

5 Ein erstes Spiel – Meteoritenalarm

Videogaming has been a normal choice of living-room entertainment; the emergence of convergent handheld devices extends this spur-of-the-moment option to people on the move.

— Steve Poole

5.1 Das Spielkonzept

Da wir bereits gelernt haben, wie man kleine UFOs über das Display fliegen lassen kann, handelt es sich bei unserem ersten Handyspiel naheliegend um einen einfachen Space-Shooter. Das Spiel soll den Einsatz der bisher besprochenen Bausteine demonstrieren. Bevor wir mit der Programmierung loslegen, überlegen wir uns ein geeignetes Spielkonzept:

- *Sie sind der Pilot eines Weltraumgleiters, der in der Nähe der Erdatmosphäre Erkundungsflüge durchführt. Dabei trifft der tapfere Pilot immer wieder auf entgegenkommende Meteoriten, denen er ausweichen muss. Daher wollen wir das Spiel auch schlicht "Meteors" nennen.¹*
- *Um den Schwierigkeitsgrad anzuheben, sollen feindliche Raumschiffe seitwärts über den Screen fliegen und den Spieler mit Bomben attackieren. Schließlich soll der Spieler die Möglichkeit haben, selbst Lasergeschosse abzufeuern, um sich verteidigen zu können.*
- *Meteoriten sind unzerstörbar, abgeschossene UFOs werden mit einem Score von 50 Punkten belohnt. Der Spieler hat drei Leben. Jede Berührung mit einem Meteor, einem feindlichen UFO oder einer Bombe führt zu einer Schadensmeldung und dem Abzug eines Lebens. Sind keine Leben mehr vorhanden, ist das Spiel vorbei. Durch das Drücken einer beliebigen Taste kann das Spiel erneut gestartet werden.*

¹ Im Gegensatz zu Asteroiden bezeichnet man mit Meteoriten diejenigen Materiebrocken, die nach Eintritt in die Erdatmosphäre durch die dadurch entstehende Reibung zunehmend erhitzt werden und mitunter einen Feuerschweif nach sich ziehen. Unser Pilot befindet sich an der äußersten Schicht der Erdatmosphäre.

Wir werden das Spiel schrittweise entwickeln und beginnen zunächst damit, die verwendeten Grafiken vorzustellen:

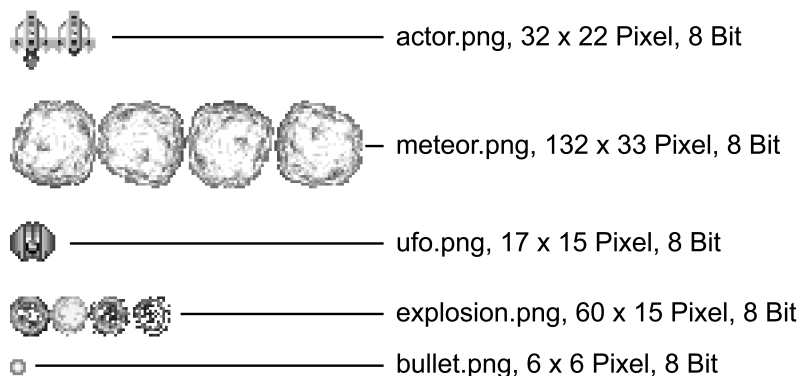


Abbildung 5.1 Übersicht der verwendeten Grafiken für Meteors

- **"actor.png"**: Das Raumschiff des Spielers besteht aus zwei Einzelgrafiken, jeweils mit/ohne Antriebsschweif.
- **"Meteor.png"**: Unsere Meteoriten sollen sich drehen, deshalb besteht diese Grafik aus vier Einzelbildern, die jeweils um 90° Grad gedreht sind.
- **"ufo.png"**: Das gegnerische Raumschiff.
- **"explosion.png"**: Bei einer Kollision des Raumschiffs soll eine Explosion zu sehen sein. Die gleiche Explosion verwenden wir für getroffene UFOs.
- **"bullet.png"**: Sowohl der Spieler als auch die UFOs können Lasergeschosse bzw. Bomben abfeuern.

5.2 Hilfsmittel bündeln – die Toolbox

Wir haben bereits ein ganzes Bündel an hilfreichen Methoden kennengelernt. Diese sollen nun nicht bestimmten Klassen zugeordnet, sondern zentral abgelegt werden, sodass wir jederzeit bei Bedarf darauf zugreifen können und redundanten Code vermeiden. Wir wollen dieser zentralen Klasse den Namen "Tb" geben (als Abkürzung für "Toolbox") und zunächst die folgenden Methoden dort unterbringen:

```
checkColPoint(...) {...} //Kollisionserkennung
checkCollision(...) {...} //Kollisionserkennung
checkColRect(...) {...} //Kollisionserkennung
checkColRectTol(...) {...} //Kollisionserkennung
drawOutlineString(...) {...} //Textausgabe
plotFrame(...) {...} //Animationen
getRnd(...) {...} //Zufallszahlen
```

Alle Methoden wurden im Einführungskapitel vorgestellt: Neben dem vollständigen Arsenal an Kollisionsmethoden wird in der Toolbox die Outline-Text-Methode untergebracht,

sodass wir jederzeit – unabhängig von der jeweiligen Vorder-/Hintergrundfarbe – lesbaren Text ausgeben können. Ebenso nützlich ist die `getRnd()`-Methode, um Zufallszahlen zu erzeugen. Schließlich wird noch die `plotFrame()`-Methode als Basis für animierte Grafiken hinzugefügt. Methodenkorpus und -signatur werden jeweils unverändert übernommen. Damit ergibt sich für die Toolbox-Klasse `Tb` die folgende Grundstruktur:

Listing 5.1 Toolbox `Tb.java`

```
import java.util.Random;
import javax.microedition.lcdui.Font;
import javax.microedition.lcdui.Graphics;
import javax.microedition.lcdui.Image;

//Tb = "Toolbox"
public class Tb {

    static int w; //Abmessungen des Screens
    static int h;
    static byte anchor=Graphics.TOP | Graphics.LEFT;
    static byte anchorM=Graphics.HCENTER|Graphics.TOP;
    static Random random;
    static Font font; //Unser Standardfont
    static int fontH; //Höhe des Standardfonts

    ...
    checkColPoint(...) {...}
    checkCollision(...) {...}
    checkColRect(...) {...}
    checkColRectTol(...) {...}
    drawOutlineString(...) {...}
    plotFrame(...) {...}
    getRnd(...) {...}
    ...
}
```

Als Klassenvariablen definieren wir über die Variablen `w` und `h` die Abmessungen des Displays. Dadurch können wir jederzeit bequem mittels `Tb.w` bzw. `Tb.h` die aktuelle Screengröße abfragen; die entsprechenden Abmessungen werden wir später über den `ActionCanvas`-Konstruktor direkt nach Start des Programms zuweisen, da wir erst *nach* der Initialisierung der Leinwand-Klasse die Auflösung des Displays abfragen können. Zur einfachen Positionierung von Grafiken und Texten deklarieren wir außerdem die Variablen `anchor` (Standard Positionierung) und `anchorM` (mittige Positionierung) fest. Schließlich legen wir für die Erzeugung von Zufallszahlen noch eine `Random`-Variable an und einen Standardfont (`font`), zusammen mit der Font-Höhe `fontH`.

Bisher haben wir die benötigten Bilder stets über den Konstruktor der Leinwand-Klasse geladen. Es bietet sich an, den Ladevorgang ebenfalls in die Toolbox zu verlagern:

```
static Image[] img;
static byte actorPNG=0;
static byte ufoPNG=1;
static byte meteorPNG=2;
static byte bulletPNG=3;
static byte explosionPNG=4;
static byte imgAnz=5;

public static void loadImages(){
    try{
        img = new Image[imgAnz];
    }
```

```

img[actorPNG] = Image.createImage("/actor.png");
img[ufoPNG] = Image.createImage("/ufo.png");
img[meteorPNG] = Image.createImage("/meteor.png");
img[bulletPNG] = Image.createImage("/bullet.png");
img[explosionPNG] = Image.createImage("/explosion.png");
} catch (Exception e){ System.out.println("Bildladefehler!"); }
}

```

Der Ladevorgang unterscheidet sich von den bisherigen Beispielen dadurch, dass wir als Neuerung das `Image`-Array `img` anlegen: Um statt Zahlen "sprechende" Namen für den Zugriff auf die einzelnen Indizes des Arrays zu bekommen, definieren wir für jedes Bild eine eigene Variable. Die Bildindizes werden fortlaufend durchnummeriert: So können wir über `imgAnz` die Anzahl der verwendeten Images ablesen. Über die `loadImages()`-Methode laden wir dann wie bisher die Bilder, dimensionieren aber zuvor über

```
img = new Image[imgAnz];
```

die Größe des `Image`-Arrays. Wozu nun überhaupt ein `Image`-Array?

Später werden wir weitere Klassen definieren, welche die einzelnen Sprites repräsentieren sollen, z.B. den Spieler, das UFO und die Meteoriten. Wir wollen vermeiden, dass pro `Sprite`-Klasse ein eigenes `Image`-Objekt definiert wird, um Platz zu sparen. Dennoch sollen pro Klasse die benötigten Bilder festgelegt werden, deshalb wählen wir den Umweg über die `Byte`-Bezeichner. So können wir der `UFO`-Klasse das `UFO-PNG` zuordnen, ohne gleich ein `Image`-Objekt zuweisen zu müssen. Denken Sie daran, dass es nicht nur ein `UFO` geben wird: Alle `UFO`-Objekte sollen daher auf ein einziges `Image`-Objekt zugreifen. Außerdem schadet es der Übersichtlichkeit Ihres Codes nicht, die Grafiken an einem Ort gebündelt zu sammeln. Wir werden diesen Sachverhalt später noch einmal aufgreifen. Eine Änderung muss allerdings jetzt schon vollzogen werden: Der `plotFrame()`-Methode der `Toolbox`-Klasse `Tb` darf nun kein `Image`-Objekt mehr übergeben werden, sondern nur noch der neue `Byte`-Referenzierer auf das `Image`. In der `drawImage()`-Methode wird über diesen Index das `Image` innerhalb des `img`-Arrays eindeutig identifiziert und kann angezeigt werden:

```

public static void plotFrame(
    byte image, //Verweis auf das Image-Objekt
    int width,
    int height,
    int frame,
    int x,
    int y,
    Graphics g
){
    g.setClip(x,y,width,height);
    g.drawImage(img[image], x-frame*width, y, anchor);
    g.setClip(0,0,w,h);
}

```

Den Ladevorgang der Bilder stoßen wir über eine weitere neue Methode an, der wir den bezeichnenden Namen `preloader()` geben:

```

public static boolean preloaderReady=false;
public static void preloader(){
    font = Font.getFont(Font.FACE_SYSTEM,Font.STYLE_BOLD,Font.SIZE_SMALL);
    fontH = font.getHeight();
    random = new Random();
}

```

```
Tb.loadImages();
Game.init(); //Game initialisieren
preloaderReady=true;
}
```

Die `preloader()`-Methode wird später über die `paint()`-Methode einmalig aufgerufen werden. Wir wollen in dieser Methode alles unterbringen, was mit Rechenzeit verbunden sein könnte. Wie Sie wissen, sind viele Handys nicht mit besonders schnellen CPUs ausgerüstet, sodass z.B. das Laden von Bildern eine gewisse Zeit in Anspruch nehmen kann. Wir definieren deshalb den statischen Flag `preloaderReady`, der auf `false` gesetzt wird. Nachdem alle Aufgaben in der `preloader()`-Methode abgearbeitet sind, wird dieser Flag auf `true` gesetzt. So können wir jederzeit extern auf einfache Weise feststellen, ob die Ladevorgänge bereits abgeschlossen sind. Die genaue Funktionsweise werden wir später anhand der Leinwand-Klasse vorstellen. Außerdem werden wir gegen Ende des Buches eine etwas elegantere Lösung eines Preloaders mit Fortschrittsanzeige präsentieren.

Innerhalb der `preloader()`-Methode weisen wir zunächst den Font zu, initialisieren das `Random`-Objekt und laden die Bilder – der eigentliche Zweck der `preloader()`-Methode. Außerdem stoßen wir auf den Aufruf `Game.init()`: Wir werden hierüber die spielbezogenen Daten initialisieren. Dazu später mehr.

Weiterhin bringen wir in der `Toolbox`-Klasse die bekannte `getFrame()`-Methode unter, zusammen mit der neuen `pumpTimer()`-Methode:

```
static byte timer=0;
public static void pumpTimer(){
    timer++;
    if (timer>127) timer=0;
}

getFrame(...){}
```

Wie Sie wissen, benötigt die `getFrame()`-Methode eine Zählervariable, die kontinuierlich hochgezählt werden muss. Dies regeln wir mithilfe der `pumpTimer()`-Methode, die wir später an einer zentralen Stelle aufrufen werden.

Abschließend gönnen wir der `Toolbox`-Klasse noch eine `clearScreen()`-Methode:

```
public static void clearScreen(int bgColor, Graphics g){
    g.setColor(bgColor);
    g.fillRect(0,0,w,h);
}
```

Damit können wir jederzeit den Screen "löschen". Die Methode greift auf die aktuellen Screenshotdimensionen zu, die in den statischen `Tb.w`- und `Tb.h`-Variablen abgelegt sind.

Damit ist die `Toolbox`-Klasse bereits gut gefüllt mit nützlichen Methoden; Sie können nach dem Patchwork-Prinzip die Klasse jederzeit um eigene Methoden ergänzen – beachten Sie dabei, dass wir keine Instanz der Klasse erzeugen werden, um Ressourcen zu sparen. Deshalb sollten alle Methoden und Variablen als `static` deklariert sein, auch um ein einfaches Zugreifen über `Tb.meineVariable` bzw. `Tb.meineMethode()` zu ermöglichen. Statische Variablen und Methoden können schneller ausgeführt werden als objektbezogene Variablen und Methoden.

5.3 Meteors-Grundgerüst

Die nötigen Bausteine haben wir jetzt beisammen. Nun müssen wir uns noch eine geeignete Form für die Organisation der Quelltexte des Spiels überlegen. Das Grundgerüst soll auf dem bisher Gelernten aufbauen und neben der zentralen MIDlet-Klasse natürlich auch über ein Canvas verfügen: Über die Leinwand-Klasse `ActionCanvas` bzw. `Ac` wird naheliegender Input Handling gesteuert sowie eine einfache Möglichkeit integriert, das Spiel über eine "Exit"-Taste zu verlassen. Daneben wollen wir die neue `Game`-Klasse hinzufügen, in der wir die Steuerung des Programmablaufs unterbringen. Die für die zahlreichen Sprites benötigten Methoden werden über eine eigene Klasse namens `SpriteTool` organisiert. Außerdem gehört zu unserem Grundgerüst die bereits vorgestellte `Toolbox`-Klasse `Tb`, in der wir die nützlichen Methoden aus den vergangenen Kapiteln untergebracht haben. Sie finden das komplette Grundgerüst des Spiels auch im SRC-Ordner unter "MobileGames_10_Meteors_Grundgeruest".

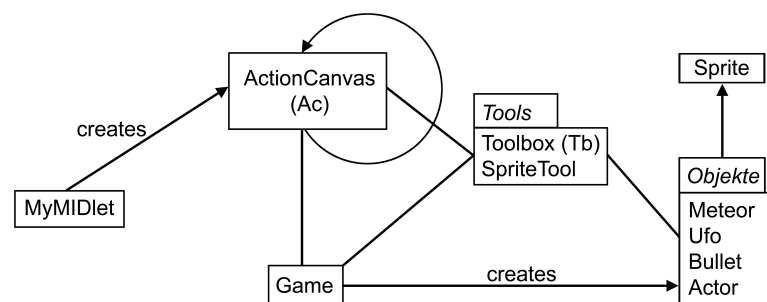


Abbildung 5.2 Klassenübersicht Meteors

Die `MyMIDlet`-Klasse bleibt gegenüber den bisherigen Beispielen unverändert. Auch die `ActionCanvas`-Klasse dürfte uns bekannt vorkommen, insbesondere zwecks Tastaturabfrage werden wir häufig auf diese Klasse zugreifen müssen. Deshalb verwenden wir für die Klasse den kürzeren Namen "Ac", um uns etwas Tipparbeit zu ersparen. Die Initialisierung der Klasse über die `MyMIDlet`-Klasse muss natürlich entsprechend angepasst werden:

```
Ac ac = new Ac(this);
```

Zusätzlich übergeben wir dem Konstruktor der Leinwand-Klasse eine Instanz des MIDlets, um über die `notifyDestroyed()`-Methode der MIDlet-Klasse die Anwendung jederzeit verlassen zu können. Wir setzen das Grundgerüst der `Ac`-Klasse wie folgt auf:

```
import javax.microedition.lcdui.Canvas;
import javax.microedition.lcdui.Graphics;

public class Ac extends Canvas implements Runnable{
    static MyMIDlet midlet;
    static final int MAX_GAME_SPEED=40;
    static boolean up=false,
                 down=false,
                 left=false,
```