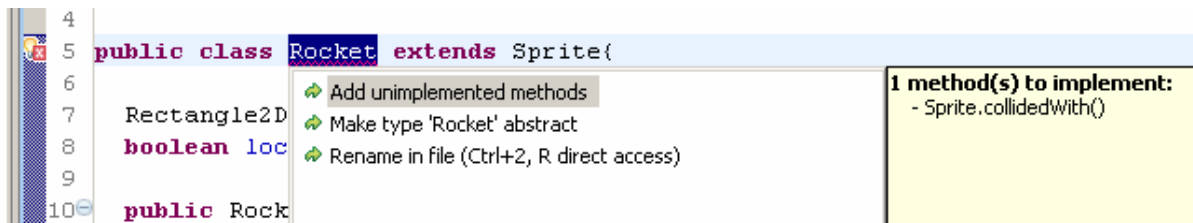
Abstrakte Methoden

Da die Kollisionsüberprüfung für die meisten Objekte notwendig sein wird, wollen wir uns zwingen, diese in alle Objekte einzubauen, die von Sprite erben. Daher definieren wir zunächst nur einen abstrakten Methodenrumpf, den wir überall einbauen müssen:

```
68  
69  
70 public abstract boolean collidedWith(Sprite s);  
71
```

Zeile 70: Definition der abstrakten Methode in Klasse Sprite.

Je nach verwendetem Editor bzw. verwendeter IDE werden wir jetzt benötigt, diese Methode in allen Objekten, die von Sprite erben zu realisieren:



Entsprechend lassen wir uns in allen Objekten die benötigte Methode generieren bzw. fügen Sie von Hand ein (hier z. B. in Klasse Rocket):

```
56 @Override
57 public boolean collidedWith(Sprite s) {
58
59
60
61     return false;
62 }
```

Jetzt füllen wir diese Methode auch gleich mit Leben:

```
66 @Override
67 public boolean collidedWith(Sprite s) {
68
69     if(this.intersects(s)){
70         System.out.println("Kollision Rakete");
71         return true;
72     }
73
74     return false;
75 }
```

Der Code dazu ist denkbar kurz, da wir das Ereignis zunächst nur der Konsole bekannt geben (Zeile 70). Hier sparen wir uns erneut einige Mühe, da wir alle Objekte von Rectangle2D.Double erben lassen und uns somit schon die geeignete Methode zur Verfügung steht.

Die wiederholen wir auch für die Klasse Heli:

```
35 @Override
36 public boolean collidedWith(Sprite s) {
37
38     if(this.intersects(s)){
39         System.out.println("Kollision Heli");
40         return true;
41     }
42
43     return false;
44 }
45
46
47 }
```

Nun fehlt uns nur noch der Code, um die Objekte gegenseitig zu überprüfen, dies packen wir in unsere Logik-Methode im GamePanel:

```
140 private void doLogic(){
141
142     Vector<Sprite> trash = new Vector<Sprite>();
143
144     for(Movable mov:actors){
145         mov.doLogic(delta);
146         Sprite check = (Sprite)mov;
147         if(check.remove){
148             trash.add(check);
149         }
150     }
151
152
153     for(int i = 0; i < actors.size(); i++){
154         for(int n = i+1; n < actors.size(); n++){
155
156             Sprite s1 = actors.elementAt(i);
157             Sprite s2 = actors.elementAt(n);
158
159             s1.collidedWith(s2);
160
161         }
162     }
163
164     if(trash.size() > 0){
165         actors.removeAll(trash);
166         trash.clear();
167     }
168
169 }
```

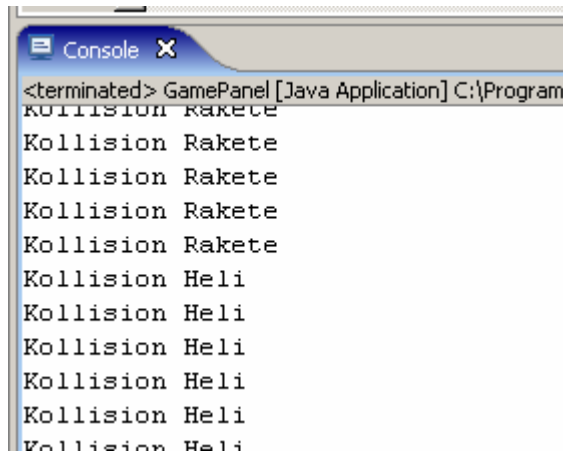
Zeile 153: 1. Schleife über alle Objekte des Vektors.

Zeile 154: 2. Schleife – Ermittlung aller Objekte die auf das Objekt aus Zeile 153 folgen.

Zeile 156 – 157: Die beiden zu prüfenden Sprites aus dem Vector holen

Zeile 159: Kollisionsprüfung durchführen. Die einzelnen Objekte schreiben bei erfolgter Kollision in die Konsole.

Bei einem Test des Programms erhalten wir folgende Konsolenausgabe:

A screenshot of a Java console window titled "Console". The window shows the output of a Java application. The first line is "<terminated> GamePanel [Java Application] C:\Program". Below this, there are several lines of text: "Kollision Rakete", "Kollision Rakete", "Kollision Rakete", "Kollision Rakete", "Kollision Heli", "Kollision Heli", "Kollision Heli", "Kollision Heli", "Kollision Heli", and "Kollision Heli". The text is displayed in a monospaced font, typical of a console window.

```
<terminated> GamePanel [Java Application] C:\Program
Kollision Rakete
Kollision Rakete
Kollision Rakete
Kollision Rakete
Kollision Heli
Kollision Heli
Kollision Heli
Kollision Heli
Kollision Heli
Kollision Heli
```

Jedes Objekt erzeugt wiederholt Einträge solange es sich mit einem anderen überschneidet, da mit jeder Spielschleife die Prüfung wiederholt wird.

Daher wollen wir jetzt die Objekte nach erfolgter Kollision löschen. In die Klassen Heli und Rocket fügen wir in der Methode `collidedWith(Sprite s)` jeweils folgenden Code ein:

```
66 @Override
67 public boolean collidedWith(Sprite s) {
68
69     if(remove){
70         return false;
71     }
72
73     if(this.intersects(s)){
74
75
76         if(s instanceof Heli){
77             remove = true;
78             s.remove = true;
79         }
80
81         if(s instanceof Rocket){
82             remove = true;
83             s.remove = true;
84         }
85
86         return true;
87     }
88
89     return false;
90 }
```

Zeile: 69 – 71: Ist das Objekt schon zum Entfernen vor gemerkt, führen wir keine weiteren Prüfungen durch.

Zeile 76 – 79: Wenn das Objekt mit dem kollidiert wird eine Instanz der Klasse Heli ist, setzen wir das „Entfernen-Flag“ für beide Objekte. Diese erste Bedingung können wir in der Klasse Heli natürlich weglassen, da wir keine 2 Heli-Objekte haben, die zusammen stoßen könnten.

Zeile 81 – 84: Kollidiert unserem aktuellen Objekt mit einem Objekt der Klasse Rocket, setzen wir bei beiden Objekten das „Entfernen-Flag“.

Wenn wir nun testen, werden alle kollidierten Objekte entfernt. Dummerweise kratzt dies unser Spiel aber momentan noch nicht, es läuft einfach ohne Hubschrauber weiter.

So sieht's aus:



Spiel-Ende signalisieren

Damit das Spiel beendet wird, wenn unser Hubschrauber „stirbt“ müssen wir etwas Code hinterlegen.

Zunächst definieren wir uns eine Klassenvariable gameover vom Typ long in GamePanel:

```
10 public class GamePanel extends JPanel implements Runnable, KeyListener, ActionListener{
11
12     private static final long serialVersionUID = 1L;
13     boolean game_running = true;
14     boolean started = false;
15     boolean once = false;
16
17     long delta = 0;
18     long last = 0;
19     long fps = 0;
20     long gameover = 0;
21 }
```

Diese müssen wir auch bei jedem Aufruf von `doInitializations()` wieder auf 0 setzen:

```
52 private void doInitializations() {  
53  
54     BufferedImage[] heli = loadPics("pics/heli.gif", 4);  
55     rocket = loadPics("pics/rocket.gif", 8);  
56     background = loadPics("pics/background.jpg", 1)[0];  
57  
58     last = System.nanoTime();  
59     gameover = 0;  
60  
61     actors = new Vector<Sprite>();  
62     copter = new Heli(heli, 400, 300, 100, this);  
63     actors.add(copter);  
}
```

Wird unser Helikopter nun zerstört, übergeben wir dieser Variable die Systemzeit. Dies geschieht in der Methode `collidedWith(Sprite s)` unsere Klasse `Heli`:

```
35 @Override  
36 public boolean collidedWith(Sprite s) {  
37  
38     if(remove){  
39         return false;  
40     }  
41  
42     if(this.intersects(s)){  
43  
44         if(s instanceof Rocket){  
45             remove = true;  
46             s.remove = true;  
47         }  
48  
49         if(remove){  
50             parent.gameover = System.currentTimeMillis();  
51         }  
52  
53         return true;  
54     }  
55  
56     return false;  
57 }  
58
```

Nun müssen wir in der Logikprüfung unseres GamePanels auch noch etwas Code hinterlegen, um dies abzu prüfen:

```
142 private void doLogic() {
143
144     Vector<Sprite> trash = new Vector<Sprite>();
145
146     for(Movable mov:actors) {
147         mov.doLogic(delta);
148         Sprite check = (Sprite)mov;
149         if(check.remove) {
150             trash.add(check);
151         }
152     }
153
154
155     for(int i = 0; i < actors.size(); i++) {
156         for(int n = i+1; n < actors.size(); n++) {
157
158             Sprite s1 = actors.elementAt(i);
159             Sprite s2 = actors.elementAt(n);
160
161             s1.collidedWith(s2);
162
163         }
164     }
165
166     if(trash.size() > 0) {
167         actors.removeAll(trash);
168         trash.clear();
169     }
170
171     if(gameover > 0) {
172         if(System.currentTimeMillis() - gameOver > 3000) {
173             stopGame();
174         }
175     }
176
177 }
178
179 private void stopGame() {
180     timer.stop();
181     setStarted(false);
182 }
183
```

Zeile 171: Prüfung, ob die Variable mit einem Wert belegt wurde

Zeile 172: Wir prüfen, ob die Zeitdifferenz seit dem Setzen 3 Sekunden bzw. 3000 Millisekunden her ist.

Zeile 137: War die Prüfung erfolgreich, rufen wir die Methode stopGame() auf.

Zeile 179 – 181: Die Methode stopGame() enthält den gleichen Code, wie unser KeyListener, wenn wir während des laufenden Spiels die Escape-Taste drücken. Dementsprechend ändern wir unseren KeyListener hier auch noch etwas ab:

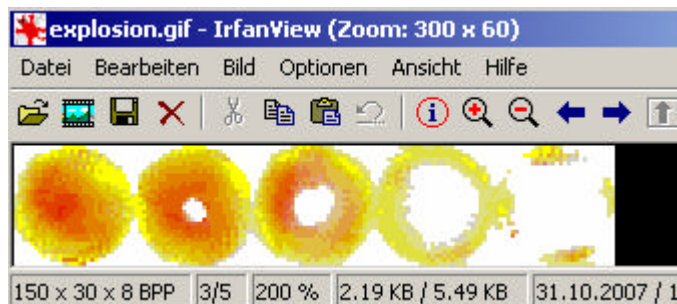
```
320     if(e.getKeyCode()==KeyEvent.VK_ESCAPE){
321         if(isStarted()){
322             setStarted(false);
323             timer.stop();
324             stopGame();
325         }else{
326             setStarted(false);
327             System.exit(0);
328         }
329     }
```

Die beiden rot markierten Zeilen ersetzen wir durch die grün markiert.

Ab jetzt wird das aktuelle Spiel, 3 Sekunden nachdem der Hubschrauber zerstört wurde, beendet. Das Zeitintervall ist willkürlich gewählt. Es soll sicherstellen, dass die Explosionen, die wir gleich noch einbauen auch bis zum Ende animiert werden.

Explosionen

Für den Einbau der Explosionen benötigen wir nichts Neues mehr, daher werde ich dies etwas kürzer halten. Zunächst benötigen wir wieder eine Grafik-Datei mit der Animationsfolge, die wir am üblichen Ort speichern:



Außerdem definieren wir uns einen Klasse Explosion, die

- von Sprite erbt
- in der wir die Logik-Methode überschreiben und
- in der wir die abstrakte collidedWith(.) implementieren.

Uns so sieht sie aus:

```
1 import java.awt.image.BufferedImage;
2
3
4 public class Explosion extends Sprite{
5
6     int old_pic = 0;
7
8     public Explosion(BufferedImage[] i, double x, double y, long delay, GamePanel p) {
9         super(i, x, y, delay, p);
10    }
11
12
13    @Override
14    public void doLogic(long delta) {
15        old_pic = currentpic;
16        super.doLogic(delta);
17        if(currentpic==0 && old_pic!=0){
18            remove = true;
19        }
20    }
21
22
23
24    @Override
25    public boolean collidedWith(Sprite s) {
26
27        return false;
28    }
29
30 }
31
```

Als Besonderheit ist hier anzumerken, dass in der Logik-Methode etwas mehr Code hinterlegt wird, um zu garantieren, dass nach dem ersten Durchlaufen der Animation, das remove-Flag gesetzt wird.

In diesem Fall wird geprüft, ob current_pic das zweite Mal den Wert Null hat. Diese etwas umständliche Prüfung ist notwendig, da die Logik-Methode aus doLogic() unserer Spielschleife aufgerufen wird und dort auch direkt im Anschluss alle Objekte, die das Lösch-Flag gesetzt haben, gelöscht werden. Daher kann man nicht abfragen, ob die Animation beim letzten Bild angelangt ist, da das Objekt sonst gelöscht würde, ohne das letzte Bild je gezeichnet zu haben.

Die Methode collidedWith(Sprite s) lassen wir leer. Wollten wir die anderen Objekte beeinflussen, wenn sie durch die Explosionswolke fliegen, könnten wir hier Code hinterlegen.

In unserer Hauptklasse GamePanel definieren wir uns wieder eine Klassenvariable, um die Bilddaten immer vorrätig zu haben:

```
10 public class GamePanel extends JPanel implements Runnable, KeyListener, ActionListener{
11
12     private static final long serialVersionUID = 1L;
13     boolean game_running = true;
14     boolean started = false;
15     boolean once = false;
16
17     long delta = 0;
18     long last = 0;
19     long fps = 0;
20     long gameover = 0;
21
22     Heli copter;
23     Vector<Sprite> actors;
24
25     boolean up = false;
26     boolean down = false;
27     boolean left = false;
28     boolean right = false;
29     int speed = 50;
30
31     Timer timer;
32     BufferedImage[] rocket;
33     BufferedImage background;
34     BufferedImage explosion;
```

Und füllen Sie in unserer Initialisierungs-Methode mit Leben:

```
53 private void doInitializations(){
54
55     BufferedImage[] heli = loadPics("pics/heli.gif", 4);
56     rocket = loadPics("pics/rocket.gif",8);
57     background = loadPics("pics/background.jpg",1)[0];
58     explosion = loadPics("pics/explosion.gif",5);
59
60     last = System.nanoTime();
61     gameover = 0;
62
63     actors = new Vector<Sprite>();
64     copter = new Heli(heli,400,300,100,this);
65     actors.add(copter);
66
67     createClouds();
68
69     timer = new Timer(3000,this);
70     timer.start();
71
72     if(!once){
73         once = true;
74         Thread t = new Thread(this);
75         t.start();
76     }
77 }
```


Zusätzlich fügen wir noch eine Methode ein, die es uns ermöglicht, eine Explosion an einer bestimmten Stelle zu erzeugen:

```
92 public void createExplosion(int x, int y){
93
94     Explosion ex = new Explosion(explosion,x,y,100,this);
95     actors.add(ex);
96
97 }
```

Danach fügen wir in die Klassen Heli und Rocket noch den Code zum Erzeugen der Explosionen ein:

```
66 @Override
67 public boolean collidedWith(Sprite s) {
68
69     if(remove){
70         return false;
71     }
72
73     if(this.intersects(s)){
74
75
76         if(s instanceof Heli){
77             parent.createExplosion((int)getX(),(int)getY());
78             parent.createExplosion((int)s.getX(),(int)s.getY());
79             remove = true;
80             s.remove = true;
81         }
82
83         if(s instanceof Rocket){
84             parent.createExplosion((int)getX(),(int)getY());
85             parent.createExplosion((int)s.getX(),(int)s.getY());
86             remove = true;
87             s.remove = true;
88         }
89
90         return true;
91     }
92
93     return false;
94 }
```

Dies wird für jeweils beide Objekte durchgeführt, da bei diesen im Anschluss das Entfernen-Flag auf true gesetzt wird und das „Gegner-Objekt“ durch die Bedingung in Zeile 69 dann nicht mehr abgearbeitet würde, wenn es im Rahmen der Kollisionsprüfung an die Reihe käme (bedingt durch den Code in Zeile 69).

Und so sieht's aus:



Hier der Zusammenstoß zweier Raketen.

Pixelgenaue Kollisionsermittlung

Wie zu Beginn der Kollisionsermittlung beschrieben, ist es sehr ungenau, wenn die Kollision nur über die Methoden der Klasse Rectangle2D durchgeführt wird. Gerade wenn man Grafiken verwendet, die relativ viel transparenten Hintergrund enthalten kann so ein Spiel schnell frustrierend werden.

Um dies zu vermeiden, wollen wir den bisherigen Codes so abändern, dass eine genauere Prüfung durchgeführt wird, sobald eine potentielle Kollision erkannt wurde. Hierfür ist jedoch einiges an zusätzlichem Code notwendig.

Da wir den Code für alle Objekte zur Verfügung haben wollen, fügen wir diesen in die Klasse Sprite ein:

Die Methode checkOpaqueColorCollisions(Sprite s):

```
101 public boolean checkOpaqueColorCollisions(Sprite s){
102
103     Rectangle2D.Double cut = (Double) this.createIntersection(s);
104
105     if((cut.width<1) || (cut.height<1)){
106         return false;
107     }
108
109     // Rechtecke in Bezug auf die jeweiligen Images
110     Rectangle2D.Double sub_me = getSubRec(this,cut);
111     Rectangle2D.Double sub_him = getSubRec(s,cut);
112
113     BufferedImage img_me = pics[currentpic].getSubimage((int)sub_me.x,(int)sub_me.y,
114                                                         (int)sub_me.width,(int)sub_me.height);
115     BufferedImage img_him = s.pics[s.currentpic].getSubimage((int)sub_him.x,(int)sub_him.y,
116                                                             (int)sub_him.width,(int)sub_him.height);
117
118     for(int i=0;i<img_me.getWidth();i++){
119         for(int n=0;n<img_him.getHeight();n++){
120
121             int rgb1 = img_me.getRGB(i,n);
122             int rgb2 = img_him.getRGB(i,n);
123
124
125             if(isOpaque(rgb1) && isOpaque(rgb2)){
126                 return true;
127             }
128
129         }
130     }
131
132     return false;
133 }
```

Zur Ermittlung der pixelgenauen Kollision erzeugen wir die Methode oben, die das gegnerische Objekt übergeben bekommt.

Zeile 103: Hier ermitteln wir zunächst das Rechteck, welches die Schnittmenge der beiden kollidierenden Rechtecke darstellt

Zeile 105: Prüfung, dass in Zeile 103 auch wirklich eine verwertbare Schnittmenge erzeugt wurde.

Zeile 110 – 111: Hier errechnen wir mit einer selbstgestellten Methode 2 Rechtecke in Bezug auf die aktuellen Sprites/Images bzw. deren aktuellen Bildschirmposition. Der Code dieser Methode wird weiter unten dargestellt

Zeile 113 – 115: Aus dem aktuell angezeigten Bild der beiden Sprites wird die oben ermittelte Schnittmenge mit `getSubImage(..)` ausgeschnitten

Zeile 118 – 130: Danach wird für jedes einzelne Pixel auf den beiden erzeugten Bildern geprüft, ob es eines gibt, dass auf beiden Bildern nicht transparent ist. Diese Prüfung wird mit der selbst erstellten Methode `isOpaque(..)` durchgeführt (Inhalt siehe weiter unten). Wurde ein Pixel gefunden, dass auf beiden Bildern nicht transparent ist, wird eine Kollision zurück gemeldet.

Die Methode `getSubRectangle(..)`:

```
135 protected Rectangle2D.Double getSubRec(Rectangle2D.Double source, Rectangle2D.Double part) {  
136  
137     //Rechtecke erzeugen  
138     Rectangle2D.Double sub = new Rectangle2D.Double();  
139  
140     //get X - compared to the Rectangle  
141     if(source.x>part.x){  
142         sub.x = 0;  
143     }else{  
144         sub.x = part.x - source.x;  
145     }  
146  
147     if(source.y>part.y){  
148         sub.y = 0;  
149     }else{  
150         sub.y = part.y - source.y;  
151     }  
152  
153     sub.width = part.width;  
154     sub.height = part.height;  
155  
156     return sub;  
157 }
```

Die Methode `getSubRectangle(..)` bekommt 2 `Rectangle2D.Double` übergeben: als erstes das Rechteck des Sprites aus dem die Bilddaten der Schnittmenge ermittelt werden sollen und als 2. eben diese Schnittmenge der beiden Rechtecke.

Die Methode errechnet dann die Koordinaten der Schnittmenge bezogen auf die Position des ersten Rechtecks bzw. des ersten Sprites. Damit ist es später dann möglich, die Schnittmenge der beiden Sprites in grafischer Form zu ermitteln.

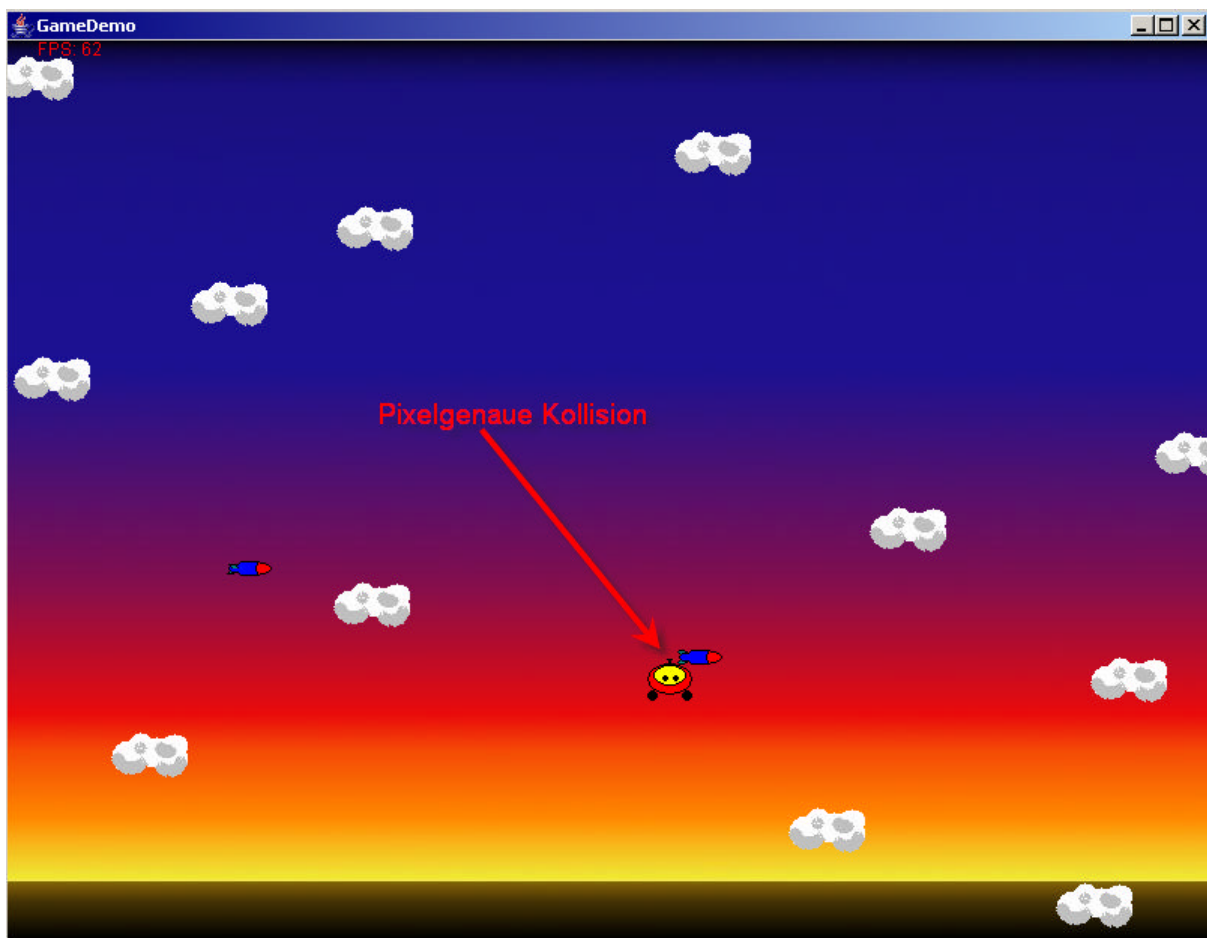
Die Methode isOpaque(int rgb):

```
159 protected boolean isOpaque(int rgb) {  
160  
161     int alpha = (rgb >> 24) & 0xff;  
162     //red    = (rgb >> 16) & 0xff;  
163     //green  = (rgb >>  8) & 0xff;  
164     //blue   = (rgb ) & 0xff;  
165  
166     if(alpha==0) {  
167         return false;  
168     }  
169  
170     return true;  
171  
172 }
```

Die Methode bekommt einen RGB-Wert übergeben und ermittelt den Alpha-Wert durch Bit-Verschiebung.

Farb-Informationen innerhalb des rgb-Wertes lt. API:
(Bits 24-31 = alpha, 16-23 = rot 8-15 = grün, 0-7 = blau).

Und so sieht's aus:



Bzw. auf diesem Bild eben keine Kollision wg. Pixelgenauer Prüfung ☺

Sound

Zu guter Letzt fehlt uns noch der Sound. Ohne diesen ist so ein Spiel nur halb programmiert. In diesem Anfänger-Tutorial wollen wir uns auf eine relative einfache Lösung beschränken. Gerade zum Thema Sound gibt es sehr viele, auch komplexe, Lösungsmöglichkeiten und entsprechende externe Bibliotheken.

Es gibt jedoch auch eine einfache Standard-Lösung die für kleinere Spiele durchaus ausreichend ist.

Diesmal wollen wir eine eigene Klasse zum Verwalten der Sounds verwenden, was letzten Endes viel komfortabler ist – auch unter dem Gesichtspunkt der Wiederverwendung des Codes in späteren Projekten. Zudem wird dann auch der Unterschied z. B. zu den Bilddaten deutlicher, wo wir auf eine eigene Klasse zur Verwaltung der Daten verzichtet haben.

Basis für unsere Sound-Ausgabe ist das Interface AudioClip. Instanzen dieses Interfaces können über die statische Methode `newAudioClip(URL u)` der Klasse `Applet` geladen werden (vgl. API). Es ist möglich, mehrere AudioClip's gleichzeitig abzuspielen, sowie diese wiederholt ablaufen zu lassen (loop), was z. B. bei Fahrzeuggeräuschen sehr nützlich ist. (wiederum siehe API)

Für dieses Anfänger-Tutorial wollen wir uns auf 3 Sounds beschränken, die wir, analog zu den Bilddateien, in einem eigenen Verzeichnis ablegen.



Die Klasse SoundLib

```
1 import java.applet.*;
2 import java.net.*;
3 import java.util.*;
4
5 public class SoundLib {
6
7     Hashtable<String, AudioClip> sounds;
8     Vector<AudioClip> loopingClips;
9
10    public SoundLib() {
11        sounds = new Hashtable<String, AudioClip>();
12        loopingClips = new Vector<AudioClip>();
13    }
14
15    public void loadSound(String name, String path) {
16
17        if(sounds.containsKey(name)) {
18            return;
19        }
20
21        URL sound_url = getClass().getClassLoader().getResource(path);
22        sounds.put(name, (AudioClip)Applet.newAudioClip(sound_url));
23    }
24
25    public void playSound(String name) {
26        AudioClip audio = sounds.get(name);
27        audio.play();
28    }
29
30    public void loopSound(String name) {
31        AudioClip audio = sounds.get(name);
32        loopingClips.add(audio);
33        audio.loop();
34    }
35
36    public void stopLoopingSound() {
37        for(AudioClip c:loopingClips) {
38            c.stop();
39        }
40    }
41
42 }
```

Zeile 7: HashTable um die AudioClips abzuspeichern

Zeile 8: Vector, in dem die AudioClips gespeichert werden, die sich permanent wiederholen. Damit diese später beendet werden können.

Zeile 10 – 13: Konstruktor. Die Collections werden hier instanziiert.

Zeile 15 – 23: Methode zum Laden der AudioClips. Die Methode bekommt einen Namen zum Speichern im HashTable und die Pfadangabe übergeben. Wenn der AudioClip schon im HashTable existiert wird abgebrochen, andernfalls wird der AudioClip geladen und im HashTable abgelegt.

Zeile 25 – 28: Methode zum einmaligen Abspielen eines AudioClips. Der AudioClip wird anhand des übergebenen Namens im HashTable gesucht und mit play() gestartet.

Zeile 30 – 34: Methode zum permanenten Abspielen eines AudioClips. Die entsprechenden AudioClips werden in einem Vector gespeichert.

Zeile 36 – 40: Über diese Methode können alle AudioClips, die permanent abgespielt werden, beendet werden.

Verwenden der Klasse SoundLib

Zunächst definieren wir uns eine Klassenvariable:

```
10 public class GamePanel extends JPanel implements Runnable, KeyListener, ActionListener{
11
12     private static final long serialVersionUID = 1L;
13     boolean game_running = true;
14     boolean started = false;
15     boolean once = false;
16
17     long delta = 0;
18     long last = 0;
19     long fps = 0;
20     long gameover = 0;
21
22     Heli copter;
23     Vector<Sprite> actors;
24
25     boolean up = false;
26     boolean down = false;
27     boolean left = false;
28     boolean right = false;
29     int speed = 50;
30
31     Timer timer;
32     BufferedImage[] rocket;
33     BufferedImage background;
34     BufferedImage[] explosion;
35
36     SoundLib slib;
```


In unserer Initialisierungs-Methode instanziiieren wir die Klasse und versorgen Sie auch gleich mit den wav-Dateien:

```
55 private void doInitializations(){
56
57     BufferedImage[] heli = loadPics("pics/heli.gif", 4);
58     rocket = loadPics("pics/rocket.gif",8);
59     background = loadPics("pics/background.jpg",1)[0];
60     explosion = loadPics("pics/explosion.gif",5);
61
62     slib = new SoundLib();
63     slib.loadSound("bumm", "sound/boom.wav");
64     slib.loadSound("rocket", "sound/rocket_start.wav");
65     slib.loadSound("heli", "sound/heli.wav");
66
67     last = System.nanoTime();
68     gameover = 0;
69
70     actors = new Vector<Sprite>();
71     copter = new Heli(heli,400,300,100,this);
72     actors.add(copter);
73
74     if(isStarted()){
75         slib.loopSound("heli");
76     }
77
78     createClouds();
79
80     timer = new Timer(3000,this);
81     timer.start();
82
83     if(!once){
84         once = true;
85         Thread t = new Thread(this);
86         t.start();
87     }
88 }
```

Zeile 62 – 65: Instanziiieren des Objektes und laden der 3 Sounddateien

Zeile 74 – 76: Wenn die Instanzierungs-Methode aufgerufen wird, um das Spiel zu starten, soll der Helikopter-Sound ständig abgespielt werden.

Nun müssen die Sounds noch den Ereignissen zugeordnet werden:

```
103 public void createExplosion(int x, int y){
104
105     Explosion ex = new Explosion(explosion,x,y,100,this);
106     actors.add(ex);
107     slib.playSound("bumm");
108
109 }
```

Zeile 107: Wenn eine Explosion erzeugt wird, wird der entsprechende Sound ausgelöst

Beim Erzeugen eines Rocket-Objekts lassen wir einen Warnton erzeugen (der im Beispiel zugegebenermaßen sehr nervig ist ☺):

```
273 private void createRocket() {
274
275     int x = 0;
276     int y = (int) (Math.random() * getHeight());
277     int hori = (int) (Math.random() * 2);
278
279     if(hori==0) {
280         x = -30;
281     } else {
282         x = getWidth()+30;
283     }
284
285
286     Rocket rock = new Rocket(rocket, x, y, 100, this);
287     if(x<0) {
288         rock.setHorizontalSpeed(100);
289     } else {
290         rock.setHorizontalSpeed(-100);
291     }
292     actors.add(rock);
293     slib.playSound("rocket");
294
295 }
```

Und schließlich müssen wir noch beim Beenden des Spiels, die Soundschleife für den Helikopter beenden.

```
200 private void stopGame() {
201     timer.stop();
202     slib.stopLoopingSound();
203     setStarted(false);
204 }
```

Das war's ab sofort haben wir ein „richtiges“ Spiel.

Hier gibt's jetzt kein „So sieht's aus“-Kapitel. Sound lässt sich halt leider nicht grafisch abbilden ☺. Aber das Ergebnis kann durch einen Programm-Test selbst begutachtet werden.

Teil2 – tile-basierte Karten

Auf den folgenden Seiten, möchte ich ein Beispiel für ein Spiel darstellen, bei dem über eine tile-basierte Karte gescrollt wird. (engl. Tile = Kachel, Platte).

Da das Beispiel aus dem 1. Teil dafür aber von der grundsätzlichen Konzeption nicht geeignet ist, werde ich hier ein neues Spiel beginnen. Ich möchte mich jedoch auf die grundsätzliche Technik beschränken und das Spiel **nicht bis zu Ende programmieren**. So werde ich z. B. nicht noch einmal auf das Thema Sound eingehen.

Außerdem setze ich voraus, dass jeder eine API hat und dort ggf. Klassen und Methoden nachliest, so dass ich diese großteils nur sehr rudimentär behandle!

Als Beispiel möchte ich hier ein „Rennspiel“ erstellen, wobei ich nur die Bewegung eines Autos über eine tile-basierte Strecke realisieren möchte. Die Karte soll für dieses Beispiel sehr einfach sein, d. h.

- nur 1 Tile pro Position
- keine Animationen für die Tiles
- nur eine Karte (keine zusätzliche Hinter- oder Vordergrund-Map (z.B. Wolken, etc.))
- ...

Dazu sind aber erst einige Grundlagen notwendig. Diese werde ich aus dem Teil1 entnehmen und ggf. etwas anpassen.

GamePanel wird abstract

Zunächst einige Änderungen für die Basics, die ich in relativ groben Zügen erläutern möchte. Das meiste sollte schon bekannt sein, wenn man sich an die Erstellung eines Spiels wagt (z. B. abstrakte Klassen) und den Rest habe ich aus Teil1 entlehnt.

Um nicht alles neu schreiben zu müssen und die Klassen möglichst wieder verwenden zu können, habe ich zunächst das GamePanel aus Teil1 abstract gesetzt und alles was individuell für Teil1 programmiert war raus geworfen. Außerdem habe ich alle Methoden, die für ein Spiel verwendet werden abstract gesetzt, so dass diese in einer Klasse, die von GamePanel erbt, übernommen werden müssen.

Und so sieht sie jetzt aus:

```
1 package game;
2
3 import java.awt.Color;
4
15
16 public abstract class GamePanel extends JPanel implements Runnable, KeyListener, ActionListener{
17
18     private static final long serialVersionUID = 1L;
19     protected boolean game_running = true;
20     protected boolean started = false;
21     boolean once = false;
22
23     protected long delta = 0;
24     protected long last = 0;
25     protected long fps = 0;
26     public long gameover = 0;
27
28     protected boolean up = false;
29     protected boolean down = false;
30     protected boolean left = false;
31     protected boolean right = false;
32
33     Timer timer;
34
35     public GamePanel(int w, int h){
36         this.setPreferredSize(new Dimension(w,h));
37         JFrame frame = new JFrame("GameDemo");
38         frame.setLocation(100,100);
39         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
40         frame.addKeyListener(this);
41         frame.add(this);
42         frame.pack();
43         frame.setVisible(true);
44         doInitializations();
45     }
46
47     protected void doInitializations(){
48
49         timer = new Timer(3000,this);
50         timer.start();
51
52         if(!once){
53             once = true;
54             Thread t = new Thread(this);
55             t.start();
56         }
57     }
58
59
```

....

```

61 public void run() {
62
63     while (game_running) {
64
65         computeDelta();
66
67         if (isStarted()) {
68             checkKeys();
69             doLogic();
70             moveObjects();
71         }
72
73         repaint();
74
75         try {
76             Thread.sleep(10);
77         } catch (InterruptedException e) {}
78
79     }
80
81 }
82
83 protected abstract void checkKeys();
84
85 protected abstract void doLogic();
86
87 protected void stopGame() {
88     timer.stop();
89     setStarted(false);
90 }
91
92 protected abstract void moveObjects();
93
94 @Override
95 protected void paintComponent(Graphics g) {
96     super.paintComponent(g);
97
98     g.setColor(Color.red);
99     g.drawString("FPS: " + Long.toString(fps), 20, 10);
100
101     if (!isStarted()) {
102         return;
103     }
104
105     paintAll(g);
106 }
107
108 public abstract void paintAll(Graphics g);
109
110 protected void computeDelta() {
111     delta = System.nanoTime() - last;
112     last = System.nanoTime();
113     fps = ((long) 1e9) / delta;
114 }
115
116 protected BufferedImage[] loadPics(String path, int pics) {
117
118     BufferedImage[] anim = new BufferedImage[pics];
119     BufferedImage source = null;
120
121     URL pic_url = getClass().getClassLoader().getResource(path);
122
123     try {
124         source = ImageIO.read(pic_url);
125     } catch (IOException e) {}
126
127     for (int x = 0; x < pics; x++) {
128         anim[x] = source.getSubimage(x * source.getWidth() / pics, 0,
129             source.getWidth() / pics, source.getHeight());
130     }
131
132     return anim;
133 }
134
135
136
137 public boolean isStarted() {
138     return started;
139 }
140
141 public void setStarted(boolean started) {
142     this.started = started;
143 }
144
145
146 }

```

Die neue Hauptklasse

Diese abstrakte Klasse wollen wir jetzt gleich einmal verwenden, um die Hauptklasse für das neue Spiel zu schreiben.

Diese werde ich der Einfachheit halber einmal ScrollGame nennen. Nachdem alle abstrakten Methoden implementiert wurden, sieht die Grundform so aus:

```
1 package main;
2
3 import game.GamePanel;
4
5 import java.awt.Graphics;
6 import java.awt.event.ActionEvent;
7 import java.awt.event.KeyEvent;
8
9 public class ScrollGame extends GamePanel {
10
11     private static final long serialVersionUID = 1L;
12
13
14     public static void main(String[] args) {
15
16     }
17
18     public ScrollGame(int w, int h) {
19         super(w, h);
20     }
21
22     @Override
23     protected void checkKeys() {
24         // TODO Auto-generated method stub
25     }
26
27
28
29     @Override
30     protected void doLogic() {
31         // TODO Auto-generated method stub
32     }
33
34
35
36     @Override
37     protected void moveObjects() {
38         // TODO Auto-generated method stub
39     }
40
41
42
43     @Override
44     public void paintAll(Graphics g) {
45
46     }
47
48 }
```

```

49 public void keyTyped(KeyEvent arg0) {
50     // TODO Auto-generated method stub
51 }
52
53
54
55 public void actionPerformed(ActionEvent arg0) {
56     // TODO Auto-generated method stub
57 }
58
59
60
61 public void keyPressed(KeyEvent e) {
62     if (e.getKeyCode() == KeyEvent.VK_UP) {
63         up = true;
64     }
65
66     if (e.getKeyCode() == KeyEvent.VK_DOWN) {
67         down = true;
68     }
69
70     if (e.getKeyCode() == KeyEvent.VK_LEFT) {
71         left = true;
72     }
73
74     if (e.getKeyCode() == KeyEvent.VK_RIGHT) {
75         right = true;
76     }
77 }
78
79
80
81 public void keyReleased(KeyEvent e) {
82     if (e.getKeyCode() == KeyEvent.VK_UP) {
83         up = false;
84     }
85
86     if (e.getKeyCode() == KeyEvent.VK_DOWN) {
87         down = false;
88     }
89
90     if (e.getKeyCode() == KeyEvent.VK_LEFT) {
91         left = false;
92     }
93
94     if (e.getKeyCode() == KeyEvent.VK_RIGHT) {
95         right = false;
96     }
97
98
99
100    if (e.getKeyCode() == KeyEvent.VK_ENTER) {
101        if (!isStarted()) {
102            setStarted(true);
103            doInitializations();
104        }
105    }
106
107    if (e.getKeyCode() == KeyEvent.VK_ESCAPE) {
108        if (isStarted()) {
109            stopGame();
110        } else {
111            setStarted(false);
112            System.exit(0);
113        }
114    }
115
116 }
117
118
119 }

```

Wichtig: Die Methoden keyPressed(..) und keyReleased(..) werden natürlich nicht automatisch mit dem oben gezeigten Code gefüllt. Diesen habe ich der Bequemlichkeit halber aus dem GamePanel von Teil1 des Tutorials heraus kopiert.

Ein erster Test

Jetzt noch ein paar Modifikationen, dann können wir testen, ob soweit alles funktioniert:

Zuerst muß die main-Methode mit Leben gefüllt werden:

```
12 public static void main(String[] args) {  
13     new ScrollGame(800, 600);  
14 }
```

Hier wird der Konstruktor des GamePanels aufgerufen. Ich habe hier subjektiv eine Größe von 800 x 600 Pixel gewählt.

Weiterhin überschreiben wir die Methode doInitializations, damit diese auch für unsere Initialisierungen zur Verfügung steht:

```
21 @Override  
22 protected void doInitializations() {  
23     super.doInitializations();  
24  
25  
26 }
```

Wichtig ist, dass nicht vergessen wird, die Methode der Vaterklasse mit super aufzurufen, so dass der dort hinterlegte Code ebenfalls ausgeführt wird.

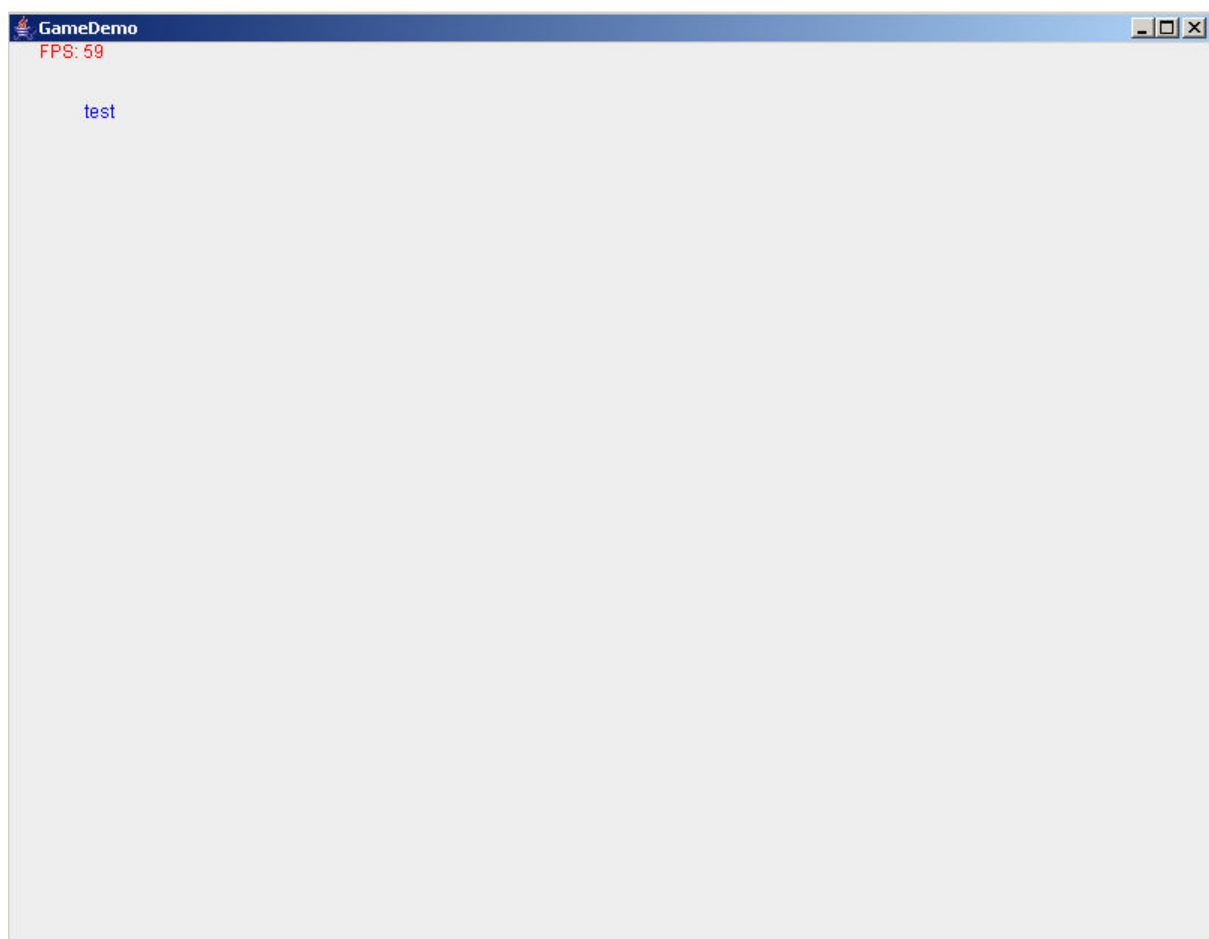
Temporär fügen wir in die paint-Methode noch etwas Code ein:

```
52 @Override  
53 public void paintAll(Graphics g) {  
54  
55     g.setColor(Color.blue);  
56     g.drawString("test", 50, 50);  
57  
58 }
```


Außerdem soll das nervige Türkis nicht mehr Hintergrundfarbe des ungestarteten Spiels sein. Also weg damit. In der Klasse GamePanel entfernen wir noch diese Zeile:

```
35 public GamePanel(int w, int h) {  
36     this.setPreferredSize(new Dimension(w,h));  
37     this.setBackground(Color.Cyan);  
38     JFrame frame = new JFrame("GameDemo");  
39     frame.setLocation(100,100);  
40     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
41     frame.addKeyListener(this);  
42     frame.add(this);  
43     frame.pack();  
44     frame.setVisible(true);  
45     doInitializations();  
46 }
```

Und so sieht's aus:



Nicht anderes, als wir es auch aus dem 1. Teil kennen, mit dem Unterschied, dass die Klasse GamePanel ab sofort wieder verwendbar ist. Für dauerhaften Einsatz wäre natürlich noch etwas Feinschliff notwendig, so dass z. B. der Titel für den JFrame übergeben werden kann. Für unser Demo hier soll es aber ausreichend sein.

Wiederverwendung bereits erstellter Klassen:

Um uns weitere Arbeit zu sparen, kopieren wir uns einige Klassen, die wir im Teil 1 erstellt haben in dieses Projekt. Hierbei sind u. U. einige Import-Anweisungen anzupassen. Selbstverständlich wäre es möglich, auf die Klassen von Teil 1 zu referenzieren, für dieses Tutorial möchte ich allerdings alle Klassen in einem Paket haben, daher habe ich mich dafür entschieden, diese in das neue Projekt zu kopieren.

Eine SpriteLib

Im Teil 1 haben wir auf eine eigene Klasse zur Verwaltung der Sprites verzichtet. Da uns dies zwar Arbeit erspart hat, uns aber auch Bequemlichkeit gekostet hat, wollen wir dies jetzt nachholen und uns analog zur Sound-Bibliothek eine Klasse für die Sprites basteln.

Um bequem von überall her auf die SpriteLib zugreifen zu können, soll diese als Singleton konstruiert werden:

```
1 package sprite;
2
3 import java.awt.image.BufferedImage;
4 import java.net.URL;
5 import java.util.HashMap;
6
7 public class SpriteLib {
8
9     private static SpriteLib single;
10    private static HashMap<URL, BufferedImage> sprites;
11
12    public static SpriteLib getInstance() {
13        if(single==null){
14            single = new SpriteLib();
15        }
16        return single;
17    }
18
19    private SpriteLib(){
20        sprites = new HashMap<URL, BufferedImage>();
21    }
22
23
24
25 }
26
```

Hierzu wird der Konstruktor als private deklariert. Eine Instanziierung der SpriteLib erfolgt dann über die statische Methode getInstance(), die das statische SpriteLib-Objekt, das wir als Klassenvariable deklariert haben übergibt bzw. instanziiert. Zusätzlich definieren wir noch eine HashMap, in der wir über die URL unsere Images ablegen können. Somit ist es eigentlich eher eine Image-Bibliothek als eine Sprite-

Bibliothek. Da aber die Images der Teil sind, der nicht verändert wird, wollen wir uns hier nicht auf Haarspaltereien einlassen ☺

Laden eines einzelnen Images

Zum Laden eines einzelnen Bildes wollen wir die folgende Methode verwenden, die als Übergabeparameter die URL des zu ladenden Bildes übergeben bekommt:

```
26 public BufferedImage getSprite(URL location) {
27
28     BufferedImage pic = null;
29
30     pic = (BufferedImage) sprites.get(location);
31
32     if (pic != null) {
33         return pic;
34     }
35
36     try {
37         pic = ImageIO.read(location); // Über ImageIO die GIF lesen
38     } catch (IOException e1) {
39         System.out.println("Fehler beim Image laden: " + e1);
40         return null;
41     }
42
43     sprites.put(location, pic);
44
45     return pic;
46 }
47
```

Zeile 28: Wir definieren eine Variable vom Typ BufferedImage.

Zeile 30: Es wird versucht, die Bilddaten aus unserer HashMap zu lesen.

Zeile 32 – 34: Haben wir Bilddaten gefunden, können wir diese zurück geben.

Zeile 36 – 41: Wurden in unserer HashMap für die URL kein Eintrag gefunden, versuchen wir, das Bild zu laden.

Zeile 43: Anschließend speichern wir das Bild in unserer HashMap.

Zeile 45: Abschließen geben wir das Bild an den Methoden-Aufrufer zurück.

Um das Ganze komfortabler gestalten zu können, wollen wir es noch möglich machen, dass anstelle einer URL die Pfadangabe als String übergeben zu können. Hierfür fügen wir noch folgende Methode ein:

```
48 public URL getURLfromRessource(String path){
49     return getClass().getClassLoader().getResource(path);
50 }
```

Diese sollte noch aus Teil1 bekannt sein, wo wir damit unsere Pfadangaben in URL's umgewandelt haben.

Dementsprechend ist es jetzt möglich, die oben erstellte Methode zu überladen und eine zusätzliche komfortablere Lösung zum Laden von BufferedImages bereit zu stellen.

Somit müssen wir die oben erstellte Methode nur noch einmal kopieren und 2 Änderungen vornehmen, um diese zu Überladen:

```
48 public BufferedImage getSprite(String path) {  
49  
50     BufferedImage pic = null;  
51     URL location = getURLfromRessource(path);  
52  
53     pic = (BufferedImage) sprites.get(location);  
54  
55     if (pic != null) {  
56         return pic;  
57     }  
58  
59     try {  
60         pic = ImageIO.read(location); // Über ImageIO die GIF lesen  
61     } catch (IOException e1) {  
62         System.out.println("Fehler beim Image laden: " + e1);  
63         return null;  
64     }  
65  
66     sprites.put(location, pic);  
67  
68     return pic;  
69 }
```

Zeile 48: Anstelle einer URL wird jetzt eine Pfadangabe als String übergeben

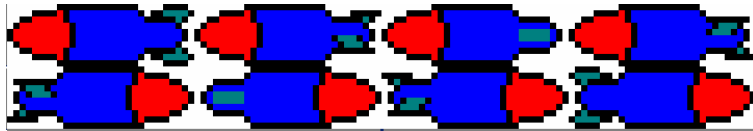
Zeile 51: Diese Pfadangabe wird über die zuvor erstellte Methode in eine URL umgewandelt.

Laden von Animationen

Da es auch möglich sein soll, Animations-Sequenzen, die in einem Image hinterlegt sind auszulesen, wollen wir eine weitere Methode implementieren. Diese Methode soll Sub-Images einer festen Größe auslesen und als eindimensionales Array übergeben.

Die Verwendung eines eindimensionalen Arrays entspricht z. T. einer persönlichen Gewohnheit, allerdings habe ich bis jetzt auch noch nicht das Bedürfnis nach einer zweidimensionalen Lösung verspürt.

Mit der gleich vorgestellten Methode wird es möglich sein, eine Animation übersichtlicher zu zeichnen, wie z. B. auf folgendem Bild dargestellt:



Die o. a. Animation hatten wir im Teil1 noch hintereinander gezeichnet. Dies würde bei komplexeren Animationen aber sehr schnell unübersichtlich. Hier haben wir je eine Zeile für die Bewegung nach links und eine Zeile für die Bewegung nach rechts. Mit der im Folgenden dargestellten Methode würde daraus ein Bilder-Array mit den Hausnummer 0 – 7, so dass wir die Animation wie im Teil 1 einstellen können, nämlich so, dass bei einer Bewegung nach links die Bilder 0 – 3 verwendet werden und bei einer Bewegung nach rechts die Bilder 4 – 7.

```

72 public BufferedImage[] getSprite(String path, int column, int row) {
73
74     URL location = getURLfromResource(path);
75
76     BufferedImage source = null;
77
78     source = (BufferedImage) sprites.get(location);
79
80     if (source == null) {
81         try {
82             source = ImageIO.read(location); // Über ImageIO die GIF lesen
83         } catch (IOException e1) {
84             System.out.println(e1);
85             return null;
86         }
87
88         sprites.put(location, source);
89
90     }
91
92     int width = source.getWidth() / column;
93     int height = source.getHeight() / row;
94
95     BufferedImage[] pics = new BufferedImage[column * row];
96     int count = 0;
97
98     for (int n = 0; n < row; n++) {
99         for (int i = 0; i < column; i++) {
100             pics[count] = source.getSubimage(i * width, n * height, width, height);
101             count++;
102         }
103     }
104
105     return pics;
106
107 }

```

Die dargestellte Methode bekommt die Pfadangabe als String übergeben (wir werden dies im Anschluss aber auch überladen). Zusätzlich erhalten wir noch die Information aus wie viel Zeilen und Spalten die Animation besteht.

Zeile 72 – 90: Bis hier entspricht der Code der oben vorgestellten Methode zum Laden eines einzelnen Bildes.

Zeile 92 – 93: Hier werden die Dimensionen der „Unter-Bilder“ (Subimages klingt doch besser 😊) anhand der übergebenen Parameter errechnet.

Zeile 95 – 96: Hier wird das Array aus BufferedImages erzeugt und ein Zähler für „Hausnummer“ initialisiert.

Zeile 98 – 103: Mit den beiden Schleifen werden jetzt Bildstücke ausgeschnitten und in das Array „verpackt“. Zum Erzeugen der Bildstücke wird die Standard-Methode `getSubimage(..)` der Klasse `BufferedImage` zurück gegriffen.

Um die Methode zu überladen und die URL direkt im Methoden-Aufruf einzugeben, müssen wir wieder nur 2 Zeilen ändern. Der grün markierte Bereich wird überschrieben, der rot markierte Bereich gelöscht:

```
115 public BufferedImage[] getSprite(URL location, int column, int row) {
116
117     URL location = getURLfromRessource(path);
118
119     BufferedImage source = null;
120
121     source = (BufferedImage) sprites.get(location);
122
123     if (source == null) {
124         try {
125             source = ImageIO.read(location); // Über ImageIO die GIF lesen
126         } catch (IOException e1) {
127             System.out.println(e1);
128             return null;
129         }
130
131         sprites.put(location, source);
132
133     }
134
135     int width = source.getWidth() / column;
136     int height = source.getHeight() / row;
137
138     BufferedImage[] pics = new BufferedImage[column * row];
139     int count = 0;
140
141     for (int n = 0; n < row; n++) {
142         for (int i = 0; i < column; i++) {
143             pics[count] = source.getSubimage(i * width, n * height, width, height);
144             count++;
145         }
146     }
147
148     return pics;
149
150 }
```

Unsere SpriteLib enthält bis jetzt folgende Methoden:

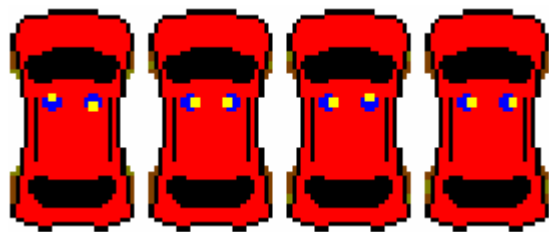
```
1 package sprite;
2
3 import java.awt.image.BufferedImage;
4
5
6
7
8
9 public class SpriteLib {
10
11     private static SpriteLib single;
12
13     private static HashMap<URL, BufferedImage> sprites;
14
15     public static SpriteLib getInstance() {}
16
17
18
19
20
21     private SpriteLib() {}
22
23
24
25
26     public BufferedImage getSprite(URL location) {}
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46     public BufferedImage getSprite(String path) {}
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71     public BufferedImage[] getSprite(String path, int column, int row) {}
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108     public URL getURLfromRessource(String path) {}
109
110
111
112     public BufferedImage[] getSprite(URL location, int column, int row) {}
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147 }
148
```

Anmerkung: Ich habe hier der Übersichtlichkeit halber, das Coding in Eclipse „eingeklappt“, so dass nur die Methodenköpfe zu sehen sind.

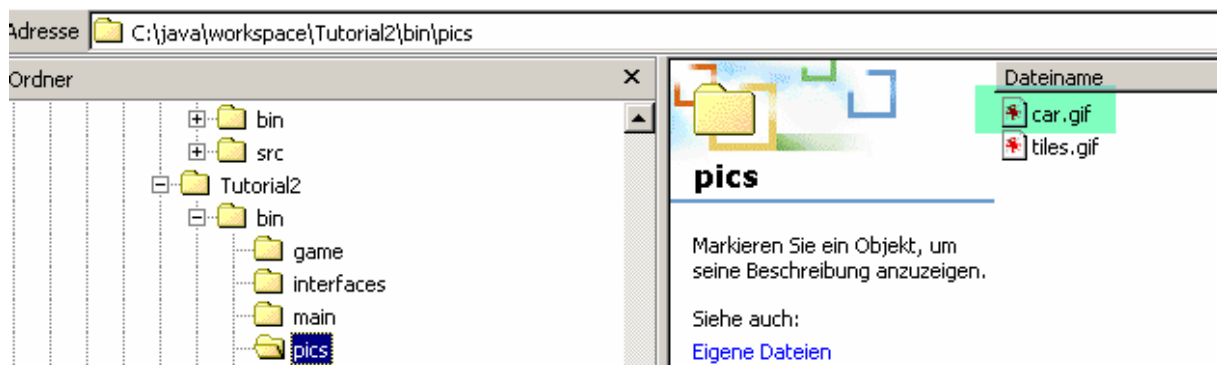
Das Spieler-Objekt

Als nächstes wollen wir ein Spieler-Objekt einbauen. Damit können wir dann auch gleich die SpriteLib testen. Zunächst einmal brauchen wir eine Grafik-Datei, die ein animiertes Objekt enthält.

Hier ist es:



Wie schon im 1. Teil legen wir es in einem eigenen Ordner ab:



Als nächstes definieren wir uns eine Klasse Car, die von Sprite erbt (analog zum 1. Teil).

Da wir Sprite schon im 1. Teil als abstrakt definiert haben, sieht die Klasse Car nach dem Anlegen und implementieren der abstrakten Methoden so aus:

```
1 package player;
2
3 import game.GamePanel;
4
5 import java.awt.image.BufferedImage;
6
7 import sprite.Sprite;
8
9 public class Car extends Sprite {
10
11     public Car(BufferedImage[] i, double x, double y, long delay, GamePanel p) {
12         super(i, x, y, delay, p);
13     }
14
15     @Override
16     public boolean collidedWith(Sprite s) {
17         return false;
18     }
19
20 }
```

Für den ersten Test wollen wir jetzt nur ein paar kleine Änderungen vornehmen:

```

1 package player;
2
3 import game.GamePanel;
4 import java.awt.image.BufferedImage;
5 import main.ScrollGame;
6 import sprite.Sprite;
7
8 public class Car extends Sprite {
9
10     ScrollGame parent;
11
12     public Car(BufferedImage[] i, double x, double y, long delay, GamePanel p) {
13         super(i, x, y, delay, p);
14         parent = (ScrollGame) p;
15     }
16
17     public void doLogic(long delta) {
18         super.doLogic(delta);
19     }
20
21
22     @Override
23     public boolean collidedWith(Sprite s) {
24         return false;
25     }
26
27 }

```

Zeile 10 und 14: Referenz auf unser GamePanel

Zeile 17 – 19: Da wir diese später noch benötigen, überschreiben wir schon mal die doLogic-Methode.

Nun füllen wir das Ganze in unserem GamePanel bzw. vielmehr in der Klasse ScrollGame mit Leben:

```
1 package main;
2
3 import game.GamePanel;
4 import java.awt.Graphics;
5 import java.awt.event.ActionEvent;
6 import java.awt.event.KeyEvent;
7
8 import player.Car;
9 import sprite.SpriteLib;
10
11 public class ScrollGame extends GamePanel {
12
13     private static final long serialVersionUID = 1L;
14
15     SpriteLib lib;
16     Car car;
17
18     public static void main(String[] args) {
19         new ScrollGame(800,600);
20     }
21
22     public ScrollGame(int w, int h) {
23         super(w, h);
24     }
25
26
27     @Override
28     protected void doInitializations() {
29         super.doInitializations();
30
31         lib = SpriteLib.getInstance();
32         car = new Car(lib.getSprite("pics/car.gif", 4, 1),400,300,100,this);
33
34     }
35
36     @Override
37     protected void checkKeys() {
38         // TODO Auto-generated method stub
39
40     }
```

Zeile 15: Unsere SpriteLib als Klassenvariable

Zeile 16: ein Car-Objekt als Klassenvariable

Zeile 31: Hier holen wir uns eine Instanz für unsere SpriteLib ab

Zeile 32: Hier füllen wir den Konstruktor des Car-Objektes. Das Ganze ist etwas komprimiert dargestellt. Für Java-Anfänger ist das Ganze weiter unten etwas einfacher dargestellt.

Weitere Änderungen in ScrollGame:

```
42
43 @Override
44 protected void doLogic() {
45     car.doLogic(delta);
46
47 }
48
49
50 @Override
51 protected void moveObjects() {
52     car.move(delta);
53
54 }
55
56
57 @Override
58 public void paintAll(Graphics g) {
59     car.drawObjects(g);
60 }
61
```

Zeile 45: Für unser Car-Objekt rufen wir die doLogic-Methode auf

Zeile 52: Analog zu Zeile 45, die move-Methode

Zeile 59: Und schließlich müssen wir das Ganze noch zeichnen

Zur Erinnerung:

delta ist die Zeit, die für den letzten Schleifendurchlauf benötigt wurde.

Anmerkung:

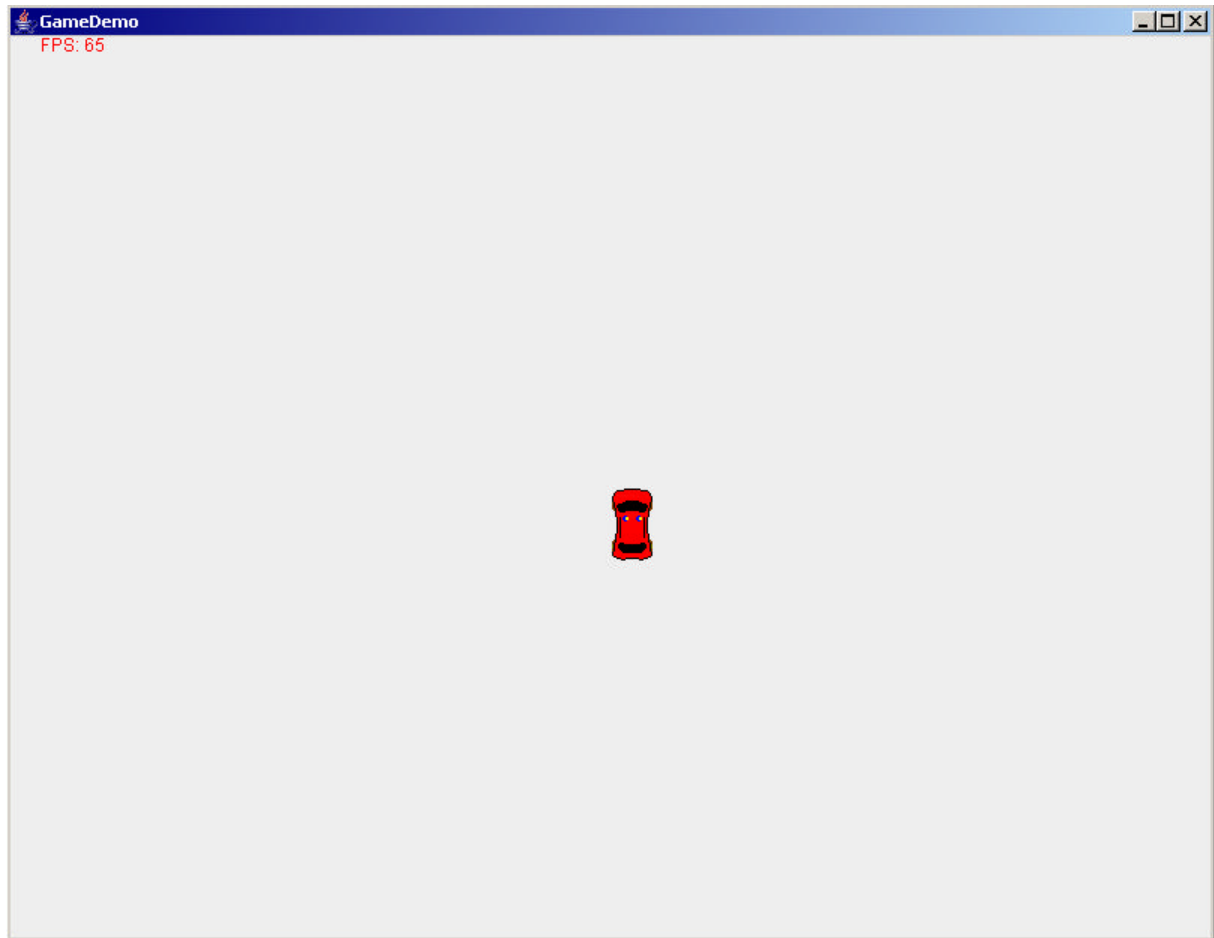
Die folgende grün markierte Zeile

```
lib = SpriteLib.getInstance();
car = new Car(lib.getSprite("pics/car.gif", 4, 1), 400, 300, 100, this);
```

ist natürlich eine Kurzform für:

```
lib = SpriteLib.getInstance();
BufferedImage[] animation = lib.getSprite("pics/car.gif", 4, 1);
car = new Car(animation, 400, 300, 100, this);
```

Und so sieht's aus:

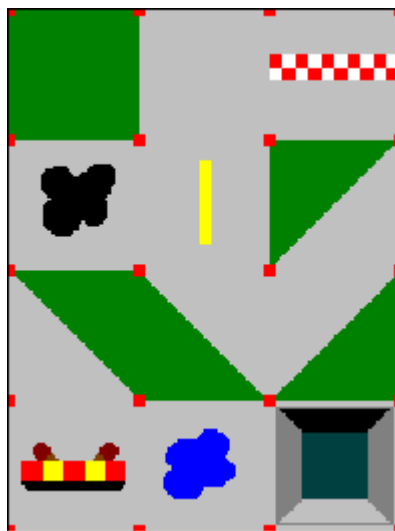


Die Karte

Zunächst möchte ich auf einige Vorbedingungen und Grundlagen eingehen.

Die Tiles

Grundlage für die tilebasierte Karte soll ein sehr einfaches Tilesset sein, das gerade mal 12 einfach gestrickte Tiles enthält. Die Verarbeitung wird über die neue SpriteLib erfolgen, so dass die einzelnen Tiles von links nach rechts zeilenweise durchnummeriert werden. Somit wird das erste Tile links oben die Nr. 0 und das letzte Tile rechts unten die Nr. 11 erhalten.



Die Tiles für das Tutorial wurden der Einfachheit halber mit MS Paint erstellt und haben eine Größe von 50 x 50 Pixel, was für uns hier ausreichend sein soll. Je nach Art des Spiels, gewünschtem Detailgrad, Variierbarkeit, etc. ist eine andere Tile-Größe sinnvoll. Dies sollte jeder vor der Realisierung eines Spiels selbst durchdenken.

Anmerkung: Die kleinen, roten Vierecke sind meine persönliche Hilfe beim Entwerfen eines Tilessets und markieren die Eckpunkte der Tiles. Üblicherweise entferne ich diese erst, wenn ich sicher bin, dass alles wie gewünscht funktioniert. Darunter fällt dann z.B. das Aufteilen des Tilessets, das Verschieben der Karte innerhalb des Spiels, etc..

Die Datendatei

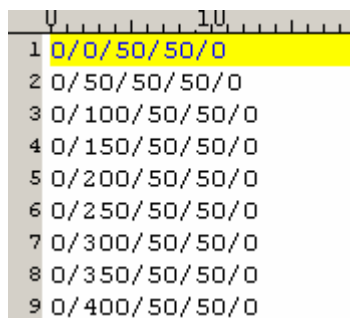
Für die Datendatei möchte ich ein normales Textfile mit sehr einfacher Struktur verwenden. Für komplexere Karten ist hier eine komplexere Struktur denkbar (evtl. xml-basiert). Für unser kleines Projekt mit gleichförmiger Datenstruktur soll die folgende, sehr einfache Struktur ausreichen, die dafür den Vorteil hat, dass sie sehr

einfach zu verarbeiten ist (auch wenn mir xml-begeisterte Leser jetzt widersprechen würden ☺)

In der Datei sind pro Zeile Informationen für 1 Tile hinterlegt:

- die x-Position
- die y-Position
- die Breite (bei einheitlichen Tiles ist diese Info eigentlich unnötig)
- die Höhe (dito)
- die Nummer des Tiles aus dem Tileset

Hier ein Beispiel:



1	0/0/50/50/0
2	0/50/50/50/0
3	0/100/50/50/0
4	0/150/50/50/0
5	0/200/50/50/0
6	0/250/50/50/0
7	0/300/50/50/0
8	0/350/50/50/0
9	0/400/50/50/0

Gemäß diesem Beispiel würde aus der ersten Zeile folgendes Tile entstehen:

Position: (0/0) = linke obere Ecke des Bildschirms

Dimension: Breite x Höhe = 50 x 50

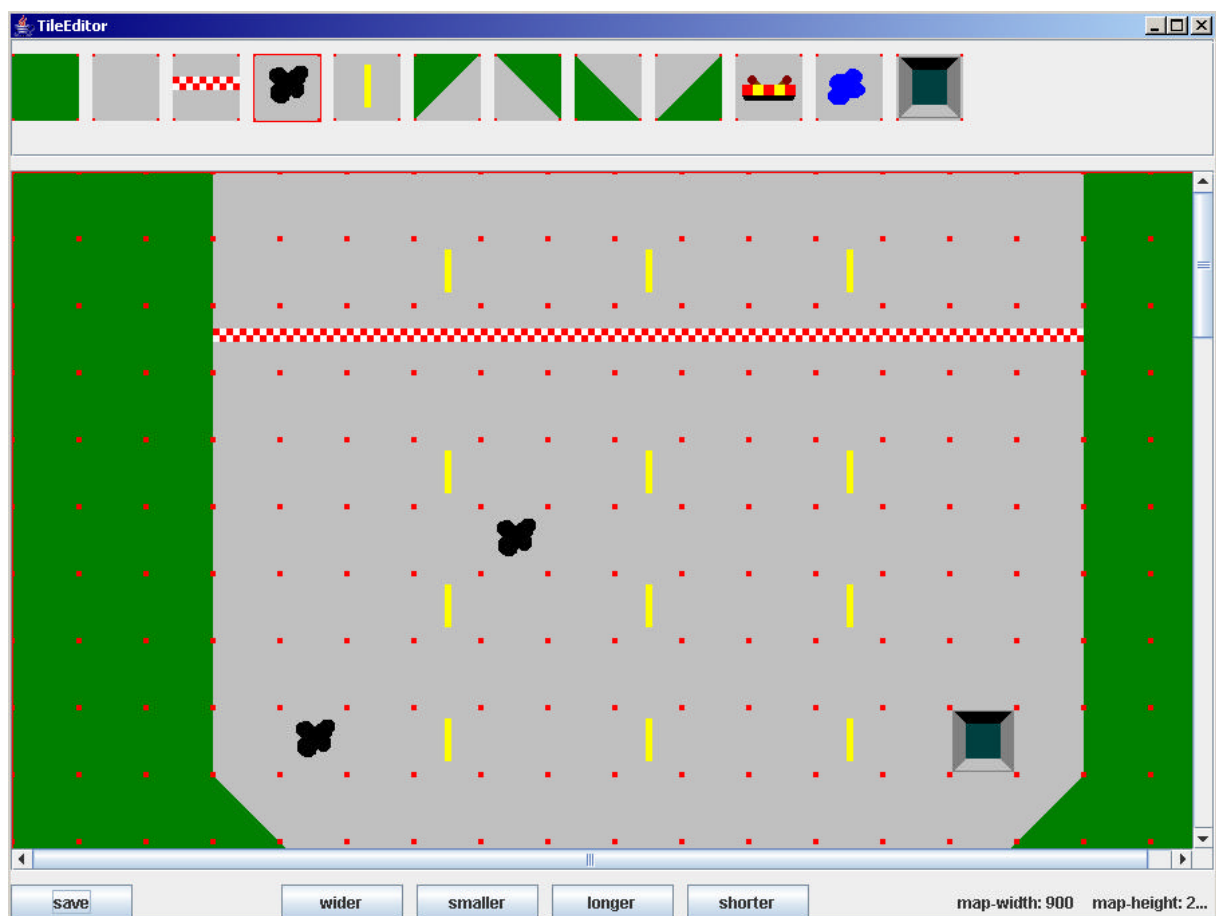
Bildnummer: 0 = die „grüne Wiese“ aus unserem Tileset.

Auf die Erstellung der Datendatei möchte ich hier nur kurz eingehen. Prinzipiell gibt es 3 Möglichkeiten:

1. Von Hand erstellen. Ein sehr aufwändige und fehleranfällige Herangehensweise, die wohl niemand ernsthaft in Betracht ziehen wird.
2. Verwendung entsprechender Freeware. Google ist hier Dein Freund☺. Es gibt im Internet einige gute Editoren, bei denen sich zum Teil auch das Ausgabe-File individuell anpassen lässt. Dafür muss dann die Routine zum Einlesen des Files angepasst werden.
3. Selbst einen Editor schreiben. Relativ aufwändig, hat aber den Vorteil, dass man eine Datei erhält, die exakt den eigenen Vorstellungen entspricht. Außerdem ist dieser später wieder verwendbar und je nach Konzept könnte man diesen u. U. sogar in das Spiel einbinden.

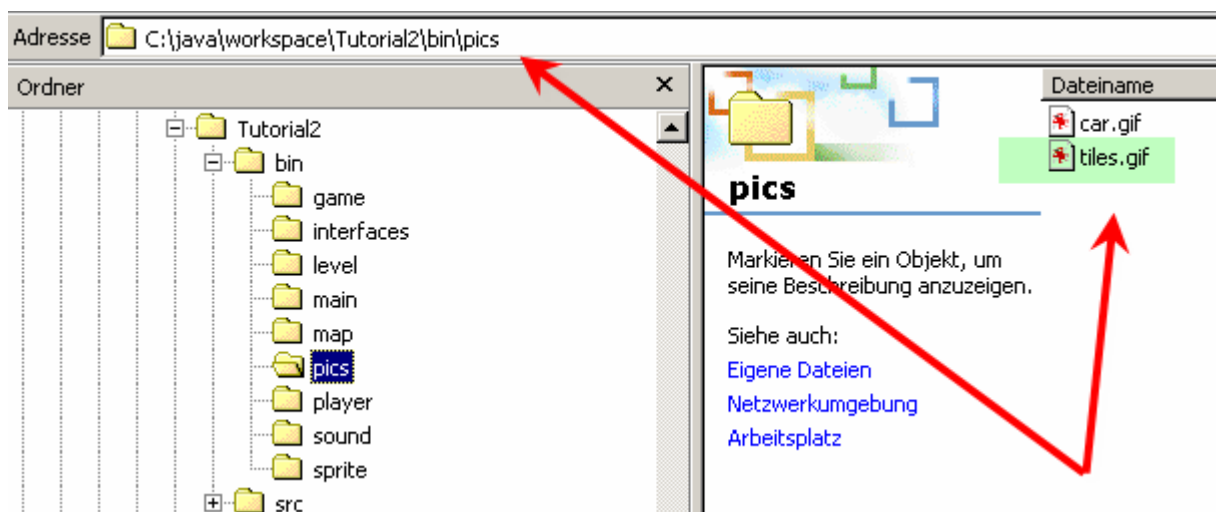
Ich selbst benutze einen einfachen, selbstgeschriebenen grafischen Editor, der mir eine Datei in der oben angezeigten, einfachen Struktur erstellt. Auf die Programmierung des Editors werde ich nicht eingehen und er ist auch nicht Teil des Quellcodes zum Tutorial. Von der Funktionsweise her, arbeitet er in etwa, wie die im Folgenden dargestellte Vorgehensweise für tilebasierte Karten.

Hier ein Beispiel für den selbsterstellten Map-Editor:



Die aus diesen Vorarbeiten resultierenden Daten werden wie folgt gespeichert:

Das Tileset am selben Ort, wo wir schon die Grafikdaten für das Auto hinterlegt haben.



Die Datei für die Erzeugung der Tiles analog dazu in einem eigenen Ordner.



Verwaltung der Bilddaten

Vorteil einer tilebasierten Karte ist, dass die abgespeicherten Bilddaten im Vergleich zur Größe der Karte relativ wenig Speicherplatz benötigen. Um zu gewährleisten, dass dies so bleibt und um das Spiel nicht durch unnötigen Speicherplatz-Verbrauch zu verlangsamen, wollen wir möglichst vermeiden, die Bilddateien in jedem Tile einzeln zu hinterlegen.

Dazu ein Beispiel: Die Leveldatei zu diesem Tutorial enthält ca. 3200 Zeilen, was 3200 Tiles entspricht. Damit ist diese Datei noch relativ klein. 280 Einträge davon stehen für das 1. Tile (die grüne Wiese). Wenn wir nun die Bildinformationen bei jedem Tile hinterlegen würden, würden wir 280 mal das gleiche Bild in ein Objekt packen, was den Speicherbedarf des Spiels erheblich aufblähen würde.

Aus diesem Grund wollen wir uns zunächst eine Klasse zur Verwaltung der Bilddaten anlegen. Dort werden wir die Einzelbilder durchnummerieren. Durch Verwendung unserer SpriteLib ist sichergestellt, dass die Nummerierung der Einzelbilder, der Nummerierung aus der Level-Datei entspricht (ein weiterer Vorteil, wenn man seinen Editor selbst schreibt☺).

Die Klasse zur Verwaltung der Bilder möchte ich ImageControl nennen. Da diese auch des Öffern aus verschiedenen Klassen aufgerufen wird, werde ich sie auch als Singleton konzipieren.

Die Klasse ImageControl:

```
3 import java.awt.image.BufferedImage;
5
6 public class ImageControl {
7
8     BufferedImage[] tiles;
9     private static ImageControl instance;
10
11     public static ImageControl getInstance(){
12         if(instance==null){
13             instance = new ImageControl();
14         }
15
16         return instance;
17     }
18
19     private ImageControl(){
20         tiles = null;
21     }
22
23
24 }
```

Hier zunächst der Teil, der dafür sorgt, dass die Klasse als Singleton verwendet wird. Der Konstruktor ist private und das eigentliche Objekt erhält man über die getInstance()-Methode.

Zusätzlich habe ich als Klassenvariabel ein Array aus BufferedImages zum Speichern der Bilddaten angelegt.

Im Anschluss daran noch die Methode zum Laden der Bilddaten und einige Methoden zum Abrufen zusätzlicher Informationen:

```

3+ import java.awt.image.BufferedImage;
5
6 public class ImageControl {
7
8     BufferedImage[] tiles;
9     private static ImageControl instance;
10
11 public static ImageControl getInstance(){
12     if(instance==null){
13         instance = new ImageControl();
14     }
15
16     return instance;
17 }
18
19 private ImageControl(){
20     tiles = null;
21 }
22
23 public void setSourceImage(String path, int col, int row){
24     SpriteLib lib = SpriteLib.getInstance();
25     tiles = lib.getSprite(path, col, row);
26 }
27
28 public BufferedImage getImageAt(int num){
29     return tiles[num];
30 }
31
32 public int getTileWidth(int n){
33     return tiles[n].getWidth();
34 }
35
36 public int getTileHeight(int n){
37     return tiles[n].getHeight();
38 }

```

Zeile 23 – 26: `setSourceImage(..)` füllt mit Hilfe unserer `SpriteLib` das `BufferedImage`-Array mit Werten. Diese Methode muß nach dem ersten `getInstance()` aufgerufen werden, damit die Klasse auch wirklich Bilddaten enthält.

Zeile 28 – 30: Methode zum Abruf eines einzelnen Images aus dem Array.

Zeile 32 – 38: Methoden zur Abfrage von Länge und Breite eines Einzelbildes

Das eigentliche Tile

Nun, da wir die Bilddaten zentral zur Verfügung haben, wollen wir uns die Klasse Tile ansehen. Diese soll die Position des Einzelbildes speichern und zum Zeichnen des Bildes aufgerufen werden.

Die grundsätzlichen Informationen sind somit: Position, Länge, Breite (und die Referenz auf das Einzelbild). Somit schreitet diese Klasse eigentlich danach, von Rectangle zu erben (wie so viele in diesem Tutorial ☺), so dass wir hier durch die Verwendung von vorhanden Klassen wieder einmal Arbeit sparen.

```
5 public class Tile extends Rectangle{
6
7     private static final long serialVersionUID = 1L;
8     int image_num;
9     ImageControl control;
10
11 public Tile(int x, int y, int width, int height, int num){
12     super(x,y,width,height);
13     image_num = num;
14     control = ImageControl.getInstance();
15 }
16
17 public int getImageNumber(){
18     return image_num;
19 }
20
21 public void drawTile(Graphics g){
22     g.drawImage(control.getImageAt(image_num), x, y, null);
23 }
24
25
26 }
```

Zeile 5: Die Klasse Tile erbt, wie schon erwähnt von Rectangle

Zeile 8: Integer-Variable zum Speichern der Bildinformation. Das eigentlich Bild ist in der Klasse ImageControl gespeichert

Zeile 9: Referenz auf die ImageControl-Klasse

Zeile 11: Der Konstruktor der Klasse, er bekommt alle Information für ein Rechteck und die Nummer des Einzelbildes übergeben.

Zeile 12: super-Aufruf. Damit wird die Vaterklasse Rectangle mit Werten versorgt

Zeile 13: Übergabe der Bildreferenz

Zeile 14: Abholen der Referenz auf die ImageControl-Klasse

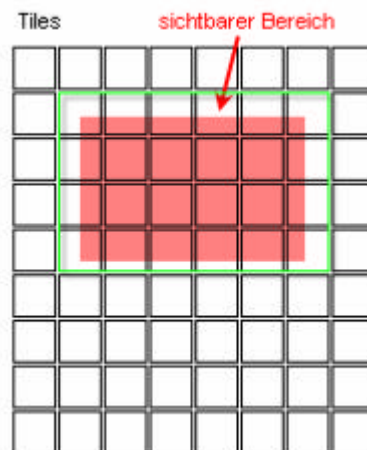
Zeile 17 – 19: Methode um die Bildreferenz abzufragen

Zeile 21 – 23: Methode zum Zeichnen des Tiles. Analog z. B. zur Klasse Sprite wird hier ein Graphics-Objekt übergeben, in welches das Einzelbild aus ImageControl hineingezeichnet wird.

Zusammenführung der einzelnen Komponenten zur Karte

Kommen wir jetzt zur wichtigsten Klasse: MapDisplay. Die im Folgenden vorgestellte Klasse soll unsere Tiles verwalten und mit Daten füllen und zusätzlich auch sämtliche Logik zum Bewegen und Zeichnen der Karte zur Verfügung stellen.

Zur Funktionsweise: Von der Masse der existierenden Tiles, in unserem Falle, einer Rennstrecke, muß jeweils nur ein kleiner Teil angezeigt werden. Für die Anzeige sind jeweils nur die Tiles interessant, die sich ganz oder teilweise mit dem sichtbaren Bereich überschneiden.



Die oben angezeigte Grafik stellt dies dar: Die schwarzen Rechtecke stellen die Summe aller Tiles für die Karte dar, sozusagen unseren Vector. Der rote Kasten repräsentiert den sichtbaren Bereich, d. h. den Bildschirm bzw. das Hauptfenster des Spiels. Der grüne Kasten enthält dann die Tiles, die notwendig sind, um den aktuell benötigten Teil der Karte anzuzeigen. Diese Schnittmenge an Tiles soll unsere Klasse zur Anzeige bereitstellen. Außerdem muß es möglich sein, den sichtbaren Bereich entsprechend der Bewegungen des Spielers zu verändern.

Beginnen wir mit dem Erstellen der neuen Klasse: MapDisplay.

```
3 import game.GamePanel;
4 import java.awt.*;
5 import java.awt.geom.Rectangle2D;
6 import java.util.*;
7 import main.ScrollGame;
8
9 public class MapDisplay extends Rectangle{
10
11     private static final long serialVersionUID = 1L;
12
13     ImageControl control;
14     Vector<Tile> tiles;
15     ScrollGame parent;
16     Rectangle2D display;
17
18     public MapDisplay(String level, String picpath, int col, int row, GamePanel p){
19         tiles = new Vector<Tile>();
20         parent = (ScrollGame) p;
21
22         loadLevelData(level);
23
24         control = ImageControl.getInstance();
25         control.setSourceImage(picpath, col, row);
26         display = new Rectangle2D.Double(0,0,parent.getWidth(),parent.getHeight());
27     }
28
29     private void loadLevelData(String level){
30
31     }
32
33
34
35 }
36
```

Zeile 9: Die Klasse erbt von Rectangle – mal wieder 😊. Damit sind gleich Variablen vorhanden, um Breite und Höhe der Karte zu speichern und Abfragen zu können.

Zeile 13: Referenz auf ImageControl

Zeile 14: Ein Vector zum Speichern der einzelnen Tiles

Zeile 15: Referenz auf unser GamePanel bzw. auf die Klasse, die von GamePanel erbt

Zeile 16: Ein Rectangle2D-Objekt mit Namen display.

Zeile 18: Konstruktor der Klasse. Er bekommt folgende Informationen übergeben:

- die Pfadangabe für die Leveldaten
- die Pfadangabe für die Bilddaten (das Tielset)
- die Anzahl der Bilder pro „Zeile“ im Tilesset
- die Anzahl der „Zeilen“ im Tilesset

- die Referenz auf das GamePanel in unserem Fall wird das die abgeleitete Klasse ScrollGame sein

Zeile 19: Instanziieren des Vectors, der die Tiles speichert

Zeile 20: Übergabe der Referenz auf die Hauptklasse

Zeile 22: Methodenaufruf zum Laden der Leveldaten. Die Methode selbst ist in Zeile 29 – 32 definiert, aber im Moment noch leer.

Zeile 24 – 25. Erzeugen/Abrufen der Instanz von ImageControl und Übergabe der Bilddaten.

Zeile 26: Hier wird das Rectangle2D-Objekt display erzeugt. Es erhält Breite und Höhe unseres GamePanels. Es repräsentiert, wie anfangs beschrieben den sichtbaren Bereich – in diesem Fall unser Fenster.

Als nächstes wollen wir das Einlesen der Level-Daten programmieren:

```
32 private void loadLevelData(String level) {  
33  
34     try {  
35  
36         InputStreamReader isr = new InputStreamReader(getClass().getClassLoader()  
37             .getResourceAsStream(level));  
38         BufferedReader bufread = new BufferedReader(isr);  
39  
40         String line = null;  
41  
42         do {  
43  
44             line = bufread.readLine();  
45  
46             if (line == null) {  
47                 continue;  
48             }  
49  
50             String[] split = line.split("/");  
51             int posx = Integer.parseInt(split[0]);  
52             int posy = Integer.parseInt(split[1]);  
53             int width = Integer.parseInt(split[2]);  
54             int height = Integer.parseInt(split[3]);  
55             int num = Integer.parseInt(split[4]);  
56  
57             if ((posx + width) > this.width) {  
58                 this.width = posx + width;  
59             }  
60  
61             if ((posy + height) > this.height) {  
62                 this.height = posy + height;  
63             }  
64  
65             Tile t = new Tile(posx, posy, width, height, num);  
66             tiles.add(t);  
67  
68         } while (line != null);  
69  
70         bufread.close();  
71         isr.close();  
72  
73     } catch (IOException e) {  
74         e.printStackTrace();  
75     }  
76  
77 }
```

Zeile 32: Die Methode bekommt die Pfadangabe als String übergeben. Die Pfadangabe erfolgt analog, wie wir es vom Laden der Bilddaten bereits kennen („level/level.txt“).

Zeile 36 – 38: Hier basteln wir uns einen BufferedReader zum zeilenweisen Einlesen unserer Tile-Daten

Zeile 40: String zum Einlesen der Einzelzeilen.

Zeile 42 – 68: doWhile-Schleife zum Einlesen der Level-Daten

Zeile 44 – 48: Einlesen einer einzelnen Zeile. Weitere Verarbeitung nur, wenn diese nicht null ist.

Zeile 50 – 55: Das String-Objekt (enthält z. B. 100/350/50/50/0) wird an den Schrägstrichen aufgespalten und in Integer-Werte geparkt.

Zeile 57 – 63: Wenn ein Tile die absolute Breite oder Höhe der Karte erhöht, wird die entsprechende Dimension neu berechnet.

Zeile 65 – 66: Das Tile-Objekt wird erzeugt und im Vector gespeichert.

Zeile 70 – 71: Alle Reader wieder schließen

Zwischentest

Nachdem wir jetzt schon einiges an Code neu geschrieben haben, sollten wir einen kleinen Test machen, ob bisher alles soweit funktioniert. Dazu bauen wir MapDisplay schon mal in unser Programm ein und fügen temporär 2 Zeilen Code in unsere Klassen ein:

Zunächst der Code, den wir beibehalten wollen:

```
15 public class ScrollGame extends JPanel {
16
17     private static final long serialVersionUID = 1L;
18
19     SpriteLib lib;
20     Car car;
21     MapDisplay map;
22
23     public static void main(String[] args) {
24         new ScrollGame(800,600);
25     }
26
27     public ScrollGame(int w, int h) {
28         super(w, h);
29     }
30
31
32     @Override
33     protected void doInitializations() {
34         super.doInitializations();
35
36         lib = SpriteLib.getInstance();
37         car = new Car(lib.getSprite("pics/car.gif", 4, 1),400,300,200,this);
38
39         map = new MapDisplay("level/level.txt","pics/tiles.gif",3,4,this);
40     }
```

In Zeile 21 fügen wir MapDisplay als Klassenvariable ein und erzeugen in unserer Methode doInitializations() das Objekt indem wir über den Konstruktor die Datendatei und die Imagedatei zuordnen.

Temporär zum Testen des bisher erzeugten Codes fügen wir noch 2 Zeilen Code ein, die wir anschließend wieder löschen:

In der Klasse MapDisplay, in der Methode loadLevelData() nach dem Erzeugen des Tiles:

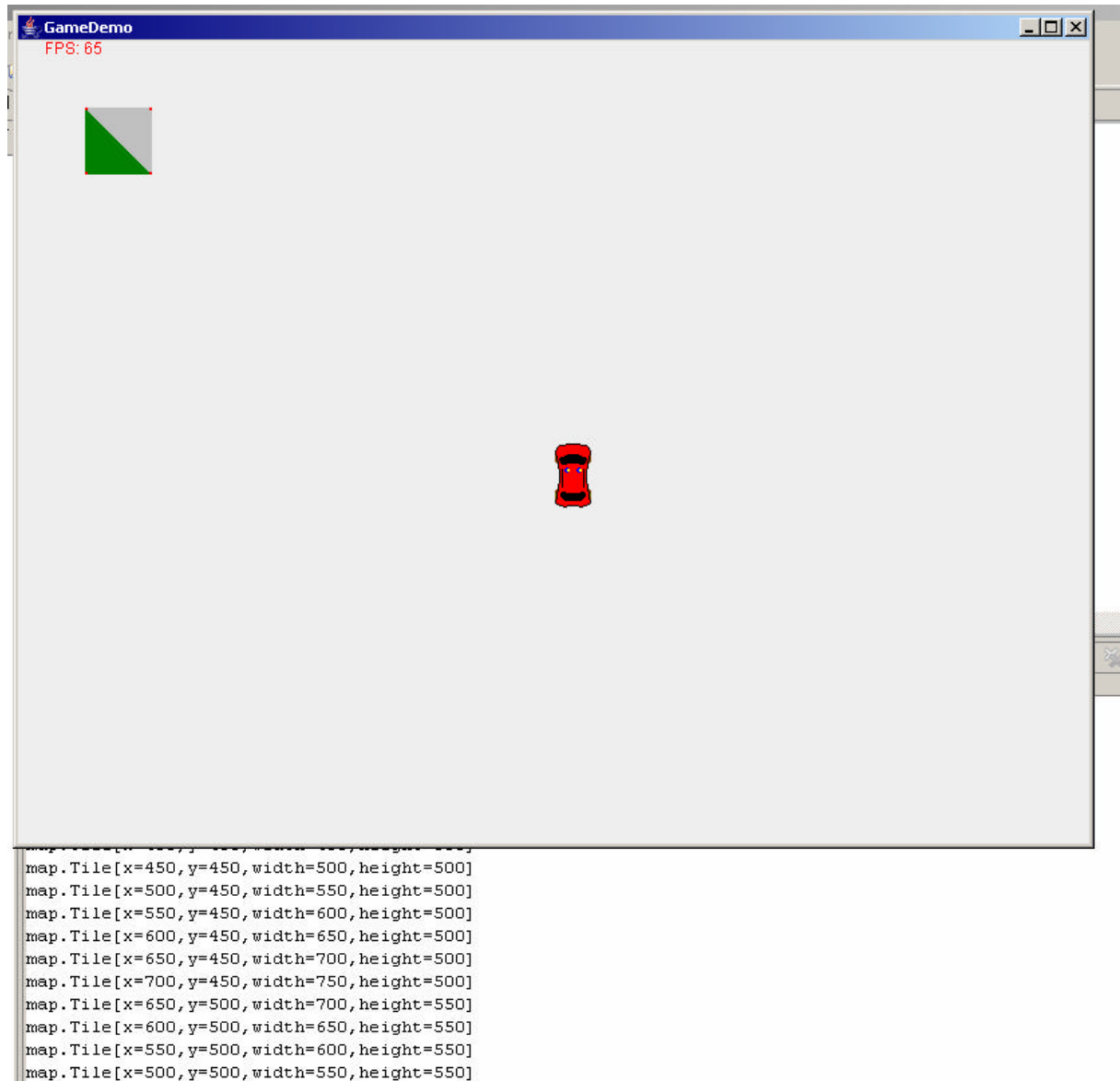
```
64         Tile t = new Tile(posx, posy, width, height, num);
65         tiles.add(t);
66         System.out.println(t);
67     }
```

In der Hauptklasse ScrollGame in der Methode paintAll():

```
63 @Override
64 public void paintAll(Graphics g) {
65     car.drawObjects(g);
66     g.drawImage(ImageControl.getInstance().getImageAt(7), 50, 50, this);
67 }
```

So sieht's aus:

Nach dem Starten des Programms sieht das Ganze schon einigermaßen zufriedenstellend aus:



Wie man sieht, werden die Grafikdaten offensichtlich sauber getrennt und auch die Tiles, die wir in die Konsole schreiben lassen, sehen soweit gut aus.

Jetzt die beiden temporären Zeile wieder löschen und weiter geht's.

Das Zeichnen der ganzen Karte

Jetzt soll aber endlich mal die ganze Karte gezeichnet werden! Dazu fügen wir in die Klasse MapDisplay eine neue Methode ein:

```
78 public void drawVisibleMap(Graphics g){
79
80     for(Tile t:tiles){
81         if(t.intersects(display)){
82             double dx = t.x - display.getX();
83             double dy = t.y - display.getY();
84             g.drawImage(Image) (control.getImageAt(t.getImageNumber()), (int) dx, (int) dy, null);
85         }
86     }
87 }
88 }
```

Die oben dargestellte Methode rufen wir aus der paintAll-Methode des GameLoop auf und übergeben das GraphicsObject.

Zeile 80 – 81: Es wird der komplette Bestand an Tiles geprüft. Überschneidet sich ein Tile mit dem aktuell sichtbaren Bereich, soll es gezeichnet werden. Da sowohl der sichtbare Bereich (display), als auch die Klasse Tile vom Typ Rectangle sind bzw. erben, ist diese Prüfung ganz leicht mit intersects durchführbar.

Zeile 82 – 83: Da Tiles u. U. so gezeichnet werden müssen, dass sie nicht vollständig sichtbar sind, werden x- und y-Position in Abhängigkeit vom sichtbaren Fenster (display) errechnet.

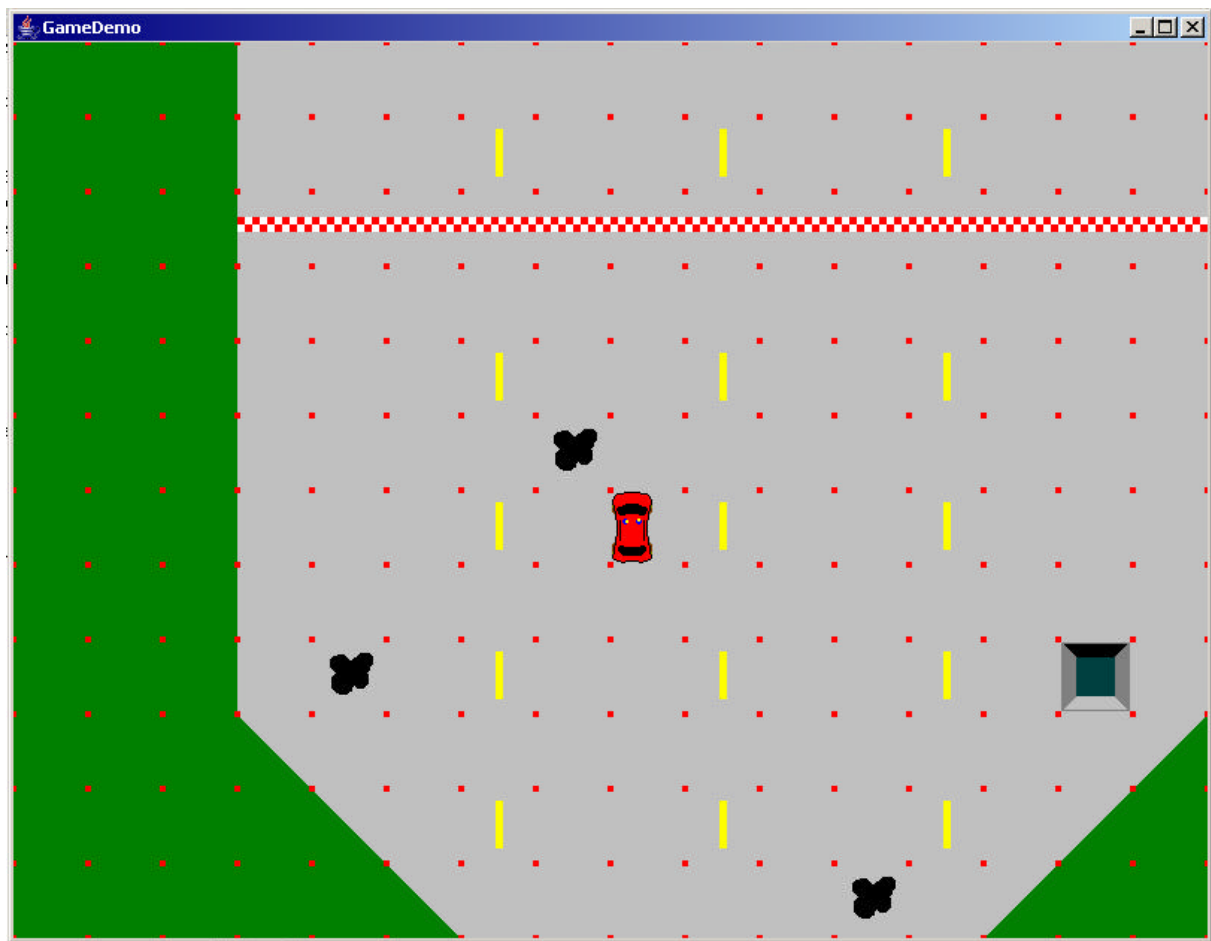
Zeile 84: ..und schließlich gezeichnet, indem die Bilddaten von ImageControl abgeholt werden.

Anschließen muß noch ein Zeile in der Klasse ScrollGame eingefügt werden:

```
62 @Override
63 public void paintAll(Graphics g) {
64     map.drawVisibleMap(g);
65     car.drawObjects(g);
66 }
```

Wichtig ist, dass die Karte vor dem Auto gezeichnet wird, damit dieses auch auf der Karte fährt!

Und so sieht's aus:



Bewegung

Bis jetzt sieht es ja schon ganz gut aus. Nun wollen wir auf der Karte auch fahren. Dazu möchten wir in unserem Fall auch am unteren Ende der Karte starten.

Dazu erweitern wir die Klasse MapDisplay um folgende Methoden:

```
90 public void setVisibleRectangle(Rectangle2D rec) {  
91     display = rec;  
92 }
```

Mit dieser Methode wollen wir das sichtbare Rechteck verschieben. Sie soll aber grundsätzlich nur zur initialen Verlagerung dienen. Zu einer flüssigen Bewegung kommen wir gleich.

In unserer Hauptklasse ScrollGame setzen wir nun noch den sichtbaren Bereich dorthin wo er gehört:

```
31 @Override  
32 protected void doInitializations() {  
33     super.doInitializations();  
34  
35     lib = SpriteLib.getInstance();  
36     car = new Car(lib.getSprite("pics/car.gif", 4, 1), 400, 300, 200, this);  
37  
38     map = new MapDisplay("level/level.txt", "pics/tiles.gif", 3, 4, this);  
39     map.setVisibleRectangle(new Rectangle2D.Double(50, map.getHeight() - getHeight(),  
40         getWidth(), getHeight()));  
41 }
```

Jetzt wollen wir die Karte bewegen. Als ersten definieren wir 2 Klassenvariablen für die horizontale und vertikale Bewegung. Um dies entsprechend genau berechnen zu können, definieren wir diese als double.

```
12 public class MapDisplay extends Rectangle {  
13  
14     private static final long serialVersionUID = 1L;  
15  
16     ImageControl control;  
17     Vector<Tile> tiles;  
18     ScrollGame parent;  
19     Rectangle2D display;  
20  
21     double dx = 0;  
22     double dy = 0;  
23  
24 public MapDisplay(String level, String picpath, int col, int  
25     tiles = new Vector<Tile>();
```

Als nächste klauen wir die Getter- und Setter-Methoden aus der Klasse Sprite mit copy & paste ☺:

```
94- public void setVisibleRectangle(Rectangle2D rec){
95     display = rec;
96 }
97
98- public void setVerticalSpeed(double d) {
99     dy = d;
100 }
101
102- public void setHorizontalSpeed(double d) {
103     dx = d;
104 }
105
106- public double getVerticalSpeed(){
107     return dy;
108 }
109
110- public double getHorizontalSpeed(){
111     return dx;
112 }
```

Und auch für die Berechnung der Bewegung bedienen wir uns bei der Klasse Sprite. Allerdings sind hier ein paar kleine Änderungen notwendig:

```

110 public double getHorizontalSpeed() {
111     return dx;
112 }
113
114 public void moveVisibleRectangle(long delta) {
115
116     double x = display.getX();
117     double y = display.getY();
118
119     if(dx!=0) {
120         x = x + (dx*delta/1e9);
121     }
122
123     if(dy!=0) {
124         y = y + (dy*delta/1e9);
125     }
126
127     if((x+display.getWidth())>width) {
128         x = width - display.getWidth();
129     }
130
131     if(x<0) {
132         x = 0;
133     }
134
135     if((y+display.getHeight())>height) {
136         y = height - display.getHeight();
137     }
138
139     if(y<0) {
140         y = 0;
141     }
142
143     display.setRect(x, y, display.getWidth(), display.getHeight());
144
145 }

```

Zeile 116 – 117: x- und y-Position des sichtbaren Bereichs packen wir der kürzeren Schreibweise wegen in 2 eigene Variablen.

Zeile 119 – 121: Soll eine horizontale Bewegung stattfinden, berechnen wir diese in Abhängigkeit der Zeit, die für den letzten Schleifendurchlauf benötigt wurde.

Zeile 123 – 125: Das Gleiche für vertikale Bewegungen

Zeile 127 – 141: Mit diesen Bedingungen verhindern wir, dass die Karte über ihren Rand hinaus angezeigt wird.

Zeile 143: Hier setzen wir die neue Position des sichtbaren Bereichs.

Jetzt noch ein paar Anpassung in unserer Hauptklasse und wir können losfahren. Zunächst wollen wir noch nicht steuernd eingreifen und setzen die Geschwindigkeit in unserer Klasse ScrollGame in doInitializations().

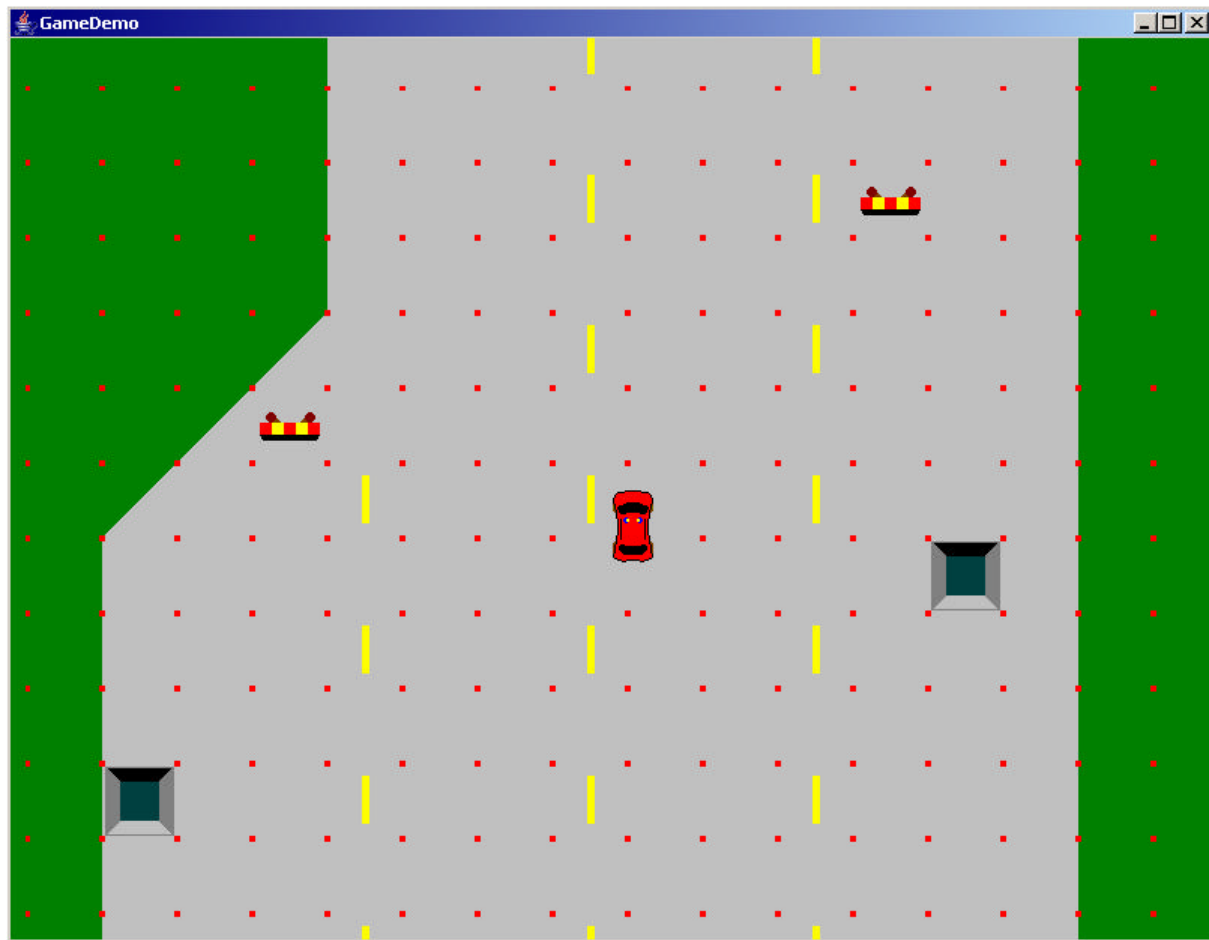
```
31 @Override
32 protected void doInitializations() {
33     super.doInitializations();
34
35     lib = SpriteLib.getInstance();
36     car = new Car(lib.getSprite("pics/car.gif", 4, 1), 400, 300, 200, this);
37
38     map = new MapDisplay("level/level.txt", "pics/tiles.gif", 3, 4, this);
39     map.setVisibleRectangle(new Rectangle2D.Double(50, map.getHeight() - getHeight(),
40         getWidth(), getHeight()));
41     map.setVerticalSpeed(-150);
42     map.setHorizontalSpeed(10);
43 }
```

Außerdem müssen wir in der move-Methode des GameLoop noch die Bewegung berechnen lassen:

```
58 @Override
59 protected void moveObjects() {
60     map.moveVisibleRectangle(delta);
61
62 }
```

Und schon fährt die Karre☺:

So sieht's aus:



Hier natürlich etwas statisch...

Insgesamt ist das Ganze aber noch etwas ruckelig. Darum werden wir uns aber gleich noch kümmern.

Steuerung

Zunächst mal wollen wir unser Auto selbst steuern können, dazu entfernen wir folgende Zeilen Code:

```
35 @Override
36 protected void doInitializations() {
37     super.doInitializations();
38
39     lib = SpriteLib.getInstance();
40     car = new Car(lib.getSprite("pics/car.gif", 4, 1), 400, 300, 200, this);
41
42     map = new MapDisplay("level/level.txt", "pics/tiles.gif", 3, 4, this);
43     map.setVisibleRectangle(new Rectangle2D.Double(50, map.getHeight() - getHeight(),
44         getWidth(), getHeight()));
45     map.setVerticalSpeed(-150);
46     map.setHorizontalSpeed(10);
47 }
```

Außerdem definieren wir uns zwei neue Instanzvariablen in der wir die Geschwindigkeit festlegen. Zunächst der Einfachheit halber als festen Wert:

```
14 public class ScrollGame extends JPanel {
15
16     private static final long serialVersionUID = 1L;
17
18     SpriteLib lib;
19     Car car;
20     MapDisplay map;
21
22     int vspeed = 100;
23     int hspeed = 50;
```

Danach klauen wir aus unserem ersten Teil Code aus den KeyListener-Methoden:

```
105 public void keyPressed(KeyEvent e) {
106
107     if (e.getKeyCode() == KeyEvent.VK_UP) {
108         up = true;
109     }
110
111     if (e.getKeyCode() == KeyEvent.VK_DOWN) {
112         down = true;
113     }
114
115     if (e.getKeyCode() == KeyEvent.VK_LEFT) {
116         left = true;
117     }
118
119     if (e.getKeyCode() == KeyEvent.VK_RIGHT) {
120         right = true;
121     }
122
123 }
```

```

125 public void keyReleased(KeyEvent e) {
126
127     if (e.getKeyCode() == KeyEvent.VK_UP) {
128         up = false;
129     }
130
131     if (e.getKeyCode() == KeyEvent.VK_DOWN) {
132         down = false;
133     }
134
135     if (e.getKeyCode() == KeyEvent.VK_LEFT) {
136         left = false;
137     }
138
139     if (e.getKeyCode() == KeyEvent.VK_RIGHT) {
140         right = false;
141     }
142
143
144     if (e.getKeyCode() == KeyEvent.VK_ENTER) {
145         // ...

```

Zur Erinnerung: Die notwendigen boolean-Werte haben wir bereits in unserem GamePanel definiert!

Anschließend passen wir noch die Methode checkKeys() an:

```
49 @Override
50 protected void checkKeys() {
51
52     if(left){
53         map.setHorizontalSpeed(-hspeed);
54     }
55
56     if(right){
57         map.setHorizontalSpeed(hspeed);
58     }
59
60     if(!left && !right){
61         map.setHorizontalSpeed(0);
62     }
63
64     if(up){
65         map.setVerticalSpeed(-vspeed);
66     }
67
68     if(down){
69         map.setVerticalSpeed(0);
70     }
71
72 }
```

Und schon ist das Ganze steuerbar, auch wenn es sich hier um eine sehr primitive Steuerung handelt. Aber wer das Tutorial bis hierher gelesen hat, für den sollte es kein Problem darstellen, z. B. eine Beschleunigung einzubauen 😊

Anzeigeoptimierung

Momentan ist die Animation noch etwas ruckelig. Das wollen wir jetzt möglichst minimieren.

Zuerst einmal fügen wir eine kleine Änderung in unserem GamePanel ein:

```
61 public void run() {
62
63     while(game_running) {
64
65         computeDelta();
66
67         if(isStarted()){
68             checkKeys();
69             doLogic();
70             moveObjects();
71         }
72
73         repaint();
74
75         try {
76             Thread.sleep(5);
77         } catch (InterruptedException e) {}
78
79     }
80
81 }
```

Diese Verringerung der Zeitangabe in Thread.sleep() sollte schon eine spürbare Verbesserung bringen.

Jetzt wollen wir auf aktives Rendern und VolatileImages umstellen.

Für das aktive Rendern müssen wir auf AWT-Komponenten zurückgreifen, da nicht alle Swing-Komponenten die benötigten Eigenschaften besitzen.

Die AWT-Entsprechung zu unserem JPanel ist ein Canvas, daher ändern wir unsere Vaterklasse entsprechend ab:

```
16 public abstract class GamePanel extends Canvas implements Runnable, Ke
17
18     private static final long serialVersionUID = 1L;
19     protected boolean game_running = true;
20     protected boolean started = false;
```

Da Canvas keine eingebaute Doppelbufferung besitzt, müssen wir das selbst übernehmen. Die Klasse, die uns hierfür Mechanismen zur Verfügung stellt heißt `BufferStrategy`.

```
29  protected boolean down = false;
30  protected boolean left = false;
31  protected boolean right = false;
32
33  BufferStrategy strategy;
34
35  public GamePanel(int w, int h) {
36      this.setPreferredSize(new Dimension(w,h));
37      JFrame frame = new JFrame("GameDemo");
38      ...
39  }
```

Anschließend haben wir im Konstruktor einige Änderungen durchzuführen:

```
33  public GamePanel(int w, int h) {
34
35      this.setPreferredSize(new Dimension(w,h));
36      Frame frame = new JFrame("GameDemo");
37      frame.setLocation(100,100);
38      frame.addKeyListener(this);
39      frame.add(this);
40      frame.pack();
41      frame.setIgnoreRepaint(true);
42      frame.setVisible(true);
43
44      createBufferStrategy(2);
45      strategy = getBufferStrategy();
46      doInitializations();
47  }
48
```

Zeile 36: Da das Mischen von AWT- und Swing-Komponenten nach Möglichkeit unterlassen werden sollte, verwenden wir statt eines `JFrame` jetzt die AWT-Entsprechung `Frame`.

Zeile 41: Da wir das Zeichnen jetzt selber steuern, weisen wir den `Frame` an, `repaint()`-Aufrufe zu ignorieren.

Zeile 44: Hier erzeugen wir über die Methode `createBufferStrategy(int num)` die Möglichkeit einer Multi-Bufferung für die Canvas-Komponente, bzw. in unserem Falle ein Doppelbufferung.

Zeile 45: Hier weisen wir unserer Instanzvariablen die Referenz auf die eben erzeugte `BufferStrategy` zu.

Danach müssen wir unsere Spielschleife modifizieren. Den repaint-Aufruf können wir jetzt auskommentieren oder löschen. Außerdem fügen wir einen neuen Methodenaufruf doPainting() ein:

```
75 public void run() {
76
77     while(game_running){
78
79         computeDelta();
80
81         if(isStarted()){
82             checkKeys();
83             doLogic();
84             moveObjects();
85         }
86
87         doPainting();
88         repaint();
89
90         try {
91             Thread.sleep(5);
92         } catch (InterruptedException e) {}
93
94     }
95
96 }
```

Die Methode doPainting() fügen wir gleich ein. Vorher wollen wir noch das Zeichnen auf VolatileImages umstellen. VolatileImages sind Hardware-beschleunigt, so dass man sich hier einen Performance-Schub versprechen kann.

Wir wollen im Folgenden ein VolatileImage für eine selbsterstellte Doppelpufferung verwenden.

Dazu sind zunächst einige Vorarbeiten zu leisten und einige neue Instanzvariablen hinzu zu fügen:

```
23 protected boolean up    = false;
24 protected boolean down  = false;
25 protected boolean left  = false;
26 protected boolean right = false;
27
28 VolatileImage backbuffer;
29 GraphicsEnvironment ge;
30 GraphicsConfiguration gc;
31 BufferStrategy strategy;
32
33 public GamePanel(int w, int h){
34     ge = GraphicsEnvironment.getLocalGraphicsEnvironment();
35     gc = ge.getDefaultScreenDevice().getDefaultConfiguration();
36
37     this.setPreferredSize(new Dimension(w,h));
38
39     Frame frame = new JFrame("GameDemo");
```


Zeile 28: Hier definieren wir unser `VolatileImage` namens `backbuffer`

Zeile 29 – 30: Hier definieren wir uns Variablen in denen wir Informationen über die Hardware-Umgebung speichern (vgl. API)

Zeile 34 – 35: Im Konstruktor unseres Spiels ermitteln wir die notwendigen Informationen über die „grafischen Gegebenheiten“

Das `VolatileImage` erzeugen wir in unserer Methode `doInitializations()`:

```
52 protected void doInitializations(){
53
54     createBackbuffer();
55
56     if (!once){
57         once = true;
58         Thread t = new Thread(this);
59         t.start();
60     }
61 }
62
63 private void createBackbuffer(){
64     if (backbuffer != null) {
65         backbuffer.flush();
66         backbuffer = null;
67     }
68     //GraphicsConfiguration für VolatileImage
69     ge = GraphicsEnvironment.getLocalGraphicsEnvironment();
70     gc = ge.getDefaultScreenDevice().getDefaultConfiguration();
71     backbuffer = gc.createCompatibleVolatileImage(getWidth(), getHeight());
72
73 }
```

Zeile 54: Wir rufen eine neue Methode auf: `createBackbuffer()`

Zeile 63 – 71: In dieser Methode wird das `VolatileImage` nötigenfalls initialisiert und anschließend über die entsprechende Methode der Klasse `GraphicsConfiguration` neu erzeugt.

Jetzt können wir die neue Methode doPainting erzeugen:

```
98 //doPainting wird periodisch aus dem GameLoop aufgerufen
99 private void doPainting(){
100
101     checkBackbuffer(); //Prüf-Methode für VolatileImage
102
103     Graphics g = backbuffer.getGraphics(); //GraphicsObject vom VolatileImage holen
104     render(g); //alle Zeichenoperationen: Map, Player, etc.
105     g.dispose(); //Graphics-Objekt verwerfen
106
107     Graphics g2 = strategy.getDrawGraphics(); //Zeichenobjekt der BufferStrategy holen
108     g2.drawImage(backbuffer,0,0,this); //VolatileImage in den Buffer zeichnen
109     g2.dispose(); //GraphicsObject verwerfen
110
111     strategy.show(); //Bufferanzeigen.
112 }
113
114 private void checkBackbuffer(){
115     if(backbuffer==null){
116         createBackbuffer();
117     }
118     if(backbuffer.validate(gc)==VolatileImage.IMAGE_INCOMPATIBLE){
119         createBackbuffer();
120     }
121 }
```

Zeile 101: Als erstes rufen wir die Methode checkBackbuffer() auf, die wir gleich selbst erstellen (vgl. Zeile 114 – 121). Lt. API kann es vorkommen, dass die Inhalte eines VolatileImage verloren gehen. In der Methode checkBackbuffer wird geprüft, ob dies der Fall ist. Wenn ja, rufen wir createBackbuffer() auf, um ein neues VolatileImage zu erzeugen (ausführliche Informationen dazu stehen in der API). Ich habe während der Entwicklung dieser Methode versucht, den Inhalt eines VolatileImages zu „verlieren“, damit diese Methode einmal aufgerufen wird, es ist mir allerdings nicht gelungen.

Zeile 103: Hier wollen wir uns das Graphics-Objekt des VolatileImages. Hier im Rahmen des aktiven Renderns ist es tatsächlich einmal erlaubt, getGraphics() zu verwenden. ☺

Zeile 103: Jetzt rufen wir eine neue Methode auf: render(g). Diese Methode ist in unserem GamePanel abstract und ersetzt den paintComponent(..)-Aufruf, den wir jetzt nicht mehr brauchen (und den es für einen AWT-Frame auch nicht gibt) und löschen werden. Dazu weiter unten mehr. Somit zeichnen wir unsere Objekte jetzt direkt in das VolatileImage.

Zeile 104 -1 05: Hier geben wir die Ressource auf das GraphicsObject wieder frei.

Zeile 107: Hier holen wir uns das GraphicsObject der BufferStrategy ab.

Zeile 108: Anschließend zeichnen wir das VolatileImage in den Buffer der BufferStrategy

Zeile 109: Das GraphicsObject wird jetzt nicht mehr benötigt

Zeile 111: Hier wechseln wir die Buffer der BufferStrategy und bringen den, in den wir eben gezeichnet haben, zur Anzeige.

Da wir wie oben schon gesagt, die `paintComponent(..)`-Methode nicht mehr verwenden können, löschen wir diese und ersetzen sie durch die Methode `render(Graphics g)`, die wir hier in der Klasse `GamePanel` abstract setzen.

```
133  
134 protected abstract void render(Graphics g);  
135
```

Damit müssen wir sie in der Klasse `ScrollGame` mit Leben füllen:

```
92 public void render(Graphics g) {  
93  
94     if(isStarted()){  
95         map.drawVisibleMap(g);  
96         car.drawObjects(g);  
97     }else{  
98         g.setColor(Color.red);  
99         g.drawString("Press Enter!", 50, 50);  
100     }  
101  
102     g.setColor(Color.red);  
103     g.drawString(Long.toString(fps), 20, 20);  
104  
105 }
```

Zeile 92 – 105: Die Methode `render(Graphics g)` ersetzt jetzt unsere überschriebenen `paintComponent(Graphics g)`-Methode. Der Inhalt bleibt prinzipiell gleich.

Weitere Verbesserungen:

Um die Performance unseres Programms nach Möglichkeit weiter zu verbessern wollen wir die Klasse SpriteLib noch etwas modifizieren:

```
13 public class SpriteLib {
14
15     private static SpriteLib single;
16     private static GraphicsEnvironment ge;
17     private static GraphicsConfiguration gc;
18     private static HashMap<URL, BufferedImage> sprites;
19
20     public static SpriteLib getInstance() {
21         if (single == null) {
22             single = new SpriteLib();
23         }
24         return single;
25     }
26
27     private SpriteLib() {
28         //GraphicsConfiguration für VolatileImage
29         ge = GraphicsEnvironment.getLocalGraphicsEnvironment();
30         gc = ge.getDefaultScreenDevice().getDefaultConfiguration();
31
32         sprites = new HashMap<URL, BufferedImage>();
33     }
34 }
```

Zeile 16 -17: Auch in der Klasse SpriteLib fügen wir analog zur Klasse GamePanel Instanzvariablen ein, die uns Informationen über GraphicsEnvironment und GraphicsConfiguration bereit stellen.

Zeile 29 – 30: Die Inhalte dieser Variablen ermitteln wir im Konstruktor.

Anschließend modifizieren wir alle Methoden, die unsere Sprites einlesen:

```
35 public BufferedImage getSprite(String path) {
36
37     BufferedImage pic = null;
38     URL location = getURLfromRessource(path);
39
40     pic = (BufferedImage) sprites.get(location);
41
42     if (pic != null) {
43         return pic;
44     }
45
46     try {
47         pic = ImageIO.read(location); // Über ImageIO die GIF lesen
48     } catch (IOException e1) {
49         System.out.println("Fehler beim Image laden: " + e1);
50         return null;
51     }
52
53     BufferedImage better = gc.createCompatibleImage(pic.getWidth(), pic.getHeight(), Transparency.BITMASK);
54     Graphics g = better.getGraphics();
55     g.drawImage(pic, 0, 0, null);
56
57     sprites.put(location, better);
58
59     return better;
60 }
61 }
```

Zeile 53: Über die Methode `createCompatibleImage(...)` erzeugen wir uns ein `BufferedImage`, das von Layout und ColorModel her optimal angezeigt werden kann (vgl. API).

Zeile 54: Von diesem Image holen wir uns das `GraphicsObject`

Zeile 55: Anschließend zeichnen wir das eben eingelesene `BufferedImage` (pic) in das „bessere“ `BufferedImage` (better).

Zeile 57: Jetzt speichern wir das „bessere“ `BufferedImage` in unserer `HashTable`

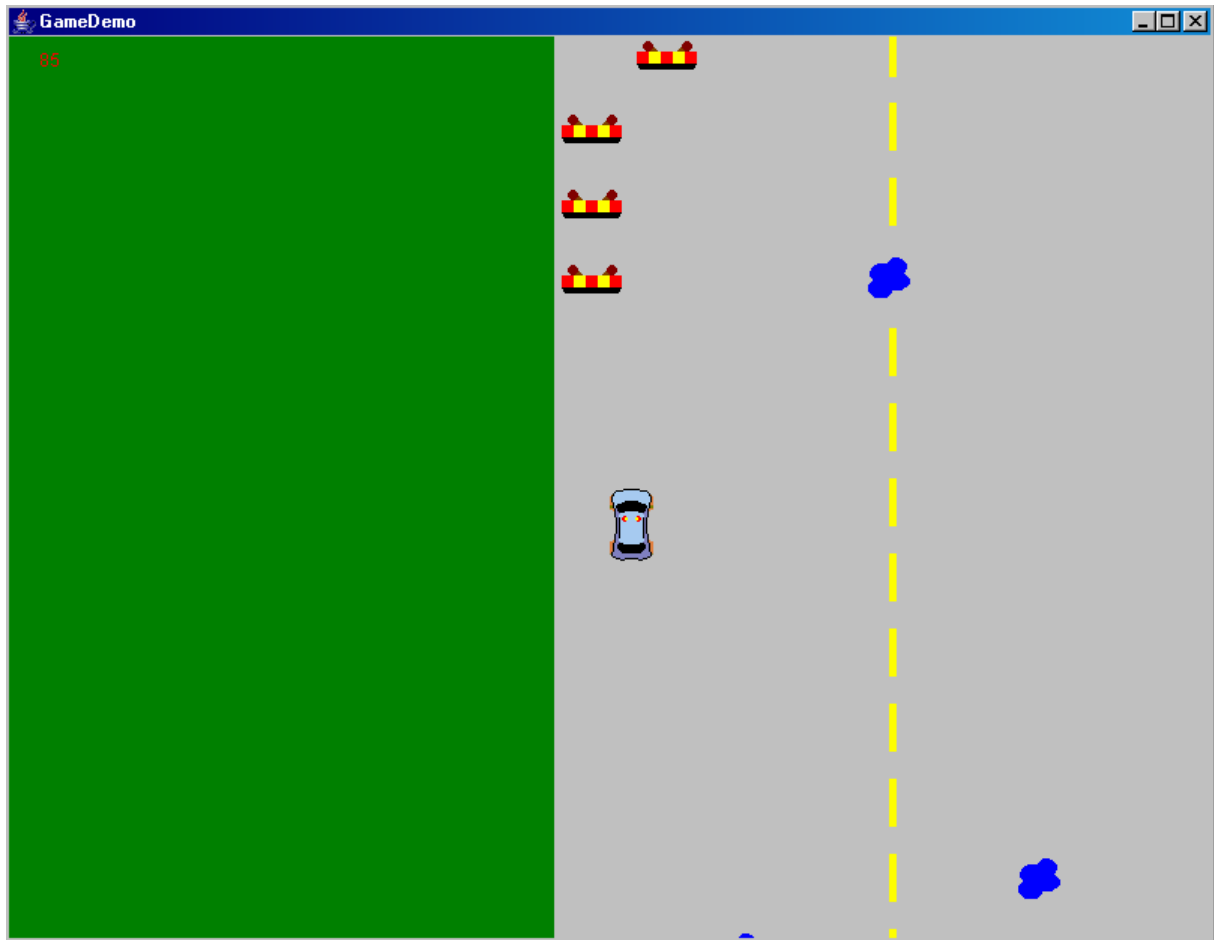
Zeile 59: Anschließend geben wir das `BufferedImage` zurück.

Hier das Ganze analog und unkommentiert für die Methode, die unsere Image-Arrays einliest:

```
62 public BufferedImage[] getSprite(String path, int column, int row) {
63
64     URL location = getURLfromResource(path);
65
66     BufferedImage source = null;
67
68     source = (BufferedImage) sprites.get(location);
69
70     if (source == null) {
71         try {
72             source = ImageIO.read(location); // Über ImageIO die GIF lesen
73         } catch (IOException e1) {
74             System.out.println(e1);
75             return null;
76         }
77
78         sprites.put(location, source);
79
80     }
81
82     BufferedImage better = gc.createCompatibleImage(source.getWidth(), source.getHeight(), Transparency.BITMASK);
83     Graphics g = better.getGraphics();
84     g.drawImage(source, 0, 0, null);
85
86     int width = source.getWidth() / column;
87     int height = source.getHeight() / row;
88
89     BufferedImage[] pics = new BufferedImage[column * row];
90     int count = 0;
91
92     for (int n = 0; n < row; n++) {
93         for (int i = 0; i < column; i++) {
94             pics[count] = source.getSubimage(i * width, n * height, width, height);
95             count++;
96         }
97     }
98
99     return pics;
100
101 }
```

Mit den jetzt gemachten Modifikationen hat unser Programm (zumindest auf meinen Rechnern) einen deutlichen Performance-Schub erhalten.

So sieht's aus:



Ein paar Änderungen sind hier noch unbemerkt eingeflossen:

1. Die roten Punkte der Tiles habe ich mittlerweile entfernt
2. Das Auto wurde neu lackiert 😊

Hier auf dem Rechner auf dem das Tutorial entwickelt wurde, haben die Modifikationen eine deutliche Leistungssteigerung ergeben. Links oben sieht man die FPS (frames per second) die mit 85 einen guten Wert haben. Hier war vor der Modifikation nur ein gutes Drittel erreicht worden!

Kollisionen

Nun wollen wir unser Auto mit dem Hintergrund interagieren lassen. Hierfür nehmen wir zunächst eine Veränderung an der Steuerung vor.

Änderung der Steuerung

Zunächst 2 neue Instanzvariablen in der Klasse MapDisplay:

```
3+ import game.GamePanel;
12
13 public class MapDisplay extends Rectangle{
14
15     private static final long serialVersionUID = 1L;
16
17     ImageControl control;
18     Vector<Tile> tiles;
19     ScrollGame parent;
20     Rectangle2D display;
21
22     double dx = 0;
23     double dy = 0;
24
25     int vspeed = 0;
26     final int VMAX = -250;
27
```

Zeile 23: hier speichern wir die aktuelle Geschwindigkeit

Zeile 24: hier speichern wir die max. Geschwindigkeit. Da sich die Karte sozusagen von unten nach oben bewegt, muss diese negativ sein.

Anschließend erzeugen wir noch 4 neue Methoden, die es uns ermöglichen, die Geschwindigkeit der Karte schrittweise zu verändern.

```

148 public void driveToLeft() {
149     setHorizontalSpeed(vspeed/2);
150 }
151
152 public void driveToRight() {
153     setHorizontalSpeed(-vspeed/2);
154 }
155
156 public void speedUp() {
157     vspeed-=5;
158     if(vspeed<VMAX) {
159         vspeed = VMAX;
160     }
161     setVerticalSpeed(vspeed);
162 }
163
164 public void speedDown() {
165     vspeed+=5;
166     if(vspeed>0) {
167         vspeed = 0;
168     }
169     setVerticalSpeed(vspeed);
170 }

```

Zeile 148 – 154: Mit den Methoden driveToLeft() und driveToRight() wird die horizontale Geschwindigkeit in Abhängigkeit von der Fahrzeuggeschwindigkeit gesetzt.

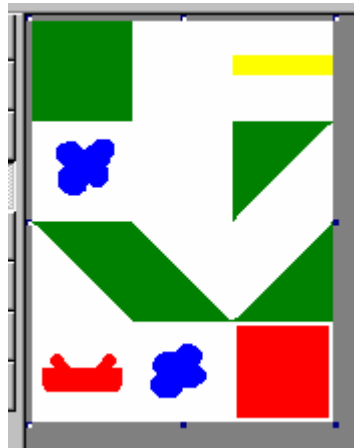
Zeile 156 – 169: Mit speedUp() und speedDown() realisieren wir eine schrittweise Beschleunigung bzw. Verzögerung bis zu einem Maximum bzw. bis 0.

Weiterhin passen wir die Methode `checkKeys()` an, so dass die in `MapDisplay` hinterlegte Logik jetzt verwendet werden kann:

```
58 @Override
59 protected void checkKeys() {
60
61     if(left){
62         map.driveToLeft();
63     }
64
65     if(right){
66         map.driveToRight();
67     }
68
69     if(!left && !right){
70         map.setHorizontalSpeed(0);
71     }
72
73     if(up){
74         map.speedUp();
75     }
76
77     if(down){
78         map.speedDown();
79     }
80
81 }
```

Die „Schatten-Karte“

Damit wir wissen, auf welchem Untergrund unser Auto fährt, basteln wir uns eine „Schatten-Karte“. Diese wird parallel zum Tileset geladen, aber niemals angezeigt. Außerdem enthält sie Farbinformationen, über die der Untergrund bzw. die Art des Untergrunds ermittelt werden kann. Dafür nehmen wir die Datei mit dem Tileset und malen dieses aus:



Für dieses Tutorial habe ich eine ganz einfache Farbcodierung gewählt:

Gelb: für die Realisierung einer Zeitmessung

Grün: Bereich außerhalb der Straße

Blau: Objekte, die unser Fahrzeug behindern

Rot: Objekte, die unser Fahrzeug zerstören

Ich habe hier „Standard-Farben“ verwendet, wie z. B. in MS Paint vorhanden, da diese mit den statischen Color-Objekten (z. B. Color.red) übereinstimmen.

Die Schatten-Karte speichern wir ab üblichen Ort. (vgl. weiter oben).

Da die Schatten-Karte mit dem Tileset übereinstimmt, bietet es sich an, diese Informationen zusammen mit dem Tileset zu verwalten, dafür bohren wir jetzt die Klasse ImageControl etwas auf:

```
8 public class ImageControl {
9
10     BufferedImage[] tiles;
11     BufferedImage[] shadow;
12     private static ImageControl instance;
13
14     public static ImageControl getInstance(){
15         if(instance==null){
16             instance = new ImageControl();
17         }
18
19         return instance;
20     }
21
22     private ImageControl(){
23         tiles = null;
24     }
25
26     public void setSourceImage(String path, int col, int row){
27         SpriteLib lib = SpriteLib.getInstance();
28         tiles = lib.getSprite(path, col, row);
29     }
30
31     public void setShadowImage(String path, int col, int row){
32         SpriteLib lib = SpriteLib.getInstance();
33         shadow = lib.getSprite(path, col, row);
34     }
35
36     public BufferedImage getImageAt(int num){
37         return tiles[num];
38     }
39
40     public BufferedImage getShadowImageAt(int num){
41         return shadow[num];
42     }
```

Zeile 11: ein neues Image-Array für die „Schatten-Tiles“

Zeile 31 – 34: im Prinzip eine Kopie der Methode setSourceImage(...) um das Array für die Schatten-Tiles zu füllen.

Zeile 40 – 42: Kopie der Methode getImageAt(int num), um die Bildinformation für ein Schatten-Tile abzurufen.

Danach erweitern wir den Konstruktor der Klasse MapDisplay, um die Daten hier gleich mit einzulesen, wenn wir die Karte initialisieren:

```
28 public MapDisplay(String level, String picpath, String shadowpath, int col, int row, GamePanel p){
29     tiles = new Vector<Tile>();
30     parent = (ScrollGame) p;
31
32     loadLevelData(level);
33
34     control = ImageControl.getInstance();
35     control.setSourceImage(picpath, col, row);
36     control.setShadowImage(shadowpath, col, row);
37
38     display = new Rectangle2D.Double(0,0,parent.getWidth(),parent.getHeight());
39 }
40
```

Zeile 28: Erweiterung des Konstruktors

Zeile 36: Methodenaufruf zum Laden der Schattenkarte

Damit müssen wir dann selbstverständlich auch die Klasse ScrollGame anpassen:

```
41 @Override
42 protected void doInitializations() {
43
44     lib = SpriteLib.getInstance();
45     car = new Car(lib.getSprite("pics/car.gif", 12, 1),400,300,200,this);
46
47     map = new MapDisplay("level/level.txt","pics/tiles.gif","pics/shadow.gif",3,4,this);
48
49     map.setVisibleRectangle(new Rectangle2D.Double(50,map.getHeight()-getHeight(),
50         getWidth(),getHeight()));
51
52     startscreen = lib.getSprite("pics/startscreen.gif");
53
54     checktime = false;
55     racetime = 0;
56     stop = 0;
57 }
```

Wenn wir jetzt das Programm starten, merken wir erst einmal keinen Unterschied. Jetzt gilt es zu überprüfen, auf welchem Tile unser Auto gerade fährt. Dazu müssen wir an geeigneter Stelle Code hinterlegen. Ich habe mich dafür entschieden, dies in der Klasse Car zu tun. Zwar bewegen wir im Grunde genommen nicht das Auto, sondern die Karte unter dem Auto und so könnte der folgende Code auch in MapDisplay hinterlegt werden. Da ich aber die Klasse MapDisplay möglichst variabel einsetzen können will, habe ich mich entschieden, dies etwas flexibler zu gestalten.

Daher habe ich eine Methode in der Klasse MapDisplay hinterlegt, die mir für einen bestimmten Punkt den Farbcode aus der ShadowMap zurück gibt:

```
104 public Color getColorForPoint(Point p){
105
106     for(Tile t:tiles){
107
108         double dx = t.x - display.getX();
109         double dy = t.y - display.getY();
110
111         Rectangle temp2 = new Rectangle((int)dx, (int)dy,
112             (int)t.getWidth(), (int)t.getHeight());
113
114         if(temp2.contains(p)){
115             int px = (int) (p.x - dx);
116             int py = (int) (p.y - dy);
117
118             Color c = new Color(ImageControl.getInstance().
119                 getShadowImageAt(t.getImageNumber()).getRGB(px, py));
120             return c;
121         }
122     }
123
124     return null;
125 }
126
127 }
```

Zeile 104: Die Methode soll ein Color-Objekt für den Punkt p zurück geben

Zeile 106: Dafür ist es notwendig über alle Tiles zu nudeln

Zeile 108 – 109: Unser Tile hat eigentlich eine feste Position und wird nie bewegt. Was wir tatsächlich bewegen ist der sichtbare Bereich. Daher muss hier sozusagen die (virtuelle) Position errechnet werden, wenn wir das Tile und nicht den sichtbaren Bereich bewegen würden.

Zeile 111: Für die eben errechnete „virtuelle“ Position erzeugen wir ein temporäres Rechteck, damit wir wieder einmal von den Vorteilen profitieren wollen, die man durch die Verwendung von Basisklassen erhält ☺

Zeile 114: Das geschieht hier: Durch die Methode contains(Point p) der Klasse Rectangle prüfen wir, ob der gesuchte Punkt im virtuellen Tile enthalten ist.

Ziele 115 – 116: Haben wir einen Treffer gelandet. Errechnen wir die Position unseres Punktes im Tile. Dadurch könnten wir bei komplexeren Karten unterschiedliche Farbinformationen in einem einzelnen Tile der Schattenkarte hinterlegen.

Zeile 118: Danach holen wir uns das dazugehörige BufferedImage und lesen dort die Farbinformation aus, um damit ein Color-Objekt zu erzeugen. Die Zeile 118 stellt durch verschachtelte Methodenaufrufe folgenden Code dar:

```
122         int num = t.getImageNumber();  
123         BufferedImage buf = ImageControl.getInstance().getShadowImageAt(num);  
124         int rgb = buf.getRGB(p.x, p.y);  
125         Color c = new Color(rgb);
```

Nun müssen wir diese Farbinformation nur noch abrufen – in der Klasse Car. Dazu brauchen wir in der Klasse Car Zugriff auf das MapDisplay-Objekt: Dazu fügen wir in der Klasse ScrollGame folgenden Code neu ein:

```
150     public MapDisplay getMap(){  
151         return map;  
152     }
```

Abfrage der Farbinformationen

Danach können wir die Klasse Car modifizieren:

```
9 public class Car extends Sprite {
10
11     ScrollGame parent;
12
13
14     public Car(BufferedImage[] i, double x, double y, long delay, GamePanel p) {
15         super(i, x, y, delay, p);
16         parent = (ScrollGame) p;
17         setLoop(0, 3);
18     }
19
20     public void doLogic(long delta) {
21         super.doLogic(delta);
22
23         Color col1 = parent.getMap().
24             getColorForPoint(new Point((int) (getX()), (int) getY()));
25
26         Color col2 = parent.getMap().
27             getColorForPoint(new Point((int) (getX()+getWidth()/2), (int) getY()));
28
29         Color col3 = parent.getMap().
30             getColorForPoint(new Point((int) (getX()+getWidth()), (int) getY()));
31
32         checkColor(col1);
33         checkColor(col2);
34         checkColor(col3);
35
36     }
37
38     private void checkColor(Color col) {
39
40         if(col.equals(Color.yellow)) {
41             System.out.println("gelb");
42         }
43
44         if(col.equals(Color.blue)) {
45             System.out.println("blau");
46         }
47
48         if(col.equals(Color.green)) {
49             System.out.println("grün");
50         }
51
52         if(col.equals(Color.red) && currentpic<4) {
53             System.out.println("aua");
54         }
55
56     }
```

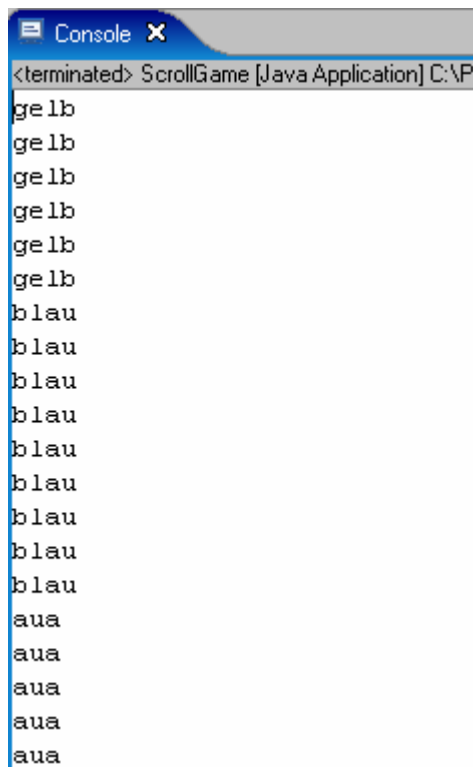
Zeile 23 – 30: Jetzt holen wir uns während des Spiels für einige ausgesuchte Punkte den Farbwert der Schatten-Karte ab. Ich habe diese hier als col1 – col3 benannt und mich entschieden, dass die Prüfung der folgenden Punkte ausreichend ist: col = links oben, col2 = die Mitte des oberen Randes, col 3 = rechts oben. Ich habe hierbei auch keine Rücksicht auf transparente Bereiche genommen, da dies in diesem Fall

vernachlässigbar ist. Bei Bedarf könnte man die Abfrage dieser Punkte dann entsprechend anpassen.

Zeile 32 – 34: Für jeden Farbwert wird die Methode `checkColor(..)` aufgerufen, die wir gleich betrachten.

Zeile 38 – 54: Hier prüfen wir das `Color`-Objekt gegen die am Anfang definierten Farben. Dadurch, dass ich Standardfarben verwendet habe, ist es mir möglich die statischen `Color`-Objekte der Klasse `Color` zu verwenden und etwas Code zu sparen.

Wenn wir das Programm jetzt starten, erhalten wir in der Konsole z. B. folgende Ausgabe:



```
<terminated> ScrollGame [Java Application] C:\P
gelb
gelb
gelb
gelb
gelb
gelb
gelb
blau
blau
blau
blau
blau
blau
blau
blau
blau
blau
blau
aue
aue
aue
aue
aue
```

Dabei ist zu beachten, dass jeder Farbwert während eines Schleifendurchlaufs mehrmals ermittelt werden kann. Dies müssen wir bei der weiteren Programmierung berücksichtigen.

Verarbeitung der Informationen

Jetzt sollen die Farbinformationen verarbeitet werden. Fangen wir mit der Farbe gelb an. Dieser Farbcode ist für Start und Ziel hinterlegt und soll das Ende des Spiels signalisieren, sowie zur Zeitmessung dienen.

Dazu definieren wir uns zunächst eine Instanz-Variable zur Zeitmessung:

```
11 public class Car extends Sprite {
12
13     ScrollGame parent;
14     long racetime;
15
16     public Car(BufferedImage[] i, double x, d
17         super(i, x, y, delay, p);
18         parent = (ScrollGame) p;
19         setLoop(0, 3);
20         racetime = 0;
21 }
```

Anschließend ersetzen wir die Ausgabe in die Konsole durch einen Methodenaufruf:

```
40 private void checkColor(Color col){
41
42     if(col.equals(Color.yellow)){
43         checkTime();
44     }
45 }
```

Und erzeugen die dazugehörige Methode:

```
60 private void checkTime(){
61     if(racetime==0){
62         racetime = System.currentTimeMillis();
63     }
64 }
```

Zeile 61: Da die Methode mehrfach aufgerufen würde (entspricht der Konsolenausgabe gelb, gelb, gelb,... von oben) prüfen wir hier ob wir unsere Variable schon mal gesetzt haben. Wenn nicht, erhält sie die aktuelle Zeit in Millisekunden.

Damit wir die gefahrene Zeit auch anzeigen lassen können, sind zwei weitere Modifikationen notwendig:

```

66 public String getRaceTime(){
67     double time = (double) (System.currentTimeMillis()-racetime);
68     time/=1000;
69     java.lang.Double d = new java.lang.Double(time);
70     return ("Time: " + d.toString());
71 }
72
73 @Override
74 public void drawObjects(Graphics g) {
75     super.drawObjects(g);
76     if(racetime>0){
77         g.setColor(Color.orange);
78         g.drawString(getRaceTime(), 250, 20);
79     }
80 }

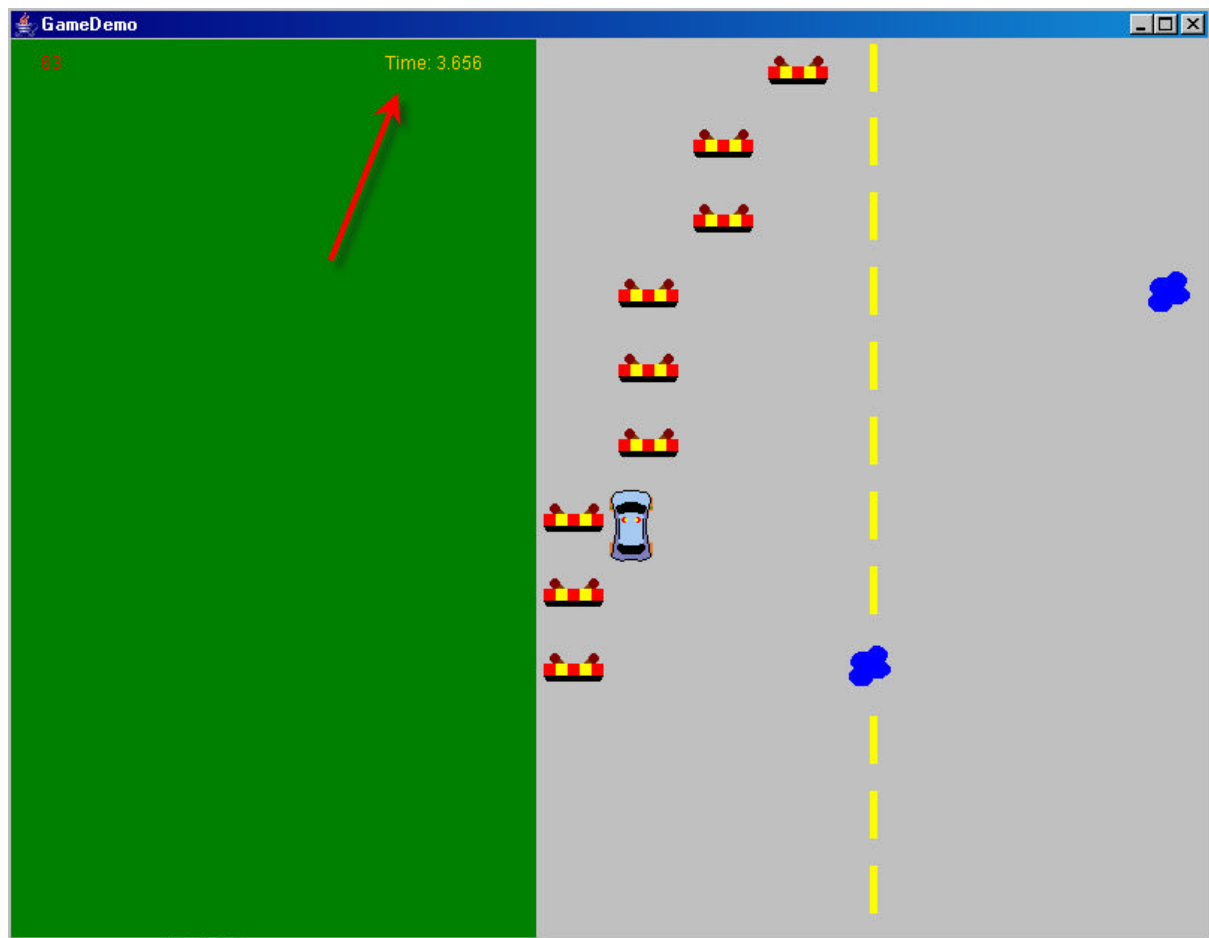
```

Zeile 66: Mit der Methode getRaceTime() errechnen wir die Differenz unseres Zeitstempels und der aktuellen Zeit in Sekunden und liefern diese als String zurück.

Zeile 69: Hier hat Eclipse steif und fest behauptet, dass Double vom Typ Rectangle2D wäre (vermutlich weil unsere Klasse Sprite von dieser Klasse erbt). Daher war es notwendig hier die lange Notation zu verwenden.

Zeile 74: Hier überschreiben wird die drawObjects-Methode von Sprite (super-Aufruf nicht vergessen) und lassen die Zeit auf dem Bildschirm ausgeben, sobald unsere Instanzvariable racetime nicht mehr Null ist, d. h. verwendet wird.

Und so sieht's aus:



Allerdings wird das Überfahren der 2. Linie noch nicht erkannt, das wollen wir jetzt hinzufügen. Dazu definieren wir eine neue Instanzvariable:

```
11 public class Car extends Sprite {
12
13     ScrollGame parent;
14     long racetime;
15     long stoprace;
16
17     public Car(BufferedImage[] i, double
18         super(i, x, y, delay, p);
19         parent = (ScrollGame) p;
20         setLoop(0, 3);
21 }
```

Diese neue Variable setzen wir, wenn das 2. Mal „gelb“ überfahren wird:

```
67 private void checkTime() {  
68     if(racetime==0){  
69         racetime = System.currentTimeMillis();  
70     }else{  
71         if(System.currentTimeMillis()-racetime>3000){  
72             stoprace = System.currentTimeMillis();  
73         }  
74     }  
75 }
```

Zeile 70: checkTime() wird immer dann aufgerufen, wenn ein gelbes Tile der Shadow-Map überfahren wird. Wie wir gesehen haben, geschieht dies für ein Tile mehrfach, weshalb wir die if-Bedingung eingebaut haben.

Zeile 71: Da in unserem Fall die gelben Tiles sehr weit auseinander liegen, nämlich nur am Anfang und Ende der Karte, prüfen wir hier im else-Zweig, ob das Überfahren eines gelben Tiles schon eine Weile her ist. Hier wurde diese Prüfung willkürlich auf 3000 Millisekunden (= 3 Sekunden) gesetzt. Wenn diese Bedingung greift, setzen wir in der Variable stoprace unseren Zeitstempel.

Jetzt noch eine Erweiterung der doLogic-Methode, damit das Überfahren der Ziellinie berücksichtigt wird:

```
23 public void doLogic(long delta) {  
24     super.doLogic(delta);  
25  
26     Color col1 = parent.getMap().getColorForPoint(new Pc  
27     Color col2 = parent.getMap().getColorForPoint(new Pc  
28     Color col3 = parent.getMap().getColorForPoint(new Pc  
29  
30     checkColor(col1);  
31     checkColor(col2);  
32     checkColor(col3);  
33  
34     if(stoprace>0){  
35         if(System.currentTimeMillis()-stoprace>1000){  
36             parent.setStarted(false);  
37         }  
38     }  
39 }
```

Zeile 34: Wir fügen eine Bedingung ein, die nur greift, wenn die stoprace-Variable ungleich Null ist.

Zeile 35: Da wir dann das Spiel nicht sofort abwürgen wollen, lassen wir es noch eine Sekunde weiterlaufen. Hier könnte natürlich noch mehr Logik eingebaut werden, für eine Bonus-Animation, etc. . Unser Code hier gewährleistet eigentlich nur, dass die Ziellinie noch vollständig überfahren wird. Mehr soll in diesem Tutorial erst einmal nicht eingebaut werden.

Als nächstes kommen die Öl- und Wasserflecken an die Reihe. Hier soll nur ein kleinerer Effekt hinterlegt werden. Wir wollen nur ein bisschen „am Auto ruckeln“. Ich habe bei Erstellung des Tutorials überlegt, wo ich diesen Code hinterlegen soll, da wir ja eigentlich nicht das Auto, sondern den sichtbaren Bereich der Karte bewegen. Ich habe mich nach einigem Hin- und Her dazu entschieden, diesen Code in der Klasse Car zu hinterlegen, da dort auch die Farbinformation ausgewertet wird. Da wir in dieser Klasse Zugriff auf die Karte benötigen, habe ich in der Klasse ScrollGame die entsprechende Getter-Methode erzeugt:

```
112 |
113 | public MapDisplay getMap(){
114 |     return map;
115 | }
```

Danach wird in der Klasse Car die Behandlung für „blaue Objekte“ geändert:

```
48 | private void checkColor(Color col){
49 |
50 |     if(col.equals(Color.yellow)){
51 |         checkTime();
52 |     }
53 |
54 |     if(col.equals(Color.blue)){
55 |         pushCar();
56 |     }
```

Hier wird der Methodenaufruf für die neu zu erzeugende Methode hinterlegt. Diese wollen wir gleich erzeugen, zuvor noch eine weitere Instanzvariable:

```
11 | public class Car extends Sprite {
12 |
13 |     ScrollGame parent;
14 |     long racetime;
15 |     long stoprace;
16 |     long lastpush;
17 | }
```

Jetzt die Methode:

```
68 private void pushCar() {  
69  
70     if(lastpush != 0 && System.currentTimeMillis()-lastpush<200){  
71         return;  
72     }  
73  
74     lastpush = System.currentTimeMillis();  
75  
76     int rnd = (int) (Math.random()*2);  
77     if(rnd==0){  
78         parent.getMap().setHorizontalSpeed(-500);  
79     }else{  
80         parent.getMap().setHorizontalSpeed(500);  
81     }  
82     parent.getMap().moveVisibleRectangle(500000);  
83 }  
84  
85
```

Zeile 70 – 72: Wie schon angemerkt wird für ein und dasselbe Objekt diese Methode mehrfach aufgerufen. Um diesem Effekt zu begegnen hier wieder eine Zeitmarke, die bewirkt, dass die Ausführung der Methode in gewissen Abständen erfolgen muss. Hier wurden 200 ms eingestellt.

Zeile 74: Zeitstempel des letzten Methodenaufrufs.

Zeile 76 – 81: Zufallsgesteuert verschieben wir das Auto ein Stück nach links oder rechts bzw. richtiger: Wir verschieben die Karte unter dem Auto. Hier verwenden wir daher die getMap()-Methode, die wir eben in der Klasse ScrollGame neu angelegt haben.

Zeile 82: Hier wird die Methode die das sichtbare Rechteck bewegt ausgeführt und so die Bewegung des Fahrzeugs simuliert. Als Zeitwert wird hier eine halbe Sekunde übergeben (Berechnung in ScrollGame bzw. GamePanel erfolgt in ns!). Durch diesen Methodenaufruf, der sozusagen außer der Reihe statt findet, wird ein zusätzlicher Schub simuliert, den die Öl- und Wasserflecken bewirken wollen.

Nachdem auch dies umgesetzt ist, wollen wir das Fahrzeug jetzt auch zerstören können. Dazu erweitern wir die Animationssequenz in car.gif um einige Images. Bei mir sieht das so aus:



Dies zieht natürlich einige Änderungen nach sich. Zunächst müssen wir beim Laden der Bilder jetzt mitteilen, dass die Datei mehrere Einzelbilder enthält. Dies machen wir beim Laden in der Methode doInitializations() in der Klasse ScrollGame:

```
35 @Override
36 protected void doInitializations() {
37
38     lib = SpriteLib.getInstance();
39     car = new Car(lib.getSprite("pics/car.gif", 12, 1), 400, 300, 200, this);
```

Außerdem müssen wir im Konstruktor der Klasse Car festlegen, dass nicht mehr über alle Bilder für die Animation des fahrenden Autos verwendet werden sollen, sondern vorerst nur über die ersten 4:

```
19 public Car(BufferedImage[] i, double x, double y, long delay, GamePanel p) {
20     super(i, x, y, delay, p);
21     parent = (ScrollGame) p;
22     setLoop(0, 3);
23 }
```

Danach funktioniert das Ganze wie bisher, ohne dass wir einen Unterschied feststellen können.

Jetzt werden wir in der Klasse Car aber noch einige Modifikationen vornehmen.

Zunächst definieren wir ein Instanzvariable, die uns anzeigt, ob das Auto gerade explodiert:

```
11 public class Car extends Sprite {
12
13     ScrollGame parent;
14     long racetime;
15     long stoprace;
16     long lastpush;
17
18     boolean explode = false;
19 }
```

Als nächstes hinterlegen wir in der Methode checkColor(..) neuen Code für die Farbe rot:

```
50 private void checkColor(Color col){
51
52     if(col.equals(Color.yellow)){
53         checkTime();
54     }
55
56     if(col.equals(Color.blue)){
57         pushCar();
58     }
59
60     if(col.equals(Color.green)){
61         slowDownCar();
62     }
63
64     if(col.equals(Color.red)){
65         explodeCar();
66     }
67
68 }
69
70 private void slowDownCar(){
71
72
73
74 private void pushCar(){
75
76
77
78 private void explodeCar(){
79     if(!explode){
80         explode = true;
81     }else{
82         return;
83     }
84     setLoop(4,11);
85
86 }
```

Zeile 91: In dieser Methode werden die Weichen für die Explosion gestellt. Da beim Überfahren eines Tiles die Bedingung (Farbe == rot) mehrfach aufgerufen werden kann, wird hier die neue Instanzvariable gesetzt und nicht weiter gemacht, wenn diese schon gesetzt wurde.

Zeile 97: Hier „zünden“ wir die Explosion indem wir das anzuzeigende Bild-Array verändern.

Danach erweitern wir die doLogic-Methode:

```
26 public void doLogic(long delta) {
27     super.doLogic(delta);
28
29     Color col1 = parent.getMap().getColorForPoint(new Point((int)getX(), (int)getY()));
30     Color col2 = parent.getMap().getColorForPoint(new Point((int)(getX()+getWidth()), (int)getY()));
31     Color col3 = parent.getMap().getColorForPoint(new Point((int)(getX()+getWidth()/2), (int)getY()));
32
33     checkColor(col1);
34     checkColor(col2);
35     checkColor(col3);
36
37     if(stoprace>0){
38         if(System.currentTimeMillis()-stoprace>1000){
39             parent.setStarted(false);
40         }
41     }
42
43     if(currentpic==11 && !remove){
44         remove = true;
45         checkTime();
46     }
47
48 }
```

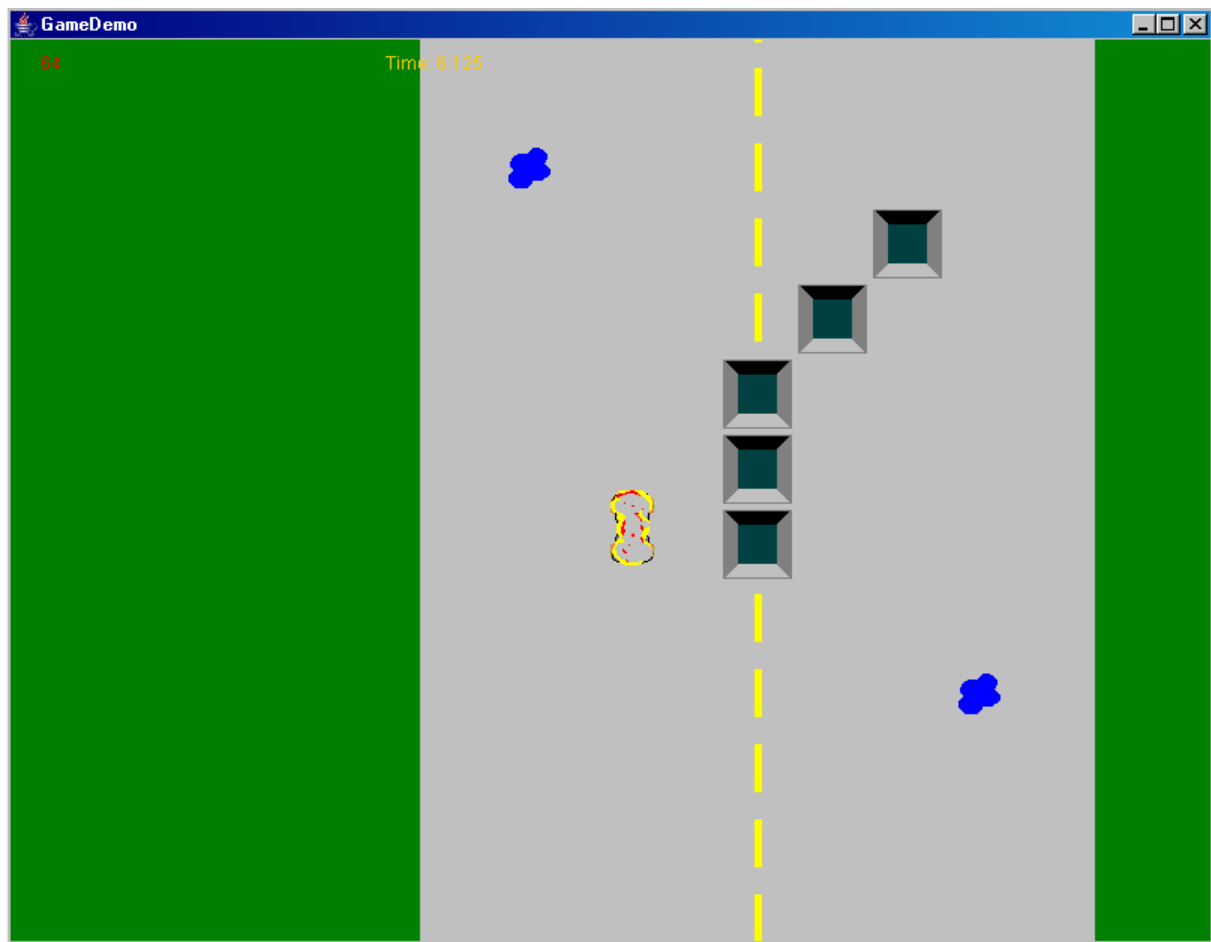
Sobald das letzte Bild angezeigt wird, setzen wir die von Sprite geerbte Variable `remove` auf `true`. Außerdem rufen wir die Methode `checkTime()` auf, die wir weiter oben für das Überfahren der gelben Linie geschrieben haben und die uns das Spiel nach 3 Sekunden beendet.

(Anmerkung: Die Abfrage auf `currentpic == 11` ist natürlich etwas fehlerträchtig, falls wir noch einmal etwas ändern. Mit `currentpic == pics.length-1` wären wir auf der sicheren Seite).

Jetzt müssen wir nur noch sicherstellen, dass unsere Auto nach der Anzeige des 11. Bildes nicht mehr angezeigt wird:

```
119 @Override
120 public void drawObjects(Graphics g) {
121
122     if(remove){
123         return;
124     }
125
126     super.drawObjects(g);
127     if(racetime>0){
128         g.setColor(Color.orange);
129         g.drawString(getRaceTime(), 250, 20);
130     }
131 }
```

Und so sieht's aus:



Das war's

Wie eingangs schon erwähnt wollte ich dieses Beispiel nicht bis ins Letzte ausprogrammieren und u. a. den Sound weglassen. Daher ist hier jetzt Schluss. Alles Weitere darf jeder selbst einprogrammieren. ☺

Nachträgliche Änderung

Eine Änderung hat sich jetzt doch noch ergeben. Während der Tests musste ich feststellen, dass der Thread der unseren GameLoop steuert nicht zufrieden stellend beendet wird. Daher habe ich 2 Modifikationen in der Klasse GamePanel eingebaut:

```
76 public void run() {  
77  
78     while(frame.isVisible()){  
79  
80         computeDelta();  
81  
82         if(isStarted()){  
83             checkKeys();  
84             doLogic();  
85             moveObjects();  
86         }  
87  
88         doPainting();  
89  
90         try {  
91             Thread.sleep(10);  
92         } catch (InterruptedException e) {}  
93  
94     }  
95     System.exit(0);  
96  
97 }
```

Zum einen habe ich die while-Bedingung so geändert, dass auf die Sichtbarkeit des Fensters abgefragt wird, was prinzipiell auch das Ende der while-Schleife bewirkt hat. Trotzdem war der Thread weiterhin aktiv. Daher habe ich in Zeile 95 ein `System.exit(0)` eingebaut, auch wenn hier jetzt mancher schimpfen mag, dass das kein guter Programmierstil ist. Wenn mir eine bessere Lösung einfällt, trage ich diese hier nach. ☺

Fertig

So, das war's – im Großen und Ganzen. Ich habe beim Schreiben festgestellt, dass man in dieses Thema noch viel mehr reinpacken könnte, wie z. B jar-Dateien, Webstart, etc.. Aber bis hier soll es das jetzt mal gewesen sein.

Dieses Tutorial ist letzten Endes sehr viel umfangreicher geworden als anfangs gedacht und eigentlich wollte ich kein Buch schreiben ☺ und irgendwann muß mal Schluss sein. Im Bezug auf dieses Tutorial ist das jetzt der Fall.

Natürlich sind die Beispiele im eigentlichen Sinn noch lange nicht fertig: Es gibt keine Punkte-Anzeige, keine Highscoreliste, etc.. Aber es war auch nicht die Absicht, das alles bis zum Ende umzusetzen. Hauptgrund war der Wunsch eine funktionierende Methode zur Erstellung eines Spiels in Java aufzuzeigen. Jeder der das Tutorial tatsächlich bis hierher gelesen hat, sollte jetzt hoffentlich keine Probleme mehr haben, noch fehlende Funktionalitäten selbst einzubauen.

Zum Schluss noch ein Dank an alle Mitglieder des Forums www.java-forum.org, die sich die Mühe gemacht haben, über den Entwurf dieses Tutorials zu lesen und mir Fehler und Verbesserungsvorschläge aufgezeigt haben.